ДМИТРИЙ ОСИПОВ

bhv®

Базы данных и Delphi

Теория и практика

+ ПРОБНЫЕ ВЕРСИИ ПО

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

- ВВЕДЕНИЕ В ТЕОРИЮ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ
- ПРОЕКТИРОВАНИЕ ЛОКАЛЬНЫХ И КЛИЕНТ-СЕРВЕРНЫХ БАЗ ДАННЫХ
- СТРУКТУРИРОВАННЫЙ ЯЗЫК ЗАПРОСОВ SQL
- РАСШИРЯЕМЫЙ ЯЗЫК РАЗМЕТКИ XML
- ТЕХНОЛОГИИ DBEXPRESS, INTERBASE EXPRESS И ADO
- МНОГОУРОВНЕВЫЕ БАЗЫ ДАННЫХ НА ОСНОВЕ DATASNAP
- НЕСТАНДАРТНЫЕ ПРИЕМЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ БАЗ ДАННЫХ

Дмитрий Осипов

Базы данных и Delphi

Теория и практика

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.06
ББК 32.973.26-018.2
О-74

Осипов Д. Л.

О-74 Базы данных и Delphi. Теория и практика. — СПб.: БХВ-Петербург, 2011. — 752 с.: ил. + DVD — (Профессиональное программирование)

ISBN 978-5-9775-0659-5

Книга основана на материалах лекций и практических занятий, разработанных автором, и объединяет теоретические основы и практические аспекты разработки реляционных баз данных. В первой части рассмотрена концепция реляционных баз данных: реляционная модель данных, жизненный цикл информационной системы, концептуальное и логическое моделирование БД, нормализация отношений, обеспечение многопользовательского доступа к данным, вопросы обеспечения безопасности БД, языки SQL и XML и др. Во второй части описаны возможности современных версий Delphi в области разработки приложений баз данных: подробное описание технологий dbExpress, Interbase Express и ADO, особенности использования компонентов управления данными визуальной библиотеки Delphi, механизм разработки многоуровневых приложений на основе технологии DataSnap, порядок создания отчетов для печати и многое другое. На DVD размещены дополнительные главы, а также материалы и пробные версии ПО компании Embarcadero, включая Delphi XE.

Для студентов и программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.01.11.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 60,63.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.005770.05.09
от 26.05.2009 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0659-5

© Осипов Д. Л., 2011
© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Введение **1**

ЧАСТЬ I. ВВЕДЕНИЕ В РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ **5**

Глава 1. Системы, основанные на файлах **7**

Принцип построения систем, основанных на файлах 9
Недостатки систем, основанных на файлах 10
Пути устранения недостатков систем, основанных на файлах 13
Резюме 14

Глава 2. Эволюция моделей реализации данных **15**

Необходимость моделирования 17
Иерархическая модель 18
Сетевая модель 20
Попытки разработки стандарта БД 21
Реляционная модель 23
Объектно-ориентированная модель 25
Резюме 26

Глава 3. Функции и компоненты СУБД **27**

Функциональные обязанности СУБД 27
Компоненты СУБД 29
Архитектурные решения доступа к БД 32
Файл-сервер 32
Клиент-сервер 34
Многоуровневые решения 36
Резюме 37

Глава 4. Реляционная модель данных **38**

Сущность и атрибуты 39
Тип данных и домен 41

Связь ... 43
Реляционная таблица ... 45
Ключ ... 47
Целостность данных ... 48
Целостность доменов ... 48
Целостность сущностей ... 49
Ссылочная целостность ... 50
Корпоративная целостность ... 50
Реляционная алгебра ... 51
Резюме ... 56

Глава 5. Модель "сущность-связь" ... 57

Сущности и атрибуты в ER-модели ... 58
Подтипы сущностей ... 61
Связи в ER-модели ... 63
Сильные и слабые связи ... 66
Рекурсивная связь ... 67
Связи высокого порядка ... 68
Вариации ER-моделей ... 70
Резюме ... 72

Глава 6. Нормализация ... 73

Первая нормальная форма ... 76
Функциональная зависимость атрибутов ... 79
Порядок определения первичного ключа ... 81
Вторая нормальная форма ... 82
Третья нормальная форма ... 84
Нормальная форма Бойса—Кодда ... 86
Четвертая нормальная форма ... 87
Пятая нормальная форма ... 88
Резюме ... 90

Глава 7. Индексирование ... 91

Индексы на основе хеширования ... 93
Хеш-функции ... 95
Хеширование текстовых данных ... 96
Борьба с коллизиями ... 96
Индексы на основе B-деревьев ... 98
Битовые индексы ... 103
Правила назначения вторичных индексов ... 103
Резюме ... 104

Глава 8. Транзакции и параллельная обработка данных 105

Понятие транзакции 106
Проблемы параллельного доступа к данным 108
Потерянные обновления 108
Неактуальные чтения ("грязное чтение") 109
Несогласованная обработка 110
Чтение строк-фантомов 111
Управление параллельными транзакциями 112
Метод блокировок 113
Метод временных меток 115
Метод двухфазной блокировки 116
Оптимистический подход 116
Уровни изоляции SQL-транзакций 117
Резюме 118

Глава 9. Проектирование БД 119

Понятие информационной системы 119
Этапы жизненного цикла базы данных 122
Планирование разработки БД 123
Определение и анализ требований к системе 123
Проектирование БД 127
Выбор СУБД 130
Создание прикладного программного обеспечения 131
Тестирование 132
Реализация 133
Эксплуатация и сопровождение 133
Резюме 134

Глава 10. Защита БД 135

Откуда исходят угрозы 136
Правила защиты БД 139
Аутентификация и авторизация 140
Криптографическая защита 141
Резервное копирование 143
Аудит событий безопасности 144
Модернизация системного и прикладного ПО 145
Доступ к данным только при посредничестве представлений и хранимых процедур 145
Резюме 146

Глава 11. Знакомство с SQL 147

Назначение SQL 149
Типы данных SQL 150
Предопределенные типы данных 152
Непредопределенные типы данных 157
Массив 158
Мультимножество 158
Пользовательский тип 159
Другие типы 160
Определение констант 160
Преобразование данных 161
Операторы 162
Встроенные функции 163
Резюме 164

Глава 12. Построение запросов 165

Порядок сортировки — *ORDER BY* 167
Условие отбора данных — *WHERE* 167
Сравнение 168
Попадание в диапазон — *BETWEEN* 169
Соответствие шаблону — *LIKE* 169
Проверка неопределенного значения — *IS NULL* 170
Принадлежность множеству — *IN*, *ALL*, *ANY*, *SOME* 170
Предикат существования — *EXISTS* 171
Многотабличные запросы 171
Слияние *UNION* 173
Объединение *ON* 173
Объединение *USING* 176
Агрегирующие функции 176
Группировка данных — *GROUP BY* 177
Дополнительная фильтрация группы строк — *HAVING* 178
Оконные функции 178
Рекурсивные запросы 182
Резюме 184

Глава 13. Манипулирование данными и управление транзакциями 185

Язык манипулирования данными DML 185
Вставка, инструкция *INSERT* 185
Редактирование, инструкция *UPDATE* 187
Удаление, инструкция *DELETE* 188
Слияние данных, инструкция *MERGE* 190

Транзакции ... 191
Диагностирование ошибок в работе транзакции ... 194
Настройка уровня изоляции ... 196
Резюме ... 197

Глава 14. Определение данных в SQL ... 198

Базы данных (схемы) ... 198
Таблицы ... 200
Индексы ... 205
Домены ... 208
Представления (виртуальные таблицы) ... 209
Хранимые процедуры ... 210
Триггеры ... 212
Курсоры ... 217
Управление доступом к данным ... 220
Управление наборами привилегий ... 221
Предоставление привилегий ... 222
Лишение привилегий ... 224
Резюме ... 225

Глава 15. Основы XML ... 226

Правильность и допустимость документа ... 229
Построение простейшего документа XML ... 229
Элементы ... 230
Специальные символы ... 232
Атрибуты ... 232
Пространство имен ... 233
Определение документа ... 236
DTD ... 236
Хранение DTD во внешнем файле ... 240
Резюме ... 242

Глава 16. XML Schemas ... 243

Определение элемента <*element*> ... 246
Тип данных ... 248
Производные типы <*simpleType*> ... 248
Глобальное и локальное объявление ... 252
Квалифицирование элемента ... 252
Ограничения на число элементов ... 253
Значение по умолчанию и фиксированное значение ... 253
Создание сложных структур <*complexType*> ... 254

Определение атрибута <*attribute*> ... 256
Подключение XML-схемы к документу ... 257
Пример схемы computers.xsd ... 258
Пример документа computers.xml ... 261
Резюме ... 262

ЧАСТЬ II. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ БД В DELPHI ... 263

Глава 17. Концепция приложения БД Delphi ... 265

Пример простейшего приложения БД ... 265
Архитектура приложения БД ... 269
Общая характеристика компонентов соединения с БД ... 271
Общая характеристика компонентов наборов данных ... 271
Источник данных ... 273
Общая характеристика компонентов управления данными ... 274
Резюме ... 275

Глава 18. Универсальный набор данных *TDataSet* ... 276

Функционал набора данных ... 278
Открытие и закрытие набора данных ... 278
Обновление набора данных ... 279
Перемещение по набору данных ... 280
Создание закладок и переход к закладке ... 282
Состояние набора данных ... 284
Редактирование записей в наборе ... 286
Организация доступа к отдельному полю ... 288
Вычисляемые поля ... 290
Агрегатное поле ... 290
Фильтрация набора данных ... 291
Организация поиска данных ... 292
Обработка событий ... 294
Кэширование данных ... 295
Взаимодействие с элементами управления ... 296
Резюме ... 297

Глава 19. Общая характеристика полей набора данных ... 298

Поле таблицы — класс *TField* ... 298
Классификация полей по функциональному назначению ... 301
Классификация полей по типу хранимых данных ... 301

Обращение к полю ... 303
Доступ к данным поля ... 304
Низкоуровневый доступ к данным ... 306
Преобразование типа данных ... 307
Размер поля ... 308
Значение по умолчанию ... 309
Ограничения на ввод данных ... 310
Маска ввода ... 311
Индексные поля ... 312
Отображение данных ... 312
Обработка событий ... 314
Поля подстановки ... 315
Вычисляемые поля ... 322
Числовые поля, класс *TNumericField* ... 324
Поля целых чисел *TLargeintField*, *TIntegerField*, *TSmallintField*, *TWordField*, *TAutoIncField* ... 326
Поля вещественных чисел *TFloatField* и *TCurrencyField* ... 326
Бинарно-кодированные десятичные поля *TBCDField* и *TFMTBCDField* ... 327
Текстовые поля, *TStringField* ... 328
Поле глобального идентификатора, *TGuidField* ... 329
Логическое поле, *TBooleanField* ... 330
Бинарные поля *TBinaryField*, *TBytesField* и *TVarBytesField* ... 330
Дата и время, поля *TDateTimeField*, *TDateField* и *TTimeField* ... 330
Дата и время, поле *TSQLTimeStampField* ... 331
Поля больших бинарных объектов, *TBlobField*, *TGraphicField*, *TMemoField* и *TWildMemoField* ... 331
Композитные поля, *TObjectField* ... 338
Резюме ... 340

Глава 20. Вспомогательные классы набора данных ... 341

Коллекция *TParams* и динамический SQL ... 341
Параметр *TParam* ... 344
Описание структуры таблицы и ее индексов ... 347
Коллекция полей структуры таблицы *TFieldDefs* ... 349
Определение поля, класс *TFieldDef* ... 351
Коллекция структуры индексов таблицы, класс *TIndexDefs* ... 352
Определение индекса, класс *TIndexDef* ... 353
Пример создания таблицы ... 354
Резюме ... 356

Глава 21. Клиентский набор данных *TClientDataSet* ... 357

Проекты БД, основанные на файлах ... 358
Загрузка и сохранение данных ... 361

Требования к структуре XML ... 365
Хранение данных в компоненте ... 366
Управление данными ... 367
Работа с индексами ... 370
Определение диапазона ... 373
Ограничения ... 376
Применение агрегирующих функций ... 376
Агрегат, класс *TAggregate* ... 377
Агрегатные поля *TAggregateField* ... 378
Место *TClientDataSet* в многоуровневых проектах БД ... 379
Работа в клиентских приложениях DataSnap ранних версий ... 380
Сохранение данных на сервере ... 381
Отмена изменений ... 382
Применение точек сохранения ... 383
Обработка ошибок ... 383
Оптимизация пакета с данными ... 385
Обновление данных ... 386
Выполнение команд SQL ... 387
Резюме ... 387

Глава 22. Введение в технологию ADO ... 388

Взаимодействие ADO и компонентов VCL ... 389
Строка соединения ADO ... 392
Соединение с хранилищем данных, компонент *TADOConnection* ... 393
Установка соединения ... 393
Пример соединения без регистрации пользователя ... 397
Регистрация пользователя ... 402
Разрыв соединения ... 408
Информирование о соединении ... 408
Отправка команд ... 410
Управление подчиненными компонентами dbGo ... 413
Транзакции ... 413
Кэширование ... 415
Сервисные методы модуля ADODB ... 416
Резюме ... 418

Глава 23. Наборы данных ADO ... 419

Командный объект — *TADOCommand* ... 419
Коллекция ошибок *Errors* и объект ошибки *Error* ... 424
Набор данных ADO, компонент *TADODataSet* ... 425
Интерфейс множества записей *_Recordset* ... 426
Доступ к хранилищу данных ... 427

Выбор библиотеки курсора ... 428
Редактирование данных ... 429
Перемещение по множеству строк ... 430
Особенности применения поля BCD ... 432
События *TCustomADODataSet* ... 432
Работа с индексами ... 436
Сортировка записей ... 436
Поиск данных ... 436
Особенности изоляции транзакций ... 437
Фильтрация множества записей ... 438
Кэширование записей ... 438
Фильтрация записей в кэше ... 440
Организация отложенного обновления данных в ADO ... 441
Портфельный режим обработки данных ... 443
Управление данными ADO в стиле Delphi ... 444
Таблица *TADOTable* ... 445
Организация отношения "главная – подчиненная таблица" ... 446
Запрос *TADOQuery* ... 448
Хранимая процедура *TADOStoredProc* ... 450
Резюме ... 452

Глава 24. Доступ к БД InterBase ... 453

Доступ к базе данных, компонент *TIBDatabase* ... 454
Выбор диалекта ISQL ... 455
Создание и уничтожение базы данных ... 456
Соединение с базой данных ... 457
Регистрация пользователя ... 459
Разрыв соединения ... 460
Информирование о составе БД ... 462
Управление транзакциями ... 463
Контроль за событиями ... 464
Совместная работа с SQL монитором ... 464
Транзакция, компонент *TIBTransaction* ... 465
Управление транзакцией ... 466
Тайм-аут транзакции ... 468
Диагностика состояния транзакции ... 469
Параметры транзакции ... 469
Информация об объектах БД, компонент *TIBExtract* ... 471
События InterBase, компонент *TIBEvents* ... 473
Информация о БД, компонент *TIBDatabaseInfo* ... 474
Монитор SQL, *TSQLMonitor* ... 476
Файл инициализации БД, *TIBDataBaseINI* ... 476
Резюме ... 478

Глава 25. Наборы данных InterBase 479

Инструкция SQL, компонент *TIBSQL* 480
Подготовка к работе 480
Обслуживание полученного набора данных 482
Наборы данных InterBase, компонент *TIBDataSet* 483
Подготовка к работе 483
Обработка событий 485
Генератор значений 486
Особенности редактирования данных 488
Работа в режиме кэширования обновлений 488
Перемещение по записям 490
Фильтрация данных 490
Запрос, компонент *TIBQuery* 491
Редактирование данных, доступных только для чтения 492
Хранимая процедура, компонент *TIBStoredProc* 492
Таблица, компонент *TIBTable* 494
Экспорт-импорт данных 494
Модифицируемый запрос, компонент *TIBUpdateSQL* 496
Диалог фильтрации, компонент *TIBFilterDialog* 498
Резюме 500

Глава 26. Администрирование сервера InterBase 501

Свойства сервера, *TIBServerProperties* 505
Сервис лицензирования, *TIBLicensingService* 510
Конфигурирование сервера, *TIBConfigService* 511
Ведение журнала транзакций 514
Протокол работы сервера, *TIBLogService* 517
Статистика, *TIBStatisticalService* 518
Проверка БД, *TBDValidationService* 520
Управление учетными записями, *TIBSecurityService* 522
Резервное копирование и восстановление, *TIBBackupService* и *TIBRestoreService* .. 528
Резюме 532

Глава 27. Технология доступа к данным dbExpress 533

Соединение с сервером БД, компонент *TSQLConnection* 534
Настройка компонента 535
Управление соединением 537
Создание БД 539
Создание подключения в Data Explorer 539
Пример подключения 542
Управление подчиненными наборами данных 547

Управление транзакциями ... 547
Выполнение SQL-инструкций ... 548
Ограничение числа выполняющихся инструкций ... 549
Информирование о БД ... 549
Аутентификация пользователя в DataSnap ... 551
Мониторинг работы программы, *TSQLMonitor* ... 551
Резюме ... 553

Глава 28. Наборы данных dbExpress ... 554

Базовый класс *TCustomSQLDataSet* ... 555
Формирование инструкций SQL ... 556
Получение системной информации ... 557
Набор данных dbExpress, компонент *TSQLDataSet* ... 561
Особенности обслуживания BLOB-полей ... 563
Таблица *TSQLTable* ... 563
Запрос *TSQLQuery* ... 565
Хранимая процедура *TSQLStoredProc* ... 565
Простой набор данных *TSimpleDataSet* ... 567
Резюме ... 569

Глава 29. Интерфейс приложения и компоненты Data Access ... 570

Источник данных — компонент *TDataSource* ... 570
Общие черты компонентов отображения данных ... 572
Сетка базы данных — компонент *TDBGrid* ... 573
Одновременный выбор нескольких строк ... 575
Колонки сетки ... 576
Коллекция колонок — класс *TDBGridColumns* ... 576
Колонка — класс *TColumn* ... 578
Обработка событий ... 582
События прорисовки данных ... 583
Статический текст — компонент *TDBText* ... 585
Строка ввода — компонент *TDBEdit* ... 586
Многострочный редактор — *TDBMemo* ... 587
Редактор расширенного формата — *TDBRichEdit* ... 588
Изображение — компонент *TDBImage* ... 588
Список — *TDBListBox* ... 589
Комбинированный список — *TDBComboBox* ... 590
Группа переключателей — *TDBRadioGroup* ... 590
Флажок — *TDBCheckBox* ... 591
Компонент — *TDBCtrlGrid* ... 591
Поля подстановки ... 594
Список подстановки — *TDBLookupListBox* ... 595
Комбинированный список подстановки — *TDBLookupComboBox* ... 596

Навигатор — *TDBNavigator* ... 596
Резюме ... 598

Глава 30. Нестандартные решения для стандартных компонентов ... 599

Компоненты-списки ... 599
Компонент *TListView* ... 604
Сетка, компонент *TStringGrid* ... 607
Иерархические данные ... 610
Многотабличное представление иерархических данных ... 610
Рекурсивная связь ... 611
Инициализация проекта ... 614
Новая запись ... 615
Сбор данных ... 617
Очистка данных ... 618
Редактирование записи ... 619
Удаление записи ... 620
Сортировка узлов ... 621
Переподчинение узлов ... 623
Резюме ... 628

Глава 31. Многоуровневые БД на основе DataSnap ... 629

Архитектура трехзвенного проекта БД DataSnap ... 630
Сервер *TDSServer* ... 633
Класс сервера *TDSServerClass* ... 637
Обмен данными между клиентом и сервером, компоненты *TDSTCPServerTransport* и *TDSHTTPService* ... 638
Аутентификация, *TDSHTTPServiceAuthenticationManager* ... 641
Метод сервера *TSqlServerMethod* ... 641
Пример проекта DataSnap ... 642
Регистрация службы ... 648
Подготовка клиентского приложения ... 649
Подключение сервера приложений к БД ... 651
Получение данных клиентским приложением ... 652
Реализация на сервере метода вставки новой записи ... 654
Доступ к методу вставки записи на стороне клиента ... 655
Архитектура DataSnap, совместимая со старыми клиентскими приложениями ... 655
Интерфейс *IAPPServer* ... 656
Провайдер набора данных, компонент *TDataSetProvider* ... 658
Подключение к провайдеру набора данных, компонент *TDSProviderConnection* ... 664
Клиентское приложение БД на основе *IAppServer* ... 665
Механизм обратного вызова ... 666
Резюме ... 669

Глава 32. Управление службой сервера приложений DataSnap 670

Менеджер управления службами 671
Работа со службой 672
Пример управляющего приложения SCP 673
Доработка сервиса DataSnap 680
Создание модуля панели управления 682
Резюме 686

Глава 33. Отчеты Rave Reports 687

Обзор компонентов Rave Reports 687
Соединение *TRvCustomConnection* 688
Проект *TRvProject* 688
Системный компонент *TRvSystem* 689
Компоненты экспорта отчета в файл 690
Пример работы с редактором Rave Reports 690
Вызов отчета из приложения 696
Резюме 697

Глава 34. Разработка динамических библиотек для проектов БД 698

Общая характеристика DLL 698
Создание шаблона динамической библиотеки в Delphi 700
Экспортирование функций DLL 702
Пример простой DLL 703
Взаимодействие динамической библиотеки с проектом 704
Размещение файла DLL 705
Явная загрузка DLL 705
Неявная загрузка DLL 706
Пример DLL универсального генератора отчетов 707
Резюме 712

Заключение 713

Список литературы 715

Содержимое DVD 717

Предметный указатель 719

МАТЕРИАЛ НА ДИСКЕ

Генератор отчетов своими руками

Технические характеристики принтера
Описание принтера в Delphi, класс TPrinter
Выбор принтера
Управление страницей документа
Формирование и отправка задания на печать
Отмена задания
Печать многострочного текста
Пример универсального отчета
Окно предварительного просмотра
Вывод отчета на печать
Резюме

Диаграммы и графики

Компонент TChart
Хранение графиков в диаграмме
Базовый класс графиков — TChartSeries
Точки графика, класс TChartValueList
Методы вставки новой точки
Удаление точек, очистка графика
Оформление графика
Взаимодействие с мышью
Метки графика — TSeriesMarks
Легенда диаграммы — TChartLegend
Координатные оси диаграммы — TChartAxis
Масштабирование
Многостраничные диаграммы
Экспорт диаграмм
Печать диаграммы
Упорядочивание графиков внутри диаграммы
Обеспечение объемного вида диаграммы
Особенности компонента TDBChart
Графики диаграммы в качестве набора данных
Резюме

ОТЧЕТЫ В MICROSOFT OFFICE

Константы Microsoft Office
Обращение к серверу автоматизации
Понятие коллекции
Текстовый процессор Microsoft Word
Приложение Word — Application
Коллекция документов Documents и документ Document
Параметры страницы, объект PageSetup
Область документа Range
Объект выделения Selection
Коллекция абзацев Paragraphs и абзац документа Paragraph
Коллекция списков Lists и список List
Форматирование списка
Коллекция списков
Коллекция разделов Sections и раздел документа Section
Коллекция таблиц Tables и таблица Table
Внедрение в документ OLE-объектов
Оформление документа
Граница области
Пример универсального генератора отчетов
Электронные таблицы Microsoft Excel
Приложение Excel — Application
Коллекция WorkBooks и книга Workbook
Листы Excel
Универсальная коллекция листов Sheets
Общие свойства листов Worksheet и Chart
Параметры страницы — объект PageSetup
Особенности листа электронной таблицы Worksheet
Коллекция диаграмм Charts и лист диаграммы Chart
Область ячеек Range
Слияние ячеек
Оформление ячеек
Заливка области — Interior
Усовершенствование универсального генератора отчетов
Резюме

Введение

Интересно, что послужило первопричиной столь страстного желания человека систематизировать большое количество информации? Быть может, толчком стала необходимость строгого учета сбитых птеродактилей или поверженных динозавров? Почему бы и нет? Коротая время между схватками с саблезубыми тиграми и себе подобными, первобытный человек кроме чувства голода, возможно, испытывал еще одно — тягу к систематизации и учету. Куда деваться, когда первобытная супруга постоянно твердила на ухо: "А у Петровых из 35-й пещеры на ужин филе из звероящера". Не верите? Доказательством тому служат многочисленные наскальные рисунки, до сих пор находимые на стоянках наших пращуров. Несчастные мужчины еще не знали систем счисления, посему учет им приходилось вести в графической форме. Поймал динозавра — нарисовал, поймал второго — нарисовал еще одну картинку. Не поймал ни одного — нарисовал трех… В итоге по наскальным пиктограммам можно выяснить, сколько было съедено антилоп, бизонов и упомянутых динозавров за отчетный период.

Минуло несколько тысячелетий. Мамонты в страхе попасть на ужин к нашим предкам бежали на север, а человек перебрался из уютных, хорошо проветриваемых пещер в каменные дома древнего Вавилона. Первая же попытка вести учет по старинке столкнулась с противодействием лучшей половины человечества. Только представьте себе, как по улицам ночного Вавилона жены гоняли скалками первых разработчиков БД, попытавшихся по "пещерной" привычке начертать полугодовой отчет (в виде стада овечек) на недавно побеленной стене. Загнанным в угол мужчинам не оставалось больше ничего, кроме как вновь двинуть вперед научный прогресс. В результате была изобретена первая система счисления, а записи отныне стали наносить на каменные скрижали. Сами понимаете, что камни ворочать довольно тяжело, зато наверняка все программисты тех времен были в прекрасной физической форме. Представьте себе вавилонскую картотеку на пару-другую тысяч каменных плит, среди которых вы второй месяц ищете приходный ордер от 2000 года до рождества Христова! Хочешь, не хочешь, а мускулы станут как сталь. Впрочем, ни стали, ни, тем более физической культуры, в то время еще не знали, но прототип первой базы данных уже сформировали. Таким образом, предлагаю считать, что профессия проектировщика баз данных входит в десятку древнейших профессий… Ну да ладно, мы немного отвлеклись.

Практически рядом с древним Вавилоном (как в пространстве, так и во времени) над вопросами учета и контроля бились египтяне. Их система хранения данных базировалась на папирусах и росписях пирамид. Такого размаха не достигала еще ни одна из цивилизаций. Древнеегипетских наработок хватило вплоть до нашего времени. И по сей день, археологи пытаются восстановить целостность данных, разыскивая утраченные богатства и дешифруя иероглифы.

Небывалых успехов достигли математики из далекой Индии, примерно два тысячелетия назад они придумали цифру ноль. Ноль — не только число в современной системе счисления, но и предвестник краеугольного для БД понятия — пустоты NULL. Без NULL невозможно корректно представить данные ни в одной из современных систем управления базами данных. Я докажу вам это на страницах книги, которую вы держите в руках.

Мощнейшие цивилизации Европы и Азии несколько тысячелетий развивали системы хранения и представления данных. До сих пор ходят легенды об одном из самых великих хранилищ древности — Александрийской библиотеке. Несколько веков ее полки пополнялись новой информацией, пока бóльшая часть книг не погибла в пожаре войны между Клеопатрой и ее братом Птолемеем XIII. Если бы тогда существовало понятие создания резервной копии данных! Человечество бы не утратило бесценные знания.

Годы сменялись десятилетиями, десятилетия складывались в века. Человечество изобрело печатный станок, покорило просторы океана, научилось парить в небесах, побывало на Луне, и лишь потом открыло величайшую тайну — научилось хранить информацию в базах данных.

На сегодняшний день не осталось ни одной области знаний, где не применялись бы компьютерные базы данных (БД). Наука, образование, экономика, электронная коммерция, медицина, статистика, военное дело, — список можно продолжать до бесконечности. Базы данных сопровождают человека на протяжении всей жизни. Появление на свет ребенка, учеба в школе и вузе, получение паспорта или водительских прав, посещение врача, покупки в магазинах, поиск книги в библиотеке, открытие счета в банке, в современном информационном обществе ни одно из этих и многих других событий не обходится без появления очередной электронной пометки в памяти многочисленных компьютеров. Одним словом, базы данных так глубоко вошли в нашу повседневную жизнь, что мы зачастую воспринимаем их как что-то само собой разумеющееся.

В перечне современного программного обеспечения базы данных входят в десятку самых востребованных программных продуктов и служат источником неплохого заработка для профессиональных программистов. Судите сами, без хранения и учета данных сегодня обойтись весьма сложно. Многочисленные магазины, склады, страховые агентства, отделы кадров, бухгалтерии, учебные заведения и много других предприятий и организаций остро нуждаются в разработанных специально для них БД. И спрос все еще превышает предложение.

Создать эффективную базу данных весьма непросто, даже если она предназначена для обслуживания незначительных объемов данных и подлежит эксплуатации на домашнем компьютере. Сложность проекта возрастает на порядок, когда возникает задача разработать жизнеспособный коммерческий продукт, с которым будут одновременно работать десятки пользователей. Именно поэтому главная задача этой книги — вооружить читателя знаниями о порядке проектирования реляционных баз данных и разработке приложений БД на одном из самых совершенных и успешных языков программирования — Delphi.

Столь популярный сегодня язык программирования своим рождением обязан профессору Цюрихской высшей технической школы, блестящему ученому, обладателю премии Тюринга — Никлаусу Вирту (*Niklaus Wirth*). Язык изначально предназначался для обучения студентов основам структурного программирования. Уже с самых первых дней своего существования язык Pascal (именно так назывался в те времена предшественник Delphi) был обречен на широчайшую популярность благодаря ряду своих неоспоримых достоинств: гибкости и надежности; простоте и наглядности конструкций; контролю правильности исходного кода на этапе компиляции; возможности построения новых типов данных и многим другим качествам, с которыми мы далее познакомимся.

Книга состоит из двух частей. В первой части излагаются теоретические основы современной реляционной модели данных. Здесь вы узнаете:

- историю появления реляционной модели данных;
- функциональные обязанности и архитектуру систем управления базами данных (СУБД);
- особенности организации доступа к БД;
- основы реляционной модели данных;
- особенности концептуального моделирования БД;
- порядок нормализации реляционных таблиц;
- правила индексирования данных;
- обеспечение многопользовательского доступа к данным и правильную организацию транзакций;
- особенности всех этапов проектирования БД;
- вопросы обеспечения безопасности данных;
- структурированный язык запросов SQL;
- расширяемый язык разметки XML.

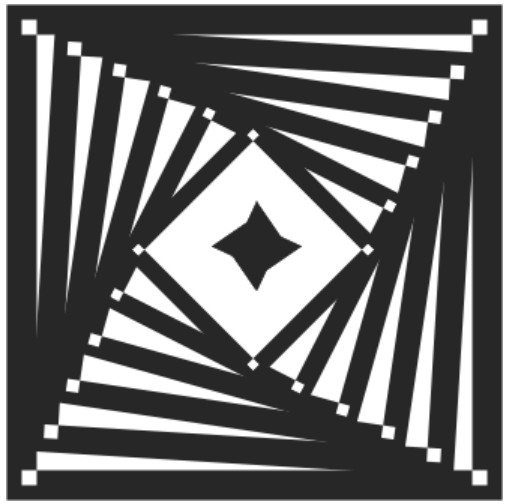
Вторая часть книги посвящена вопросам проектирования приложений баз данных на языке программирования Delphi. Здесь вы найдете исчерпывающую информацию о:

- концепции построения приложения БД Delphi;
- универсальном наборе данных `TDataSet` и полях набора данных;
- технологиях доступа к данным ADO, InterBase и dbExpress;
- порядке построения настольных и клиент-серверных БД;

- применении технологии DataSnap в проектах БД;
- организации взаимодействия с приложениями Microsoft Office.

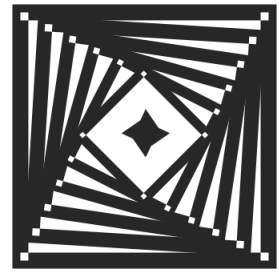
На прилагаемом к книге DVD находятся дополнительные главы, в которых рассмотрена подготовка отчетов в Microsoft Office, создание собственного генератора отчетов, а также построение диаграмм и графиков.

Так как основная направленность этой книги — практическая разработка БД, то читатель, имеющий хотя бы небольшой опыт программирования в Delphi, найдет здесь все необходимое, чтобы самостоятельно спроектировать реляционную базу данных и разработать клиентское приложение для совместной работы с БД. Самое главное, что получит читатель после изучения предложенного материала, — владение методологией работы с любой БД. Книга не ограничивает читателя в выборе целевой СУБД, поэтому вашу базу данных можно развернуть как на основе простейших настольных систем (например, Microsoft Access), так и на фундаменте профессиональных многопользовательских клиент-серверных систем (таких как InterBase, FireBird, Blackfish SQL, Microsoft SQL Server, Oracle, MySQL и т. п.).



ЧАСТЬ I

Введение в реляционные базы данных



Глава 1

Системы, основанные на файлах

Разговор о первых прототипах современных баз данных — системах, основанных на файлах, начнем с небольшого, но весьма поучительного экскурса в историю. В пятидесятых годах XX века вокруг электронно-вычислительных машин постоянно "вился рой" инженеров, математиков и программистов. Инженеры меняли радиодетали, математики выдумывали формулы, а программисты замыкали круг — закладывали в ЭВМ программы, которые жгли радиодетали. Огромное количество обслуживающего персонала, "роящегося" вокруг первых вычислительных устройств, в некоторой степени роднило обслуживание ЭВМ с возведением Вавилонской башни. Процесс разработки и загрузки прикладного программного обеспечения тех времен был достоин кисти Сальвадора Дали. Наивный заказчик расчетов приходил к программисту-алгоритмисту и просил, чтобы машина ответила, сколько на ее взгляд будет равняться 2+2. Алгоритмист в глубокой задумчивости рисовал алгоритм и передавал его программисту, перекладывающему алгоритм на низкоуровневый язык машинных команд, чтобы ЭВМ смогла "усвоить" программу, которую техники набивали на перфоленту или на перфокарты. Данные с перфокарт полдня загружались в память лампового монстра, затем он пару-другую раз зависал, шипел, кряхтел и, наконец, к всеобщей радости выдавал распечатку. К этому моменту времени результат уже никого не интересовал, т. к. заказчик все успел посчитать в уме…

К шестидесятым годам XX века картина приобрела больше мажорных нот. ЭВМ подверглись усовершенствованию настолько, что уже были способны не только воодушевлять писателей-фантастов и согревать воздух, но и выполнять некоторые полезные операции, в первую очередь связанные с утомительными математическими расчетами. Повысилась не только производительность процессоров, но и на качественно другой уровень перешли периферийные устройства. Теперь почти отпала необходимость вставки между пользователем и машиной гильдии разношерстных посредников. Пользователь просто садился за терминал и получал возможность прямого общения с ЭВМ. Заказчики — люди разные, и далеко не всех интересовали расчеты траекторий баллистических ракет. Многие искали применения вычислительных машин в другой не менее важной области — хранении и обработке больших массивов данных. Судите сами: XX век — век информации; в офисах корпораций, в правительственных учреждения, в архивах и библиотеках

хранились миллионы тонн бумаги с данными о чем угодно, начиная с роста поголовья пингвинов в Антарктике и заканчивая налоговыми декларациями от горячо любимых сограждан. Все это необходимо не просто хранить, а еще и максимально быстро обрабатывать с возможностью получения аналитических выводов, графиков и статистики. До сих пор все попытки систематизации колоссальных объемов данных, представленных на бумажных носителях, были в некотором родстве с сизифовым трудом. И вдруг, о чудо, наконец, у человечества появился шанс вкатить камень на вершину горы!

В обязанности вычислительных машин, кроме проведения расчетов, вменили еще один элемент — хранение и обработку больших объемов данных. Только бесконечно наивный пользователь (метко называемый в народе "чайником") может предположить, что ЭВМ сразу же подставила человечеству свое плечо и в мановение ока все бумажные архивы превратились во всех отношениях в совершенные базы данных. Не тут-то было! Вычислительные машины — лишь инструмент, который работает ровно так, как его научат программисты. Поэтому примерно на стыке 50-х и 60-х годов XX века перед программистами была поставлена очередная задача — научить машины обслуживать большие объемы данных.

Мы с вами все учились сами, а некоторые из нас даже пытались учить других. Любой участник образовательного процесса, находящийся в здравом уме и доброй памяти, понимает, что научить можно только тому, что в совершенстве знаешь сам. А теперь ответьте на вопрос: "Что в начале шестидесятых программисты знали о хранении больших банков данных?" Прав окажется тот, кто скажет — практически ничего! Не удивительно, что на первых этапах становления такой области знаний, как базы данных, все пошли по особому пути, который получил название *системы, основанные на файлах* (file-based system) или просто *системы файлов*. К чему столько скепсиса? Поймете к концу главы, а пока рассмотрим историю болезни прототипов современных БД — файловых систем.

Особенность человеческого образа мышления заключается в том, что при поиске решений новых проблем в первую очередь мы пытаемся применить подходы, которые были придуманы ранее. Примеров в истории предостаточно. Вспоминайте сами, разработчики первых летательных аппаратов весьма настойчиво пытались обучить их махать крылами. Эффективность подобного решения они обычно проверяли сами, поэтому долго не жили. В большинстве своем программисты также народ прямолинейный. Разработчики прообразов баз данных при поиске решения посмотрели вокруг и увидели, что везде, где есть бумажные архивы, данные хранятся в картотеках. Возьмем обычную городскую библиотеку тех лет. Здесь каждой книге соответствует отдельная бумажная карточка, в которой отражены данные об авторе, названии книги, годе издания, месте хранения и т. д. Карточки систематизированы по областям знаний и упорядочены по алфавиту. Любой грамотный посетитель, придя в библиотеку, достаточно быстро (за пару-тройку часов перелопатив с тысячу карточек) выяснял, что интересующей его книги там нет, и с гордо поднятой головой уходил восвояси... Плохо это или хорошо, но на тот момент времени другого, более рационального решения проблемы архивного дела не существовало. Поэтому воодушевленные программисты, не теряя ни одной секун-

ды на "лишние" размышления, самоотверженно приступили к обучению самолетов махать крыльями — взялись за "лобовое" проектирование электронных аналогов бумажных картотек.

Замечание

Почему системам, основанным на файлах, мы уделяем столько времени и не переходим сразу к изучению баз данных? По двум причинам. Во-первых, понимание сути недостатков, присущих файловым системам, позволяет избежать их в дальнейшем. Во-вторых, даже сегодня в спектре программного обеспечения есть место для приложений, построенных на основе файлов.

Шутки шутками, а пионером в области разработки файловых систем стала фирма IBM, разработавшая операционную систему OS/360. В те годы это был мощный эволюционный скачок, т. к. прежде в качестве устройств внешней памяти в основном использовали стримеры — накопители на магнитной ленте, а машины 360-й серии были снабжены новейшими контроллерами жестких дисков. С тех пор в тезаурус пользователей и программистов вошло новое понятие — файл.

Принцип построения систем, основанных на файлах

Рассмотрим в общих чертах идею построения систем, основанных на файлах. Допустим, что мы планируем создать программку, хранящую сведения об именах и днях рождений наших сотрудников. Получивший столь ответственное задание программист поступает следующим образом.

Во-первых, он готовит структуру, подходящую для хранения указанных данных. Для этого программист просматривает список персонала и выясняет максимальное число символов в фамилии, имени и отчестве (допустим, это значения: 20, 15 и 15 байтов). Затем программист выделяет какое-то число байтов для хранения даты (пусть это будет 8 байтов). Узнав все необходимые размерности, разработчик описывает структуру непосредственно в коде программы (рис. 1.1).

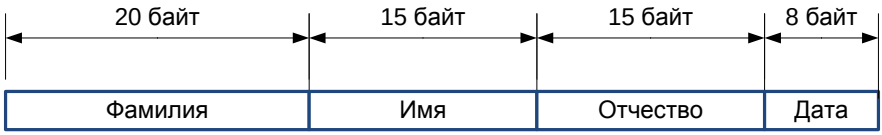


Рис. 1.1. Структура для хранения одной записи в файле

Во-вторых, программист разрабатывает процедуры, осуществляющие основные операции по работе с файлом. Как минимум это добавление новой записи, редактирование, удаление и чтение записи. Ни одну из этих операций невозможно осуществить без знания размерности и состава полей исходной структуры. При вставке

новой строки в файл к нему следует добавить 20+15+15+8=58 байтов, причем процедура добавления должна знать, что фамилия начинается с 1-го байта, имя с 21-го и т. д. При просмотре файла необходимо осуществлять последовательные операции чтения порциями по 58 байтов, опять же процедура чтения должна иметь информацию, сколько байт отводится тому или иному полю. Как вы догадываетесь, операции редактирования и удаления не являются исключением из правил и также нуждаются в знаниях об исходной структуре. К счастью, эти сведения искать не нужно — все данные о составе полей нашей программе хорошо известны, ведь описание структуры спрятано внутри нее.

Недостатки систем, основанных на файлах

Поначалу у разработчиков систем, основанных на файлах, дела шли весьма неплохо. Пользователям очень нравилось, что работа с электронными картотеками была схожа с работой с бумажными архивами. В свою очередь, программисты были довольны, что они сравнительно легко зарабатывают себе на жизнь.

Идиллия продолжалась недолго. Очень скоро на, казалось, безоблачном горизонте забрезжили грозовые тучи — стали проявляться отрицательные стороны "лобового" подхода разработчиков первых прототипов баз данных. Из всего сонма недостатков особо выделяются пять проблем (рис. 1.2).

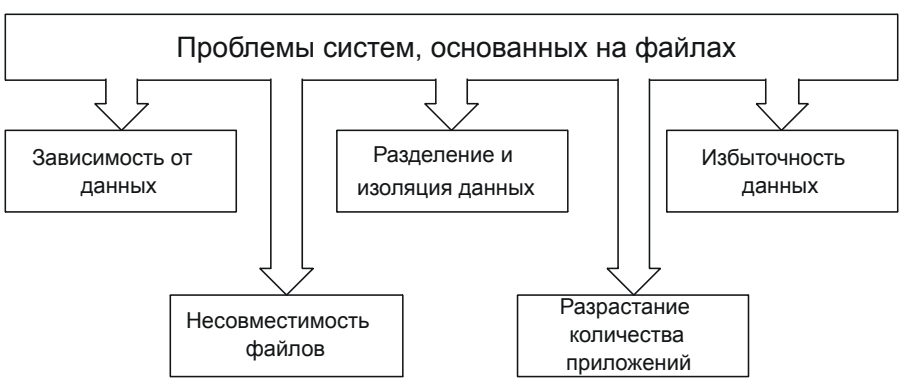


Рис. 1.2. Проблемы систем, основанных на файлах

- *Зависимость от данных*. Уже первая проблема, с которой столкнулись разработчики файловых систем, не предвещала ничего хорошего. Допустим, что нам требуется осуществить элементарную операцию — увеличить на один символ размер поля, отвечающего за хранение фамилии. Оказалось, что даже незначительное усовершенствование структуры влечет за собой ком дополнительных трудностей. Судите сами, изменение размера или состава полей записи типизи-

рованного файла вынуждает нас не только перекомпилировать исполняемый файл, но и написать одноразовую программу-конвертор, которая должна преобразовать старые данные к новому формату. То же самое произойдет не только при изменении размерностей, но и в случае, если мы попытаемся добавить новые или удалить лишние поля с данными.

Таким образом, первой болезнью, поразившей первые прототипы баз данных, стала зависимость системы, основанной на файлах от данных.

- *Разделение и изоляция данных*. Файловые системы объединяют в себе десятки отдельных файлов с данными. В одном файле хранится информация о сотрудниках фирмы, в другом — сведения о клиентах, в третьем — перечень предоставляемых услуг, в четвертом — список заказов и т. д. Для извлечения логически связанных данных (допустим о клиентах и их заказах) программисту приходилось писать сложные алгоритмы синхронного чтения из двух файлов. С увеличением числа файлов, вовлекаемых в итоговый отчет, сложность задачи возрастала в арифметической прогрессии. Задача извлечения взаимоувязанных данных из десятка файлов могла стать непосильной не только для программиста, но и для вычислительных машин тех времен. Масла в огонь дополнительно подливал тот факт, что данные могли быть разделены между отделами и службами предприятия — в отделе кадров находились данные о сотрудниках, в отделе продаж — о заказах и т. п. В 1960-х годах удельный вес предприятий, чьи машины были объединены локальными вычислительными сетями, стремился к нулю, поэтому данные были еще и изолированы друг от друга. А теперь представьте себе программиста тех времен, мечущегося по организации, в надежде объединить данные в единое целое…
- *Избыточность (дублирование) данных*. С усовершенствованием микропроцессорной техники многие руководители предприятий стали отказываться от покупок больших ЭВМ и начали отдавать свои предпочтения более дешевым мини-ЭВМ, расставляя их по отделам и службам своих организаций. Подобное, во многом правильное решение имело и свои отрицательные стороны, одна из них — вынужденный отказ от централизованного хранения данных. Децентрализованное хранение данных систем, основанных на файлах, на нескольких машинах приводила к тому, что одна и та же информация повторялась на магнитных носителях многих мини-ЭВМ, разбросанных по учреждению. В этом случае страшна не столько избыточность данных, сколько нарушение непротиворечивости данных предприятия — на всех машинах должны храниться идентичные копии данных.

  Избыточность данных порождает целый букет проблем и проблемок.
- *Противоречивость данных*. На одной из рабочих станций предприятия хранится устаревший номер телефона вашего контрагента. На второй этого номера вовсе нет. На третьем компьютере, за счет ошибки оператора, там находится телефон его бабушки. В результате ни один из звонков не достигает цели. Как следствие, фирма терпит убытки.

- *Аномалии данных*. Некорректные данные влекут за собой шлейф дополнительных неприятностей: аномалий добавления, редактирования и удаления записей. Какая из аномалий способна принести больше печали в наш офис, судите сами. Допустим, у нашей фирмы появился новый, не жалеющий денег, оптовый покупатель. Данные нового клиента следует ввести сразу в несколько систем файлов (отел сбыта, личная картотека главного менеджера, бухгалтерия и т. д.). Если все сделано безошибочно, то порядок обеспечен… Но если таких покупателей несколько, то я могу гарантировать, что где-нибудь, кто-нибудь спутает пару цифр в номерах счетов (только вслушайтесь в названия "БИК", "ИНН", "ОКПО", услышав эти аббревиатуры, не грех и ошибиться). В результате платеж в пару миллионов уходит на чужой расчетный счет. После долгого судебного разбирательства и уплаты неустоек вы, наконец, выясняете, в чем причина сбоя, но к этому времени вам уже все равно... С редактированием данных в избыточных системах дела также обстоят далеко не лучшим образом. В идеале следует нанять отдельного сотрудника, задачей которого станет регулярная "пробежка" по всем структурным подразделениям компании, чтобы исправить почтовый адрес (номер телефона, дату рождения, номер счета или что-нибудь в этом духе) главного спонсора фирмы. В результате поздравительная открытка, отправленная в канун очередного юбилея, не попадет к адресату, а в отместку "благодарный" юбиляр не перечислит вашей компании давно обещанные (и так необходимые сейчас) финансовые вливания. Впрочем, считайте, что вам крупно повезло, ведь уязвленное самолюбие может привести и к более серьезным последствиям… Аномалия удаления в состоянии принести не меньшие неприятности. Как вы думаете, как скажется на финансовом состоянии фирмы тот факт, что она станет выплачивать ежегодные премии давно уволенному менеджеру? Это печальное событие произойдет только потому, что в бухгалтерии забудут вычеркнуть всего одну строку с данными.

  В "доисторических" системах файлов избыточность данных вынужденно присутствовала и в рамках одного-единственного проекта. Допустим, что от нас потребуют дополнить программу "Дни рождений сотрудников" еще одним информационным полем — местом работы. В результате в файле появятся многократно дублирующиеся данные, например Петров — Бухгалтерия, Иванов — Бухгалтерия и т. д. Исследования файловых систем тех времен показали, что до 60% хранящихся в них данных избыточны, в этом убедилась компания North American Rockwell, участвовавшая в 1960-х годах в программе высадки астронавтов на Луну. Учитывая астрономическую стоимость первых жестких дисков тех времен, это были весьма непродуктивные расходы.
- *Несовместимость файлов*. Структура файлов с данными определялась не только разработчиками программного обеспечения, но и языками программирования, применяемыми в тех или иных организациях. Построение файла, описанного на языке Algol, могло принципиально отличаться от структур, генерируемых программами на PL/1, ADA или каким-нибудь еще средством разработки приложений тех времен. Более того, дополнительные ограничения

вносились из-за особенностей архитектурных решений тех или иных ЭВМ. Помножьте это на специфичные черты различных операционных систем. В результате файлы, выходящие из-под "пера" программистов, зачастую становились несовместимыми, хотя и содержали практически одно и то же описание данных.

- *Разрастание количества приложений*. Сами по себе данные не представляют никакого интереса. Представьте, что у вас имеется файл с несортированными телефонными номерами жителей миллионного города — это хорошая новость. А теперь плохая — у вас нет средств упорядочивания и поиска данных. В результате цена таким данным — ломаный грош, ну-ка найдите номер телефона гражданина Иванова, затерявшегося где-то среди сотен тысяч других номеров… Данные нужно не только хранить, но и уметь представлять пользователю в удобном формате. А пожеланий у пользователей ни счесть, одним требуется, чтобы списки заказов упорядочивались по алфавиту, другим по дате заказа, третьим хотелось бы, чтобы имелась возможность сортировки записей по денежной сумме. Старуха, затребовавшая у Золотой рыбки перечень услуг (начиная от тривиального корыта и заканчивая царским троном), — просто ангел в сравнении с запросами к данным у биржевого аналитика, главного бухгалтера завода или заведующего гипермаркетом. Идеи и пожелания пользователей сыплются на программиста как из рога изобилия и никто кроме него не ведает, что каждый новый запрос приводит к цепной реакции — бесконечной переработке исходного приложения. В конце концов, головная программа обрастает скопищем утилит и "утилиток", что, в свою очередь, необратимо ведет к хаосу.

Пути устранения недостатков систем, основанных на файлах

Сегодня системы, основанные на файлах, практически не используются, исключение составляют небольшие по числу записей хранилища, состоящие из одного-двух файлов данных. Чтобы не повторять прошлые ошибки, проектировщики БД сделали нужные выводы:

Во-первых, разработчики в принципе отказались от хранения физической структуры данных в коде приложений. Вместо этого описание данных стали выносить в отдельное хранилище, называемое *системным каталогом* (system catalog). Таким образом, во всех современных БД помимо собственно хранимых в них данных еще имеются метаданные (данные о данных). Если внешняя программа обладает возможностью чтения метаданных, то она без труда сможет получить доступ к хранимой в БД информации.

Во-вторых, стали предпринимать активные попытки стандартизировать способы описания и хранения данных. Наличие стандарта, единого для всех разработчиков, значительно упростило доступ к данным.

В-третьих, возникла необходимость создания единого универсального языка, позволяющего производить с данными наиболее важные операции: (вставки, редактирования, удаления и просмотра).

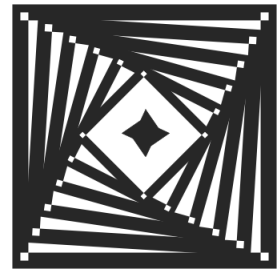
В результате на смену морально устаревшим системам файлов пришли свободные от недостатков своих предшественников базы данных.

ЗАМЕЧАНИЕ

База данных (*database*) — некоторая организованная совокупность взаимосвязанных данных, относящаяся к определенной предметной области, причем данные хранятся в таком виде, что они независимы от программ, использующих эти данные.

Резюме

При желании перечень недостатков систем, основанных на файлах, можно продолжить: сложность в администрировании, отсутствие даже элементарных понятий безопасности, отсутствие развитых процедур управления и контроля и т. п. Но и перечисленных фактов было вполне достаточно, чтобы разработчики программного обеспечения поняли, что файловые системы ведут в тупик и следует задуматься над поиском альтернативных решений организации обработки и хранения больших массивов данных. Вместе с тем, многие идеи, рожденные в период проектирования файловых систем, не канули в Лету и остаются достойными нашего внимания и сегодня. В частности, это построение малых проектов, рассчитанных на эксплуатацию на одном компьютере и не нуждающихся в услугах драйверов сторонних производителей.



Глава 2

Эволюция моделей реализации данных

Период расцвета систем, основанных на файлах, был недолгим. Все возрастающие требования к хранению и обработке больших массивов данных столкнулись с ограниченными возможностями файловых систем. Как бы это не показалось банальным, но в основе всех недостатков систем, основанных на файлах, лежит всего одна причина: описание данных прячется внутри приложений, а не хранится отдельно и независимо от них. В итоге исключается возможность корректного доступа к данным извне, единственным инструментом обработки данных остаются сами приложения, ведь только они "знают" формат структуры данных.

Нельзя сказать, что для программистов 1960-х эта причина стала откровением, просто до сих пор всех устраивало сложившееся положение вещей. Теперь же разработчикам программного обеспечения пришлось взять тайм-аут и задуматься над путями устранения возникших противоречий. Хорошим стимулом тому были грандиозные проекты шестидесятых годов XX века. Чего только стоила амбициозная цель НАСА первым отправить астронавта на Луну. Соединенные Штаты Америки, получившие поучительный щелчок по носу от СССР в космической гонке, не жалели ни денег, ни ресурсов для взятия реванша. А борьба за космос — это не только битва реактивных двигателей. Здесь в равной степени важны достижения практически во всех областях науки и техники, и среди них далеко не на последнем месте стоят компьютерные технологии. Попробуйте представить себе астронавта с логарифмической линейкой, прикидывающего в уме, сколько надо подать топлива в форсунки двигателя, чтобы в очередной раз не проскочить мимо Луны… Поэтому американскому космическому агентству потребовались мощные ЭВМ и современное программное обеспечение. В эту отрасль начали вкачивать огромные денежные суммы. В результате проектированием первых систем баз данных занялось достаточно много корпораций и научно-исследовательских лабораторий. Так что, если хотите, движущей силой современных баз данных в 1960-х годах стали уязвленное самолюбие, помноженное на отменное финансирование.

После высадки человека на Луну минуло четыре десятка лет, неплохой срок для подведения итогов развития баз данных. Беру на себя смелость утверждать, что за этот период базы данных шагнули так же далеко, как и космонавтика. На

рис. 2.1 отражены этапы развития моделей баз данных. Это так называемые модели реализации, т. е. модели, ориентированные на получение ответа на вопрос: "Каким образом следует описывать структуры данных?" Единственное исключение составляет понятийная модель "сущность-связь", это ближайший союзник реляционной модели, но отвечающий не за реализацию, а за логику будущей БД. Оговорюсь сразу, что ведущие специалисты в области баз данных до сегодняшнего дня не пришли к единому мнению даже в признании факта существования представленной на рис. 2.1 объектно-ориентированной модели.

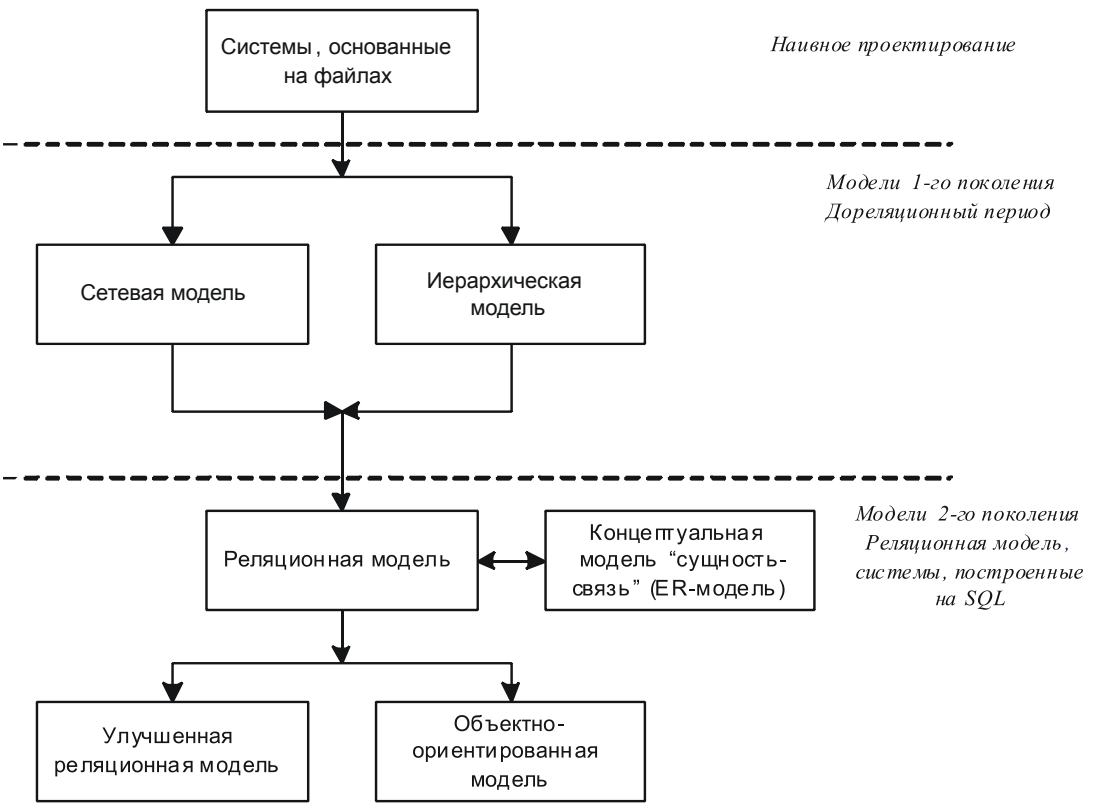


Рис. 2.1. Эволюция моделей реализации данных

Системы, основанные на файлах, мы обсудили в предыдущей главе. Файловые системы лишь с большой натяжкой можно причислять к базам данных, поэтому мы отнесем их к этапу наивного проектирования. Хотя сами понимаете, что все в этом мире относительно и спустя двадцать—тридцать лет нас с вами кто-то тоже отнесет к какому-нибудь этапу. Ну, это мы отвлеклись, вернемся к моделям. Дореляционный период заместил файловые системы, он продолжался с начала 1960-х по первую половину 1970-х годов. Началом его конца стала публикация знаменитой статьи Э. Кодда о реляционных банках данных. Закат одного дня — это лишь пре-

людия нового рассвета. Эпоха реляционной модели настала вместе с выходом в свет знаменитого прототипа реляционной базы данных — System-R.

ЗАМЕЧАНИЕ

К моделям 1-го поколения относится еще один вид БД — системы с *инвертированными списками*. Указанная модель реализации БД не получила широкого распространения, поэтому в этой книге мы их даже не рассматриваем.

Необходимость моделирования

Беседуя об эволюции систем управления базами данных, мы уже несколько раз употребили термин "модель". Почему так акцентируется на этом внимание? Для чего нужен процесс моделирования при создании БД?

Построение различного рода моделей широко применяется в различных областях науки и техники. В авиастроении, прежде чем поднять в небо дорогостоящий самолет, его уменьшенная модель проходит исследование в аэродинамической трубе. Конструкторы летательного аппарата выясняют, насколько удачны контуры будущего истребителя или пассажирского лайнера, не рискуя жизнями экипажа. Физики и астрономы, изучая особенности протекания ядерных реакций на Солнце, не имеют даже теоретической возможности потрогать светило руками. Поэтому они моделируют термоядерную реакцию в микроскопических масштабах в своих лабораториях. Экономисты ведущих стран строят математические модели поведения финансовой системы страны, на которой апробируются их законодательные инициативы, это гораздо безопаснее, чем опыты на своих гражданах. Несмотря на разную направленность, у всех рассмотренных примеров есть объединяющая черта — модель создается в том случае, когда проведение исследований на реальном объекте невозможно (по финансовым, экономическим, этическим, техническим или любым другим причинам).

Независимо от моделируемого объекта создателям модели необходимо выполнить очень важное условие — *модель должна быть адекватна реальности*, иначе результаты исследования можно выбросить в мусорную корзину. Вполне естественно, что при подготовке модели самолета для продува в аэродинамической трубе не стоит тратить время на создание миниатюрных двигателей, прокладку электропроводки и размещение кресел внутри салона. Это не столь важно в сравнении с безукоризненным соблюдением пропорций фюзеляжа, крыльев и хвостового оперения летательного аппарата. Поэтому в процессе моделирования ученые отбрасывают малозначимые детали, уделяя первоочередное внимание ключевым характеристикам модели, в последнем примере с точки зрения аэродинамики.

На страницах этой книги мы еще много раз будем говорить о моделях баз данных и, в частности, о доминирующей сегодня реляционной модели. Теперь мы с вами знаем, что модель базы данных призвана отражать реальные и виртуальные объекты окружающего мира, информацию о которых мы планируем хранить и об-

рабатывать в разрабатываемых информационных системах. Чем точнее модель отражает действительность, тем она полезнее и, как следствие, лучше база данных.

На сегодняшний момент времени подавляющее большинство современных баз данных ориентировано на реляционную модель. Это простейшие Microsoft Access и FoxPro, Paradox, dBase и сложные клиент-серверные системы Microsoft SQL Server, Informix, Oracle, InterBase, MySQL. Список можно продолжить… Но так было не всегда. До второй половины 1970-х годов рынок баз данных был поделен примерно поровну между программными продуктами, реализующими *иерархическую* и *сетевую* модели представления данных.

Иерархическая модель

Своим появлением на свет иерархическая модель данных обязана лунному проекту Apollo и двум американским компаниям: Rockwell International и IBM. Президент США Дж. Ф. Кеннеди очень хотел взять реванш у СССР и пообещал высадить американского астронавта на Луну к концу 1960-х. Для доставки человека на поверхность спутника нашей планеты требовалось не только рассчитать траекторию полета, но и обработать огромное количество вспомогательных данных, начиная от расхода кислорода астронавтами и заканчивая учетом заказов на уборку помещений НАСА. Итак, в начале 1960-х North American Aviation (именно так в те годы именовалась Rockwell) стала генеральным подрядчиком по разработке программного обеспечения для обслуживания всего проекта. В результате пятилетнего труда специалисты компании предложили заказчику программу с мудреным названием "Обобщенный метод доступа и модификации" (Generalized Update Access Method, GUAM), способную хранить данные в виде перевернутой иерархической структуры (рис. 2.2). В это же время к Rockwell присоединился субподрядчик IBM, преследовавший еще одну заветную цель — разработать первый коммерческий проект баз данных. Совместное детище двух компаний получило новое название "Информационно-управляющая система" (Information Management System, IMS). В отличие от всех своих конкурентов проект IMS приобрел одно уникальное по тем временам преимущество — система позволяла работать с данными не только на физическом, но и на логическом уровне. Новый программный продукт очень скоро стал столь популярен, что без него не обходился ни один крупный мейнфрейм вплоть до 80-х годов XX века.

Суть иерархического хранения данных заключается в применении инвертированной древообразной структуры. В наивысшей точке располагался корень дерева, ниже следовали дочерние узлы (листья дерева). Все узлы связывались друг с другом благодаря сложной системе указателей. Каждый из узлов, в свою очередь, мог являться родительским по отношению к одному или нескольким нижерасположенным узлам, но у дочернего узла не могло быть более одного родителя.

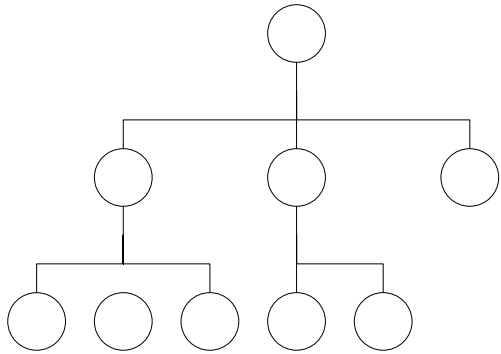


Рис. 2.2. Иерархическая модель данных

Таким образом, в иерархической модели реализована одна из наиболее часто встречающихся в реальном мире связей между сущностями — связь "один ко многим". К примеру, в городе находится много компаний, в компании образовано много отделов, в отделе работает много сотрудников. Подобное решение в некоторой степени снижает проблему избыточности данных. Например, наш проект базы данных дней рождений из *главы 1*, реализованный средствами иерархической модели, мог бы выглядеть так, как представлено на рис. 2.3.

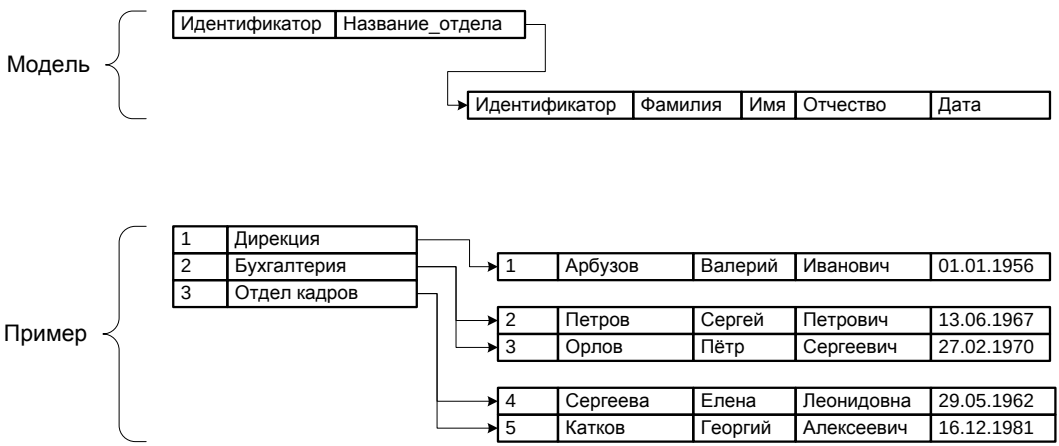


Рис. 2.3. Пример иерархической БД

Это несколько усовершенствованное решение предусматривает описание дополнительных данных — места работы сотрудника. Заметьте, что узлы сотрудников выступают в качестве подчиненных по отношению к узлам отделов. При желании схему можно развить, например, подчинив сотрудникам узлы с заказами, описанием работ, номерами счетов и т. п. Подобная модификация не потребует кардинального изменения уже существующих узлов дерева, мы лишь добавляем очередные узлы на новом уровне.

Поборов недостатки систем, основанных на файлах, иерархическая модель приобрела новые. Вот только некоторые из них:

- *Ограничения в организации отношений между сущностями*. Иерархическая модель позволяет организовать последовательную связь "один ко многим" между данными, но не в состоянии реализовать отношения "многие ко многим".
- *Структурная зависимость*. Иерархическая структура предполагала, что физически данные также станут храниться в виде дерева. Серьезное изменение структуры (например, переподчинение узлов) могло привести к тому, что прикладные приложения теряли возможность навигации по данным.
- *Сложность разработки прикладного программного обеспечения* (ППО). Разработчик программ должен знать особенности физического хранения данных, иначе он мог просто заблудиться в запутанной системе указателей.

Ко всему прочему иерархическая модель не была стандартизирована. Как следствие всегда существовала проблема переносимости данных между приложениями различных разработчиков.

ЗАМЕЧАНИЕ

Иерархическая организация очень удобна при поиске данных, когда запрашиваемая информация уже сгруппирована по соответствующим ветвям дерева и нам остается лишь спускаться от корня схемы к требуемому листу.

Сетевая модель

Практически параллельно с North American Aviation и IBM свои собственные разработки вела еще одна серьезная американская корпорация — General Electric, создавшая прототип первых сетевых баз данных IDS (Integrated Data Store). Так же, как и разработчики иерархической модели, General Electric работала в условиях отсутствия утвержденного стандарта на описание среды БД.

ЗАМЕЧАНИЕ

Говоря о сетевой модели данных, обязательно следует упомянуть заслуги руководителя проекта IDS Чарльза Вильяма Бэчмэна (Charles William Bachman), за свой вклад в науку в 1973 году он был удостоен премии Тьюринга.

Сразу оговорюсь, что когда мы говорим о "сетевой" модели, в виду имеется особенность организации данных, а не способ объединения вычислительных мощностей компьютеров в сеть. В данном случае за основу была взята не древовидная модель данных, а структура, построенная на основе графа (рис. 2.4).

По сравнению с иерархической, сетевая модель имеет одно серьезное преимущество: она позволяет назначать произвольное количество связей между узлами графа. Поэтому можно создавать базы данных, более точно отражающие связи реального мира, в частности, в сетевых БД без особого труда можно было формировать отношения "многие ко многим" или замыкать связь узла на себя самого. При удалении записи в сетевой БД уничтожается только конкретная запись и связи с

ней, все остальное остается на месте. Это разительное отличие от иерархически организованной БД, ведь здесь при удалении одной записи удалению подлежала вся нижерасположенная ветвь дочерних элементов или требовалось провести относительно сложную процедуру переподчинения дочерних узлов. Впрочем, простота перестроения структуры графа имеет и свои подводные камни, ведь любое непродуманное изменение связей может привести к нарушению целостности данных.

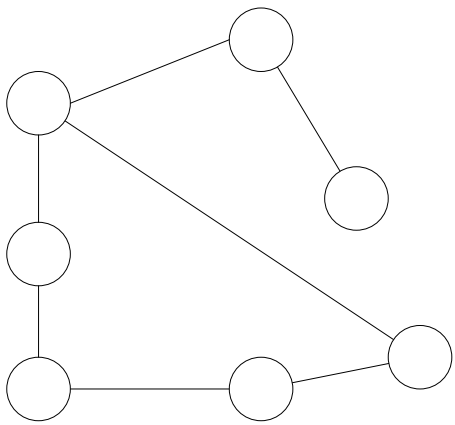


Рис. 2.4. Сетевая модель данных

Говоря о сравнительных характеристиках двух рассмотренных моделей, сразу отметим, что они свободны от недостатков систем, основанных на файлах. Все это достигнуто благодаря тому, что иерархическая и сетевая модели предусматривают существование некоторого внешнего хранилища, в котором размещаются описания данных. Подобное хранилище обычно называют *системным каталогом*. Современные системы управления базами данных размещают в нем всю информацию, необходимую для нормальной жизнедеятельности базы данных, включая описание используемых типов данных, ограничений на вводимые данные, отношений, связей между отношениями и многое-многое другое. Благодаря системной информации достигается архиважный для базы данных фактор физической независимости приложений от данных. Теперь грамотно спроектированное приложение после ознакомления с системным каталогом способно получать доступ практически к любому набору данных.

К сожалению, сетевая модель также не свободна от недостатков:

- большое количество произвольных связей повышает сложность схемы БД и, как следствие, вызывает дополнительные трудности при обеспечении целостности данных;
- сложность разработки прикладного программного обеспечения.

Попытки разработки стандарта БД

Рассуждая о судьбах иерархической и сетевой моделей баз данных, мы отмечали, что во многом их недостатки были предопределены вакуумом в области стандартов. Разработчики СУБД фактически работали на голом поле, полностью пола-

гаясь на свою интуицию. Поэтому уделим несколько минут внимания знакомству со сложившейся к середине 1960-х годов обстановкой в области стандартизации БД.

А ситуация напоминала извечный философский спор о первичности курицы или яйца. Корректный стандарт, определяющий способы хранения и описания данных в будущих БД, не мог появиться на пустом месте, для этого был необходим опыт реализации и эксплуатации подобных баз. Одновременно с этим любая попытка построения СУБД в условиях отсутствия стандартов была обречена на провал. Существовал ли выход из такого положения? Оказывается, да.

Острое желание победить СССР в "лунной гонке", помноженное на завидные финансовые возможности, позволило США пойти по пути одновременного создания и курицы, и яйца. Компании North American Aviation, IBM, General Electric и десятки их помощников уже приступили к созданию пилотных версий БД. Одновременно с этим в 1965 году при непосредственном покровительстве правительства США на конференции CODASYL (Conference on Data Systems Languages) была сформирована рабочая группа List Processing Task Force, переименованная в 1967 году в группу Data Base Task Group (DBTG). Всю свою энергию DBTG посвятила вопросам определения спецификаций среды, которая допускала бы разработку баз данных и управление данными. Группа сумела создать два научных отчета, промежуточный в 1969 году и итоговый в 1971 году. Заметьте, что к этому времени американский астронавт уже оставил свой след на спутнике Земли.

Несмотря на упорные старания группы DBTG ее труд не смог стать стандартом, отчеты DBTG не удовлетворили Национальный институт стандартизации США (American National Standard Institute, ANSI). Вместе с тем работа ни в коем случае не прошла даром и подготовила фундамент для будущих исследований. Трудно в двух предложениях резюмировать более чем пятилетний труд специалистов, но скажу следующее. Группа DBTG признала необходимость использования двухуровневого подхода, построенного на основе системного представления и пользовательских представлений. Если уж совсем просто — БД должны создаваться так, чтобы пользователю даже не имело смысла интересоваться, каким образом данные хранятся физически, ему нужно лишь обладать возможностью ими пользоваться.

Идеи DBTG не были зарыты в землю. В 1975 году Комитет планирования стандартов и норм (Standards Planning and Requirements Committee, SPARC) Национального института стандартизации США предложил свое видение модели БД — трехуровневую архитектуру ANSI/X3/SPARC. Ближе всего к данным предлагалось ввести первый уровень абстракции — внутренний (рис. 2.5). На *внутренний уровень* возлагались задачи управления физическим хранением данных в памяти ЭВМ. Над внутренним уровнем расположился очередной уровень абстракции — *концептуальный*. Этот уровень уже не нес ответственности за хранение данных на жестких дисках. Комитет SPARC предлагал возложить на концептуальный уровень описание логической структуры всех данных (логический смысл хранимых в БД данных, связи между данными, ограничения на данные, правила поддержки целостности и т. п.). Наконец третий по счету слой архитектуры ANSI/X3/SPARC получил название *внешнего уровня*. Внешний уровень отвечал за организацию интерфейса между человеком и БД. Конечный пользователь нашего программного продукта может быть высококлассным программистом, а может оказаться и "чай-

ником". Уровень должен уметь предоставить и тому, и другому соответствующие его потребностям и умениям инструменты по доступу и управлению данными.

Как видите, архитектура ANSI/X3/SPARC развивает задумку группы DBTG. И там и здесь во главу угла ставится обеспечение независимости от данных. Независимость означает, что изменения на нижних уровнях никак не влияют на верхние уровни. Только в последнем случае комитет SPARC пошел дальше и предложил три слоя абстракции. Благодаря этому модель обеспечивала не только физическую, но и логическую независимость от данных.

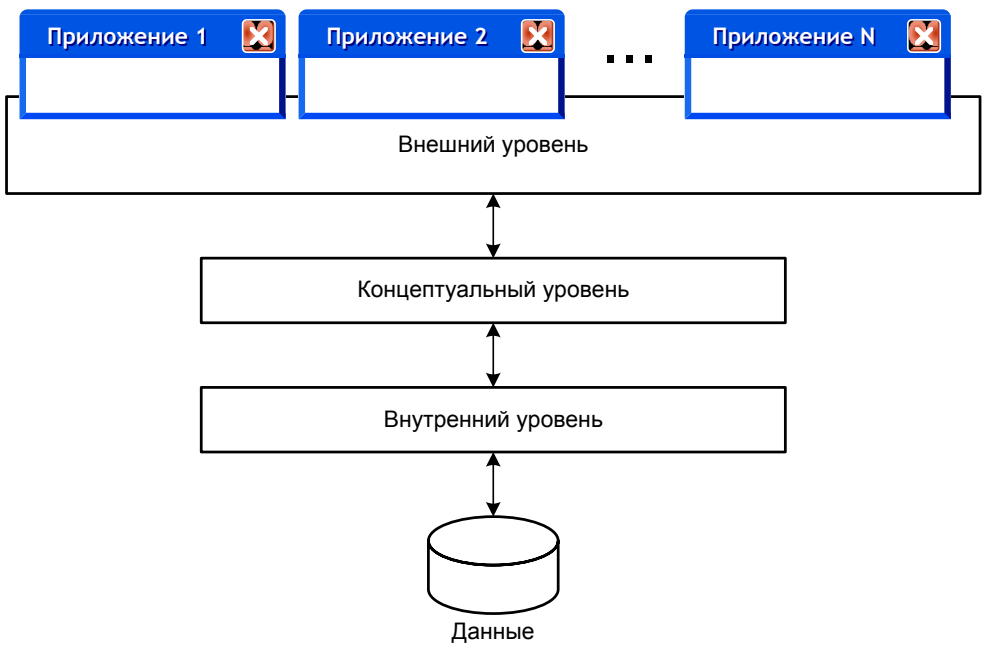


Рис. 2.5. Трехуровневая архитектура ANSI/X3/SPARC

Впрочем, дополнительный уровень не принес требуемого результата, модель ANSI/X3/SPARC (даже несмотря на то, что она вышла из стен Национального института стандартизации США) тоже не сумела стать стандартом.

Хотя труды группы DBTG и комитета SPARC не принесли разработчикам запланированного результата и не стали стандартом, предложенные ими модели оказали значительное влияние на специалистов-практиков, в особенности на создателей первой реляционной БД, получившей название System-R.

Реляционная модель

Довольно скоро на смену иерархической и сетевой моделям пришла принципиально новая модель данных — реляционная. Реляционная модель — главная "ге-

роиня" этой книги — в следующих главах будет рассмотрена весьма подробно, поэтому сейчас скажу лишь то, что она основана на принципе выявления подлежащих описанию в БД сущностей и связей между ними. По сравнению со своими "товарками" она отличается высокой гибкостью, здесь программист гораздо в меньшей степени ломает голову над механизмом управления данными. Эта задача автономно решается ядром СУБД. Реляционные БД снабжены специализированным языком SQL, который достаточно легок в освоении. Таблицы реляционной модели жестко структурированы, что упрощает их обслуживание.

Работа над реляционной моделью объединяла как теоретические, так и практические изыскания всех предшественников. Первый существенный результат пришел в 1976 году, когда в исследовательской лаборатории корпорации IBM, расположенной в городе Сан-Хосе штата Калифорния, на свет появился прототип современных реляционных моделей — проект System-R. Руководителем проекта был Мортон Астрахан (Morton M. Astrahan).

Цель упомянутого проекта — доказать практичность реляционной модели, что достигалось посредством реализации предусмотренных ею структур данных и требуемых функциональных возможностей. На основе этого проекта был разработан структурированный язык запросов (в ту пору названный SEQUEL), который несколько позднее (в 1986 г.) стал стандартом SQL.

На базе System-R впоследствии (в 1975–1979 гг.) был создан первый успешный коммерческий реляционный продукт фирмы IBM — DB2. Говоря о DB2, нельзя не упомянуть одного из ее авторов — Криса Дж. Дейта (Chris J. Date). На сегодняшний день это ведущий специалист по реляционной модели данных. В России широко известна и многократно переиздавалась его книга "Введение в системы баз данных" [7], на которой выросло не одно поколение разработчиков БД.

Как и все в нашем мире, реляционная модель данных далеко не идеальна. По существу, это компромиссное решение между потребностью отражать сущности реального мира и связи между ними и ограниченными возможностями математической теории множеств переложенной на программный код. Любой компромисс предполагает появление множества малоприятных ограничений. Так, из-за борьбы с избыточностью данных в реляционной БД (процесс нормализации) информация "размазывается" по нескольким отношениям (таблицам). Как следствие, для получения сводной информации приходится собирать данные по крохам и осуществлять множество соединений. Чем больше соединений, тем больше временных и ресурсных затрат. Другая проблема реляционной модели заключается в особенностях организации связи между отношениями. Для моделирования всего многообразия взаимодействия между сущностями реального мира в нашем распоряжении имеется лишь одна конструкция "один ко многим". Для создания отношения "многие ко многим" приходится выкручиваться за счет введения дополнительной соединительной таблицы и применения двух типов связей "один ко многим". Но на этом проблема построения связей не исчерпывается. Реляционная модель попросту бессильна, когда следует отразить смысловую нагрузку связи. Что делать, когда семантика связи между сущностями различается? Мы с вами видим разницу между глаголами "обладает", "подчиняется" или "управляет", а реляционная модель нет…

Замечание

В отличие от однотипных записей реляционных БД, записи в иерархической и сетевой моделях могут обладать разной структурой, т. е. содержать различное число разнотипных полей.

Спустя пять лет после того, как мир впервые услышал о реляционной модели, ученый Питер Пин Шань Чен дополнил теорию моделирования данных доктора Кодда весьма удобным описанием, получившим название ER-модель. Об этой модели мы поговорим в *главе 5*, а сейчас отметим, что Питер Чен предложил простой и эффективный способ представления сущностей, атрибутов и связей между ними в виде наглядных диаграмм. В отличие от сетевой, иерархической и реляционной моделей (относящихся к классу моделей реализации), модель Чена принадлежит к разряду концептуальных (понятийных). Другими словами, ER-модель отражает логическую природу представления данных.

Объектно-ориентированная модель

В середине 80-х годов XX века на рынке программных продуктов стали активно появляться программные средства, построенные на основе объектно-ориентированной технологии. Все разработчики как заклинание повторяли новые термины: абстрагирование, инкапсуляция, модульность, иерархичность. Теперь на верхушку пирамиды вознесен Его Величество Объект. Объект в технологии объектно-ориентированного программирования выступает удобной абстракцией объектов из реального мира. Он позволяет описывать все, начиная от авторучки, заканчивая тепловозом. Главное, чтобы программист грамотно сконструировал исходный класс. Программный объект обладает свойствами, методами и способен реагировать на события. Одним словом достоинств столько, что многие начали говорить о замене реляционной модели данных на перспективную объектно-ориентированную.

Желание создать что-то новое было столь острым, что в первой половине 1990-х годов группой под названием Object Database Management Group были предложены рекомендации по созданию объектно-ориентированных баз данных ODMG-93. Примерно тогда начали появляться мнения о закате реляционной модели данных. Однако до сегодняшнего дня реализовать рекомендации ODMG-93 в полном объеме не удалось. Основная причина этого — отсутствие развитого математического аппарата, способного описывать данные в формате, приемлемом объектно-ориентированной теории.

С точки зрения точности моделирования реального мира объектно-ориентированная модель данных по всем статьям сможет превзойти реляционную, ведь в идеале она должны быть способной адекватно выразить информационные структуры любой сложности. То же самое можно сказать и о реализации связей между объектами — вариантов не счесть. Самым главным достоинством СУБД очередного поколения может стать тот факт, что проведение логических операций над данными можно многократно усилить за счет возможностей ООП. Но, к сожалению,

даже спустя два десятилетия мы не увидели полностью работоспособного программного продукта с этикеткой "объектно-ориентированная СУБД". Так что, пока можно говорить только о взаимной интеграции реляционной модели и технологии ООП.

Замечание

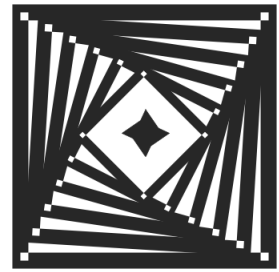
Наибольшие противоречия между специалистами по базам данных вызывает объектно-ориентированная модель баз данных (ООМБД). Ряд авторов утверждает, что ООМБД (Object Oriented Database Model, OODM) приходит на смену реляционной и представляет собой СУБД 3-го поколения, а другие, например, Крис Дж. Дейт вместо термина "объектно-ориентированная модель" используют термин "объектно-реляционная система" [7], тем самым, подчеркивая, что объектно-ориентированные системы ни в коем случае не пришли на смену СУБД 2-го поколения, а лишь дополнили реляционную модель. Так что слухи о кончине реляционной модели оказались сильно преувеличенными. И на сегодняшний день реляционная модель остается доминирующим направлением развития всех современных систем баз данных. Практическое подтверждение слов К. Дейта — реляционная СУБД Oracle 11.0, поддерживающая элементы объектно-ориентированного подхода. Так что Oracle 11.0 можно назвать объектно-реляционной СУБД.

Весьма существенный вклад в разработку объектно-реляционной системы внес американец Майкл Стоунбрэкер (Michael Stonebraker). В 1970-е он работал над проектом Ingres, в 80-е возглавлял проект Postgre, в 90-е ряд его идей был реализован в СУБД Informix Universal Server. Стоит отметить, что Стоунбрэкер (так же как и Крис Дж. Дейт) с крайней осторожностью употреблял термин "объектно-ориентированная модель", предпочитая говорить о системах баз данных будущего поколения.

Резюме

За сравнительно недолгий период существования модели реализации баз данных совершили несколько эволюционных шагов — от первых систем, основанных на файлах, до реляционных и объектно-ориентированных моделей. Отказ от систем файлов и переход к базам данных позволил получить ряд существенных преимуществ при хранении и обработке данных: непротиворечивость, целостность, совместное использование, контролируемую избыточность и высокую безопасность.

Совместно с моделями реализации развивалась и теория баз данных, появлялись на свет принципиально новые языки программирования (например, ставший сегодня стандартом структурированный язык запросов SQL) и особый вид программного обеспечения — системы управления базами данных (СУБД).



Глава 3

Функции и компоненты СУБД

Вне зависимости от того, на основе какого подхода спроектирована современная БД, ее существование немыслимо без системы управления базами данных. Прямо или косвенно СУБД используют администраторы, разработчики БД, программисты и обычные пользователи. Для этого СУБД предоставляет определенный набор инструментов, упрощающих проектирование, администрирование БД и обеспечивающих доступ к данным.

Так что такое СУБД? Предлагаю нашу работу начать с разъяснения этого термина. Система управления базами данных (Database Management System, DBMS) — это комплекс программных средств, с помощью которого можно создавать и поддерживать базу данных, а также осуществлять к ней контролируемый доступ пользователей.

Существует множество показателей, по которым можно классифицировать СУБД. Основной классификационный признак — модель реализации данных, существуют сетевая, иерархическая, реляционная и объектно-ориентированная модели. Про особенности перечисленных моделей мы уже поговорили в предыдущей главе, поэтому сразу перейдем к очередному параметру. Различают персональные и многопользовательские СУБД. Персональные системы предназначены для создания небольших БД устанавливаемых на одном компьютере, поэтому их часто называют настольными. В противовес персональным, многопользовательские системы предназначены для обслуживания БД, находящихся в совместном использовании несколькими пользователями. Есть и другие классификационные признаки, например, по способам разработки приложений БД (ручное кодирование и автоматическая генерация форм), по возможностям определения данных, особенностям обработки транзакций, используемой ОС, по экономическим параметрам. В этой книге мы обсудим многие из этих показателей, но сначала попробуем разобраться с основными функциональными обязанностями СУБД и тем, как она их выполняет.

Функциональные обязанности СУБД

Ключевые функциональные обязанности СУБД были сформулированы Коддом еще в начале 1980-х годов. На первом этапе их было восемь, позднее Кодд и другие

исследователи неоднократно расширяли и уточняли этот перечень. Так что теперь можно говорить о нескольких десятках функций, служб и сервисов, которыми обязана обладать современная СУБД. В этой книге мы не станем пытаться объять необъятное и прокомментируем только основополагающие функции СУБД:

- *Доступность данных.* Предоставление пользователю возможности вставлять, редактировать, удалять и извлекать данные их БД. При осуществлении любой из операций пользователь не должен вникать в особенности физической реализации системы, т. е. все операции должны быть прозрачны.
- *Системное описание данных.* СУБД должна предоставлять системный каталог, в котором содержится: описание хранимых в БД данных; описание связей между данными; ограничения целостности данных; регистрационные данные пользователей и другая служебная информация. Благодаря метаданным БД становится доступной внешним приложениям, упрощается понимание смысла данных, усиливаются меры безопасности, может выполняться аудит информации.
- *Управление параллельностью*. Реализация механизма одновременного многопользовательского (параллельного) доступа к обрабатываемым данным с гарантией корректного обновления этих данных. Умение предоставить нескольким пользователям одновременный доступ к разделяемым ресурсам — едва ли не самая сложная задача, решаемая СУБД. СУБД должна суметь избежать конфликта совместного доступа двух или большего числа пользователей к одним и тем же строкам таблицы, или, по крайней мере, исключить какие-либо нежелательные последствия при возникновении конфликта.
- *Транзакции*. СУБД гарантирует, что БД будет всегда находиться в непротиворечивом состоянии вне зависимости от любых сбоев при проведении операций обновления данных. Для этого операции с данными (в первую очередь вставки, редактирования и удаления данных) объединяются в единый блок называемый транзакцией. Все операторы транзакции должны быть выполнены корректно и полностью, только тогда в БД будут зафиксированы изменения. В противном случае осуществляется автоматический откат транзакции, т. е. состояние БД будет восстановлено на момент времени, предшествующий вызову транзакции. Далее в книге мы рассмотрим много примеров транзакций, а пока ограничимся одним, наиболее близким каждому из нас. Представьте себе, что вы снимаете какую-то сумму наличных денег в одном из многочисленных банкоматов вашего города. Вы уже ввели свой код доступа, указали требуемую сумму, эта сумма денег списана с вашего электронного счета и "бежит" по проводам из банка к вам в руки. И вдруг из-за сбоя питания или отказа коммуникационного оборудования, или по какой-то другой технической причине команда на выдачу денег в банкомат не дошла. Что в результате? Вы потеряли свои деньги? К счастью, нет. Программное обеспечение банковских платежных систем увидит (при условии, что оно написано знатоком своего дела), что одна из операций транзакции завершена некорректно. Поэтому все изменения, сделанные указанной транзакцией, подлежат отмене — списанные электронные деньги вновь вернутся к вам на банковский счет. Вам остается повторить операцию получения наличных в другом банкомате.

- *Обеспечение целостности данных*. Все данные, хранимые в БД, должны быть корректными и непротиворечивыми. Это означает, что данные в таблицах могут модифицироваться только в соответствии с установленными правилами. В самом общем случае можно говорить о существовании трех правил поддержания целостности данных: целостность доменов, целостность отношений, целостность связей между отношениями. Кроме того, разработчик имеет возможность описывать свои собственные бизнес-правила, которые мы будем называть корпоративными ограничениями.
- *Восстановление данных*. В случае непредвиденных ошибок и сбоев, приведших к повреждению или разрушению данных, СУБД должна обладать возможностью восстанавливать пострадавшие данные. В первую очередь эта функция реализуется с помощью процедур резервного копирования.
- *Обмен данными*. СУБД обязана поддерживать современные сетевые технологии и предоставлять доступ к БД удаленным персональным компьютерам.
- *Контроль за доступом к данным*. Доступ к данным могут осуществлять только зарегистрированные в СУБД пользователи в соответствии с назначенными администратором СУБД им правами.

Еще раз отметим, что список обязанностей СУБД на этом далеко не заканчивается. Современные системы предоставляют нам средства мониторинга, сервисы статистического анализа, утилиты экспорта/импорта данных, развитые средства проектирования БД и прикладного программного обеспечения, инструменты реорганизации файлов данных и индексных данных, службы аудита и многое другое. Однако все дополнительные сервисы и службы выполняют вспомогательную роль, а основы жизнедеятельности СУБД определяются перечисленными функциями.

Компоненты СУБД

СУБД — сложный вид программного обеспечения, над созданием которого работают большие коллективы высококвалифицированных программистов. На современном рынке программного обеспечения конкурирует около двух десятков коммерческих СУБД. Из малых систем, рассчитанных на одного пользователя, сегодня наибольшей популярностью пользуются: Microsoft Access и Microsoft FoxPro. В перечень многопользовательских СУБД, получивших широкое признание, входят: Oracle, Microsoft SQL Server, InterBase, FireBird, MySQL, Postgre, Informix, DB3. Несмотря на то, что все перечисленные программные продукты предназначены для решения практически одних и тех же задач, входящие в СУБД программные компоненты и взаимосвязи между ними далеко не идентичны. Поэтому любая попытка обобщить все известные технические решения построения СУБД и на основе этого обобщения построить структурную схему типичной СУБД, пусть даже высокой степени абстракции, несколько наивна. Тем не менее, мы попытаемся это сделать.

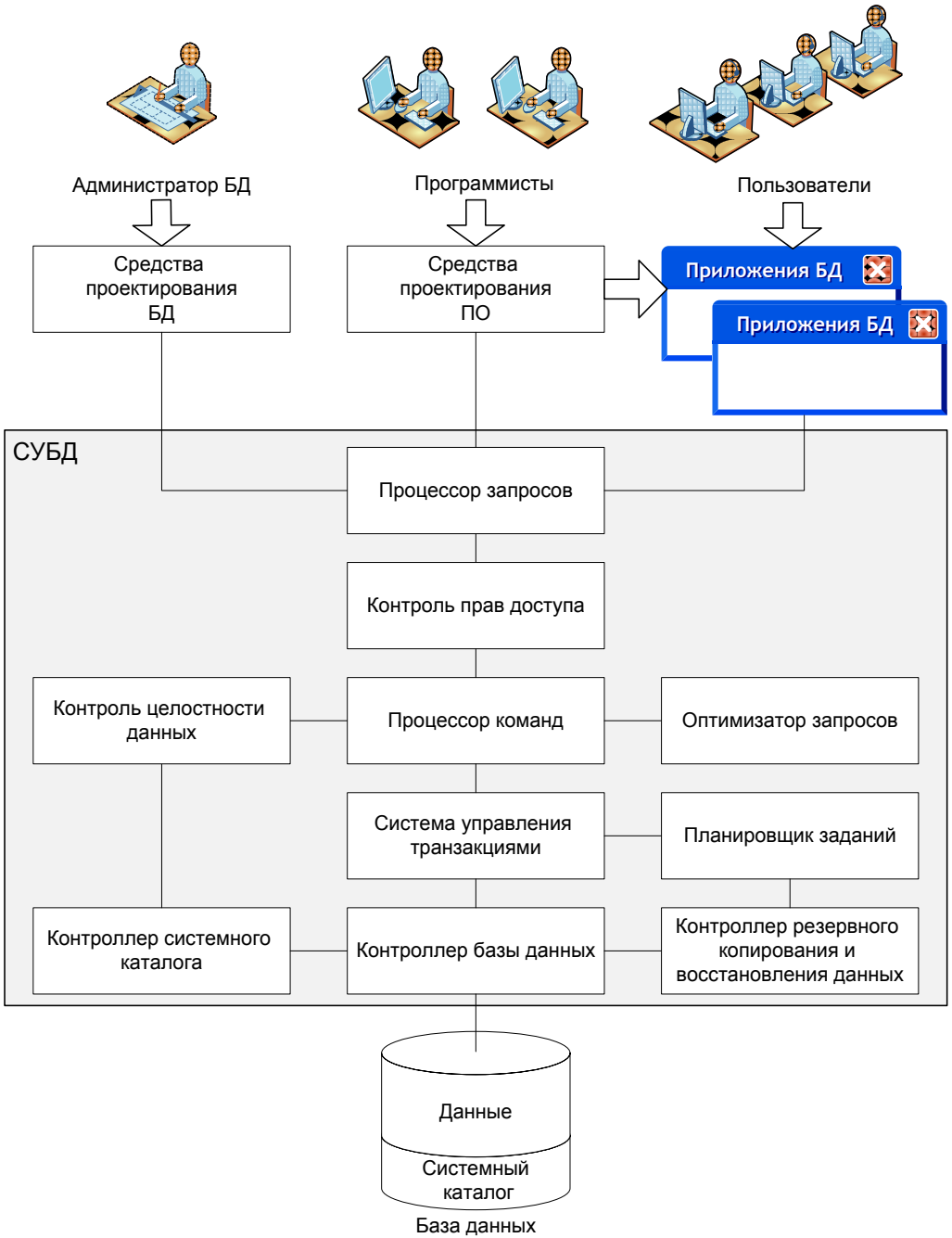


Рис. 3.1. Обобщенная структура СУБД

Для того чтобы СУБД оказалась в состоянии предоставлять минимальный набор из рассмотренных восьми сервисов, она должна состоять из набора компонен-

тов, представленных на рис. 3.1. Над верхним уровнем системы расположены потребители услуг СУБД:

- администратор БД;
- программисты;
- пользователи.

Администратор БД отвечает за планирование и физическую реализацию проекта. Он создает основные объекты БД, определяет правила поддержания целостности и непротиворечивости данных, управляет политикой безопасности, анализирует процесс эксплуатации проекта, одним словом поддерживает жизнедеятельность БД. В распоряжении администратора имеются разнообразные средства проектирования БД, эти средства могут входить в состав СУБД в виде дополнительных программных модулей, а могут быть поставлены и сторонними разработчиками.

Программисты совместно с администратором работают над физическим созданием проекта БД. Но прикладных программистов в большей степени интересует не сама концепция проекта БД, а способы донесения этой концепции до конечного пользователя. Поэтому область интересов программистов смещена в сторону разработки клиентских приложений и отчетов. Основным инструментом прикладного программиста выступают многочисленные среды проектирования, как правило, 4-го поколения (4 Generation Level, 4GL). В первую очередь это программные пакеты Embarcadero RAD Studio (в состав которого и входит язык Delphi) и Microsoft Visual Studio. Кроме того, некоторые СУБД (например, Microsoft Access и FoxPro) обладают встроенными средствами проектирования, но обычно возможности таких средств существенно ограничены.

Основным потребителем услуг СУБД выступает обычный пользователь — в его интересах создаются БД и прикладное программное обеспечение. Среди пользователей есть и неплохие специалисты, чьи советы способны помочь разработчикам БД, а на другом конце спектра находятся пользователи, с трудом находящие нужные кнопки на клавиатуре. Последние могут даже и не понимать, что они работают с БД, расположенной на другом компьютере.

Основным средством общения между людьми и базой данных выступает структурированный язык запросов SQL. Поэтому средства проектирования БД, ПО и клиентские приложения БД отправляют в адрес СУБД инструкции на языке SQL. Эти команды поступают на *процессор запросов* СУБД, который преобразует их в набор низкоуровневых команд, понятных ядру СУБД.

К базе данных могут обращаться различные категории пользователей, от руководителя предприятия до случайного посетителя. Одним из них разрешается просматривать и редактировать любые данные, другим могут быть доступны лишь выборочные сведения, третьим вообще не следует даже знать о существовании БД. Именно поэтому в составе большинства СУБД имеется модуль, отвечающий за *контроль прав доступа.* В простейшем случае модуль следит за тем, чтобы к БД присоединялись только авторизованные пользователи, для этого полученные во время регистрации пароль и логин потенциального клиента сверяются с хранящимися в системном каталоге учетными записями. В современных многопользова-

тельских СУБД система безопасности значительно сложнее и включает в себя комплекс программных, а иногда и аппаратных средств защиты.

Убедившись, что инструкция поступила от доверенного лица, модуль контроля прав доступа передает ее в распоряжение *процессора команд*. В первую очередь процессор убеждается, что поступившая команда не противоречит ограничениям целостности данных. Это зона ответственности модуля *контроля целостности данных*. На время проверки команды на соответствие ограничениям целостности доменов, сущностей и связей задействуется *контроллер системного каталога*. Именно он имеет возможность собирать метаданные, в которых прячется техническое описание нашей БД. Поняв, что угрозы целостности нет, процессор передает команду *оптимизатору запросов*. Задача оптимизатора — найти наиболее эффективный способ выполнения поступивших команд. Наконец, оптимизированная команда компилируется и передается во власть *системе управления транзакциями*. Система управления транзакциями, во-первых, отвечает за полное и корректное выполнение блока команд и, во-вторых, совместно с планировщиком заданий обеспечивает параллельную многопользовательскую обработку данных. Наконец блок команд передается в распоряжение *контроллеру баз данных*. Задача модуля заключается в организации взаимодействия СУБД с файлами БД и файлами системного каталога. При этом для осуществления стандартных операций ввода-вывода задействуются возможности операционной системы.

Архитектурные решения доступа к БД

Поиски эффективных архитектурных решений организации доступа к БД начались с первых дней работы с БД. Все решения, так или иначе, отталкивались от текущего состояния электронной вычислительной техники. Во времена доминирования больших вычислительных машин доступ к самым первым БД базировался на принципе телеобработки. База данных и СУБД размещалась в памяти центральной ЭВМ, к этой ЭВМ подключались терминалы. Так как терминальные устройства особым умом и сообразительностью не отличались (они не обладали собственными вычислительными возможностями), то вся обработка данных осуществлялась только в процессоре центральной ЭВМ. Системы на основе телеобработки доминировали до начала 1970-х годов, но с изобретением микропроцессоров стали постепенно уступать свои позиции и сейчас практически не используются.

Файл-сервер

Бурное развитие микропроцессорной техники в 70-х годах XX века привело к появлению сравнительно дешевых микро-ЭВМ. Теперь руководители предприятий вместо покупки одной большой дорогостоящей ЭВМ начали приобретать несколько недорогих машин, менее производительных, но вполне приемлемых для реше-

ния задач, стоящих перед их коллективами. Микро-ЭВМ объединялись в простейшие одноранговые локальные сети, в которых каждая из машин обладала равными правами со своими соседями. Одна из ЭВМ (с накопителями на жестких магнитных дисках наибольшего размера) назначалась файловым сервером. На выданном в общее пользование сетевом каталоге сервера кроме обычных документов размещали и файлы баз данных (рис. 3.2). Для того чтобы этой БД могла воспользоваться какая-нибудь из рабочих станций, она обращалась к файловому серверу, перекачивала все файлы БД в свою в память, вносила правки и возвращала файлы на прежнее место. Такой способ многопользовательского доступа к БД обладал всего одним преимуществом — простотой реализации и целым букетом недостатков:

- На каждой рабочей станции необходимо устанавливать полную копию СУБД.
- Во время работы с данными сетевой трафик возрастает до пиковых значений.
- Управление параллельностью обработки данных и поддержку целостности реализовать было практически невозможно, в то время как один пользователь сохранял свою порцию данных, другой уже вносил в нее изменения.

Замечание

Существуют весьма совершенные файл-серверные решения. В них сервер предоставляет клиентам доступ таким образом, что они могут осуществлять операции чтения и записи с дисками и памятью сервера так, будто это их собственные диски и ОЗУ. Примерами таких сервисов могут стать NFS, NetBEUI и Named Piped.

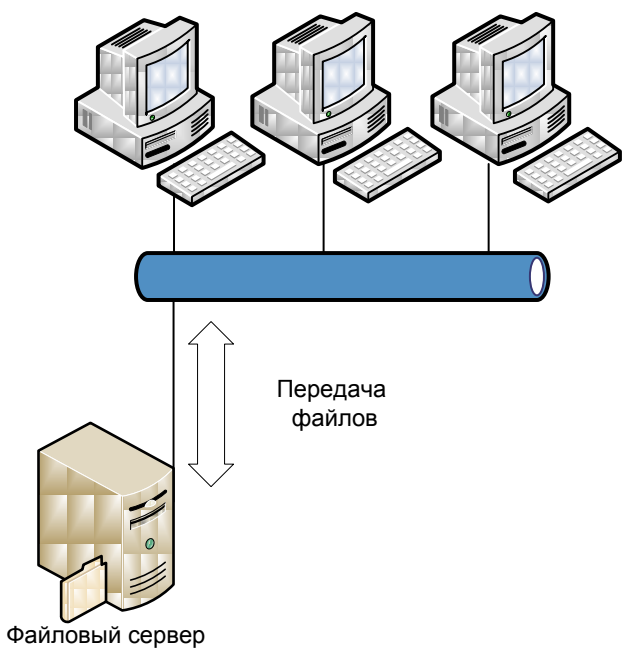


Рис. 3.2. Архитектура "файл-сервер"

В качестве примера программных продуктов, способных работать в режиме файл-серверных систем, можно привести такие настольные СУБД, как Access и FoxPro корпорации Microsoft, а также различные версии dBase.

Клиент-сервер

В настоящее время архитектурное решение "файл-сервер" применяется лишь для поддержки малых баз данных с числом пользователей, не превышающим 5–6 человек. В больших предприятиях доминирует архитектура "клиент-сервер". Появление клиент-серверных систем тесным образом связано с парадигмой открытых систем, активно эволюционирующих с начала 1980-х годов. Программисты пришли к выводу о необходимости распределения задач между элементами системы, в данном случае между двумя типами процессов, которые могут выполняться на различных компьютерах, объединенных в вычислительную сеть. *Процесс сервера* отвечает за предоставление каких-либо услуг клиентскому процессу. *Клиентский процесс* запрашивает у сервера определенные услуги и ресурсы.

В клиент-серверных системах БД размещается на отдельном, как правило, наиболее производительном компьютере, на этом же компьютере разворачивается сервер СУБД. На клиентских станциях достаточно установить сравнительно нетребовательное к ресурсам пользовательское ПО и настроить сетевой доступ к серверу СУБД. Работа клиент-серверных систем принципиально отличается от работы в системах "файл-сервер". Теперь, вместо перекачки файлов с БД, клиентский компьютер отправляет серверу запрос, построенный на основе языка SQL. Получив и обработав инструкцию SQL, сервер возвращает клиентскому компьютеру результаты ее выполнения, например выборку определенных данных (рис. 3.3).

К настоящему времени существует более десятка СУБД, предназначенных для работы по модели клиент-сервер. В России наибольшей популярностью пользуются: InterBase фирмы Embarcadero Technologies, Oracle, Microsoft SQL Server, Postgre, MySQL, Informix Universal Server, NetWare SQL фирмы Novell и т. д.

На сегодняшний день доминирует клиент-серверное построение БД. Причин тому несколько:

1. Значительно повышается доступность БД. Сервер представляет собой открытую систему, поэтому клиентские компьютеры могут функционировать под управлением различных ОС и с различным ПО.
2. Выделенный сервер СУБД в состоянии обеспечить параллельную многопользовательскую обработку данных.
3. Основные правила поддержки целостности и непротиворечивости данных описываются на одном сервере СУБД, клиентские приложения никаким образом не в состоянии обойти эти правила.
4. Экономно расходуется пропускная способность компьютерных сетей.

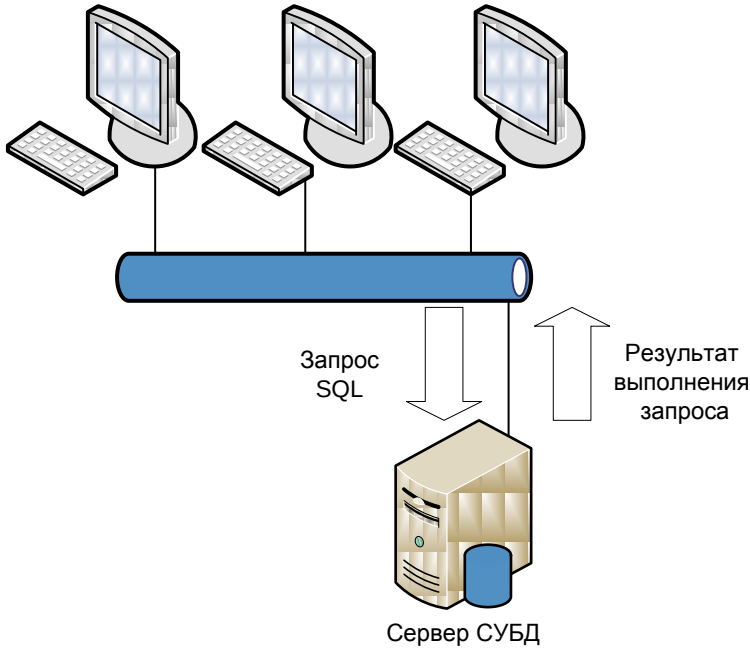


Рис. 3.3. Архитектура "клиент-сервер"

5. Благодаря централизованному хранению данных сравнительно просто поддерживать единые для всех правила безопасности БД.
6. Наличие стандарта на основной язык общения SQL обеспечивает широкие возможности доступа к серверу БД из программного обеспечения различных производителей.
7. Упрощены вопросы обслуживания и администрирования БД.

Предусмотрено несколько подходов к распределению обязанностей между сервером и клиентом. В простейшем случае (рис. 3.4, *а*) работает модель *тонкий клиент* (thin client). Это вариант, когда клиентское ПО максимально упрощено и представляет собой только интерфейсную часть, состоящую из форм ввода и просмотра данных. Тонкий клиент фактически отвечает только за презентацию данных, поэтому можно сказать, что он не умеет думать, и вся программная логика сосредоточена на стороне сервера. Есть и обратное решение — *толстый клиент* (fat client). В противовес своему тонкому собрату, толстый клиент старается максимально облегчить работу сервера (рис. 3.4, *б*), например, реализует дополнительные бизнес-правила, сортирует полученные данные на клиентской стороне, выполняет вспомогательные расчеты, проверяет синтаксис инструкций SQL и т. п. Проектирование полнофункционального толстого клиентского приложения значительно сложнее, чем разработка его упрощенного собрата. Но, в конечном счете, мы получаем существенную прибавку в производительности, а это весьма важно в больших многопользовательских БД.

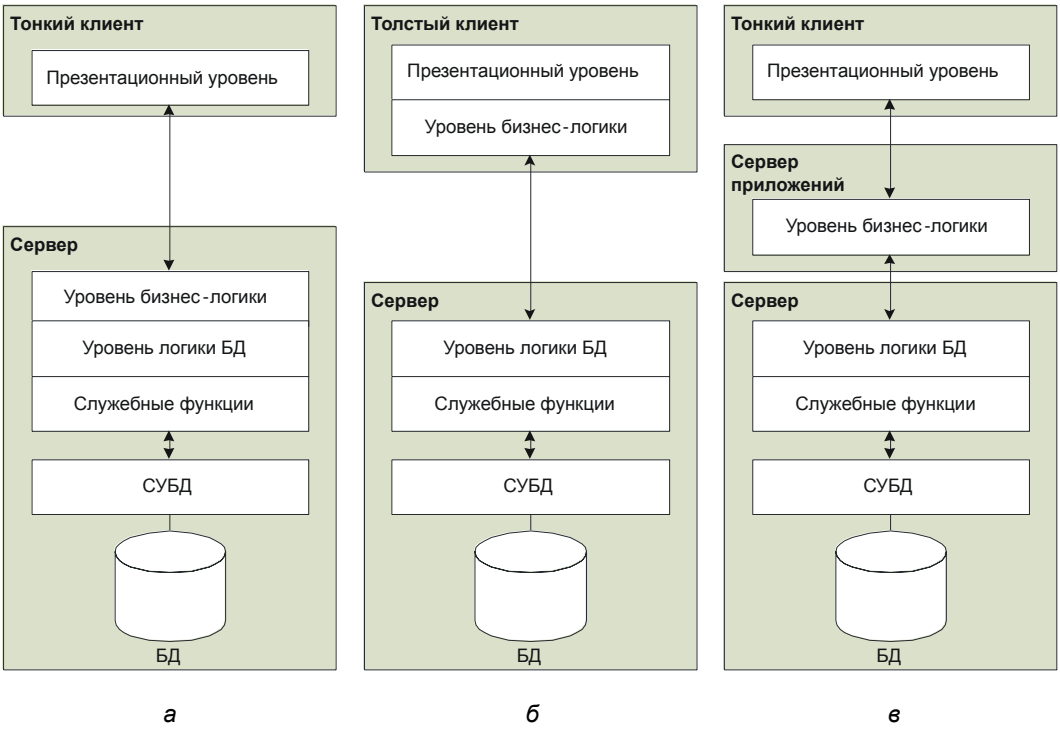


Рис. 3.4. Популярные модели распределения функций между сервером и клиентом

Многоуровневые решения

Предусмотрены и более сложные модели распределения функций между сервером и клиентом. Например, между сервером СУБД и клиентом может появиться еще одно дополнительное звено — *сервер приложений* (рис. 3.4, *в*). Сервер приложений обычно отвечает за соблюдение бизнес-правил БД, для этого реализуется несколько прикладных функций, которые представляются клиенту в виде отдельных служб. Клиент не может обратиться к СУБД напрямую, вместо этого он запрашивает интересующий его сервис у сервера приложений.

Многоуровневые проекты БД могут создаваться на фундаменте нескольких технологий. Сегодня огромной популярностью пользуется существенно усовершенствованная в Delphi 2010 распределенная модель DataSnap. В Delphi имеется целый ряд компонентов, специализирующихся на построении многозвенных проектов DataSnap, о них мы поговорим во второй части книги.

При создании промежуточного звена между клиентом и сервером разработчики часто пользуются технологией *Архитектура брокера общих объектных запросов* (Common Object Request Broker Architecture, CORBA). Технология CORBA создавалась специально для систем с распределенной обработкой информации. Ядром

системы выступает *брокер объектных запросов* (Object Request Broker, ORB), выполняющий посредническую функцию между клиентом и сервером.

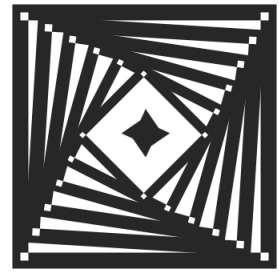
Изюминка CORBA в том, что интерфейсная часть всех описанных в ней компонентов отделена от их реализации. Благодаря этому связывающиеся при посредничестве CORBA сервер и клиентские приложения могут разрабатываться на любых языках программирования и функционировать под управлением различных ОС, лишь бы соблюдалось единственное условие — они должны уметь общаться на *языке определения интерфейсов* (Interface Definition Language, IDL).

Многоуровневые проекты БД можно создавать и на альтернативных технологиях, в частности на основе *распределенной многокомпонентной модели объектов* (Distributed Component Object Model, DCOM), *протоколе простого доступа к объектам* (Simple Object Access Protocol, SOAP) предназначенного для обмена данными в распределенной децентрализованной среде, технологии сокетов и т. д.

Несмотря на то, что трехзвенная архитектура позволяет создавать гибкие и универсальные решения, она используется значительно реже, чем двухзвенная. Причин тому несколько, но основная из них в том, что создание многоуровневых клиент-серверных проектов под силу только хорошо подготовленным разработчикам, как следствие подобные проекты стоят значительно дороже по сравнению с классической архитектурой "клиент-сервер".

Резюме

Состав и особенности функционирования любой сложной системы, в том числе и СУБД, определяется стоящими перед ней задачами. Еще на заре разработки реляционной модели доктор Эдгар Фрэнк Кодд сформулировал функциональные обязанности классической СУБД, чем определил вектор развития этой области информационных технологий. За более чем 40 лет существования реляционных баз данных идеи Кодда совершенствовали ученые, инженеры и программисты, в результате к сегодняшнему дню разработан широкий перечень эффективных программных и аппаратных решений, применяемых в СУБД.



Глава 4

Реляционная модель данных

Создателем реляционной модели считается математик Эдгар Фрэнк Кодд (Edgar Frank Codd, 1923–2003). Однако работа талантливого ученого начиналась не с чистого листа. Научный базис теории отношений, положенной Коддом в основу реляционной модели, закладывался еще на рубеже XIX и XX веков. Авторами основ будущей реляционной теории являются два исследователя: Чарльз Содерс Пирс (1839–1914) и Эрнст Шредер (1841–1902). Несмотря на столь глубокие корни реляционной теории, первая реляционная база данных появилась на свет спустя семьдесят лет. Датой рождения реляционной БД можно считать июнь 1970 года. Именно тогда Кодд (на тот момент времени сотрудник одной из лабораторий корпорации IBM) опубликовал свою знаменитую статью "Реляционная модель данных для больших совместно используемых банков данных" (Codd E. F., A Relational Model of Data for Large Shared Data Banks. CACM 13: 6), в которой впервые прозвучал столь популярный сегодня термин "реляционная модель".

Замечание

За свой вклад в науку Эдгар Кодд был удостоен премии имени Тьюринга. Это знаменательное событие произошло в 1981 году.

Первопричиной возникновения нового по тем временам подхода к проектированию баз данных послужили существенные ограничения предыдущих моделей. Ни сетевая, ни иерархическая модели не были способны просто и доступно описывать подлежащие учету данные. Кодд сумел объединить, на первый взгляд, несовместимые вещи — с одной стороны, реляционная модель опиралась на математические выкладки, а с другой стороны, была понятна рядовому пользователю, состоящему в конфронтации даже с таблицей умножения.

Замечание

Модель данных представляет собой логически взаимоувязанную совокупность описаний объектов данных и операторов, управляющих этими объектами. Применяя термин "модель", мы акцентируем внимание на то, что это абстрактная конструкция. Чтобы модель превратилась в действующую базу данных, она должна быть физически реализована средствами какой-либо СУБД. Нетрудно догадаться, что реляционная база данных может получиться только из реляционной модели.

Модель данных не представляет существенного интереса для обычного пользователя, но без нее не обойтись разработчику БД. Ведь это не что иное, как чертеж будущей БД. Ни один строитель не начнет строить дом без чертежа, так и ни один программист не начнет создание физической БД без ее подробной схемы. Фундамент реляционной модели построен на понятии сущности и связей между сущностями.

Реляционной модели данных посвящено много фундаментальных трудов, в которых подробно изложены все ключевые аспекты. Обязательно рекомендую читателю хотя бы в обзорном порядке ознакомиться с работами ведущих специалистов в этой области [7, 11, 17]. Моя задача несколько прозаичнее, далее я попытаюсь сформулировать тот минимум информации, без которого понять реляционную модель будет просто невозможно.

Любая область знаний, будь то математика, радиоэлектроника, физика, химия или информатика по мере своего развития формирует свой набор терминов и определений. Без этого никак не обойтись, ведь в своих ученых беседах специалисты оперируют понятиями, не имеющими аналогов в окружающем мире. Зачастую размерность специфичных понятий разрастается до такой степени, что существенная часть времени отводимого на изучение интересующего нас предмета затрачивается только на формирование словарного запаса. Это утверждение подтвердят студенты, выбравшие в качестве своей профессии медицинскую стезю. Вы никогда не присутствовали при общении двух эскулапов? Даже если они говорят о банальном насморке, у стороннего наблюдателя неизбежно создается впечатление, что перед ним два иностранца с неподдающейся идентификации языковой принадлежностью.

С точки зрения сложности семантики, системы управления базами данных (в общем) и реляционная модель (в частности) не являются исключением из правил. Эта сравнительно новая область знаний практически мгновенно успела обзавестись специфичным набором слов. Многие термины перекочевали в БД из словарного запаса программистов, часть определений позаимствована из теории множеств, в тезаурусе разработчиков моделей данных даже нашлось место для понятий, обычно встречающихся в справочниках по философии.

Сущность и атрибуты

Любой более-менее подготовленный пользователь твердо знает, что базы данных предназначены для хранения и обработки информации. Если в руки такому человеку попадется база городских телефонов, то, даже не запуская программу, он сможет предположить, что в ней он обнаружит телефонные номера граждан и организаций. База данных городской библиотеки обязана содержать информацию об авторах и их бессмертных произведениях. Столкнувшись с базой данных авиакомпании с высокой степенью вероятности, можно утверждать, что в ней мы найдем информацию о расписании полетов, марках самолетов и городах, в которые они рассчитывают доставить пассажиров.

То, что кажется элементарным для стороннего наблюдателя, для профессионального разработчика БД может стать очень непростым делом. На самом первом этапе проектирования будущей БД он ломает голову над двумя ключевыми вопросами:

1. Что необходимо хранить в БД?
2. Какие связи между данными следует поддерживать в БД?

Для получения ответа на первый вопрос разработчик должен составить первоначальный перечень типов сущностей. *Тип сущности* (entity) — это класс однотипных объектов, о которых следует хранить информацию в БД. Немного забегая вперед, проговорюсь, что на заключительной стадии проектирования типы сущности превратятся в реляционные таблицы. В 90% случаев сущность представляет собой какой-то объект, отвечающий на вопрос "Кто?" или "Что?". В базе данных библиотеки на роль сущностей претендуют такие одушевленные и неодушевленные объекты, как автор, книга, издательство, читатель. Совсем не обязательно, чтобы сущность соответствовала объектам реального мира, напротив, она может моделировать и отвлеченные понятия, допустим, выраженные в особых отношениях между сущностями. Именно в таком случае сущность ловко маскируется под глагол. Как правило, это происходит, когда учету подлежат некоторые события. Например, когда читатель получает (это и есть глагол) книгу из библиотечного фонда, логика базы данных требует сохранить информацию о том, кому, когда и какая из книг была выдана. При более подробном рассмотрении выясняется, что при построении первоначального списка мы просто не выявили такой тип сущности, как читательская библиотечная карточка, в которую и делаются все перечисленные пометки. Библиотечная карточка поддерживает двустороннюю связь между сущностями "читатель" и "книга".

Если тип сущности — это класс однотипных объектов, то как же величать отдельный объект, входящий в этот класс? В реляционной модели такой объект называют *экземпляром сущности*. Экземпляр сущности (или просто сущность) — это конкретная реализация объекта. Для сущности "автор" это Уильям Шекспир и Лев Толстой, для сущности "книга" это "Ромео и Джульетта" и "Война и мир".

Замечание

Экземпляр сущности — это то, о чем (или о ком) следует хранить информацию в базе данных.

У любой сущности есть свои характеристики. У сущности "самолет" из базы данных авиакомпании таких характеристик миллион. Это марка, максимальная скорость полета, число посадочных мест, грузоподъемность, дата выпуска, цвет фюзеляжа и многое-многое другое. У сущности "читатель" отличительных признаков не меньше, а может быть и больше, чем у летательного аппарата. В счет может пойти все, начиная с цвета глаз и заканчивая размером обуви. Однако в библиотечной базе данных вряд ли стоит учитывать рост, вес и отпечатки пальцев клиента. Гораздо полезнее знать имя, домашний адрес и номер телефона. Отличительные свойства сущности называют *атрибутами*.

Атрибуты могут быть *простыми* (например, название города, в который летит самолет), а может быть и *составными* (т. е. включать в себя несколько простых атрибутов). В качестве составного атрибута выступает адрес читателя. Здесь найдется место почтовому индексу, городу, улице, дому и номеру квартиры. Атрибут может быть *производным*, т. е. полученным в результате проведения какой-то операции над другими атрибутами. Подобные атрибуты часто встречаются при построении различных бухгалтерских систем, например, значение взимаемого с сотрудников предприятия налога составляет определенный процент от заработной платы этих служащих. Можно столкнуться с *многозначными* атрибутами, они описывают те свойства сущности, в которых одновременно хранится несколько значений. Например, у одного человека может быть несколько адресов проживания, несколько контактных телефонных номеров и т. п. Есть и особая разновидность атрибутов, без которых невозможно построение реляционной модели, — атрибуты, однозначно идентифицирующие экземпляр сущности. На роль такого атрибута в состоянии претендовать индивидуальный номер налогоплательщика или уникальный заводской номер самолета. Позднее подобные атрибуты превратятся в ключевые поля реляционных таблиц.

Построив исчерпывающий список типов сущностей и подготовив перечень их атрибутов, разработчик модели базы данных должен задуматься над особенностями реализации типов сущностей. Для этого в первую очередь следует разобраться с особенностями физического представления каждого из атрибутов типа сущности, для этого предназначены тип данных и домен.

Тип данных и домен

Отправной точкой процесса построения любой системы хранения и обработки данных по праву считается выбор *типа данных*. Независимо от того, какой язык программирования вы изучали, при построении новой программы самым первым шагом становится рассмотрение типов данных, имеющихся в распоряжении системы. Программистов Delphi, Си или поклонников любого другого языка программирования не удивишь типами данных `INTEGER`, `REAL` или `CHAR`. Указанные типы данных специализируются соответственно на обслуживании целочисленных, вещественных и символьных значений.

Типы данных определяют порядок хранения данных в памяти компьютера. Физическая концепция хранения данных зависит от особенностей конкретной платформы, на базе которой функционирует программное обеспечение. Поэтому в разных операционных системах, в разных системах управления базами данных число байтов, отводимых для описания целого или вещественного числа, может отличаться. Например, в 32-разрядных ОС размерность целого числа `INTEGER` равняется 4 байтам. В 64-разрядных системах это утверждение может оказаться ошибочным, ведь здесь проще адресовать 8-байтовую переменную.

Типизация хранимых значений не просто указывает на размерность в байтах, которую должна выделить система для размещения в памяти того или иного значения. Преследуемая цель еще более значима. Типизация определяет, какие операции могут быть осуществлены с теми или иными данными. Система программирования не позволит новичку передать результаты деления в целочисленную переменную, ведь в этом случае есть риск утерять дробную часть результата. Система станет отчаянно сопротивляться, если мы попробуем просуммировать символьный и вещественный тип данных. Пусть даже в символьной переменной хранится числовое значение. В последнем случае перед проведением математической операции необходимо повести преобразование данных. Точно так же СУБД не допустит ввода текстовых данных в поля таблиц, предназначенных для хранения числовой информации.

Подытожим все сказанное. Понятие "тип данных" интегрирует в себе три компонента:

1. Ограничение множества значений, принадлежащих типу.
2. Дефиниция набора операций, применяемых к типу.
3. Определение способа отображения (внешнего представления) значений типа.

Как показала практика, при обработке данных в современных информационных системах возможностей обычной типизации становится недостаточно. Примеров тому великое множество. Формально, при определении атрибута, хранящего дату рождения сотрудников какого-нибудь предприятия, следует ограничиться типом данных `DATETIME`. Но такой подход не способен в полной мере отразить задачи отдела кадров. Стандартизированный в SQL тип данных `DATETIME` допускает ввод даты в диапазоне от 1/01/0000 г. до 12/31/9999 г. Вряд ли начальник нынешнего отдела кадров примет на работу сотрудника в несовершеннолетнем возрасте или, наоборот, ровесника Ивана Грозного. Поэтому атрибут "дата_рождения" должен уметь сопротивляться данным, некорректным, с точки зрения пользователя, но абсолютно верным с точки зрения типа данных `DATETIME`. Есть и другой пример. Допустим, что в базе данных хранится почтовый индекс. Каждый поклонник эпистолярного жанра знает, что это комбинация из шести цифр, отражающая региональную принадлежность почтового отделения. Порядок описания индекса не сложен для человека — нужно заполнить все шесть значений, не четыре, не пять и не семь, а только шесть. Теперь попробуйте подыскать стандартный тип данных, способный однозначно решить эту простейшую задачу. Тип `INTEGER` не подходит, потому что обладает существенно бóльшим диапазоном значений. Строка из шести элементов `CHAR` также не вполне адекватна, ведь она позволяет пользователю вводить любые (не только цифровые) символы. Рассмотрим еще один пример. Формально номер телефона можно типизировать как целое число — `INTEGER`. Целые числа поддерживают все основные арифметические операции. Однако никому в здравом уме и в доброй памяти не придет в голову идея суммировать значения телефонных номеров или вычитать из одного номера другой, хотя формально это обычные целочисленные значения. Просто очень сомнительна практическая ценность полученного результата. В подобных и во многих других схожих ситуациях одна лишь типизация бессильна — она в недостаточной степени поддерживает че-

ловеческую логику. Если я вас убедил, что возможностей типа данных недостаточно для описания характеристик атрибутов, закончим с примерами и поговорим о том, как это ограничение можно преодолеть.

Фундаментальная для современных языков программирования идея типизации в реляционной модели получила дальнейшее развитие. На сцену вышло новое понятие — *домен*. Домен ни в коем случае не подменяет понятие типизации, а выступает в качестве дополнительного ограничителя информации, обрабатываемой в базе данных. Ограничение накладывается за счет введения дополнительных логических правил. Так, принимая нового сотрудника на работу, при вводе даты рождения можно контролировать его возраст, путем элементарного сравнения текущей системной даты компьютера и даты рождения. Обслуживая атрибут почтового индекса, описываемого шестью символами типа `CHAR`, наложим дополнительное ограничение на диапазон приемлемых значений — наш домен должен пропускать только цифровые знаки. Надо сразу оговориться, что логические правила являются не столько описанием отдельного домена, сколько средством поддержания целостности данных в масштабах всей базы данных. Допустим, вы вводите почтовый индекс города Москвы, а в атрибут "город" передаете Монреаль. В этом случае разумная база данных должна намекнуть нам, что мы сегодня перетрудились и следует прерваться на чашечку кофе… Но для того, чтобы система смогла выявить подобную логическую ошибку, разработчик БД должен построить не только строгий набор доменных ограничений, но и реализовать развитую систему поддержания корпоративной целостности.

Замечание

Тип данных определяет особенности физического хранения данных. Домен, опираясь на концепцию типа данных, несет дополнительную нагрузку — отвечает за поддержание логических правил описания данных. Благодаря доменным ограничениям в БД реализуется одно очень важное качество — доменная целостность данных.

В профессиональных СУБД доменные ограничения могут накладываться на несколько атрибутов одновременно, таким образом, одновременно контролируя непротиворечивость хранящихся в них данных.

Связь

Реляционная модель призвана не только описать перечень типов сущностей, но и отразить особенности взаимодействия между ними. Взаимодействие возникает там, где между типами сущностей проявляются отношения, выраженные в виде глагола, например, продавец оформил заказ. Здесь "продавец" и "заказ" — это две сущности, а связь между ними просматривается в глаголе "оформил".

Различают три типа связи между сущностями: "один к одному", "один ко многим" и "многие ко многим". Появление в реляционной БД связи "один к одному" — крайне редкий случай. Чаще всего он свидетельствует о том, что разработчик не

вполне корректно определился с атрибутами и вместо одного из атрибутов создал лишний тип сущности. На практике наиболее распространена связь "один ко многим", например, автор написал много книг, завод выпустил много автомобилей, в одном отделе работает много сотрудников (рис. 4.1).

"Сотрудник"		
Ключ_сотрудник	Фамилия	Ключ_отдела
1	Григорьев	2
2	Арбузов	2
3	Самойлов	1
4	Костюков	3
5	Симонова	1

"Отделы предприятия"	
Ключ_отдела	Название
1	Отдел кадров
2	Бухгалтерия
3	Отдел планирования

Рис. 4.1. Отношение "один ко многим"

Связь "многие ко многим" встречается реже: офисы разных компаний размещены в разных городах, множество издательств опубликовало произведения различных авторов и т. п. Реляционная модель напрямую не поддерживает отношение "многие ко многим", поэтому его приходится разбивать на два отношения "один ко многим", но об этом поговорим позднее.

Есть еще один классификационный признак связи — *степень* (размерность). При выявлении связей между сущностями разработчик обычно встречается с *унарными* (unary relationship) и *двойными связями* (binary relationship), но в реальной жизни имеют место связи и большей кратности. Например, выражение "авиакомпания доставляет пассажира в город" подразумевает *тернарную связь* (ternary relationship). Здесь мы сталкиваемся с тремя существительными-сущностями "авиакомпания", "пассажир" и "город" и всего одним глаголом "доставляет". Дальнейшее развитие модели такой базы данных во многом зависит от опыта и интуиции разработчика, ведь разные пассажиры вправе летать в различные города, пользуясь услугами разных авиакомпаний. Ограничившись лишь двойными связями, можно утратить суть сохраняемой информации — какая из авиакомпаний, в какой именно из городов доставила конкретного пассажира. К вопросу построения связей класса "многие ко многим" мы вернемся в *главе 5* при описании процесса ER-моделирования, а пока обсудим еще одну особенность связи.

Реляционная модель позволяет связывать сущность саму с собой — задавать *рекурсивную связь*. Такой, на первый взгляд, несколько необычный способ взаимодействия оказывается весьма полезным при "выращивании" древообразных конструкций, например, описывающих организационную структуру предприятия, завода или, скажем, университета. Здесь четко просматривается иерархия, в которой одна и та же сущность может быть главной по отношению к подчиненным элементам и одновременно находиться в подчиненном состоянии к более высокому по рангу узлу. Так, дочерними элементами по отношению к деканату факультета выступают кафедры этого факультета, в свою очередь деканат подчиняется ректорату учебного заведения.

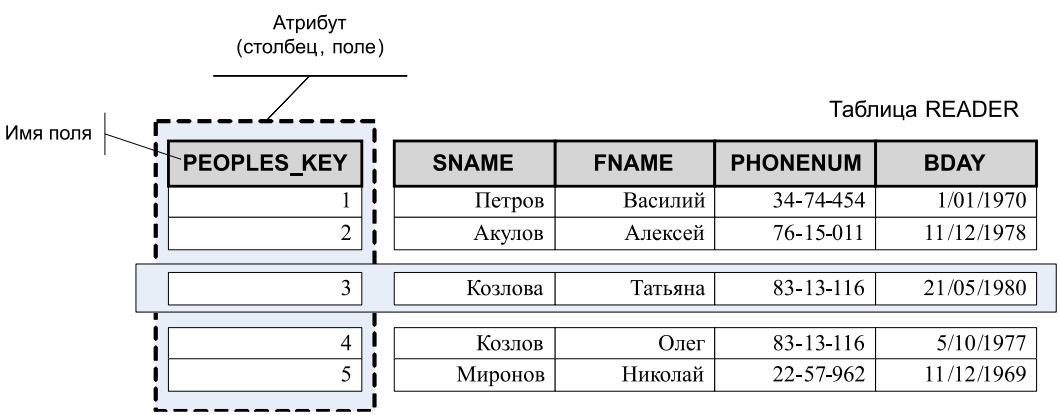
Реляционная таблица

Пользователь рассматривает всю информацию в реляционной БД как совокупность взаимосвязанных двумерных таблиц (отношений), в каждой из которых хранятся данные о строго определенном типе сущностей.

Реляционная таблица состоит из строк и столбцов. Каждая строка содержит информацию о конкретной сущности, так в таблице "Авторы" из библиотечной базы данных мы увидим строки с данными о А. С. Пушкине, М. Ю. Лермонтове, Ф. М. Достоевском и других известных (и не очень) поэтах и писателях. Но в таблице "Авторы" не должно оказаться ни единой строки о клиентах библиотеки — читателях, ведь это другой тип сущности. Каждый столбец реляционной таблицы предназначен для хранения определенного атрибута сущности. Для атрибута "Фамилия" разработчик таблицы применит текстовый тип данных размерностью в 15–20 символов. А для атрибута "Номер телефона" (которому найдется место в таблице читателей) будут наложены дополнительные доменные ограничения, запрещающие передавать в столбец нецифровые символы.

Замечание

Терминология реляционных баз данных складывалась из нескольких областей знаний, поэтому в качестве синонимов к понятию "строка" можно услышать слова "ряд" и "кортеж". У термина "столбец" также есть два тождественных понятия: "атрибут" и "поле". Ведя речь о таблице, допустимо применять термин "отношение" (рис. 4.2).



PEOPLES_KEY	SNAME	FNAME	PHONENUM	BDAY
1	Петров	Василий	34-74-454	1/01/1970
2	Акулов	Алексей	76-15-011	11/12/1978
3	Козлова	Татьяна	83-13-116	21/05/1980
4	Козлов	Олег	83-13-116	5/10/1977
5	Миронов	Николай	22-57-962	11/12/1969

Рис. 4.2. Реляционная таблица

Для того чтобы таблица (отношение) получила высокую честь называться "реляционной", необходимо ее соответствие ряду строгих требований:

- уникальность имен столбцов внутри таблицы;
- каждая строка хранит информацию об отдельной сущности, строки таблицы отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы;

- на каждом пересечении строки и столбца хранится атомарное значение;
- строки и столбцы таблицы могут обрабатываться в любом порядке, независимо от хранящихся в них данных.

Замечание

Немного забегая вперед, замечу, что о таблице, соответствующей указанным требованиям, говорят, что она приведена к первой нормальной форме.

Некоторые из перечисленных требований нуждаются в комментариях. Уникальность имен столбцов требуется соблюдать только в рамках одного отношения и ничто не запрещает использовать одинаковые имена в различных таблицах. Так, если фамилия автора будет храниться в столбце `SURNAME` таблицы `WRITER`, то логично полагать, что фамилия читателя также будет находиться в столбце с аналогичным названием, но теперь в таблице `READER`. Для обращения к требуемому столбцу средствами языка SQL требуется указать название соответствующей таблицы и ее столбца. Например, запрос

```
SELECT SURNAME FROM READER
```

возвратит все фамилии читателей. А запрос

```
SELECT SURNAME FROM WRITER
```

проинформирует нас о фамилиях авторов.

Суть запрета одинаковых строк в реляционной таблице легко объяснима. Во-первых, дублирование данных противоречит теории множеств, ведь классическое множество должно состоять из различных элементов. Во-вторых, повтор идентичных данных приводит к бессмысленному расходованию дискового пространства и оперативной памяти. В-третьих, дубликаты данных вносят существенную путаницу в работу программы. Представьте, что в студенческой группе учатся два тезки, скажем Иван Иванов и Иван Иванов. Один из них круглый отличник, а другой — патологический двоечник и прогульщик. Как следствие первого нужно перевести на следующий курс, а второго отчислить. Если записи об этих студентах не будут отличаться хотя бы на микрон (точнее на бит), то деканат окажется в патовой ситуации, руководству факультета будет непонятно, какую из строк таблицы следует удалить, а какую оставить. Обеспечение требования уникальности строки достигается путем ввода в таблицу дополнительных атрибутов, в частности отчества, даты рождения, номера и серии паспорта и т. п. На основе одного или нескольких уникальных полей формируются особые поля, называемые *ключевыми*.

На пересечении строки и столбца должно находиться единственное значение. Это значение должно быть атомарным, т. е. неделимым. Примером нарушения атомарности может стать попытка записи в одну ячейку таблицы группы значений, например, нескольких фамилий авторов или читателей. Подобный подход также затрудняет поиск и обработку информации.

Последнее из перечисленных требований об обработке строк и столбцов в любом порядке в меньшей степени заботит разработчика БД, т. к. оно реализуется автономно средствами современных СУБД. Заметим, что отсутствие требования упорядоченности не усложняет, а наоборот значительно упрощает процесс разработки

БД. Поэтому, заполняя список студентов, поступивших в вуз, не следует заботиться о соблюдении алфавитной последовательности при внесении новых строк. При необходимости получить упорядоченную по фамилии (или любому другому атрибуту/атрибутам) ведомость можно буквально в два счета, добавив в инструкцию SQL ключевое слово `ORDER BY` и перечень атрибутов сортировки. Порядок вывода столбцов точно так же назначает программист средствами SQL.

Ключ

В составе любого реляционного отношения всегда должно присутствовать поле (или группа полей), содержимое которого предназначается для хранения значений, способных уникальным образом идентифицировать строку таблицы. На роль ключевых вряд ли смогут претендовать поля, хранящие фамилию и имя человека. Если на вашем предприятии на данный момент времени не работают два полных тезки, это совсем не означает, что подобное исключено в будущем. Поэтому разработчику следует сразу включать в таблицу поля с безусловно уникальной информацией для описываемой сущности, например индивидуальный номер налогоплательщика для человека или номер двигателя и кузова для автомобиля. Но и с такими данными могут произойти неприятности. Допустим, что мы выбираем в качестве ключа поле, содержащее информацию о серии и номере паспорта гражданина. Вряд ли кто со мной не согласится, что это уникальные данные и, на первый взгляд, вполне подходят в качестве кандидатуры на ключ. Но паспорт может быть заменен (в результате потери, стихийного бедствия, плановой смены паспортов). В результате у человека появляется новый паспорт с другой серией и номером, это влечет за собой сложную процедуру изменения значений первичного ключа и связанных с ним внешних ключей во всей базе данных. Если приходится исправлять две-три строки в таблице, то ничего страшного. А если паспорта заменили у всех граждан страны?

Поэтому чаще разработчики добавляют к таблице атрибуты, предназначенные для хранения искусственной информации, обычно это целочисленные поля автоинкрементного типа. Идея подобного поля в том, что при внесении в таблицу очередной строки значение поля получало приращение (обычно плюс единица, хотя во многих СУБД это значение настраиваемое). Простая и как следствие надежная логика работы поля-счетчика исключает вероятность повтора значений в автоинкрементных полях — каждая новая строка в ключевое поле запишет новое значение. Если вы ненадолго вернетесь к рис. 4.1, на котором представлено отношение между сущностями "Отдел" и "Сотрудник", то увидите, что для сущности "Отдел" в качестве первичного ключа введен атрибут "ключ_отдела", а для сущности "Сотрудник" — "ключ_сотрудника".

Ключ, отвечающий за уникальную идентификацию строки в реляционной таблице, называют *первичным ключом* (Primary Key, PK).

Помимо первичных ключей в таблицах активно применяется еще одна разновидность ключевого поля — *внешний ключ* (Foreigner Key, FK). Благодаря внешнему ключу (еще его называют вторичным) мы получаем возможность строить реля-

ционные связи между таблицами. На рис. 4.1 внешний ключ имеется у сущности "Сотрудник", это атрибут "ключ_отдела". Если мы хотим указать, что сотрудник Григорьев не покладая рук трудится в бухгалтерии предприятия, то мы делаем следующее:

- Узнаем значение первичного ключа у строки "Бухгалтерия" (в нашем примере это 2).
- Передаем полученное значение во внешний ключ строки "Григорьев".

Таким образом, поле внешнего ключа сущности "Сотрудник" может содержать любое из значений первичного ключа сущности "Отдел".

Целостность данных

Мы уже знаем, что база данных — это не просто сосуд, который постепенно заполняется произвольной информацией. Кроме того, БД призвана на установленном разработчиком уровне абстракции отражать взаимоувязанные объекты реального мира. Чем точнее данное отражение, тем совершеннее база данных. Зеркало также призвано отражать объекты реального мира, но каждый из нас знает, что оно склонно к искажению действительности. Объект, попавший в зазеркалье, уже не тот, что находится снаружи. Поэтому, кроме функций зеркала, БД должна содержать некоторый набор правил, которые позаботятся о том, чтобы данные всегда находились в согласованном состоянии.

Специалисты выделяют три базовых класса правил, призванных поддерживать целостность данных в реляционной БД:

1. Целостность доменов.
2. Целостность сущностей.
3. Ссылочная целостность.

Кроме того, существует понятие корпоративной целостности, это не что иное, как реализация в БД бизнес-правил, присущих конкретному предприятию.

Целостность доменов

Целостность доменов поддерживается за счет механизма доменных ограничений. Мы о нем уже говорили в начале главы. Это как раз тот случай, когда описание домена включает некие логические правила, отбраковывающие некорректные значения, которые так настойчиво стремятся попасть в поля нашей таблицы. В простейшем случае допустимые значения можно просто перечислить, например "понедельник", "вторник" и т. д. Можно объявить диапазон допустимых значений, например от 2 до 5, если речь идет об оценках учеников. Все зависит от возможностей СУБД, в которой мы создаем наши проекты.

Особая роль в поддержании доменной целостности отводится специальному определителю `NULL`. `NULL` обозначает неопределенность или неизвестность. При проектировании БД разработчики выявляют обязательные атрибуты, без которых

работа БД невозможна, и атрибуты, к которым применяются пониженные требования, например, их разрешено заполнить позднее или не трогать вовсе. Допустим, что вы вводите в БД анкетные данные сотрудников, и у кого-то из них пропущена дата рождения. Что делать? Поставить дату на глазок? Это может привести к искажению фактов (БД призвана отражать, а не искажать объекты реального мира), и впоследствии отдел кадров отправит своего сотрудника раньше на пенсию. Если система способна понять, что ячейка таблицы хранит не значение, а неопределенность, то можно избежать подобной неприятности. Есть и другой пример. В вуз зачислено несколько сотен абитуриентов, но зачетные книжки и студенческие билеты на них пока не заполнены. Номера билетов и "зачеток" — это обязательные, безусловно подлежащие учету данные. Что же? Прекратить ввод уже известной информации (имена, адреса, номера телефонов и т. п.), пока первокурсникам не раздадут все документы? А может статься так, что документы невозможно оформить, пока в БД не появится информация о новых студентах. Получается замкнутый круг. Поэтому грамотно спроектированная база данных просто обязана позволить нам ввести имеющиеся на сегодня данные, отложив на завтра заполнение неопределенных полей.

Определитель `NULL` может оказаться полезен при организации связи между таблицами; если поле внешнего ключа допускает неопределенное значение, то это служит признаком, что связь необязательна.

ЗАМЕЧАНИЕ

Определитель `NULL` ни в коем случае не аналогичен нулевому значению или, например текстовой строке, заполненной пробелами! У него иная задача, `NULL` указывает программисту, что значение атрибута таблицы не определено или неизвестно.

Благодаря определителю `NULL` базы данных работают в трехзначной логике. Поэтому в БД к известным каждому программисту значениям `TRUE` (истина) и `FALSE` (ложь) добавляется `NULL` (неопределенность).

Целостность сущностей

Как недвусмысленно следует из названия, целостность сущностей направлена на обеспечение внутреннего единства отдельной сущности. Независимо от того, что мы намерены хранить в наших таблицах, необходимо соблюдать краеугольное правило — каждая строка таблицы обязана быть уникальной. А для этого следует контролировать корректность первичного ключа отношения (если вы подзабыли, то напомню, что первичный ключ — это минимальный уникальный идентификатор строки в таблице). Суть требования к первичному ключу проста: во входящих в его состав атрибутах не должно содержаться ни одного определителя `NULL`. Такое пожелание логично. Ведь если мы допускаем, что в поле, входящем в состав первичного ключа, может находиться неопределенное значение, то значит, поле перестает поддерживать механизм однозначной идентификации строки таблицы.

Замечание

Ни одно из полей, входящих в состав первичного ключа, не должно допускать ввода определителя `NULL`!

Помимо поддержания корректности первичного ключа целостность сущностей в состоянии обеспечить широкий спектр дополнительных сервисных возможностей. Их перечень определяется профессионализмом разработчика базы данных. Например, было бы хорошо научить БД контролировать непротиворечивость значения вводимого в атрибут на основе значений имеющихся в других атрибутах таблицы или даже в атрибутах других таблиц. База данных железнодорожной кассы обязана уведомить оператора, что пассажир покупает на свое имя два билета в абсолютно противоположных направлениях на одно и то же время. База данных отдела кадров металлургического завода не должна пропустить на вредную работу к мартеновской печи несовершеннолетнего. База данных в деканате не переведет на следующий курс студента, не сдавшего летнюю экзаменационную сессию.

Ссылочная целостность

Реляционная база данных — это не только таблицы с информацией, но и логические связи между ними, а нарушение связи между таблицами может разрушить всю БД. Именно поэтому на стражу реляционных связей поставлено третье правило целостности данных, получившее название ссылочной целостности. Смысл правила в следующем — внешний ключ не может "осиротеть", ему всегда должен соответствовать первичный ключ в главной таблице.

Поводов для нарушения связи предостаточно. Во-первых, удаление строки в главной таблице. Допустим (см. рис. 4.1), мы удаляем строку "Отдел кадров" в таблице "Отделы предприятия". В результате в подчиненной таблице "Сотрудники" оказываются брошенными сотрудники Самойлов и Симонова, ведь их внешний ключ по-прежнему будет хранить значение уже несуществующего первичного ключа. Во-вторых, при добавлении новой записи в таблицу "Сотрудники" во внешнее поле "ключ_отдела" может попасть значение, которому нет соответствия в главной таблице. В-третьих, ссылочная целостность может подвергнуться опасности при редактировании ключевых полей в главной или подчиненной таблицах. В современных СУБД все перечисленные проблемы предотвращаются автоматически, но это не означает, что рядовому разработчику баз данных не нужно задумываться над поддержанием ссылочной целостности в своих проектах.

Корпоративная целостность

В реальной жизни для поддержания реляционной БД в непротиворечивом состоянии перечисленных ранее правил поддержки целостности обычно не хватает. У любого предприятия, магазина, учреждения существуют свои собственные, зачастую уникальные правила. Например, в БД продуктового склада следует контро-

лировать, чтобы скоропортящиеся молочные товары попадали в холодильные камеры, а картофель и капуста — в овощехранилище. Проектируя БД агентства по продаже недвижимости, следует задуматься над тем, чтобы за агентами по продаже закреплялись дома и квартиры, находящиеся в одном районе города, иначе последним придется вместо осуществления сделок часами простаивать в автомобильных пробках. В БД библиотеки следует предусмотреть ограничения на выдачу очередной книги, если читатель просрочил возврат предыдущей. Для реализации специфичных для каждого из заказчиков бизнес-правил разработчики БД создают дополнительные правила, называемые правилами корпоративной целостности. Основным средством поддержки корпоративной целостности выступают хранимые процедуры и триггеры.

Реляционная алгебра

Являясь отличным математиком, Э. Кодд сразу понял, что наилучшим способом описания реляционной модели данных станет теория множеств. Нам, как разработчикам баз данных, может быть и не обязательно в совершенстве владеть всеми тонкостями теории множеств, но с предложениями Кодда нам познакомиться следует. Поэтому сейчас мы обсудим восемь простейших операций реляционной алгебры, определяющих особенности функционирования реляционной модели данных. Итак, нам предстоит разобраться с операциями, осуществляющими:

- Выборку (selection).
- Проекцию (projection).
- Декартово произведение (cartesian product).
- Объединение, сложение (union).
- Вычитание (set difference).
- Пересечение (intersection).
- Деление (division).
- Соединение (join).

Замечание

Реляционная алгебра представляет собой теоретический язык операций, которые на основе одного или нескольких отношений позволяют создавать другое отношение.

Операции выборки и проекции относятся к классу унарных, т. к. работают с одним отношением. При осуществлении выборки из исходного множества формируется результирующее подмножество, содержащее только удовлетворяющие определенному условию элементы. Например, с помощью операции выборки мы сможем получить список студентов, родившихся в 1990 году. Взгляните на рис. 4.3. Выборка из таблицы `STUDENTS` составила три строки.

В отличие от выборки (выделяющей горизонтальные элементы исходного множества) *операция проекции* направлена на получение вертикальной составляю-

щей множества. На этот раз возвращаются все строки из исходной таблицы, но в новое подмножество войдут не все, а только определенные пользователем столбцы таблицы. На рис. 4.4 показана проекция, полученная из таблицы `STUDENTS`. В подмножество включены поля с фамилией и именем студентов.

Таблица **STUDENTS**

S_KEY	SURNAME	FNAME	BIRTHDAY	SPECIALITY
1	Орехов	Владимир	11/04/1989	Математика
2	Петровский	Александр	21/11/1990	Математика
3	Кульгина	Оксана	5/01/1990	Ин. язык
4	Самойлов	Евгений	15/07/1991	Информатика
5	Кузьмина	Ирина	2/03/1990	Физика
6	Яковенко	Константин	12/09/1989	Химия
7	Ищенко	Владимир	09/19/1988	Астрономия

=

Выборка из таблицы **STUDENTS**

2	Петровский	Александр	21/11/1990	Математика
3	Кульгина	Оксана	5/01/1990	Ин. язык
5	Кузьмина	Ирина	2/03/1990	Физика

Рис. 4.3. Демонстрация операции выборки

Таблица **STUDENTS**

S_KEY	SURNAME	FNAME	BIRTHDAY	SPECIALITY
1	Орехов	Владимир	11/04/1989	Математика
2	Петровский	Александр	21/11/1990	Математика
3	Кульгина	Оксана	5/01/1990	Ин. язык
4	Самойлов	Евгений	15/07/1991	Информатика
5	Кузьмина	Ирина	2/03/1990	Физика
6	Яковенко	Константин	12/09/1989	Химия
7	Ищенко	Владимир	09/19/1988	Астрономия

—

Проекция из таблицы **STUDENTS**

Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Яковенко	Константин
Ищенко	Владимир

Рис. 4.4. Демонстрация операции проекции

Декартово произведение открывает набор операций, осуществляемых с двумя подмножествами. Отношение, получаемое в результате декартова произведения, представляет собой сцепление строк из одного отношения со строками из другого отношения. В результате произведения каждой строке из одной таблицы присоединяется одна строка из другой таблицы. На практике подобная операция востребована редко, ведь в результирующем множестве оказываются все возможные сочетания строк из двух таблиц. Однако из любого правила есть и исключения. Допустим, что студенты хотят получить данные о том, какие учебные дисциплины им предстоит изучать, тогда без декартового произведения не обойтись (рис. 4.5).

Операция объединения проводится только с таблицами, совместимыми по объединению, например, с идентичной структурой. В результате мы получим суммарную таблицу, содержащую строки из двух исходных отношений.

Таблица **STUDENTS**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана

×

Таблица **DISCIPLINE**

DISCIPLINE
Операционные системы
Языки программирования
Методы программирования
СУБД

=

Произведение таблиц

SURNAME	FNAME	DISCIPLINE
Орехов	Владимир	Операционные системы
Орехов	Владимир	Языки программирования
Орехов	Владимир	Методы программирования
Орехов	Владимир	СУБД
Петровский	Александр	Операционные системы
Петровский	Александр	Языки программирования
Петровский	Александр	Методы программирования
Петровский	Александр	СУБД
Кульгина	Оксана	Операционные системы
Кульгина	Оксана	Языки программирования
Кульгина	Оксана	Методы программирования
Кульгина	Оксана	СУБД

Рис. 4.5. Демонстрация операции произведения

Представленная на рис. 4.6 операция объединения суммирует записи из таблиц `STUDENTS_A` и `STUDENTS_B` в результирующее отношение. Если бы исходные таблицы включали одинаковые по содержанию строки, то в результате операции объединения из итогового множества были бы удалены записи-дубликаты.

Таблица **STUDENTS_A**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана

∪

Таблица **STUDENTS_B**

SURNAME	FNAME
Алфёров	Николай
Остапенко	Владимир
Юров	Олег

=

Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Алфёров	Николай
Остапенко	Владимир
Юров	Олег

Рис. 4.6. Демонстрация операции объединения

Операция вычитания позволяет выяснить, какие из строк первой таблицы отсутствуют во второй таблице. Как и сложение, вычитание может осуществляться только с таблицами, совместимыми по объединению. Предположим, что у нас есть две таблицы: в первой находятся данные о студентах учебной группы, а во второй — данные о студентах, успешно сдавших экзаменационную сессию. Для того чтобы вычислить, кому следует явиться на пересдачу, вычитаем из первой таблицы вторую (рис. 4.7).

Операция пересечения может проводиться с таблицами, совместимыми по объединению, действие позволяет выявить строки, общие для двух таблиц. Допустим, нам следует узнать, какие из студентов успешно сдали экзамен по высшей матема-

тике (таблица STUDENTS_MATH) и экзамен по СУБД (таблица STUDENTS_DBMS). Пересечение двух отношений даст запрашиваемый результат (рис. 4.8).

Таблица **STUDENTS_A**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Яковенко	Константин
Ищенко	Владимир
Алфёров	Николай
Остапенко	Владимир
Юров	Олег

−

Таблица **STUDENTS_C**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Ищенко	Владимир
Алфёров	Николай
Остапенко	Владимир
Юров	Олег

=

Разность

Яковенко	Константин

Рис. 4.7. Демонстрация операции вычитания

Таблица **STUDENTS_MATH**

SURNAME	FNAME
Орехов	Владимир
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Яковенко	Константин
Щукин	Александр
Алфёров	Николай
Остапенко	Владимир
Юров	Олег

∩

Таблица **STUDENTS_DBMS**

SURNAME	FNAME
Бобров	Валентин
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Ищенко	Владимир
Алфёров	Николай
Остапенко	Владимир

=

Пересечение

SURNAME	FNAME
Петровский	Александр
Кульгина	Оксана
Самойлов	Евгений
Кузьмина	Ирина
Алфёров	Николай
Остапенко	Владимир

Рис. 4.8. Демонстрация операции пересечения

Для понимания сути *операции деления* воспользуемся следующим примером. Допустим, в таблице DIVIDEND (англ. делимое) содержатся данные о том, какие виды занятий определенных дисциплин ведут педагоги университета. Нам хочется уточнить, кто ведет лабораторные работы по "СУБД" и практические занятия по

"Языкам программирования". Для этого в таблицу DIVISOR (англ. делитель) заносят две строки с указанными данными (рис. 4.9):

```
DIVIDEND/ DIVISOR = RES
```

Что окажется в результате выполнения операции деления в итоговой таблице RES?

Для того чтобы усвоить, как работает операция деления, нужно запомнить два обстоятельства.

Во-первых, результат деления исходной таблицы DIVIDEND на таблицу-делитель DIVISOR будет содержать столбцы, отсутствующие в делителе (в нашем случае это атрибут фамилии SURNAME).

Во-вторых, в качестве строк в результат войдут только те записи из делителя DIVISOR, которые при декартовом произведении результата на делитель DIVISOR × RES содержатся в делимом DIVIDEND.

Таким образом, после деления мы получаем две строки с фамилиями Орлов и Володина. Преподаватель Шуверов не попал в результирующее отношение, т. к. он не ведет лабораторные по дисциплине "СУБД".

Таблица **DIVIDEND**

SURNAME	DISCIPLINE	KIND
Орлов	СУБД	Лекция
Орлов	СУБД	Лаб. работа
Орлов	Языки программирования	Практ. занятие
Володина	СУБД	Лаб. работа
Володина	Языки программирования	Практ. занятие
Шуверов	Языки программирования	Практ. занятие

/

Таблица **DIVISOR**

DISCIPLINE	KIND
СУБД	Лаб. работа
Языки программирования	Практ. занятие

=

Результат деления **RES**

SURNAME
Орлов
Володина

Рис. 4.9. Демонстрация операции деления

Наконец, мы добрались до наиболее важной реляционной операции — *операции соединения*. Существует несколько способов соединения отношений (естественное слияние, внешнее объединение и т. д.) и все они основаны на декартовом произведении, только теперь после перемножения строк двух таблиц на результат накладываются различные дополнительные ограничения. Допустим, в нашем распоряжении имеется две таблицы, в таблице CHAIR хранится информация о кафедрах вуза, а в таблице TEACHERS — о преподавателях. Продемонстрируем наиболее распространенную операцию *естественного слияния*. Таблицы соединяются по общему столбцу, в таблице кафедр это первичный ключ CHAIR_KEY, а в таблице педагогов — одноименный столбец внешнего ключа.

В результирующей таблице не нашлось места для кафедры химии. Это особенность естественного слияния, в таблице TEACHERS нет ни одной записи с внешним ключом, ссылающимся на кафедру химии (значение 4), поэтому кафедра химии не попала в итоговую таблицу. Если же нам требуется увидеть и "брошенные" записи, то вместо естественного слияния необходимо воспользоваться *внешним слиянием*. Различают *правое* и *левое* внешние соединения. На рис. 4.11 представлено правое внешнее соединение, при котором кортежи отношения CHAIR (таблица расположе-

на справа), не имеющие совпадающих значений в общих столбцах отношения TEACHERS, также включаются в результирующее отношение.

Таблица **TEACHERS**

SURNAME	CHAIR_KEY
Орлов	1
Володина	1
Шуверов	3
Калюжный	2
Аскеров	2

⊳◁

Таблица **CHAIR**

CHAIR_KEY	CHAIR
1	Высшая математика
2	Физика
3	Информатика
4	Химия

=

Результат естественного слияния

SURNAME	CHAIR_KEY	CHAIR
Орлов	1	Высшая математика
Володина	1	Высшая математика
Шуверов	3	Информатика
Калюжный	2	Физика
Аскеров	2	Физика

Рис. 4.10. Операция естественного слияния

Таблица **TEACHERS**

SURNAME	CHAIR_KEY
Орлов	1
Володина	1
Шуверов	3
Калюжный	2
Аскеров	2

⊳⊂

Таблица **CHAIR**

CHAIR_KEY	CHAIR
1	Высшая математика
2	Физика
3	Информатика
4	Химия

=

Результат правого внешнего соединения

SURNAME	CHAIR_KEY	CHAIR
Орлов	1	Высшая математика
Володина	1	Высшая математика
Калюжный	2	Физика
Аскеров	2	Физика
Шуверов	3	Информатика
NULL	4	Химия

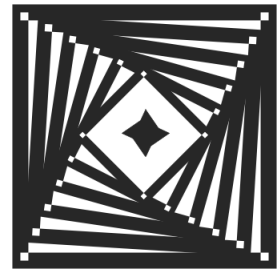
Рис. 4.11. Демонстрация правого внешнего соединения

В одной из строк результирующей таблицы в поле SURNAME мы обнаружим определитель NULL. Это произошло из-за того, что в таблице педагогов не хранится информация о сотрудниках кафедры химии.

Резюме

Своим появлением на свет реляционная модель данных обязана талантливому математику Эдгару Фрэнку Кодду. Именно он предложил четкий и выверенный математический аппарат, позволяющий описывать отношения между множествами. Физическое представление отношения имеет вид двумерной таблицы, состоящей из строк и столбцов. Доктор Кодд сформулировал требования к реляционной таблице:

- уникальность имен столбцов;
- каждая строка хранит информацию об отдельной сущности, строки таблицы отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы;
- каждая ячейка таблицы обладает свойством атомарности — содержит единственное значение;
- порядок расположения строк и столбцов не влияет на хранение данных.



Глава 5

Модель "сущность-связь"

Не секрет, что процесс разработки программного обеспечения сопряжен со многими сложностями: поиск эффективных алгоритмов, подбор подходящих структур данных, отладка и тестирование "непокорного" кода, дизайн удобного интерфейса приложения. Но ни одна из перечисленных трудностей ничто в сравнении с проблемой поиска общего языка между заказчиком и разработчиком программного продукта. Мои слова подтвердит любой программист, написавший хотя бы одну программу по просьбе третьих лиц. Проблема многогранна, но в ее базисе лежит всего-навсего одно противоречие — программист и заказчик разговаривают на разных языках. Бухгалтер, экономист, аптечный работник, строитель или любой другой узкий специалист прекрасно понимает, чего он хочет от будущей БД, но объяснить это программисту не в состоянии. Их как пропасть разделяет языковой барьер. Радиоинженер, мечтающий заполучить программу, макетирующую сверхновый процессор, сталкивается с разработчиком программного обеспечения, в совершенстве владеющим десятком языков программирования, но не имеющим ни малейшего представления не только о законах Кирхгофа, но и даже не способным по внешнему виду отличить транзистор от диода. У медали имеется и обратная сторона: разработчик базы данных, стремясь проверить адекватность построенной им модели, обращается за помощью к заказчику программного продукта. Программист рассчитывает, что последний сможет оценить степень соответствия заказанной и спроектированной информационных систем. Заказчик и рад это сделать, но он ничего не понимает в реляционной модели и в современных языках программирования…

К сожалению, реляционная модель неудобна в период проектирования будущей БД не только с точки зрения заказчика, но и даже с позиций профессионального разработчика ПО, превосходно знающего детище Эдгара Кодда. Во-первых, модель не в состоянии наглядно представлять смысл хранимых данных. Попробуйте на досуге смоделировать и изобразить на листе бумаги производственный процесс химического предприятия в виде плоских двумерных таблиц… Во-вторых, хотя физически реляционная модель превосходно поддерживает взаимосвязи между таблицами, но средств наглядного отображения особенностей этих связей у нее явно недостаточно.

Проблема упрощения процесса проектирования реляционной базы данных, так чтобы, с одной стороны, модель была понятна и неспециалисту, а с другой — вполне устраивала подготовленного программиста, не нова. Работа над этой проблемой вызвала к жизни целое семейство *семантических моделей* данных. На сегодняшний день существует несколько эффективных решений, мы познакомимся с ключевым из них — моделью, разработанной Питером Ченом (Peter Pin-Shan Chen). Указанная модель известна под названием "сущность-связь" (Entity-Relationship) или просто ER-модель. Впервые она была анонсирована в марте 1976 г., когда П. Чен опубликовал работу "The Entity-Relationship Model. Toward a Unified View of Data" в научном журнале "Transactions on Database Systems (TODS)"[1].

ЗАМЕЧАНИЕ

Модель "сущность-связь" относится к разряду концептуальных. Другими словами, она представляет общий взгляд на данные и позволяет нам на понятийном уровне разобраться с тем, что будет представлено в будущей базе данных. Ответ на вопрос, как данные должны быть реализованы на практике, нужно искать у реляционной модели.

Достоинство предложенного П. Ченом решения в том, что ER-модель представляется в виде наглядных графических диаграмм. Для того чтобы научиться читать и составлять эти диаграммы, не требуется глубокой специальной подготовки. Терминология ER-модели построена на базе понятий реляционной модели, мы уже знакомы с определениями "сущность", "атрибут" и "связь".

ЗАМЕЧАНИЕ

Доктор Чен является профессором университета штата Луизиана. Читатель, владеющий английским языком, может обратиться к его страничке в сети Интернет: **http://www.csc.lsu.edu/~chen/chen.html**

Сущности и атрибуты в ER-модели

Построение ER-модели начинается с выявления всех типов сущностей, подлежащих хранению в будущей базе данных. Различают две разновидности типов сущностей: слабую и сильную. Слабый тип находится в зависимом состоянии от сильной сущности, напротив сильный тип вполне самостоятелен и ни от кого (или чего) не зависит. На диаграммах сильный тип изображают в виде прямоугольника с названием типа сущности внутри него, для обозначения слабого типа сущности контур прямоугольника рисуют двойной линией (рис. 5.1).

Каждый тип сущности обладает некоторым набором атрибутов, в которых хранятся значения, описывающие конкретную сущность. Различают простые, составные, однозначные, многозначные и производные атрибуты. Главное отличие между

[1] Журнал TODS принадлежит семейству изданий, выпускаемых ассоциацией Association for Computing Machinery (ACM). Адрес в Интернете **http://www.acm.org/**.

простым и составным атрибутом в том, что простой состоит из одного компонента, а составной из нескольких. Примером составного атрибута может стать почтовый адрес, он включает группу простых атрибутов (индекс, страна, город, улица, дом). Однозначный атрибут содержит одно-единственное значение для сущности, например, фамилия для типа сущности "Сотрудник". Многозначный атрибут допускает одновременное хранение нескольких значений, например, у сотрудника может быть несколько телефонных номеров. Производный атрибут содержит значение, полученное на основе данных, хранящихся в других атрибутах. Например, возраст сотрудника можно легко вычислить, зная день его рождения и сегодняшнее число. Графическое представление атрибутов на ER-диаграмме приведено на рис. 5.2.

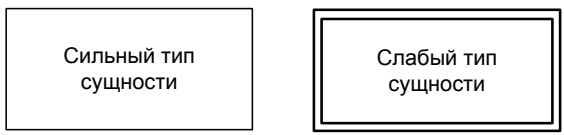


Рис. 5.1. Обозначение слабого и сильного типов сущности

Замечание

В реляционных таблицах производные атрибуты редко превращаются в реальные поля, физически хранящие значения. Ведь нет смысла занимать драгоценную память под данные, которые можно получить путем элементарных вычислений.

Рис. 5.2. Представление атрибутов на диаграммах ER-модели

На схеме атрибут изображают в виде эллипса, к своей сущности его присоединяют линией. Название атрибута записывают внутри эллипса. Если атрибут входит

в состав первичного ключа, то его название подчеркивают. Производный атрибут обводят пунктирной линией, многозначный — двойной.

Замечание

При рисовании многозначного атрибута некоторые разработчики вместо черчения эллипса с двойным контуром рисуют обычный эллипс, но соединяют его с сущностью двойной линией. То же самое можно сказать и о производном атрибуте, только на этот раз эллипс соединяют с прямоугольником пунктирной линией.

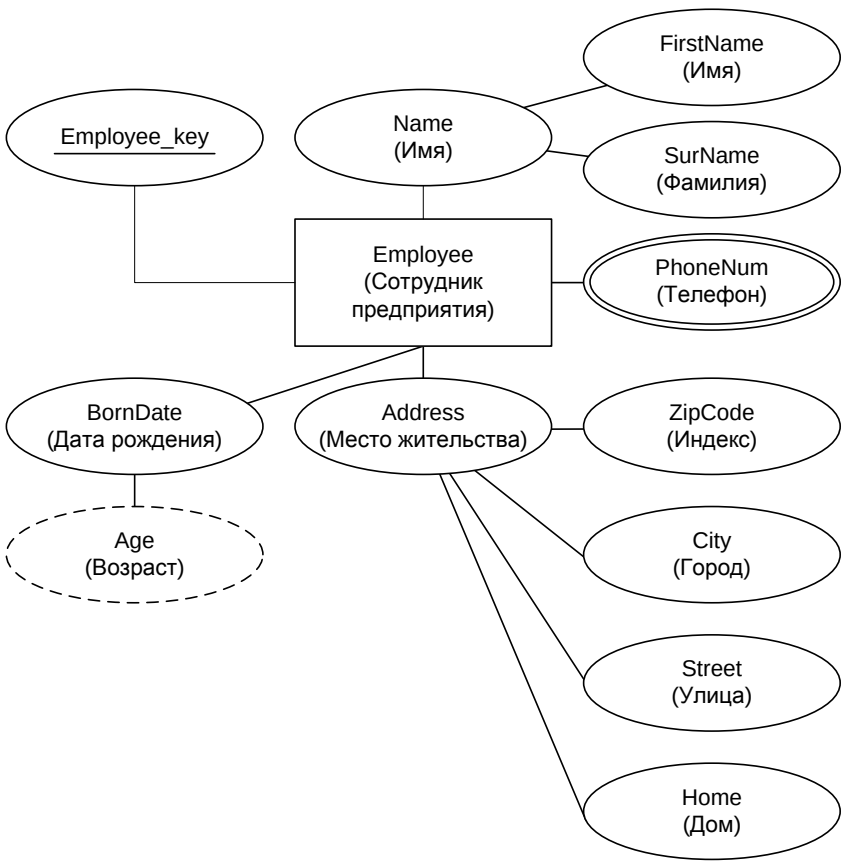


Рис. 5.3. Представление типа сущности на диаграммах ER-модели

Рассмотрим представленный на рис. 5.3 фрагмент ER-модели. Это диаграмма "Employee", описывающая тип сущности "сотрудник предприятия". Как видите, на диаграмме мы сумели представить все имеющиеся в нашем распоряжении разновидности атрибутов. Атрибут "Employee_key" выполняет функции первичного ключа сущности. Имя "FirstName" и фамилия "SurName" являются ингредиентами составного атрибута "Name". Кроме этого у нас имеется еще один составной атрибут "Address", он хранит почтовый адрес служащего. Обведенный двойной линией

многозначный атрибут "PhoneNum" говорит нам о том, что у реального человека может быть несколько контактных телефонных номеров. Производный атрибут "Age" отражает возраст сотрудника, на схеме этот атрибут соединен с датой рождения "BornDate". Такая подробность подскажет разработчику, что расчет возраста должен вестись от даты появления на свет работника предприятия. Согласитесь, что для понимания сути предложенной диаграммы не требуется специальной подготовки, поэтому, нарисовав подобную модель будущей таблицы, можно без сомнений идти к заказчику базы данных.

Подтипы сущностей

Чем больше в организации числится персонала, тем сложнее организовать учет данных о нем. Люди обладают разными специальностями и разными профессиональными навыками. Если фирма располагает научно-исследовательскими лабораториями, то нам потребуется знать ученые степени и звания сотрудников. Если в этой же фирме ведется производственная деятельность, то необходимо учитывать квалификационные разряды металлургов, фрезеровщиков и токарей. Если у нас имеется служба безопасности, то следует обладать информацией о спортивных достижениях сотрудников охраны. Список можно продолжать до бесконечности, чем шире спектр интересов предприятия, тем больше атрибутов потребуется для описания его сотрудников. Как следствие, у всех лиц, задействованных в нашей фирме, кроме общих данных (фамилия и имя, дата рождения и т. п.) появляются уникальные черты, характерные только для узких профессиональных групп. Как организовать хранение данных в таких случаях?

Лобовым решением подобных задач может стать попытка определить один-единственный универсальный тип сущности (например, как на рис. 5.3) и, пытаясь учесть все возможные случаи жизни, снабдить его всем возможными спектром атрибутов. Указанное решение вполне работоспособно, но абсолютно нерационально. В результате, принимая на работу обычную уборщицу, сотрудник отдела кадров будет вынужден указывать, что она не является доктором наук, не обладает черным поясом по каратэ и не имеет опыта пилотирования на вертолете в ночное время… В результате наш программный продукт окажется весьма неудобным в эксплуатации и займет первое место в "плеяде" плохо спроектированных приложений.

Вполне жизнеспособен еще один способ выхода из сложившегося положения. Вместо вооружения типа сущности огромным набором атрибутов, специализирующихся на хранении узконаправленной информации, мы предпринимаем попытку создать сравнительно небольшой набор разнотипных полей универсального назначения и вводим дополнительный атрибут флагов. Вся остальная бизнес-логика БД определяется состоянием флагов. Если установлено значение 0, то в универсальном поле номер `x` хранится номер водительского удостоверения, если 1 — в поле `x` мы найдем номер диплома кандидата наук, если 2 — тайный счет сотрудника в банке в Швейцарии и т. д. С точки зрения экономии памяти рассматриваемый подход вполне приемлем, ведь мы избавились от целой вереницы полей, большей

частью времени хранящих неопределенность NULL. Но зато головную боль приобретают прикладные программисты (они пишут клиентские приложения) и в не лучшем положении оказывается администратор базы данных, несущий ответственность за физическую реализацию БД. У администратора начинаются проблемы с поддержанием непротиворечивости данных, у программистов — со способами построения форм ввода, запросов, отчетов. Усугубить ситуацию могут непредвиденные обстоятельства. Например, в один прекрасный день, к вам обратится начальник отдела кадров и спросит, как ему поступить в ситуации, когда бывший пилот самолета генерального директора переходит на почетную должность директора сауны? Ведь в поле Y пилота хранилось время налета в часах, а у директора сауны — наработка котла в котельной…

Наконец мы пришли к третьему варианту действий разработчика будущей БД. Для уменьшения числа атрибутов следует ввести в ER-модель *подтипы сущности*. Это тот случай, когда общая для всех сотрудников информация хранится в головном типе сущности (*супертипе*), а нюансы и тонкости выносятся в несколько специализированных подтипов.

В нашем примере количество подтипов определяется числом профессиональных групп сотрудников предприятия (рис. 5.4). Все атрибуты супертипа в равной степени принадлежат всем подтипам, ведь у любого сотрудника есть имена и дата рождения. А информация, хранимая в подтипах сущности, специфична для каждой из групп. Обратите внимание на то, что подтипы "Manager" и "Scientific" помечены кружочком с символами "Gs", а подтип "Worker" отмечен символом "G". Символы "Gs" указывают на то, что подтипы относятся к разряду пересекающихся, т. е. ученый одновременно может быть и управленцем нашего предприятия. В таком случае мы допускаем, что данные об одном и том же человеке могут одновременно находиться в таблицах, учитывающих менеджеров и ученых. Символ "G" отмечает непересекающийся тип — рабочий не может одновременно являться ученым или менеджером.

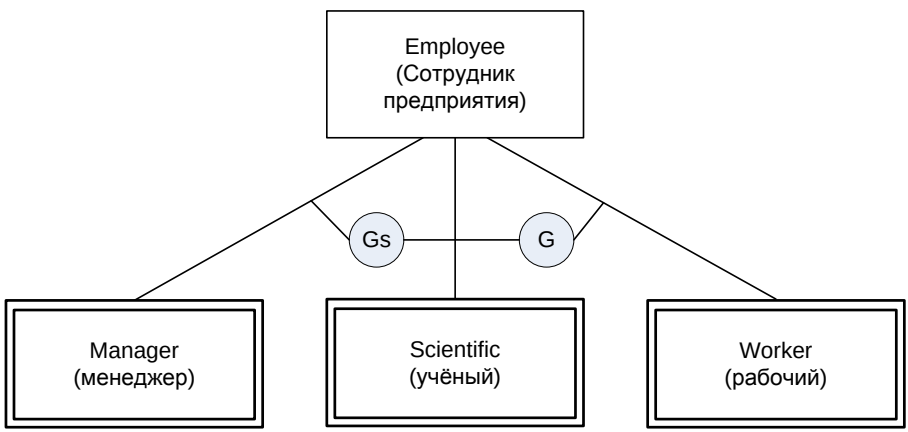


Рис. 5.4. Изображение супертипа и подтипов на диаграмме

ЗАМЕЧАНИЕ

Супертип и подтипы состоят друг с другом в отношениях "один к одному". Это едва ли не единственный случай в реляционных БД, когда появление связи "1:1" значительно упрощает процесс проектирования и сопровождения БД.

Вне всякого сомнения, появление нескольких подтипов сущности усложняет работу над проектом. В процессе проектирования выявляется великое множество проблем и проблемок: не существует единого правила поддержания целостности данных при организации взаимодействия между головной и подчиненными таблицами; возникают определенные сложности в определении первичных ключей в таблицах подтипов; средствами современного SQL весьма непросто создать запрос, объединяющий всю информацию из таблицы супертипа и всех таблиц подтипов; достаточно сложно обеспечить безопасный доступ пользователя к столбцам таблиц подтипов.

Вместе с тем, БД, построенные на основе идеи подтипов сущностей, приобретают неоспоримые преимущества:

- устраняется избыточность данных, т. к. каждый подтип хранит только необходимую информацию;
- упрощается процесс определения доменных ограничений для столбцов таблиц подтипов;
- появление новой (неучтенной в момент проектирования БД) классификационной группы записей приводит к созданию очередной таблицы подтипов. Поэтому не возникает нужды реструктурировать таблицу супертипа (добавление новых или изменение старых полей), с вытекающими отсюда проблемами кардинальной переработки всех клиентских приложений.

Связи в ER-модели

Сформировав полный перечень всех подлежащих учету типов сущностей и их атрибутов, создатель ER-модели переходит к очередному ответственному этапу. Теперь ему предстоит выявить все ассоциации между типами сущностей и на этой основе построить связи между ними. В предыдущей главе уже упоминалось, что явным признаком связи является глагол, который можно применить, характеризуя взаимоотношения между типами сущностей. Например, сотрудник *работает* в отделе, или самолет *выполняет* рейс.

Для того чтобы разработчик мог на схеме отразить смысловую нагрузку связи между сущностями, ее имя, а точнее глагол, показывающий характер взаимодействия между сущностями, записывают внутри ромба. С целью упрощения диаграммы допускается опускать атрибуты типов сущностей, ограничиваясь только первичными ключами (рис. 5.5).

В ER-моделировании различают три типа связи между типами сущностей: "один к одному" (1:1), "один ко многим" (1:M) и "многие ко многим" (M:N). При появлении на диаграмме связи типа "1:1" следует задуматься, не является ли пред-

полагаемый тип сущности всего лишь атрибутом. Исключение составляет обсужденный ранее механизм супертипов и подтипов.

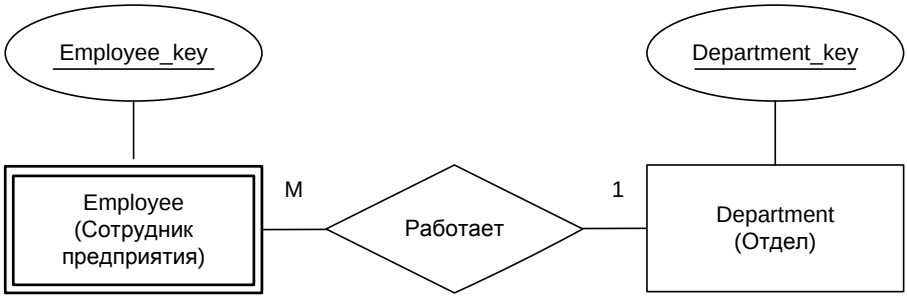


Рис. 5.5. Представление связи "1:M" на диаграммах ER-модели

На рис. 5.5 приведена наиболее часто встречающаяся связь "один ко многим" — в одном отделе работает много сотрудников. На диаграмме отдел "Department" представлен как сильная сущность, сотрудник "Employee" — как слабая. Такое решение объясняется тем, что сотрудник находится в подчиненном отношении к отделу, в котором он трудится. Часто программисты для повышения информативности диаграмм при обозначении связи вместо символа "M" явным образом указывают *мощность связи*. Например, обозначение (1, 10) говорит о том, что минимальное значение мощности соответствует 1, а максимальное 10 (другими словами, в отделах может работать от 1 до 10 сотрудников). Такие пояснения могут оказаться весьма полезными при практической разработке программного обеспечения, т. к. более подробно отражают бизнес-правила автоматизируемого учреждения или предприятия.

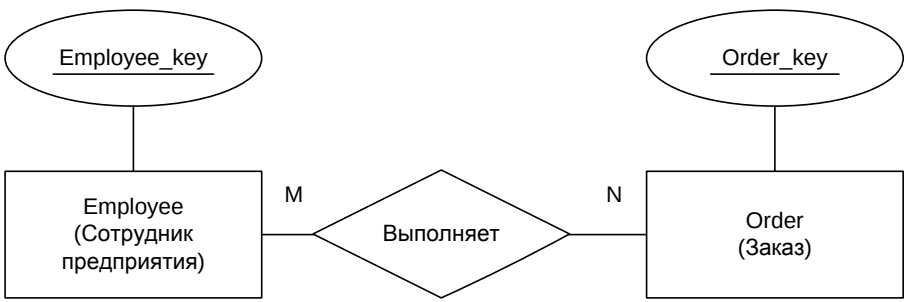


Рис. 5.6. Представление связи "M:N" на диаграммах ER-модели

Проектируя ER-модель, нам следует особое внимание уделять связи "многие ко многим". Вполне реальна ситуация, когда несколько сотрудников одновременно выполняют несколько производственных поручений "Order" (рис. 5.6). Например, один и тот же автомеханик в рабочий день может получить заявки на обслуживание нескольких автомобилей, при этом отдельно взятый автомобиль может попасть

в руки нескольких специалистов (допустим, на ремонт двигателя и на замену электропроводки).

Хотя связь "M:N" легко изобразить на диаграмме, ее далеко не просто реализовать физически, ведь реляционные базы поддерживают только отношение "один ко многим". Безусловно, рассматриваемая ситуация не безвыходная. Позднее, обсуждая процесс нормализации базы данных, мы очень подробно рассмотрим способ решения подобных задач на физическом уровне. А сейчас, пока мы находимся на концептуальном уровне моделирования, лишь запомним, что в тех случаях, когда между двумя типами сущностей возникает связь "многие ко многим", разработчик БД создает искусственный тип сущности, выполняющий функции коммутатора между основными сущностями (рис. 5.7).

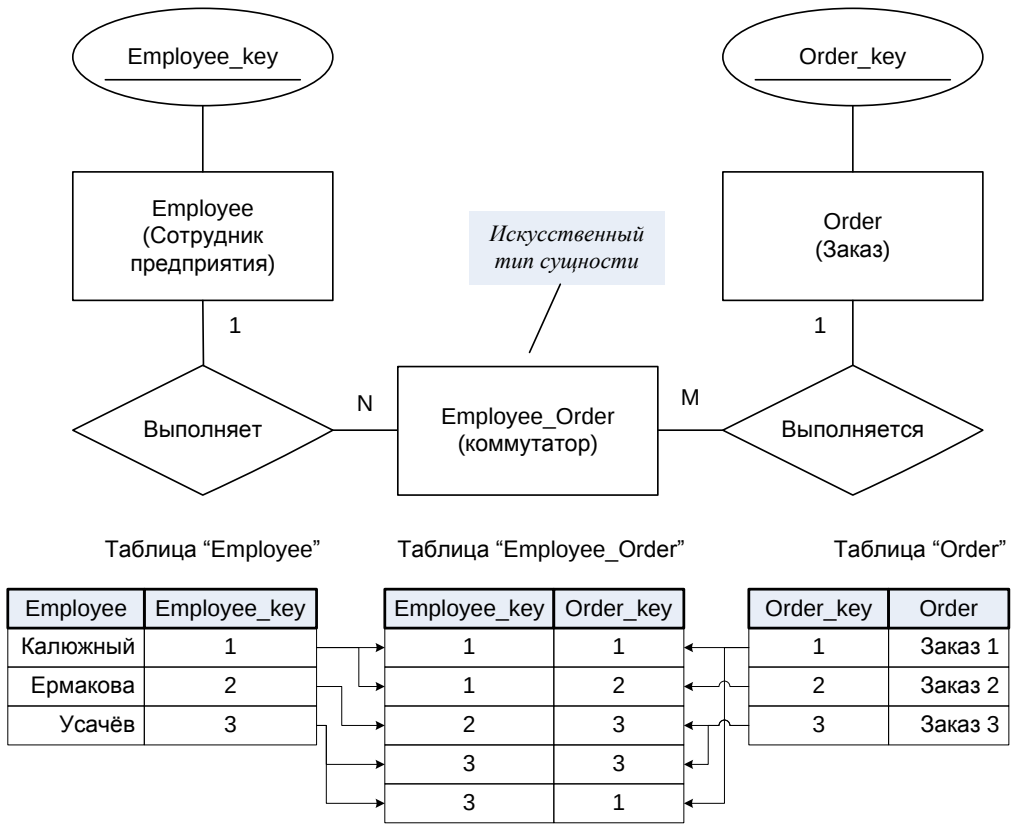


Рис. 5.7. Преобразование отношения "M:N" к двум отношениям "1:M"

Дополнительный тип сущности разрывает связь "M:N" пополам, что позволяет нам трансформировать "неподъемное" для реляционных баз данных отношение "многие ко многим" в пару обычных связей "один ко многим". Независимо от всех хитросплетений нашей ER-модели искусственно созданная сущность-коммутатор в любом случае будет содержать два атрибута внешних ключа. Они предназначены

для поддержания ассоциации между объединяемыми отношением "многие ко многим" типами сущностей, один атрибут коммутатора станет держать связь с расположенным на диаграмме слева типом "Employee", а другой — с правым типом сущности "Order". На диаграммах совсем не обязательно опускаться до степени детализации рис. 5.7. Это может оказаться излишним с точки зрения наглядности схемы, поэтому большинство разработчиков ER-модели предпочитают компромиссное решение (рис. 5.8). Суть компромисса в том, что на диаграммах искусственный тип сущности изображают в виде ромба, вписанного в прямоугольник. Такое графическое решение, с одной стороны, указывает на то, что это будущая таблица, а с другой стороны, напоминает, что данная таблица служит для коммутации между другими таблицами.

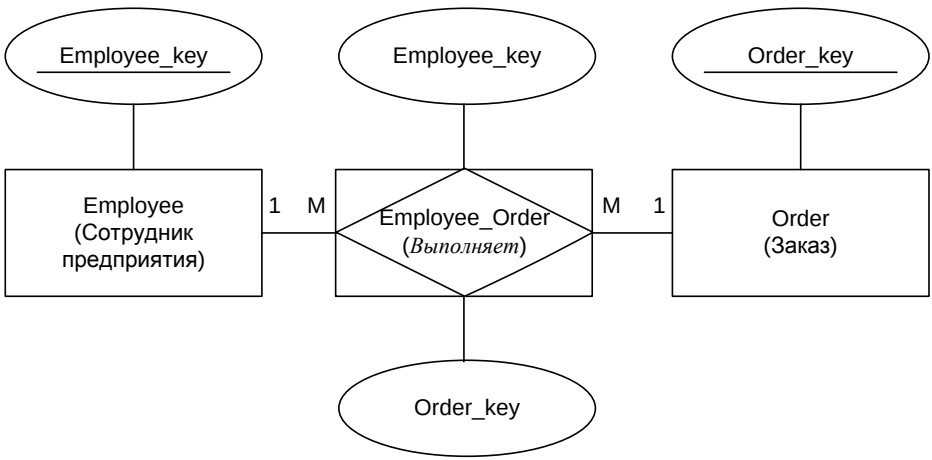


Рис. 5.8. Коммутирующий тип сущности на ER-диаграмме

Сильные и слабые связи

Рассмотрим еще одну особенность взаимодействия между типами сущностей в базе данных — наличие *сильных* и *слабых связей*. Сильная связь обычно возникает между сильным и слабым типом сущностей, в тех условиях, когда существование слабого типа невозможно без поддержки сильного. Например, слабая сущность "самолет" принадлежит сильной сущности "авиакомпания", или слабая сущность "сотрудник" не может трудиться на предприятии, не входя в штат какого-либо из отделов. Напротив, в той ситуации, когда присутствие связи между двумя типами сущности не обязательно, мы говорим о слабой связи. Слабые связи нужно искать между типами сущностей, не находящимися в прямой зависимости друг от друга и, как следствие, способными жить автономно. Например, работа над некоторыми заказами может осуществляться не только в стенах нашего предприятия, но и совместно с каким-то соисполнителем (смежной фирмой). Несмотря на то, что тип

сущности "Accomplice" (соисполнитель) относится к разряду сильных, т. к. может существовать самостоятельно, связь между заказом и смежником оказывается слабой. Ведь нам не всегда нужны помощники, и большинство заказов мы выполняем только своими силами. Об этом нас информирует маленький кружок со стороны необязательной сущности (рис. 5.9).

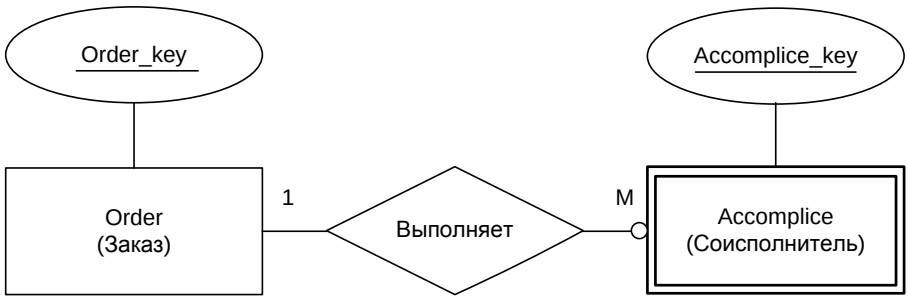


Рис. 5.9. Пример необязательной связи

В последующем, на этапе создания реляционной таблицы "Order", кружок подскажет нам, что информация о соисполнителе заказа необязательна и поэтому поле внешнего ключа допускает вставку определителя `NULL`.

ЗАМЕЧАНИЕ

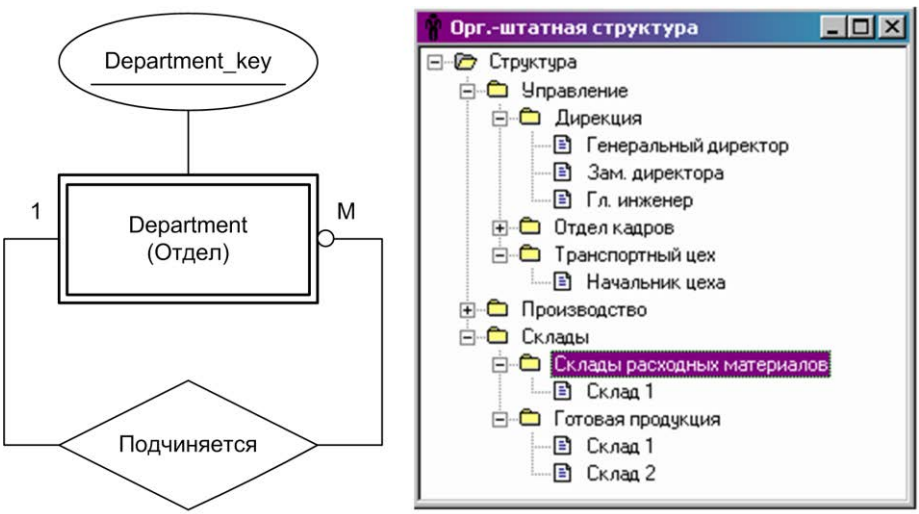
В некоторых нотациях ER-модели для обозначения сильной связи на диаграммах очерчивают ромб двойной линией.

Рекурсивная связь

До сих пор на наших диаграммах мы сталкивались с наиболее часто возникающими в реальной жизни бинарными связями. Теперь поговорим об унарных связях, которые весьма эффективны при организации рекурсивных отношений внутри одной таблицы. Замыкание типа сущности на самого себя окажется весьма полезным при описании сложных иерархических структур, подобных организационно-штатному построению современного предприятия. Взгляните на рис. 5.10, на котором представлено решение такой задачи. Здесь тип сущности "Department" хранит информацию об отделах и службах автоматизируемой фирмы, между которыми явно просматривается правило взаимодействия "главный-подчиненный".

На практике, для построения иерархии, реальной реляционной таблице потребуется как минимум три столбца: поле первичного ключа `Department_key`, поле внешнего ключа `Parent_key` и содержащее названия отделов текстовое поле `Department`. Рекурсивная связь между записями таблицы обеспечивается за счет взаимодействия внешнего и первичного ключей, с этой целью поле внешнего ключа дочернего элемента хранит значение первичного ключа родительского узла. Ес-

ли же идет речь о самом старшем элементе иерархии, который никому не подчинен (в нашем примере во главе дерева расположена папка "Структура"), то в его внешнем ключе окажется определитель NULL. Подтверждение моих слов вы обнаружите во фрагменте таблицы "Department", представленной в нижней части рис. 5.10. Например, узлу "Структура" принадлежат дочерние узлы "Управление", "Производство" и "Склады". В полях внешнего ключа всех трех подчиненных узлов вы найдете число 1, а это и есть значение первичного ключа родительского узла "Структура".



Фрагмент данных таблицы “Department”

Department_key	Parent_key	Department
1	NULL	Структура
2	1	Управление
3	1	Производство
4	1	Склады
5	2	Дирекция
6	5	Генеральный директор
7	5	Замдиректора

Рис. 5.10. Представление унарной рекурсивной связи на диаграмме ER-модели

Связи высокого порядка

Модель Питера Чена отличается особой доброжелательностью и допускает возникновение связей более высокого порядка, чем бинарные: тернарных, кватернарных и т. д. В этом есть определенная логика, ведь в реальном мире объединение

одной связью, скажем, трех сущностей далеко не редкость. Допустим, что в наше хранилище нефтепродуктов поступает бензин одной и той же марки от разных поставщиков, отсюда бензин расходится по бензозаправочным станциям, и из этого же хранилища мы отпускаем бензин ряду оптовых покупателей. Предложенная ситуация смоделирована на рис. 5.11.

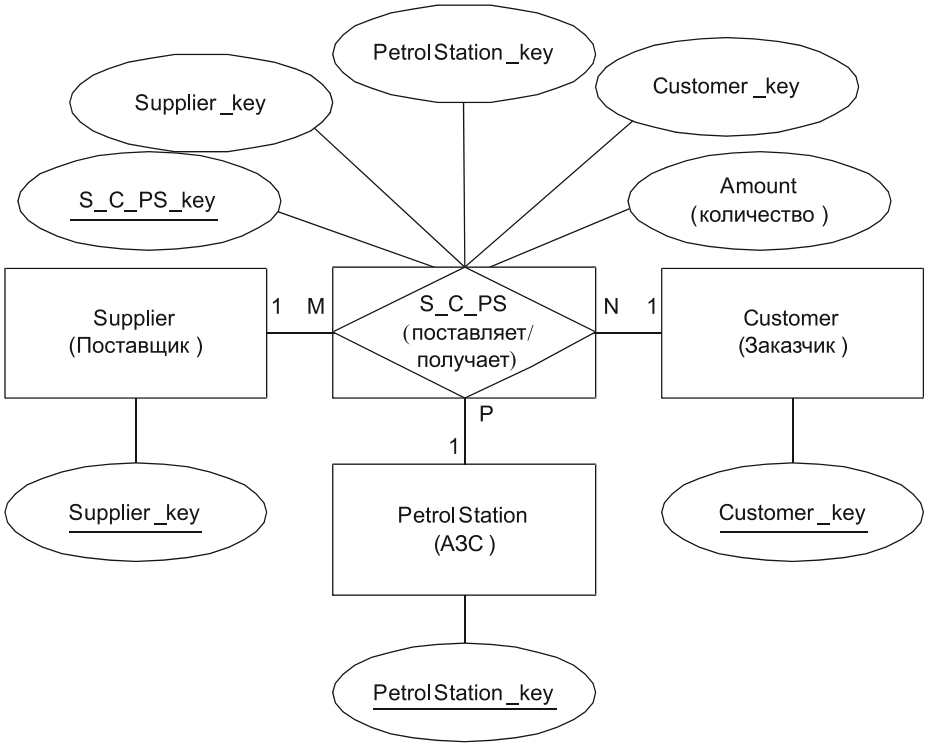


Рис. 5.11. Тернарная связь между типами сущностей

То, что разрешено ER-модели, которая работает на концептуальном уровне проектирования БД, иногда трудно реализовать на физическом уровне. Реляционная модель данных без проблем поддерживает унарные и бинарные связи, но при превышении размерности связи разработчик вынужден идти на определенные хитрости. В данном случае рецепт почти такой же, как и в подробно рассмотренной на рис. 5.6 и 5.7 ситуации с поддержанием связи "многие ко многим" между двумя типами сущностей. И тогда, и сейчас мы выйдем из положения за счет введения искусственной сущности-коммутатора. Коммутатор "S_C_PS" (название создано по первым буквам коммутируемых объектов) объединяет поставщиков, заказчиков и АЗС. Помимо трех внешних ключей, предназначенных для организации связи с интересующими нас сущностями, наша искусственная таблица снабжена первичным ключом и атрибутом "Amount", в котором мы станем хранить объем поставленного/полученного топлива.

Вариации ER-моделей

Модель Питера Чена вполне заслуживает лавры первого простого и одновременно эффективного инструмента моделирования БД. Для ее понимания вовсе не требуется глубоких знаний в области программирования и СУБД. Достаточно способности логически мыслить и немного терпения с крупицей интуиции. Благодаря своей успешности ER-модель Чена не только сразу снискала признательность у обычных программистов, но и вдохновила ряд специалистов на создание схожих по идеологии инструментов проектирования БД. Упомянем лишь наиболее значимые из них.

Один из наиболее популярных нотаций ER-модели принадлежит уже упоминавшемуся во второй главе этой книги Чарльзу Бэчмэну. Бэчман несколько видоизменил графические обозначения связей на диаграммах "сущность-связь". Связи в нотации Бэчмана также чертят в виде линии, соединяющей типы сущностей, но на ней отсутствует ромб с названием связи. Вертикальная черта на конце линии связи означает, что эта сторона представляет отношение "один". На стороне "многие" линия связи расщепляется на три луча. Некоторая схожесть связи на стороне "M" с трехпалой птичьей лапкой привела к тому, что за ER-моделью в нотации Бэчмана прочно закрепилось название "Crow’s Foot model", что дословно означает "воронья лапка" (рис. 5.12).

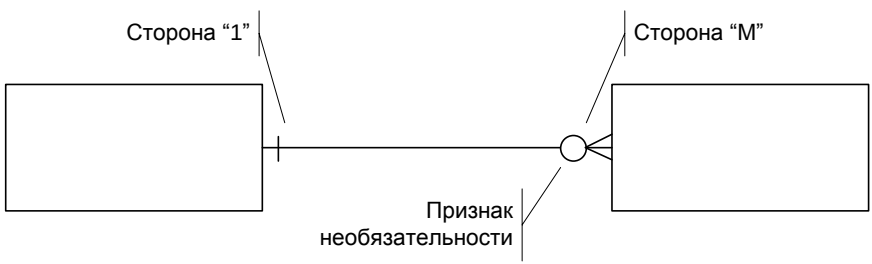


Рис. 5.12. Представление связи на модели "Crow's foot"

На диаграммах Бэчмэна атрибуты сущности рисуют не в виде отдельных эллипсов, а записывают в виде списка, под названием сущности. Благодаря этому модель приобрела более компактную форму представления.

Еще одна, заслуживающая внимания версия ER-моделей получилась в результате работы американской фирмы Hughes Aircraft. Создатели модели воспользовались наработками в области систем автоматизированного производства (Integrated Computer-Aided Manufacturing, ICAM), широко проводимыми в США в 1970-х годах, и реализовали свою модель ICAM Definition, сокращенно IDEF. Несколько позднее модель IDEF была доработана и сегодня более широко известна под именем IDEF1X. IDEF1X впитала лучшие стороны моделей Чена и Бэчмэна.

Различные разработчики создали внушительный перечень программного обеспечения, позволяющего автоматизировать этап концептуального моделирования

БД. Эти продукты объединяют под понятием CASE-системы. Термин CASE (Computer-Aided Software Engineering) можно перевести как автоматизированное проектирование и создание программ. В качестве примеров CASE продуктов стоит упомянуть: ER/Studio (корпорации Embarcadero), ERwin Data Modeler (фирмы Platinum – CA), PowerDesigner (фирмы Sybase), Visio (корпорации Microsoft). На рис. 5.13 представлен экранный снимок процесса построения ER-модели в редакторе ER/Studio, входящем в поставку Embarcadero RAD Studio. Здесь вы обнаружите диаграмму взаимодействия между сущностями "Department" и "Employee". Фактически это не что иное, как аналог нашей более ранней диаграммы (см. рис. 5.5), но изложенный на языке альтернативных моделей "сущность-связь". Как видите, корни современных CASE-систем, специализирующихся на проектировании БД, также уходят в модель Питера Чена.

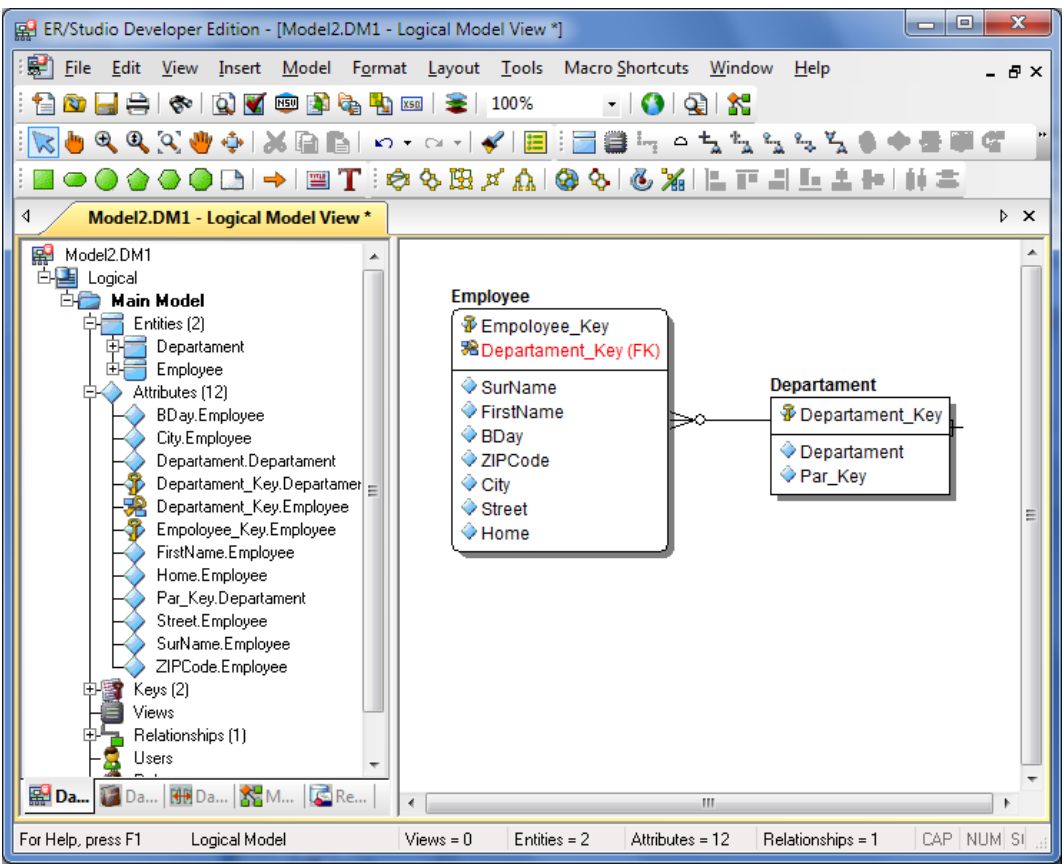


Рис. 5.13. Построение ER-модели в Embarcadero ER/Studio

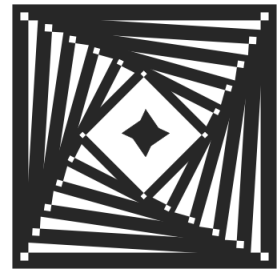
Основная цель CASE-проектирования заключается не столько в автоматизации хода разработки БД, сколько в превращении трудоемкого и малопонятного для непосвященных "ритуала" кодирования в относительно более простой процесс логи-

ческого проектирования. Разработчику зачастую даже не нужно знать языки программирования, достаточно владеть методологией концептуального проектирования БД и уметь пользоваться мышью... В результате CASE-средство оказывается доступным даже начинающему пользователю, что не только ускоряет, но и значительно удешевляет стоимость получаемого программного обеспечения. Ведь зачастую достаточно щелкнуть кнопкой, и из нарисованной концептуальной модели создается физическая БД! Но с другой стороны, качество результирующего продукта, выходящего из-под "пера" CASE-инструмента, и качество "ручной работы" профессионального программиста сегодня сравнивать не стоит.

Резюме

В результате труда разработчика баз данных должна появиться БД, представляющая собой некоторую абстракцию весьма сложной части реального мира. Вновь созданная БД должна с заданной степенью точности соответствовать настоящему миру и четко решать поставленные ей задачи по хранению и обработке данных. Именно поэтому процесс разработки БД включает в себя этап концептуального проектирования, на котором программист создает упрощенную модель будущих данных.

Наиболее показательной концептуальной моделью считается модель "сущность-связь" (ER-модель) Питера Чена. Ее диаграммы отличаются простотой и хорошей информативностью, что позволяет нам не только проектировать будущую БД, но и оценить адекватность этой модели в беседе с заказчиком программного обеспечения.



Глава 6

Нормализация

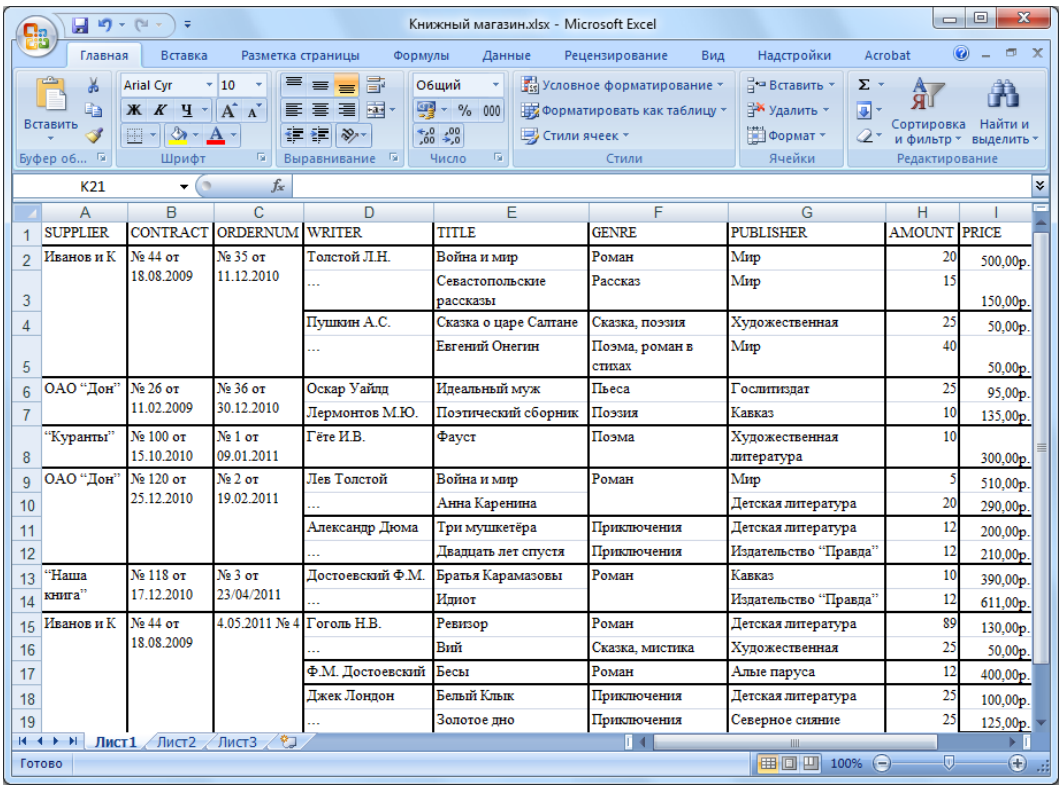
Сегодня мы обсудим процесс проектирования качественных реляционных таблиц, называемый *нормализацией*. Если вы ненадолго вернетесь к *главе 4*, то вспомните минимальный набор требований, предъявляемых к таблице, претендующей на высокое звание "реляционной". Уже тогда мы упомянули термин "нормальная форма". Нормализация преследует несколько целей. Наиболее важная из них — борьба с избыточностью данных. Устранение избыточности достигается не просто за счет рационального назначения размерности полям таблиц, а за счет формирования эффективных взаимоувязанных табличных структур, объединяющих несколько отношений. Так что процесс нормализации выступает в роли основного элемента процесса проектирования БД.

> ***Замечание***
>
> В результате нормализации БД таблицы приобретают дополнительную гибкость, что позволит нам без особых последствий изменять структуру таблиц в будущем, например в процессе модернизации действующей БД.

Победа в борьбе за лишние байты не только приводит к минимизации расходов на хранение данных, но и устраняет ряд косвенных проблем, точнее аномалий, присущих неграмотно построенным таблицам. К ним относятся аномалии вставки, редактирования и удаления данных. Для демонстрации указанных аномалий рассмотрим фрагмент складской ведомости, содержащей информацию о поступлениях печатной продукции в вымышленный книжный магазин. Данные набраны в формате электронной таблицы Microsoft Excel, содержащей колонки с именем поставщика, номером и датой контракта на поставку, номером приходного ордера с датой поступления партии, именем автора книги, названием произведения, жанром, издательством и количеством полученных экземпляров (рис. 6.1).

Для обычного покупателя или продавца вся совокупность или любая часть собранной в таблице Excel информации понятна и находит однозначное толкование. Колонка `SUPPLIER` информирует нас о названии фирмы поставщика, затем следует колонка `CONTRACT` с информацией о контракте на поставку, столбец `ODRDER` с номером приходной накладной и датой поступления книг в магазин. Далее мы обнаружим перечень авторов `WRITER`, названий книг `TITLE`, жанр произведения `GENRE`, издательство `PUBLISHER`, число принятых на склад экземпляров `AMOUNT`, и цена одного

экземпляра хранится в колонке PRICE. В результате самой первой поставки книг от компании "Иванов и К" книжные полки нашего магазина пополнились четырьмя произведениями. Работник магазина сделал об этом соответствующую запись в книге учета. Вся четверка книг поступила в соответствии с контрактом № 44 от 18.08.2009 г. и оприходована ордером под номером 35 от 11.12.2010 г.



	A	B	C	D	E	F	G	H	I
1	SUPPLIER	CONTRACT	ORDERNUM	WRITER	TITLE	GENRE	PUBLISHER	AMOUNT	PRICE
2	Иванов и К	№ 44 от 18.08.2009	№ 35 от 11.12.2010	Толстой Л.Н.	Война и мир	Роман	Мир	20	500,00р.
3				…	Севастопольские рассказы	Рассказ	Мир	15	150,00р.
4				Пушкин А.С.	Сказка о царе Салтане	Сказка, поэзия	Художественная	25	50,00р.
5				…	Евгений Онегин	Поэма, роман в стихах	Мир	40	50,00р.
6	ОАО "Дон"	№ 26 от 11.02.2009	№ 36 от 30.12.2010	Оскар Уайлд	Идеальный муж	Пьеса	Гослитиздат	25	95,00р.
7				Лермонтов М.Ю.	Поэтический сборник	Поэзия	Кавказ	10	135,00р.
8	"Куранты"	№ 100 от 15.10.2010	№ 1 от 09.01.2011	Гёте И.В.	Фауст	Поэма	Художественная литература	10	300,00р.
9	ОАО "Дон"	№ 120 от 25.12.2010	№ 2 от 19.02.2011	Лев Толстой	Война и мир	Роман	Мир	5	510,00р.
10				…	Анна Каренина		Детская литература	20	290,00р.
11				Александр Дюма	Три мушкетёра	Приключения	Детская литература	12	200,00р.
12				…	Двадцать лет спустя	Приключения	Издательство "Правда"	12	210,00р.
13	"Наша книга"	№ 118 от 17.12.2010	№ 3 от 23/04/2011	Достоевский Ф.М.	Братья Карамазовы	Роман	Кавказ	10	390,00р.
14				…	Идиот		Издательство "Правда"	12	611,00р.
15	Иванов и К	№ 44 от 18.08.2009	4.05.2011 № 4	Гоголь Н.В.	Ревизор	Роман	Детская литература	89	130,00р.
16				…	Вий	Сказка, мистика	Художественная	25	50,00р.
17				Ф.М. Достоевский	Бесы	Роман	Алые паруса	12	400,00р.
18				Джек Лондон	Белый Клык	Приключения	Детская литература	25	100,00р.
19				…	Золотое дно	Приключения	Северное сияние	25	125,00р.

Рис. 6.1. Информация о поступлениях товаров в книжный магазин

В поступлении имеется несколько произведений Льва Толстого и Александра Пушкина. При внесении оставшихся трех строк работник посчитал, что ему не требуется повторно указывать название фирмы-поставщика, номер и дату приходного ордера. Ведь и так понятно, что все четыре книги из одной партии. Этим же правилом оператор компьютера руководствовался каждый раз, сталкиваясь с повторами имени автора. Мы поймем, что если в ячейке электронной таблицы пусто, то речь идет о писателе или поэте, упомянутом несколькими строками выше. Более того, сотрудник магазина знает, что именно Антон Павлович Чехов написал рассказ "Палата № 6", а роман в стихах "Евгений Онегин" принадлежит гениальному перу Александра Сергеевича Пушкина, поэтому ему зачастую достаточно лишь названия произведения, чтобы идентифицировать его автора. Но, в отличие от нас, компьютер не читал в школе ни Чехова, ни Пушкина и не сможет правильно сопоста-

вить писателя и произведение из-за незаполненных строк. Более того, при редактировании поля авторов мы можем позволить себе еще бóльшие вольности. Например, написав: "Толстой Л. Н.", "Л. Н. Толстой", "Лев Толстой" и тому подобное. Наш мозг понимает, что речь ведется об одном и том же человеке. Для компьютера это не так. Вполне внятная с нашей точки зрения информация даже для самого совершенного процессора может оказаться некорректной. Судите сами. Наш электронный помощник мыслит категориями единичек и ноликов, поэтому с его точки зрения (и, кстати, не лишенной основания) "Толстой" и "толстой" — абсолютно разные данные, а строки без названия поставщика и номера ордера вообще могут показаться некорректными. Так что при включении в таблицу очередной строки нам требуется играть по правилам компьютера, иначе мы столкнемся с *аномалией вставки*.

Аномалию удаления продемонстрируем на примере имени поставщика `SUPPLIER` и номера приходной накладной `ODRDER`. Вновь взглянем на рис. 6.1. Заметьте, что большинство поставок не ограничивалось одним-единственным произведением и включало несколько наименований книг. Так, по самому первому в списке приходному ордеру № 35 в магазин поступило четыре наименования товара, по второму № 36 — два. Допустим, что в ордере № 36 мы ошибочно ввели книгу Оскара Уайльда и удаляем связанную с ней строку из таблицы. Вместе с пьесой "Идеальный муж" мы вынуждены удалить и все данные о номере и дате приходного ордера. В результате вторая строка с поэзией Михаила Юрьевича Лермонтова или просто потеряет связь с поставщиком и накладной или, что может оказаться еще хуже, перейдет под "опеку" предыдущей накладной. Качество стихов великого русского поэта от этого не пострадает, но нарушится весь складской учет и отчетность бухгалтерии. Поэтому при удалении строки с данными о поставщике и ордере нам потребуется перенести эти сведения в следующую за удаляемой строку. Дополнительные действия не потребовались бы, если данные о поставках были правильно структурированы.

Продолжим критику ненормализованных таблиц. Как вы заметили, в нашей таблице встречаются повторяющиеся данные. В основном они сосредоточены в столбцах `GENRE`, `PUBLISHER` и `SUPPLIER`. При классификации жанра произведения мы неоднократно применяем слова "Роман", "Приключения" и т. д. Многократный повтор данных существенно усложняет их обновление. Допустим, мы решим вместо названия "Издательство Правда" писать просто "Правда". В таком случае, чтобы внести необходимые правки, нам придется кропотливо просмотреть все строки исходной таблицы. Это пример *аномалии редактирования* ненормализованных данных.

Для устранения аномалий придется изрядно потрудиться — провести нормализацию данных. Нормализация — это процесс последовательного преобразования таблиц базы данных к виду, принятому в реляционной модели данных.

Замечание

Специалисты различают пять последовательных этапов (форм) приведения данных к реляционному виду от первой нормальной формы (First Normal Form, 1NF) до пятой

(5NF). На каждой из ступеней нормализации таблица приобретает новые черты, которые потребуются для перехода к очередному этапу. Поэтому, чтобы прийти к 5NF, нужно на каждом из этапов шаг за шагом методично совершенствовать структуру таблиц. Перепрыгнуть хотя бы через одну из ступеней нормализации нельзя!

Первая нормальная форма

В разделе "Реляционная таблица" *главы 3* мы уже вскользь поговорили о минимальном наборе требований к реляционной таблице. Среди требований был упомянут один из важнейших принципов построения реляционных отношений: одно поле — одно значение. Указанный принцип и станет героем первой части рассказа о нормализации данных.

Практически любой человек, делающий первые шаги на поприще проектирования БД, повторяет одну и ту же ошибку — собирает всю подлежащую хранению информацию в одну таблицу. Более того, видимо находясь в поиске острых впечатлений, начинающий разработчик применяет максимум стараний для того, чтобы данные в таблице расположились наихудшим способом. Две наиболее характерных ошибки, связанные с безграмотным заполнением таблицы данными, уже представлены в нашем примере книжного магазина (см. рис. 6.1). Обсудим их.

Худшее, что можно сделать с реляционной таблицей, — разместить в ней *повторяющиеся группы данных*. Повторяющаяся группа появляется каждый раз, когда в одну ячейку таблицы мы пытаемся поместить больше одного значения. В примере магазина я изрядно "постарался" и наплодил повторяющиеся группы в колонке жанров. В результате ряд ячеек столбца `GENRE` содержат по два названия жанра, что с точки зрения реляционной таблицы категорически запрещено. Ведь приходя в театр, в кино или на стадион, мы рассчитываем на отдельное место и сразу требуем администратора, обнаружив в своем кресле другого зрителя. С данными действует тот же самый закон: одна ячейка — одно значение.

Безусловно, из правил бывают и исключения. Иногда по тем или иным причинам программисту приходится идти на "преступление" и закрыть глаза на запрет повторения групп. Такое вполне вероятно при хранении в БД массивов, включающих сотни значений. Но в подобной ситуации надо готовиться к целому букету неприятностей. Как минимум разработчик столкнется с проблемами выборки данных при построении запросов. Допустим, что нам захочется найти все книги детективного содержания. Если мы строго следовали правилу "одна ячейка — одно значение", то задача решается элементарно. Но если нет, то задачка для первого класса превращается в существенную проблему. Справедливости ради отметим, что язык запросов SQL обладает средством для решения схожих задач. Для этого предназначен предикат `LIKE`, но запрос, основанный на `LIKE`, весьма медлителен.

Повторяющиеся группы — лишь одна из разновидностей ошибок начинающего разработчика. Есть еще один характерный промах, связанный с нарушением *атомарности* значения. Под атомарностью значения понимается, с одной стороны, его целостность, а с другой — неделимость. Теперь вновь посмотрим на наш пример.

Хуже всего дело обстоит с колонками ORDERNUM и CONTRACT, т. к. в каждой колонке совместно хранится два разнотипных значения: номер приходного ордера (или контракта) и дата. Это неверно, мы нарушаем правило атомарности.

ЗАМЕЧАНИЕ

Разделение данных на атомы впоследствии позволит нам вооружить БД дополнительным сервисом, в частности разнообразными способами сортировки данных и расширенными методами фильтрации и поиска данных.

Схожая, хотя не столь критичная, ситуация со столбцом WRITER (он приспособлен под хранение имени автора книги). Имя состоит из фамилии, имени и отчества писателя. Если строго следовать правилу атомарности, то исходный столбец следует немедленно разделить на три: имя, отчество и фамилия. Но в нашем случае есть место для споров. Многие программисты вправе возразить, что все три ингредиента имени человека допустимо хранить вместе в одном столбце таблицы. Я не стану противником такой идеи, если разработчик БД даст гарантию, что рядовой пользователь нашей программы (допустим, складской работник) станет всегда корректно заполнять рассматриваемое поле. Ранее уже приводился пример: один сотрудник введет строку "Толстой Лев Николаевич", а другой — "Толстой Л. Н.". С точки зрения электронно-вычислительной машины это абсолютно разные данные, и сумятица с именами впоследствии затруднит поиск и сортировку данных.

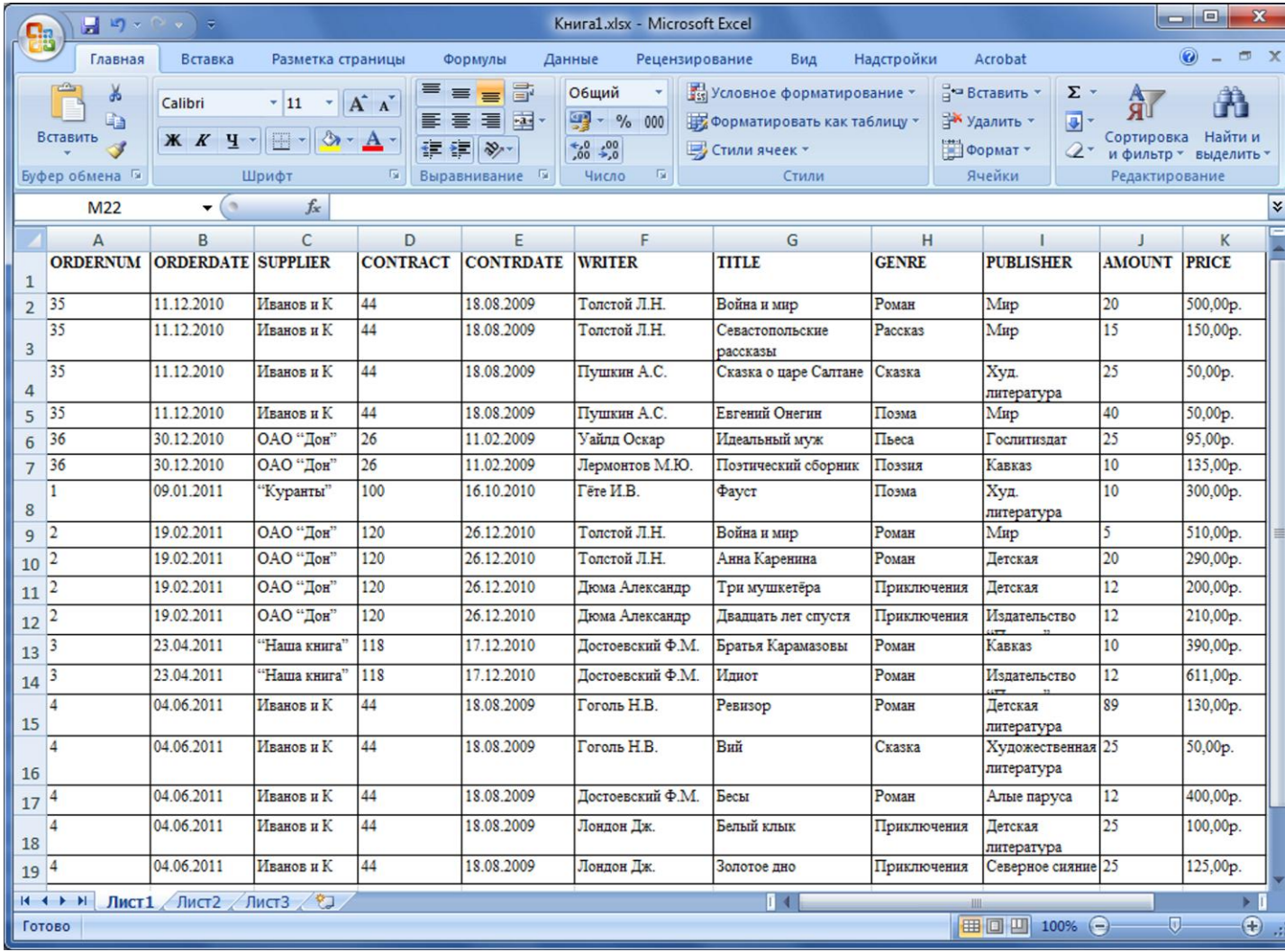
В контексте темы атомарности, стоит рассмотреть и другой пример. Дата также состоит из трех величин — года, месяца и дня. Как лучше организовать ее хранение? В виде раздельных элементов или как одно целое? Зачастую ответ на этот вопрос можно получить, только проанализировав всю логику работы БД. Впрочем, большинство разработчиков обычно считают дату (и время) единым и неделимым значением. Основная причина такого убеждения в том, что все современные СУБД для представления даты/времени ввели соответствующие типы данных, которые к тому же стандартизированы в SQL.

ЗАМЕЧАНИЕ

Таблица приведена к первой нормальной форме, если в ней отсутствуют повторяющиеся группы и все хранящиеся значения неделимы (атомарны).

В результате приведения к 1NF наша таблица стала выглядеть гораздо внушительнее (рис. 6.2). В ней все значения неделимы и нет ни одной повторяющейся группы. Правда, в целях борьбы с повторяющимися группами, нам пришлось временно упростить классификацию произведений по жанрам, ограничившись единственным значением. Например, теперь "Сказка о царе Салтане" обладает только признаком "Сказка", хотя ранее, в ведомости Excel, она была помечена еще дополнительной меткой "Поэзия". Но заверяю вас, что это временное отступление.

К сожалению, из-за многократного введения повторяющейся информации в новой таблице появились многочисленные дубликаты данных. Но на первой ступени нормализации это не столь важно, лишняя информация удалится на следующих этапах нормализации.



Книга1.xlsx - Microsoft Excel

Главная | Вставка | Разметка страницы | Формулы | Данные | Рецензирование | Вид | Надстройки | Acrobat

Вставить | Буфер обмена | Calibri | 11 | Шрифт | Выравнивание | Общий | Число | Условное форматирование | Форматировать как таблицу | Стили ячеек | Стили | Вставить | Удалить | Формат | Ячейки | Сортировка и фильтр | Найти и выделить | Редактирование

M22

	A	B	C	D	E	F	G	H	I	J	K
1	ORDERNUM	ORDERDATE	SUPPLIER	CONTRACT	CONTRDATE	WRITER	TITLE	GENRE	PUBLISHER	AMOUNT	PRICE
2	35	11.12.2010	Иванов и К	44	18.08.2009	Толстой Л.Н.	Война и мир	Роман	Мир	20	500,00р.
3	35	11.12.2010	Иванов и К	44	18.08.2009	Толстой Л.Н.	Севастопольские рассказы	Рассказ	Мир	15	150,00р.
4	35	11.12.2010	Иванов и К	44	18.08.2009	Пушкин А.С.	Сказка о царе Салтане	Сказка	Худ. литература	25	50,00р.
5	35	11.12.2010	Иванов и К	44	18.08.2009	Пушкин А.С.	Евгений Онегин	Поэма	Мир	40	50,00р.
6	36	30.12.2010	ОАО “Дон”	26	11.02.2009	Уайлд Оскар	Идеальный муж	Пьеса	Гослитиздат	25	95,00р.
7	36	30.12.2010	ОАО “Дон”	26	11.02.2009	Лермонтов М.Ю.	Поэтический сборник	Поэзия	Кавказ	10	135,00р.
8	1	09.01.2011	“Куранты”	100	16.10.2010	Гёте И.В.	Фауст	Поэма	Худ. литература	10	300,00р.
9	2	19.02.2011	ОАО “Дон”	120	26.12.2010	Толстой Л.Н.	Война и мир	Роман	Мир	5	510,00р.
10	2	19.02.2011	ОАО “Дон”	120	26.12.2010	Толстой Л.Н.	Анна Каренина	Роман	Детская	20	290,00р.
11	2	19.02.2011	ОАО “Дон”	120	26.12.2010	Дюма Александр	Три мушкетёра	Приключения	Детская	12	200,00р.
12	2	19.02.2011	ОАО “Дон”	120	26.12.2010	Дюма Александр	Двадцать лет спустя	Приключения	Издательство “...”	12	210,00р.
13	3	23.04.2011	“Наша книга”	118	17.12.2010	Достоевский Ф.М.	Братья Карамазовы	Роман	Кавказ	10	390,00р.
14	3	23.04.2011	“Наша книга”	118	17.12.2010	Достоевский Ф.М.	Идиот	Роман	Издательство “...”	12	611,00р.
15	4	04.06.2011	Иванов и К	44	18.08.2009	Гоголь Н.В.	Ревизор	Роман	Детская литература	89	130,00р.
16	4	04.06.2011	Иванов и К	44	18.08.2009	Гоголь Н.В.	Вий	Сказка	Художественная литература	25	50,00р.
17	4	04.06.2011	Иванов и К	44	18.08.2009	Достоевский Ф.М.	Бесы	Роман	Алые паруса	12	400,00р.
18	4	04.06.2011	Иванов и К	44	18.08.2009	Лондон Дж.	Белый клык	Приключения	Детская литература	25	100,00р.
19	4	04.06.2011	Иванов и К	44	18.08.2009	Лондон Дж.	Золотое дно	Приключения	Северное сияние	25	125,00р.

Лист1 | Лист2 | Лист3

Готово | 100%

Рис. 6.2. Информация о поступлениях товаров в книжный магазин, приведенная к 1NF

Чрезмерная избыточность данных — далеко не единственный недостаток таблиц, приведенных к 1NF. Проектируя таблицу, разработчик должен добиться того, чтобы хранимая в ней информация стала не только неизбыточной, но рационально взаимоувязанной. Такая рациональность в таблицах, нормализованных к 1NF, зачастую отсутствует. Поэтому на этапах приведения таблиц к более высоким нормальным формам входящие в отношение атрибуты проверяют на *функциональную зависимость* между собой.

Функциональная зависимость атрибутов

Функциональные зависимости — это те связи, которые могут возникнуть между хранящимися в БД сущностями. Каждая из сущностей, хранящихся в таблицах БД, обладает определенным набором атрибутов. Говорят, что один атрибут отношения (пусть он называется A) функционально зависит от другого (назовем его B), если каждому значению A однозначно соответствует значение B. Другими словами, если в поле A таблицы будут вноситься одинаковые значения, то в соответствующих полях B также окажутся одинаковые данные. В символической форме функциональная зависимость записывается следующим образом: $A \rightarrow B$. Левую часть такой записи называют *детерминантом*, а правую — *зависимой частью*. Степень функциональной зависимости может оказаться разной:

- частичная зависимость;
- полная зависимость;
- транзитивная зависимость;
- многозначная зависимость.

Частичная зависимость возникает при наличии взаимосвязи между отдельным атрибутом и группой атрибутов отношения, выполняющих роль ключа отношения. Это тот случай, когда какой-то из атрибутов отношения окажется в функциональной зависимости только от определенной части составного ключа, а не от всего ключа в целом. На второй ступени нормализации мы обязаны устранить все частичные зависимости и добиться полной функциональной зависимости (т. е. каждое поле, не входящее в состав ключа таблицы, должно полностью от него зависеть).

Полная зависимость — прямая противоположность частичной. Примером полной функциональной зависимости может стать, с одной стороны, индивидуальный номер налогоплательщика (ИНН), а с другой — фамилия, имя и отчество счастливого обладателя ИНН. Нетрудно догадаться, что при наличии функциональной зависимости между двумя атрибутами справедливо как прямое $A \rightarrow B$, так и обратное $B \rightarrow A$ утверждение. То есть по ИНН можно узнать Ф. И. О. человека, а по Ф. И. О. — выяснить ИНН. Полную зависимость между атрибутами обозначают следующим образом $A \leftrightarrow B$.

Транзитивная зависимость возникает между двумя атрибутами, когда они связаны друг с другом через посредника. Например, когда атрибут A зависит от B, а B,

в свою очередь, зависит от C: $A \rightarrow B \rightarrow C$, но обратная зависимость $A \leftarrow B \leftarrow C$ отсутствует.

Наконец *многозначная зависимость* имеет место в ситуации, когда одному значению атрибута A соответствует несколько значений атрибута B.

Замечание

Наличие транзитивной или многозначной зависимости между атрибутами одного отношения можно считать достаточным условием, подтверждающим необходимость разнесения данных атрибутов по разным отношениям.

Наряду с зависимыми, существуют и независимые атрибуты, между которыми нет никакой прямой связи. В нашем случае к независимым атрибутам можно отнести название поставщика и жанр произведения. Для обозначения независимости A от B применяют следующую запись: $A \neg\rightarrow B$. Соответственно, если B также не зависит от A: $B \neg\rightarrow A$. Взаимную независимость двух атрибутов принято обозначать $A \neg= B$.

На рис. 6.3 представлена схема зависимостей между атрибутами нашего проекта базы данных склада книжного магазина. На ней вы обнаружите все перечисленные разновидности зависимостей. Построение подобных схем может оказаться весьма полезным для начинающих программистов, т. к. они позволяют разобраться в хитросплетениях данных и упрощают процесс выявления зависимостей между атрибутами отношений.

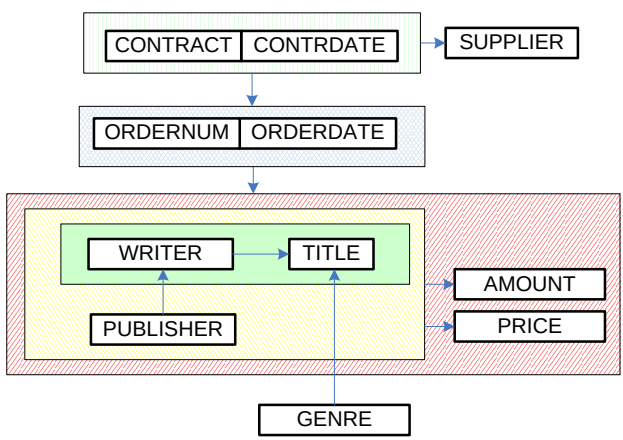


Рис. 6.3. Схема зависимостей между атрибутами

Процесс установления функциональной зависимости тесно связан с порядком назначения ключевых полей таблицы. У нашего (приведенного к 1NF) отношения пока нет атрибутов, переведенных в разряд ключевых. Но как только мы выберем ключевые поля, то все оставшиеся неключевые атрибуты станут в той или иной степени зависеть от ключа таблицы.

Порядок определения первичного ключа

Перед рассмотрением второго этапа нормализации обсудим порядок назначения ключа для отношения. Нам уже известно, что каждый кортеж таблицы нуждается в уникальной идентификации. В качестве идентификатора может использоваться одно или несколько полей таблицы. На высокое звание ключа отношения вправе претендовать набор атрибутов отношения, который полностью и однозначно определяет значения всех остальных неключевых атрибутов. Встречаются случаи, когда одна таблица содержит несколько вариантов сочетаний атрибутов, претендующих на роль первичного ключа (например, автомобиль можно идентифицировать номером кузова и номером двигателя), тогда говорят, что отношение содержит несколько *потенциальных ключей*.

Замечание

Между атрибутами, входящими в состав первичного ключа, и всеми остальными неключевыми атрибутами должна существовать полная функциональная зависимость.

Какие из полей нашей таблицы (см. рис. 6.2) вправе стать потенциальным ключом? Сразу отметаем версию о возможности построении ключа на основе всего одного атрибута. На первый взгляд может показаться, что на звание потенциального ключа претендует поле с номером ордера `ORDERNUM`, но ведь тот же номер может повториться в будущем году. Не окажется прав и тот, кто предположит, что для построения ключа достаточно столбцов с номером и датой приходного ордера. Если вы внимательно изучите таблицу 6.1, то заметите, что по контракту № 44 было две поставки партий книг, о чем свидетельствуют две накладные № 35 от 11.12.2010 и № 4 от 4.06.2011. Таким образом, в состав ключа придется включить целых четыре поля: номер контракта на поставку `CONTRACT`, дату заключения контракта `CONTRDATE`, номер приходной накладной `ORDERNUM` и дату поступления товара `ORDERDATE`. Пока согласимся с таким решением, хотя отмечу, что буквально на следующей странице я предложу вам другого кандидата на роль первичного ключа.

Построение ключей из нескольких атрибутов — крайне неблагодарное занятие. Бич таких таблиц — частичная зависимость некоторых атрибутов таблицы от части первичного ключа. Частичная зависимость может возникнуть только в той таблице, у которой ключ составной. Поэтому разработчики баз данных в качестве первичного ключа стараются использовать только один атрибут. Чаще всего для этих целей в таблицу внедряют дополнительный столбец, не несущий никакой полезной нагрузки с точки зрения представленной в таблице информации, но содержащий заведомо уникальные значения. В простейшем случае это целочисленное поле автоинкрементного типа (по сути, обычный счетчик). Первой записи, появившейся в таблице, присваивается единица, второй — двойка и т. д. Автоинкрементное поле гарантирует, что даже при удалении строки из таблицы ее уникальный номер не будет присвоен какой-либо из вновь вводимых строк.

ЗАМЕЧАНИЕ

Частичная зависимость может возникнуть только в той таблице, у которой ключ составной (построен из нескольких полей).

Современные СУБД позволяют настраивать поведение поля-счетчика. В листинге 6.1 приведен фрагмент кода создания таблицы на диалекте TRANSACT-SQL системы управления базами данных Microsoft SQL Server.

Листинг 6.1. Фрагмент кода создания таблицы средствами T-SQL

```
CREATE TABLE BOOKS
    (BOOKS_KEY INTEGER IDENTITY(1,1) PRIMARY KEY,
     SNAME CHAR(20) NOT NULL,
…,…)
```

Во второй строке кода мы указываем на то, что в таблице `BOOKS` следует создать поле первичного ключа `BOOKS_KEY` на основе счетчика. Особенности работы счетчика определяются двумя параметрами, указанными после слова `IDENTITY`. Первый параметр назначает значение, с которого стартует отсчет, а второй — шаг счетчика. Как видите, и исходное значение, и шаг приращения могут быть любыми. Для чего это нужно? Хотя бы для ситуации, когда два (или более) экземпляров идентичных БД работают в разных книжных магазинах и компьютеры не соединены сетью. Однако нам требуется предусмотреть возможность объединять информацию из двух БД в единое целое. Как же избежать конфликта первичных ключей? Очень просто! В первом экземпляре БД счетчики таблиц настраиваем только на нечетные `IDENTITY(1,2)`, а во второй — на четные значения `IDENTITY(2,2)`. В результате при слиянии БД значения первичных ключей двух таблиц никогда не пересекутся.

ЗАМЕЧАНИЕ

Автоинкрементные поля — хороший, но далеко не единственный способ определения искусственных первичных ключей. Уникальность в состоянии гарантировать поля, построенные на основе меток дата/время (стандартизированный в SQL тип данных `TIMESTAMP`), и 128-битовые поля глобальных уникальных идентификаторов `GUID` (Globally Unique Identifiers).

Надеюсь, что мне удалось вас убедить в преимуществе первичных ключей-счетчиков. Поэтому во всех остальных примерах в этой книге мы станем исходить из правила, что первичный ключ таблиц должен создаваться на основе искусственного поля автоинкрементного типа.

Вторая нормальная форма

Говорят, что таблица соответствует второй нормальной форме, если она приведена к 1NF и в ней отсутствуют частичные зависимости. Другими словами, у такой таблицы не должно быть атрибутов, зависящих только от части первичного ключа.

Вернемся к нашему проекту. Предположим, что вы не послушали моего совета и определили составной первичный ключ таблицы. В ключ вошли четыре поля: номер и дата контракта, номер и дата приходного ордера. Обсудим это решение с точки зрения частичных функциональных зависимостей (рис. 6.4). Наличие составного ключа привело к тому, что поставщик `SUPPLIER` зависит от первой половины ключа, а данные о поставке — от второй. Следовательно, наша исходная таблица не соответствует требованиям 2NF.

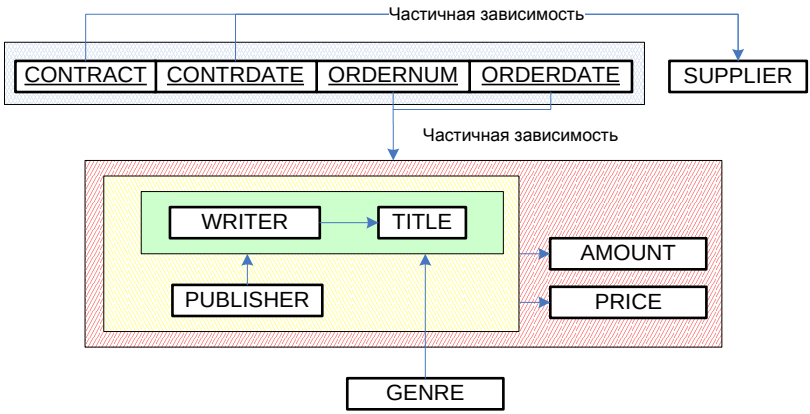


Рис. 6.4. Частичная зависимость неключевых атрибутов от составного ключа

Отказ от составного первичного ключа имеет еще один неоспоримый плюс — упрощается процесс приведения таблицы к 2NF. Доказательство на поверхности: если первичный ключ не является составным, то нет и частичных зависимостей. Нам осталось лишь провести очередную доработку таблицы магазина. Теперь в ней появился еще один столбец `KEY`, представляющий собой поле-счетчик (рис. 6.5).

Table	
PK	**KEY**
	ORDERNUM **ORDERDATE** **SUPPLIER** CONTRACT CONTRDATE WRITER TITLE **AMOUNT** GENRE PUBLISHER **PRICE**

Рис. 6.5. Состав полей таблицы, соответствующей 2NF

Как я и обещал, после приведения к 2NF между полем потенциального ключа `KEY` и любым другим полем таблицы установилось однозначное соответствие, частичные зависимости устранены.

Следует отметить, что отказ от составного ключа — наиболее простое, но не единственное и решение задачи приведения отношения к 2NF. Мы можем пойти и по более сложному пути, например, разорвав исходную таблицу на несколько таблиц, с таким расчетом, чтобы атрибуты с частичной зависимостью и элементы составного ключа разошлись по разным отношениям. В нашем случае мы можем вынести в первую таблицу данные о контракте, т. е. три поля: `CONTRACT` и `CONTRDATE` в качестве составного ключа и имя поставщика `SUPPLIER`. Во вторую таблицу включить поля `ORDERNUM` и `ORDERDATE` как составной ключ, плюс все оставшиеся поля. И сформировать третью таблицу, коммутирующую данные о договоре и о приходных ордерах. Но мы не станем сами создавать себе лишние проблемы и в принципе откажемся от составного ключа, а декомпозицию исходной таблицы (см. рис. 6.5) осуществим на очередных этапах нормализации.

ЗАМЕЧАНИЕ

Таблица приведена ко второй нормальной форме, если она соответствует 1NF и в ней нет частичных зависимостей, т. е. нет неключевых полей, зависящих от части первичного ключа.

Третья нормальная форма

Нашему проекту еще так далеко до совершенства! На втором этапе нормализации, за счет отказа от составного ключа мы в один миг избавились от частичных зависимостей. Но даже такого грандиозного успеха по-прежнему не достаточно для того, чтобы наша таблица стала идеальной, ведь в ней притаился очередной враг реляционных БД — транзитивные зависимости. Чтобы вы не утруждали себя перелистыванием страниц к тому месту главы, где мы уже говорили об этой разновидности функциональных зависимостей, позволю себе напомнить, что под транзитивной понимается такая зависимость между атрибутами отношения, когда один из атрибутов связан с другим через атрибут-посредник $A \rightarrow B \rightarrow C$.

В таблице книжного магазина посредников больше чем достаточно — она прячет в себе несколько транзитивных зависимостей. Наиболее явные из них связаны с атрибутами жанра произведения (`GENRE`) и издательством (`PUBLISHER`), выпустившим книгу. И первый и второй атрибуты напрямую взаимодействуют лишь с группой полей автор (`WRITER`) и название произведения (`TITLE`), а со всеми остальными атрибутами нашего отношения (и между собой тоже) они состоят в транзитивной зависимости. Очередной пример транзитивности — поставщик (`SUPPLIER`). Указанный атрибут непосредственно связан лишь с атрибутами, описывающими контракт, во всех остальных цепочках участвуют посредники. В транзитивной зависимости состоят атрибуты договора о поставке и данные о книге (между этими атрибутами "спрятались" атрибуты, характеризующие приходной ордер). Одним словом, до 3NF нашей таблице очень далеко!

Транзитивную зависимость $A \rightarrow B \rightarrow C$ устраняют одним-единственным способом — за счет выделения атрибутов $B \rightarrow C$ (или $A \rightarrow B$) в другую таблицу. Связи между такими таблицами строятся по ключевым полям: первичный ключ главной таблицы соединяется с внешним ключом подчиненной таблицы.

Для понимания сути третьей ступени нормализации предложу неформальное определение 3NF: внутри приведенной к 3NF таблице должны отсутствовать связи "один ко многим".

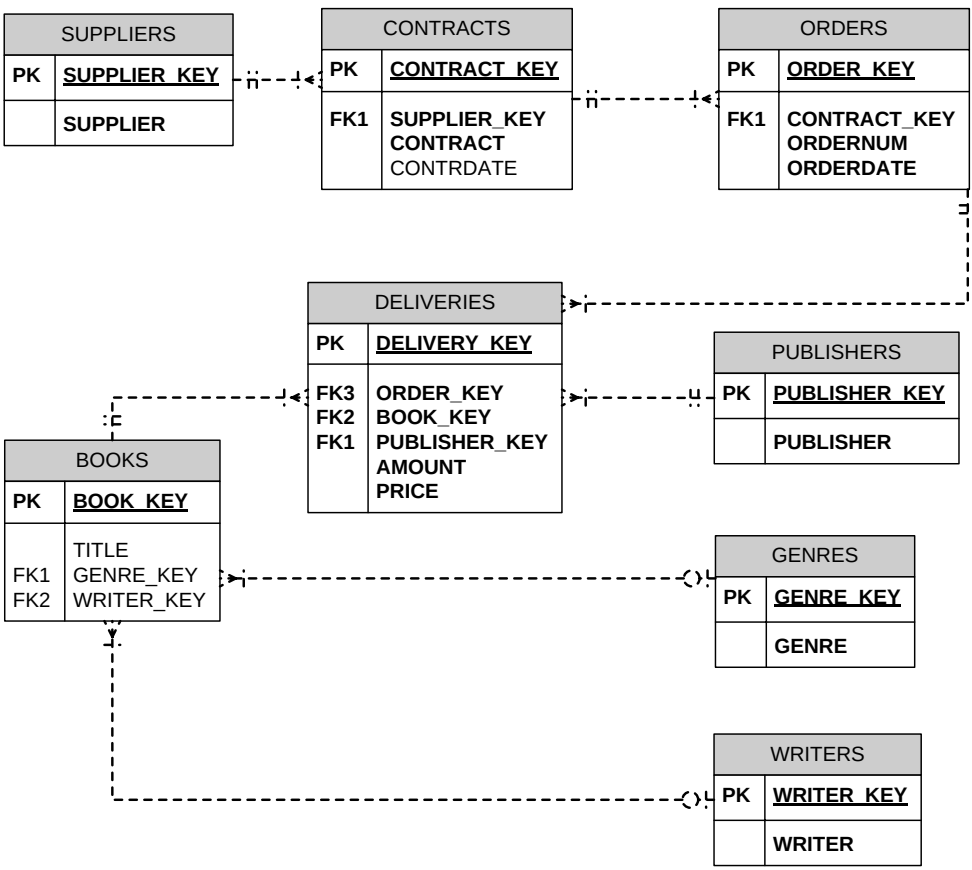


Рис. 6.6. Группа таблиц "Книжный магазин", приведенных к 3NF

В нашем случае, преобразование таблицы из 2NF в 3NF приводит к декомпозиции исходного отношения в набор из восьми отношений (рис. 6.6). Здесь можно выделить четыре главных таблицы:

- поставщики `SUPPLIERS`;
- издательства `PUBLISHERS`;
- жанры `GENRES`;
- авторы `WRITERS`.

Перечисленные таблицы называются главными потому, что они не зависят ни от каких других таблиц (специалисты по ER-моделированию сказали бы, что они представляют сильный тип сущности). Зачастую подобные таблицы называют "справочными", т. к. они являются поставщиками справочных данных для подчиненных по отношению к ним таблиц.

Справочные таблицы весьма полезны, благодаря им:

- Существенно сокращается физический размер дочерней таблицы. Потому, что теперь вместо реальных данных (названий издательств, жанров, авторов книг, поставщиков), занимающих десятки байтов, в подчиненной таблице требуется хранить значение внешнего ключа, которое обычно не превышает четырех байтов.
- При редактировании значения (например, переименование издательства) нет необходимости просматривать сотни записей, а достаточно изменить одно значение в справочной таблице.
- При вводе данных значительно снижается вероятность появления ошибки пользователя, т. к. вместо ввода с клавиатуры ему предлагаются на выбор заранее подготовленные значения.

Все остальные таблицы (контракты `CONTRACTS`, заказы `ORDERS`, поставки `DELIVERIES` и книги `BOOKS`) получают из главных таблиц справочную информацию. В центре БД помещено отношение `DELIVERIES`, оно служит связующим звеном между данными о поставщиках, договорах и заказах, с одной стороны, и данными об экземплярах книг, с другой.

ЗАМЕЧАНИЕ

Таблица приведена к третьей нормальной форме, если она соответствует 2NF и в ней отсутствуют транзитивные зависимости.

Нормальная форма Бойса—Кодда

Помимо 3NF специалисты выделяют усиленную разновидность 3NF — нормальную форму Бойса—Кодда 3NFBC (Boyce-Codd). Смысл усиления в том, что при формулировании первоначальных требований к 3NF Кодд не предусмотрел вероятность совпадения трех событий:

- в нормализуемом отношении может существовать более одного потенциального ключа;
- указанные ключи окажутся составными;
- ключи станут "счастливыми" обладателями хотя бы одного общего атрибута.

Вероятность совместного возникновения перечисленных событий крайне мала, но все-таки не исключена. Поэтому и предусмотрено более строгое определение 3NFBC. Прежде чем мы с ним познакомимся, вспомним, что левая часть символической записи $A \rightarrow B$ называется детерминантом функциональной зависимости. А теперь и дошла очередь и до определения.

Таблица приведена к третьей нормальной форме Бойса—Кодда, когда детерминанты всех ее функциональных зависимостей являются потенциальными ключами.

Вспоминать о нормальной форме Бойса—Кодда следует только в ситуации, когда в таблицах мы активно применяем составные ключи. Если же при проектировании БД разработчик опирается на идею искусственных (например, автоинкрементных) ключей, о существовании 3NFBC можно забыть.

Четвертая нормальная форма

Если третья нормальная форма призвана для борьбы с транзитивными зависимостями, то "героиня" этого раздела состоит в конфронтации с другим злом реляционной модели — многозначными зависимостями. Многозначная зависимость — это связь "многие ко многим". В нашем примере базы книжного магазина связей такого рода, по крайней мере, две.

1. Зачастую над произведением трудятся несколько авторов. Например, бессмертные книги "12 стульев" и "Золотой теленок" и множество остроумных рассказов и фельетонов создал тандем из двух талантливых писателей — Ильи Ильфа и Евгения Петрова. Учебники и научная литература почти всегда создаются коллективными усилиями. Но наша, приведенная к 3NF база данных не сможет закрепить более одного автора за произведением.
2. Каждая из продаваемых в магазине книг классифицируется по жанру. Этот атрибут позволит продавцу мгновенно рассортировать продаваемый товар по полкам, а покупателю быстрее найти интересную для него книгу. Но далеко не исключение, когда одно и то же произведение может попасть под несколько классификационных признаков. Например, "Двадцать тысяч лье под водой" Жюля Верна — не только фантастика, но еще одновременно приключения и детская литература, а роман "Преступление и наказание" Федора Михайловича Достоевского ко всему прочему еще и детектив. К подобному повороту событий наша БД также абсолютно не готова.

Итак, база данных, соответствующая четвертой ступени нормализации, обязана избавиться от многозначных зависимостей между атрибутами отношений. Но каким образом? Ранее уже упоминалась, что реляционная модель данных специализируется только на связях "один к одному" (1:1) и "один ко многим" (1:M). Для реализации связи "многие ко многим" (M:N) приходится идти на хитрость — вводить дополнительную таблицу, которая разделит связь "M:N" на две "1:M". Мы так уже поступали при изучении ER-модели.

В нашем случае потребуются две дополнительные коммуникационные таблицы. Первая из них "вклинится" между таблицей авторов и таблицей книг, новую таблицу мы назовем `WRITERS_BOOKS`. Вторая таблица (`GENRES_BOOKS`) устранит многозначные зависимости между таблицами жанров и книгами. Оба вновь созданных отношения содержат пару внешних ключей для связи с правой и левой таблицами.

Замечание

Таблица приведена к четвертой нормальной форме, если она соответствует 3NF и в ней отсутствуют многозначные зависимости ("многие ко многим").

Схему новой версии базы данных книжного магазина иллюстрирует рис. 6.7. Она представлена в формате модели IDEF1X (говоря по старинке — ER-диаграммы), включающей 10 типов сущностей.

Внимание!

Представленная на рис. 6.7 модель базы данных книжного магазина будет взята за основу для большинства примеров в этой книге.

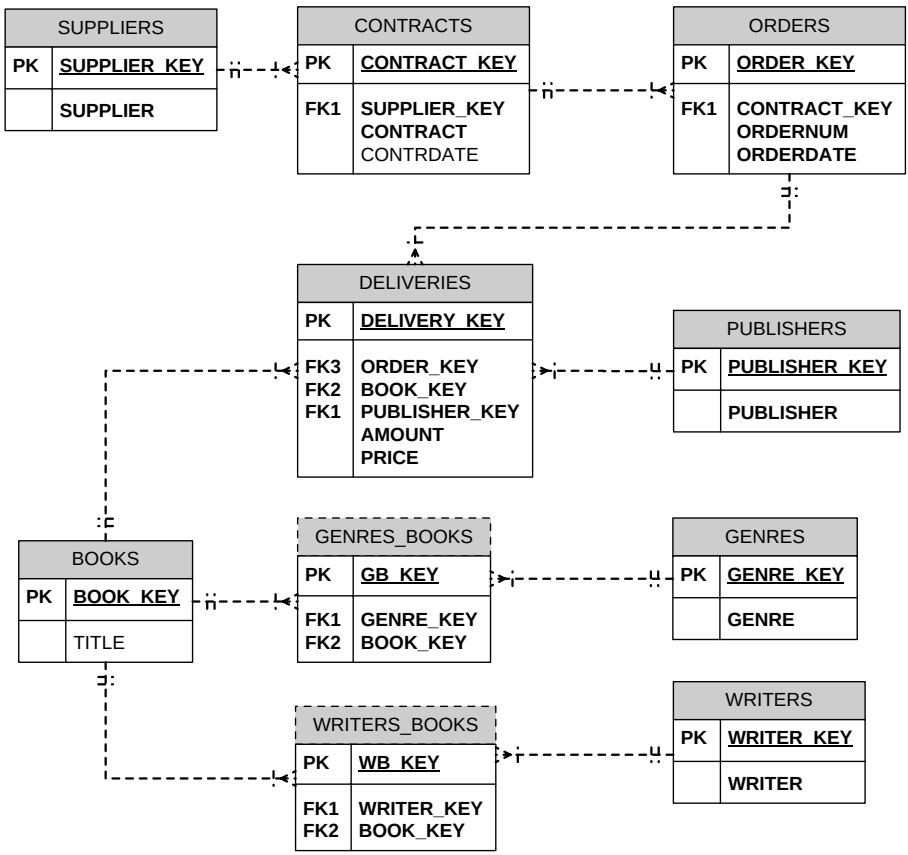


Рис. 6.7. ER-модель БД "Книжный магазин", соответствующая 4NF

Пятая нормальная форма

Для обычного разработчика БД пятая нормальная форма представляет скорее теоретический, чем практический интерес. 5NF требует обеспечения беспрепятственной возможности перестройки данных в нормализованных таблицах. Приведение таблицы к высшей степени нормализации — крайне редкий случай. Это действие имеет смысл, только если таблица содержит так называемые зависимые

сочетания. *Зависимые сочетания* — это свойство декомпозиции, которое вызывает генерацию ложных строк при обратном соединении декомпозированных отношений с помощью операции естественного соединения.

Поясню на примере. Предположим, что в нашей базе данных существует некое отношение, хранящее данные о поставщике (SUPPLIER), книге, полученной от этого поставщика, (BOOK) и издательстве, напечатавшем эту книгу (PUBLISHER) (рис. 6.8). Вполне вероятно, что в процессе работы с базой данных на основе исходной таблицы нам придется построить два запроса:

1. Первый запрос вернул данные о поставщиках и книгах (мы его сохранили в виде таблицы BOOK_SUPPLIER).
2. На базе второго запроса мы получили информацию о книгах и издательствах (таблица PUBLISHER_BOOK).

Обратите внимание, что данные в новых таблицах вполне корректны и соответствуют значениям, содержащимся в исходной таблице.

А теперь попробуем решить обратную задачу — восстановим исходную таблицу на основе данных из BOOK_SUPPLIER и PUBLISHER_BOOK. Для этого создаем обычный запрос на языке SQL, объединяющий обе таблицы по общему для них полю BOOK (листинг 6.2).

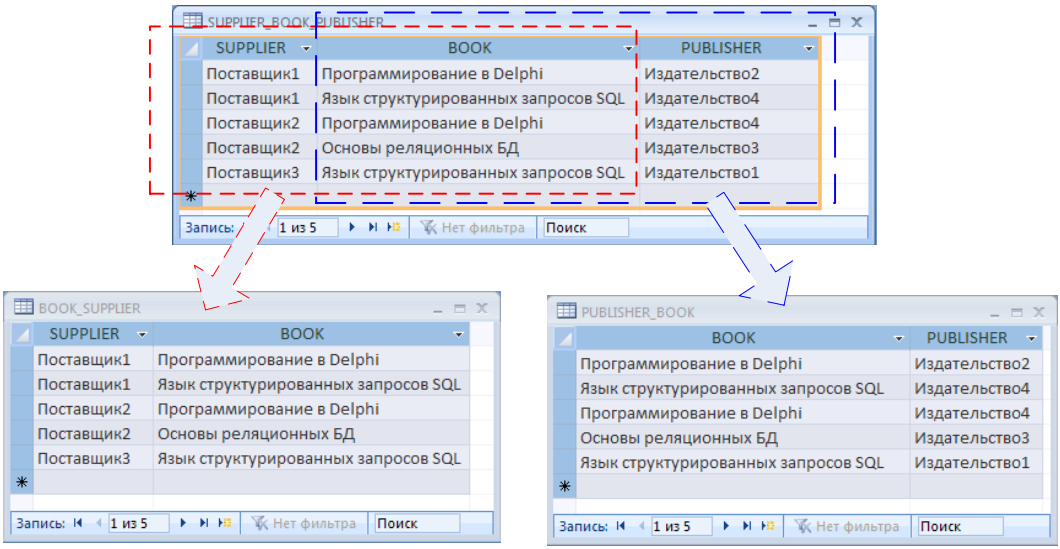


Рис. 6.8. Декомпозиция таблицы

Листинг 6.2. Пример объединения двух таблиц на языке SQL

```
SELECT BOOK_SUPPLIER.SUPPLIER, BOOK_SUPPLIER.BOOK,
       PUBLISHER_BOOK.PUBLISHER
FROM BOOK_SUPPLIER
INNER JOIN PUBLISHER_BOOK ON BOOK_SUPPLIER.BOOK = PUBLISHER_BOOK.BOOK
```

Не стану комментировать текст запроса. Для тех, кто пока не знаком с языком SQL, придется немного потерпеть до *главы 11*, с которой мы начнем разговор о базовом языке реляционных БД. Главное — результат (рис. 6.9) — после соединения двух таблиц мы получили четыре ложных (не существующих в реальной таблице) строки.

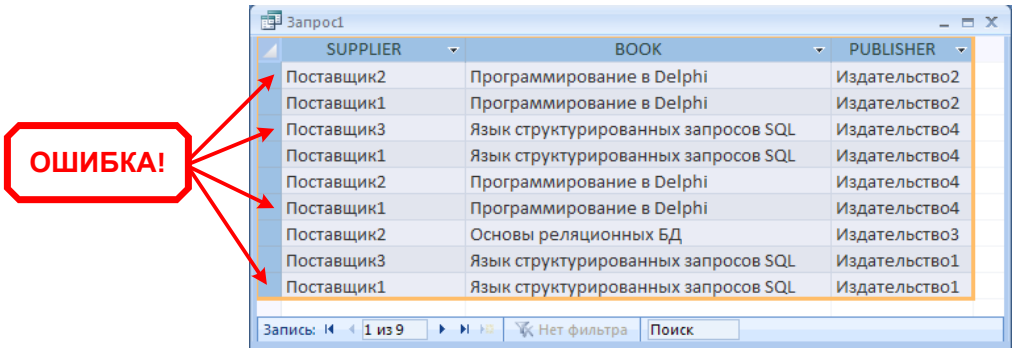


Рис. 6.9. Генерация ложных строк

Резюме

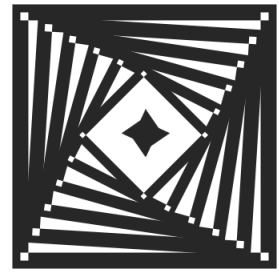
Внимательный читатель наверняка заметил, что БД, созданную в процессе нормализации таблиц, можно получить и альтернативным путем — с помощью методики Питера Чена "сущность-связь". Действительно, предложение построения концептуальной модели БД на основе ER-моделирования и методика нормальных форм Эдгара Кодда не противоречат друг другу и фактически выступают разными сторонами одной медали. Схожесть результатов не означает, что в своей профессиональной деятельности нам следует отдавать предпочтение только одной из методик моделирования, забыв о существовании другой. Напротив, программисту необходимо владеть всеми подходами, используя в своей работе сильные стороны каждой из методик.

А теперь подведем итоги главы, посвященной нормализации отношений. На практике для построения БД приемлемой структуры следует пройти четыре ступени нормальных форм:

- 1NF — все поля в таблицах неделимы и не содержат повторяющихся групп.
- 2NF — все неключевые поля в таблицах зависят от первичного ключа.
- 3NF — отсутствуют избыточные неключевые поля.
- 4NF — устранены многозначные зависимости.

Пятая нормальная форма предназначена для устранения зависимых сочетаний и учитывается только при декомпозиции отношений.

Теории реляционных баз данных и нормализации не стоят на месте. В работе одного из ведущих специалистов по теории баз данных — Криса Дейта уже упоминается шестая нормальная форма, правда с оговоркой, что если речь ведется "о классических операциях проекции и соединения, то пятая нормальная форма является последней нормальной формой" [7].



Глава 7

Индексирование

Представьте себе, что вам в руки попал телефонный справочник крупного города, допустим, Москвы. В увесистом фолианте собрано несколько миллионов телефонных номеров жителей и организаций столицы с их фамилиями, почтовыми и электронными адресами, названиями фирм и многой другой интересной информацией. Воистину бесценная находка, если бы не одна маленькая загвоздка — строки в справочнике не упорядочены по алфавиту, а напечатаны в хаотическом порядке. А ну-ка, попробуйте воспользоваться таким "подарком" и найдите в нем телефонный номер! Полагаю, что энтузиазм даже самого большого оптимиста улетучится после бесплодного пролистывания первого десятка страниц, ведь столь нужный нам телефонный номер с равной вероятностью может оказаться на любой из нескольких тысяч страниц. Парадоксальная ситуация — мы обладаем требуемыми данными, но не в силах ими воспользоваться. Ценность столь полезной информации свелась к нулю из-за ее неудобного представления. Несложно предугадать судьбу этого справочника — он окажется в мусорном ведре.

Электронная база данных и бумажный справочник преследуют единую цель — обеспечить хранение и поиск данных. Это определяет схожесть принципов хранения и объясняет родство недостатков. И рукописная книга, и электронная реляционная таблица не предусматривают автоматическое упорядочивание строк. И там, и там строки записываются в той последовательности, в которой были введены. Такой подход нисколько не противоречит требованиям к реляционным таблицам, согласно задумке разработчиков реляционной модели данных в них кортежи могут содержаться в любом порядке. В базах данных можно упорядочивать данные в момент их ввода, например, оставляя в запасе пустые строки и заполняя их позднее. Подобную идею реализовать несложно, но что делать, если возникнет необходимость изменить порядок следования строк? Сегодня нас устраивает тот факт, что телефонный справочник отсортирован по фамилии владельца, но завтра мы захотим упорядочить данные по номерам телефонов, а послезавтра — по названиям улиц и номерам домов. Может показаться, что в такой ситуации единственным решением задач поиска и сортировки записей стало бы полное сканирование всех неупорядоченных строк отношения. Компьютеру (в отличие от человека) никогда не надоест перебирать записи, и он когда-нибудь обязательно доберется до строки с интересующим нас телефонным номером. Но на операции сортировки и поиска

будут постоянно затрачиваться существенные вычислительные ресурсы. Какой выход из сложившегося положения? Ответ таков — внедрить в СУБД механизм индексирования данных.

Основная задача индекса — обеспечение быстрого доступа к данным по некоторому коду. В любой бумажной энциклопедии, справочнике, словаре эта задача частично решена. В перечисленных книгах информация упорядочена по алфавиту, а в качестве кода (индекса) в простейшем случае выступает первая буква искомого слова. При поиске необходимого слова (термина, названия и т. п.) мы почти сразу попадаем на требуемую страницу, открыв книгу на нужной букве.

В отличие от бумажной книги, реляционная таблица может одновременно обладать несколькими индексами:

- одним первичным индексом;
- одним кластерным индексом;
- несколькими вторичными индексами.

Первичный индекс создается автоматически в момент назначения первичного ключа таблицы. Перед ним стоит одна-единственная задача — первичный индекс должен гарантировать, что в полях, входящих в состав первичного ключа, будут храниться уникальные данные.

Задача *кластерного индекса* иная, он обеспечивает физическое упорядочивание записей в таблице по полям, входящим в индекс. В отличие от первичного индекса, кластерный индекс допускает совпадение значений, хранящихся в проиндексированных полях. В таблице может существовать только один кластерный индекс.

Вторичные индексы описывают дополнительные способы упорядочивания строк таблицы, ускоряют поиск данных и могут проверять данные на уникальность. В отличие от кластерного индекса (физически упорядочивающего строки), вторичные индексы определяют только логический порядок следования строк в таблице, никак не влияя на их физическое расположение. Число вторичных индексов таблицы ограничено лишь здравым смыслом проектировщика и особенностями построения той или иной СУБД.

Замечание

Основное назначение любого индекса — обеспечить эффективный прямой доступ к кортежу отношения (строке таблицы) по его коду.

Быстрый доступ к данным — главный козырь индекса, благодаря нему в БД реализуется несколько полезных сервисных возможностей:

- упорядочивание строк таблиц и отношений;
- поиск данных по полному или даже частичному совпадению;
- возможность поддержки уникальности данных за счет защиты от ввода дублирующихся значений.

За любую услугу приходится платить. Для получения существенного выигрыша в скорости доступа к данным в БД приходится создавать дополнительные структуры. В них хранится код данных и указатель на местоположение этих данных, и так для каждой строки проиндексированной таблицы. Чем больше индексов,

тем больше размерность дополнительных структур. Может случиться так, что суммарный объем служебной информации, отводимой для хранения индексов, превысит размер полезных данных, что несколько печально. Поэтому умение создавать хорошие индексы основано на интуитивной способности профессионального программиста балансировать между двумя противоречиями: поддержанием высокой скорости обработки данных, с одной стороны, и затратами на хранение индексов в памяти, с другой.

Получив общее представление о назначении индексов, перейдем к рассмотрению способов индексирования данных в современных СУБД.

На сегодня существует несколько методов построения индексов. Мы рассмотрим три наиболее популярных: индексы, базирующиеся на технологии хеширования, индексы на основе сбалансированных B-деревьев и битовые индексы. Первая разновидность индексов обычно используется в простейших СУБД, база данных которых содержит небольшое число записей. Индексы, построенные на основе B-деревьев, — прерогатива серьезных СУБД, специализирующихся на обработке значительных по размеру данных. Есть и универсальные решения (например, в Oracle и в Postgre), в них в момент создания индекса система поинтересуется у нас, какой метод построения индексов предпочтителен. Этот сервис заложен в команду `CREATE INDEX` диалектов языка SQL, например PostgreSQL предлагает три варианта алгоритма индексирования: B-дерево, R-дерево и линейное хеширование. Битовые индексы поддерживаются немногими платформами, они необходимы при индексировании столбцов с небольшим количеством различных значений.

Индексы на основе хеширования

Хеширование (hashing) полезно в том случае, когда требуется большой объем значений сохранить в сравнительно небольшой по размерности таблице (точнее хеш-таблице) и обеспечить быстрый доступ к этим значениям.

Существует множество методов хеширования данных. Их общая идея — применение к данным некоторой функции свертки (хеш-функции), по определенному алгоритму вырабатывающей значение меньшего размера (хеш-значение). Например, на вход функции поступает фамилия и инициалы автора книги, а на выходе мы получаем какое-то число. Свертка исходных данных — однонаправленный процесс, получив хеш-значение, мы не сможем восстановить по нему исходные данные, но при построении индексов в этом и нет необходимости. Главное, чтобы хеш-функция на одни и те же входные данные всегда реагировала одним и тем же выходным значением.

Для демонстрации алгоритма хеширования рассмотрим процесс индексирования поля `WRITER` таблицы писателей `WRITERS` из придуманной нами базы данных книжного магазина (рис. 7.1).

Для хранения индексных данных СУБД создает отдельную хеш-таблицу, состоящую из двух полей: поля хеш-значения и адресного поля, содержащего указа-

тель на строку индексируемой таблицы `WRITERS`. Только что созданная хеш-таблица не пуста и уже содержит какое-то число строк. Размерность таблицы определяется настройками СУБД и особенностями хеш-функции, о которых мы поговорим буквально на следующей странице. Изначально адресное поле не заполнено и хранит неопределенный указатель `nil`. Система начинает процесс индексирования — последовательно читает имена авторов книг и передает их на вход хеш-функции. Хеш-функция в соответствии с заложенным в нее математическим выражением осуществляет свертку исходных текстовых данных в числовое значение. Допустим, что хеширование текстовой строки "Толстой Л.Н." дало хеш-значение `h2`. Переходим к строке с порядковым номером `h2` в индексной таблице и в ее поле "Адрес данных" вносим указатель на физическую запись "Толстой Л.Н." исходной таблицы `WRITER`. Эта операция повторяется до тех пор, пока не построится индекс для всей таблицы авторов. По завершению индексирования в хеш-таблице мы обнаружим пары значений: хеш-значение и соответствующий этому значению указатель на строку с реальными данными таблицы `WRITER`.

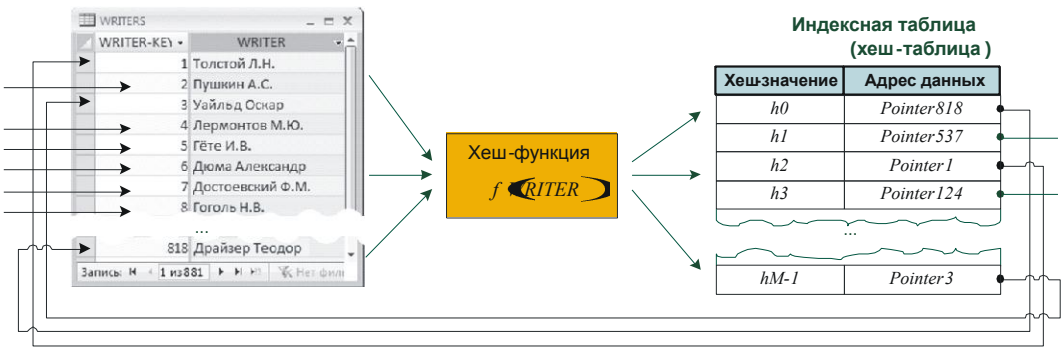


Рис. 7.1. Построение индексной таблицы на основе хеширования

Благодаря индексам существенно упрощается поиск данных, теперь вместо утомительного перебора всех строк исходной таблицы для отыскания совпадения система передаст содержание интересующей нас строки хеш-функции (рис. 7.2).

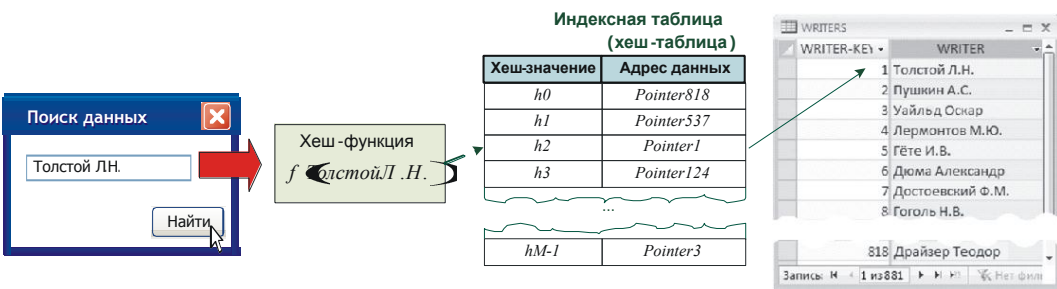


Рис. 7.2. Поиск данных по индексу

Хеш-функция вычисляет хеш-значение и возвращает номер строки в индексной хеш-таблице. После этого нам остается перейти к искомой строке по хранящемуся в индексной таблице указателю.

После создания индекса СУБД обязана поддерживать его в актуальном состоянии. Это означает, что при вводе в базовую таблицу новой записи, редактировании и удалении записи СУБД индексирует изменения и обновляет данные в индексной таблице. Объем вычислительных ресурсов, затрачиваемых на обновление индексов, прямо пропорционален количеству модифицированных записей и в отдельных случаях может существенно снизить производительность БД.

Хеш-функции

В процессе построения индекса на основе технологии хеширования ключевая роль отводится хеш-функции, поэтому уделим ей еще несколько минут нашего внимания. Как уже говорилось ранее, хеш-функция отвечает за свертку исходных данных в хеш-значение заведомо меньшей размерности. Для того чтобы это происходило с наилучшим качеством, функция хеширования должна обладать следующими чертами:

- Высокая скорость вычисления хеш-значения.
- Умение обрабатывать входные данные любого типа.
- Низкая вероятность совпадения хеш-значений для разных входных данных. Говоря точнее, для хеш-таблицы размером в M строк эта вероятность должна быть равной $1/M$.
- Функция хеширования должна быть детерминистической, т. е. для одного и того же значения всегда давать одно и то же хеш-значение.

К настоящему времени разработано достаточно много алгоритмов генерации хеш-значения, мы познакомимся лишь с двумя простейшими:

1. Метод деления (модульная хеш-функция).
2. Метод умножения (мультипликативная хеш-функция).

Метод деления наиболее простой и потому весьма быстрый, его идея сводится к получению остатка от деления входного значения W на модуль M. Модульная хеш-функция будет вырабатывать значения в диапазоне [0, M–1]:

$$f(W) = \bmod M. \tag{7.1}$$

Обычно в качестве делителя M выбирают простое число, достаточно далекое от степени двойки, это число и определяет число строк хеш-таблицы. Метод умножения развивает предыдущий метод. Здесь построение хеш-функции выполняется в два приема. На первом этапе входное значение W умножается на константу a, значение которой должно находиться в интервале от 0 до 1. На втором этапе мы получаем остаток от деления на M:

$$f(W) = [aW \bmod M]. \tag{7.2}$$

В качестве множителя a часто выбирают значение золотого сечения 0,618033…

Некоторые СУБД позволяют программистам писать собственные хеш-функции. Например, такой сервис реализован в Oracle, начиная с седьмой версии, эта СУБД разрешает заменить свою собственную функцию пользовательской.

ЗАМЕЧАНИЕ

Делитель функции хеширования определяет размер хеш-таблицы. Поэтому при расширении (уменьшении) индексной таблицы СУБД может вносить дополнительные коррективы в формулу хеширования. В свою очередь, любое изменение хеш-функции приводит к необходимости перестройки всего индекса.

Хеширование текстовых данных

Как получить входное значение для хеш-функции, обрабатывающей текстовые данные больших размерностей? Допустим, что у нас есть исходный текст: "Толстой Л.Н.". Представим текст в виде 8-разрядной ANSI-кодировки:

$$210\times256^{11}+238\times256^{10}+235\times256^{9}+241\times256^{8}+210\times256^{7}+238\times256^{6}+233\times256^{5}+32\times$$
$$\times256^{4}+203\times256^{3}+46\times256^{2}+205\times256^{1}+46\times256^{0}.$$

Во входной строке 12 символов. Только первый множитель 210×256^{11} возвращает нам 29-разрядное число, а итоговый результат составляет примерно $6{,}528\times10^{28}$. Для обработки таких данных нужны 64-разрядные типы данных. А теперь представьте себе, какое значение вы получите, если попробуете обрабатывать строки еще большей размерности! Текстовая строка длиннее 24 символов может оказаться неподъемной даже для переменной класса INT64!

Поэтому для обработки строковых значений часто применяют алгоритм Горнера (Horner) [19]. Согласно этому алгоритму можно вычислить десятичное число, соответствующее коду символов строки, при просмотре ее слева направо, умножая полученное значение на 256, а затем добавляя код следующего символа:

$$((((((((((210\times256+238)\times256+235)\times256+241)\times256+210)\times256+238)\times256+233)\times$$
$$\times256+32)\times256+203)\times256+46)\times256+205)\times256+46)\times256.$$

Со временем даже алгоритм Горнера создает число, которое вообще будет нельзя представить на компьютере. Однако полученное значение нас мало интересует, поскольку при хешировании нам требуется остаток от деления исходного числа на M. Результат можно получить, даже не сохраняя большое накопленное значение, т. к. в любое время можно отбросить число, кратное M — при каждом выполнении сложения и умножения нужно хранить только остаток деления по модулю M.

Борьба с коллизиями

В природе идеальных функций хеширования не бывает, кроме того, размер хеш-таблицы существенно меньше исходной таблицы, поэтому всегда имеется вероятность, что в результате хеширования два (или более) различных входных значения свернутся в одно и то же хеш-значение и, как следствие, будут направлены в

одну и ту же строку хеш-таблицы. Но строка хеш-таблицы обладает одним-единственным полем указателя, поэтому одновременно не способна ссылаться на несколько записей исходной таблицы.

На языке разработчиков ПО свертка различных исходных данных в одно и то же хеш-значение называется коллизией. Появление коллизий неприятно, но не фатально. На сегодня программисты разработали большое количество противоядий. Одно из наиболее простейших — *разрешение коллизий с помощью цепочек*. Это тот случай, когда предусматривается возможность прикрепления к каждой из ячеек хеш-таблицы дополнительной структуры, например связного списка. Как только в результате коллизии в уже занятую ячейку "неожиданно" попадает ссылка на еще одну запись, мы не впадаем в панику, а создаем дополнительный элемент связного списка и заносим указатель на строку с исходными данными уже не в ячейку таблицы, а во вновь созданный элемент списка (рис. 7.3). В результате выстраивается цепочка из элементов списка, а в ячейке хеш-таблицы теперь хранится указатель не на строку исходной реляционной таблицы, а на первый элемент списка.

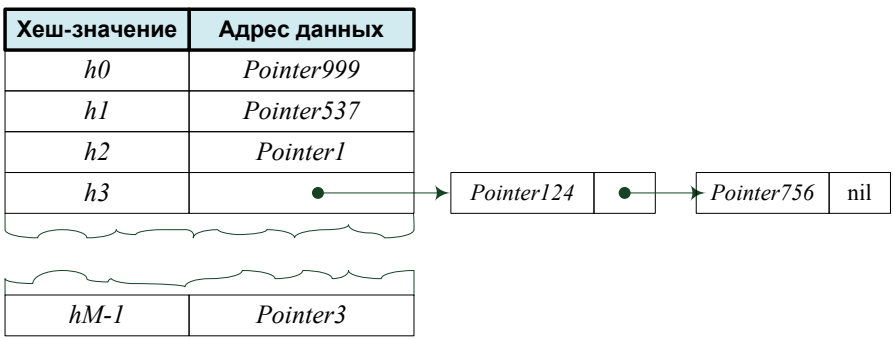


Рис. 7.3. Устранение коллизий с помощью цепочек

Наличие цепочки несколько замедляет процесс поиска данных, ведь теперь нам приходится просматривать все элементы связного списка, но это все равно лучше, чем перебирать все записи неиндексированной таблицы.

Другой метод противодействия коллизиям называется *линейным зондированием*. Алгоритм зондирования также не отличается особой изысканностью — столкнувшись с тем фактом, что требуемая ячейка уже задействована, мы переходим в режим поиска свободной ячейки. Для этого последовательно просматриваем следующие (за уже занятой) ячейки хеш-таблицы. Если, достигнув конца таблицы, мы не найдем свободного места, продолжаем просмотр с первой ячейки. Наконец, если в хеш-таблице просто не осталось свободного места, увеличиваем размер этой таблицы.

Замечание

Нужно отметить, что любое расширение индексной таблицы — это "микростресс" для СУБД, ведь увеличение размера хеш-таблицы заканчивается перерасчетом всех хеш-значений и перестановкой ее элементов.

Существуют и более сложные методы борьбы с коллизиями, например метод двойного хеширования, но все они в той или иной степени опираются на рассмотренные ранее решения.

Индексы на основе B-деревьев

Далеко не всегда когда программисты говорят о "деревьях", они подразумевают представителей флоры, которые мы с вами можем увидеть в своем саду. Что касается садов программистов, то в них произрастают деревья удивительных пород. Здесь легко найти бинарное дерево, красно-черное дерево, дерево поиска, героя этих страниц книги — B-дерево и ряд других не менее необычных растений. Деревья программистов отличаются упорядоченностью, ветвистостью, сбалансированностью, бывают вырожденными, и … все они растут корнем вверх!

В памяти компьютера деревья проще всего хранить в виде связных списков. Каждый элемент списка содержит область для данных и какое-то количество указателей, хранящих ссылки на дочерние элементы списка (рис. 7.4).

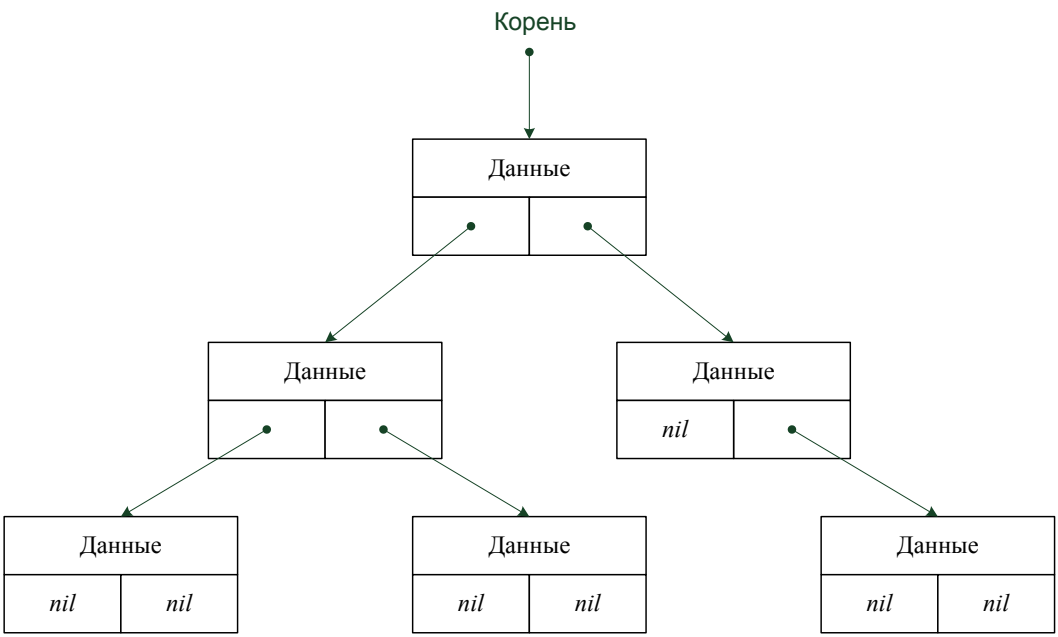


Рис. 7.4. Представление дерева в памяти

Для чего нам нужны деревья? Они очень удобны для представления различного рода иерархических структур, подобных структуре каталогов на диске вашего компьютера, организационной структуре предприятия, генеалогического древа, и, как вы уже догадались, могут применяться при индексировании таблиц.

Не вдаваясь в тонкости всех остальных выдуманными программистами древесных пород, сосредоточим наше внимание на B-деревьях. В первую очередь отметим, что своему появлению на свет они обязаны "селекционерам" Байеру (Bayer) и МакКрейту (McCreight), которые в 1972 году предложили применять B-деревья для решения задач поиска данных[1]. Выведенное ими B-дерево содержит узлы двух типов: листья (не имеющие потомков узлы самого нижнего уровня) и страницы (имеющие потомков узлы верхних уровней). Согласно Байеру—МакКрейту B-дерево должно обладать следующими качествами:

- сбалансированностью — расстояние от корня дерева до любого его листа (узла, не имеющего ссылок на другие элементы) должно быть одинаковым;
- ветвистостью — любой узел дерева должен обладать возможностью ссылаться на большое число узлов-потомков.

После краткого экскурса по азам "ботаники" сместим акцент беседы ближе к области применения B-деревьев в СУБД. Для этого вновь воспользуемся подопытной таблицей `WRITERS` из БД книжного магазина и построим индекс по полю имен авторов `WRITER`. В момент появления на свет B-дерево представляет собой скромный росток с всего одной корневой страницей и одним-двумя листами (рис. 7.5). В нашем примере у всех узлов дерева имеется по пять пар полей "данные-указатель", но на практике их может быть значительно больше.

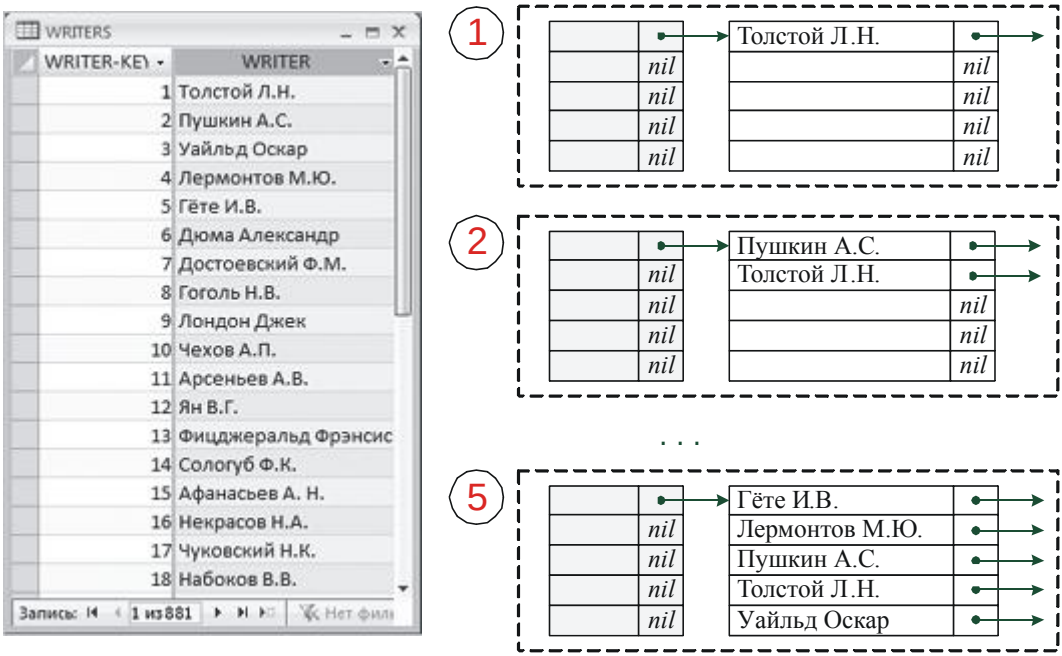


Рис. 7.5. Первые шаги заполнения B-дерева

[1] Иногда рассматриваемую структуру называют B* или B+-деревьями.

Замечание

Количество элементов в узле B-дерева зависит от двух факторов: физического размера страницы памяти в компьютере, на котором развернута СУБД, и характера данных, хранимых в полях узла. Допустим, что объем страницы памяти составляет 4096 байтов, а длина текстовой строки с именем автора равна 20 байтам. В таком случае СУБД создаст узлы, состоящие из `4096 DIV (20+4) = 176` элементов. Кстати, дополнительные 4 байта в делителе — это размер указателя в 32-разрядной ОС, в 64-разрядной ОС указателю потребуется 8 байтов.

По нашей команде система индексирования начинает последовательно считывать строки из таблицы `WRITERS`. На первом шаге в поле указателя первого элемента листа передается ссылка на первую строку таблицы (запись "Толстой Л.Н."). На рис. 7.5 показано, что, кроме поля указателя, мы заполняем и поле данных — помещаем в него строку "Толстой Л.Н.". На втором шаге в лист попадает ссылка на запись "Пушкин А.С.". Так как в алфавите буква "П" предшествует "Т", то указатель "Толстой Л.Н." смещается на вторую позицию, а первая позиция листа B-дерева передается указателю "Пушкин А.С.". На пятом шаге построения индекса лист заполняется полностью, поэтому после считывания из таблицы `WRITER` шестой по счету записи "Дюма Александр" происходит весьма неординарное событие — создается очередной лист (рис. 7.6).

После появления нового узла данные перераспределяются между листьями поровну. В первый лист попадут ссылки на три первые записи "Гете – Дюма – Лермонтов", во второй — на три заключительные "Пушкин – Толстой – Уайльд". Корневая страница также подвергается глубокой модернизации: теперь в ней будут храниться указатели на вновь созданные дочерние листы и индекс этих листов. Суть индекса точно такая же, как и в любом многотомном словаре или энциклопедии — индекс разъясняет нам, с какой буквы алфавита начинаются слова в конкретном томе энциклопедии. В примере в первом дочернем листе содержатся ссылки на авторов с именами от "А" до "О" включительно, а во второй от "П" и далее. После расщепления процесс продолжается до тех пор, пока не заполнится любой из листов, в нашем примере это произошло уже на 8-м шаге индексирования. Девятый по счету автор "Лондон Джек" должен попасть в первый лист, но в нем уже нет места. Что делать? Добавляем еще один лист и перераспределяем в него часть данных из переполненного собрата. Появление нового листа вынуждает СУБД перестроить корневую страницу: теперь в ней хранится три указателя на листы и три индекса: "А – К", "Л – П" и "П – Я".

Внимание!

Если индекс кластерный, т. е. он определяет физический порядок хранения строк в БД, то строки реляционной таблицы могут реально храниться в листьях B-дерева. В таком случае B-дерево станет выступать физическим хранилищем для таблицы. Если же речь идет об обычном вторичном индексе, то в поле данных листа действующие данные (например, текст "Толстой Л.Н.") не передаются. Вместо этого в поле данных может помещаться любая свертка данных (хеш-значение) или значение первичного ключа (в нашем примере `WRITER-KEY`) или даже ничего, ведь указатель и так ссылается на эту запись в физической таблице.

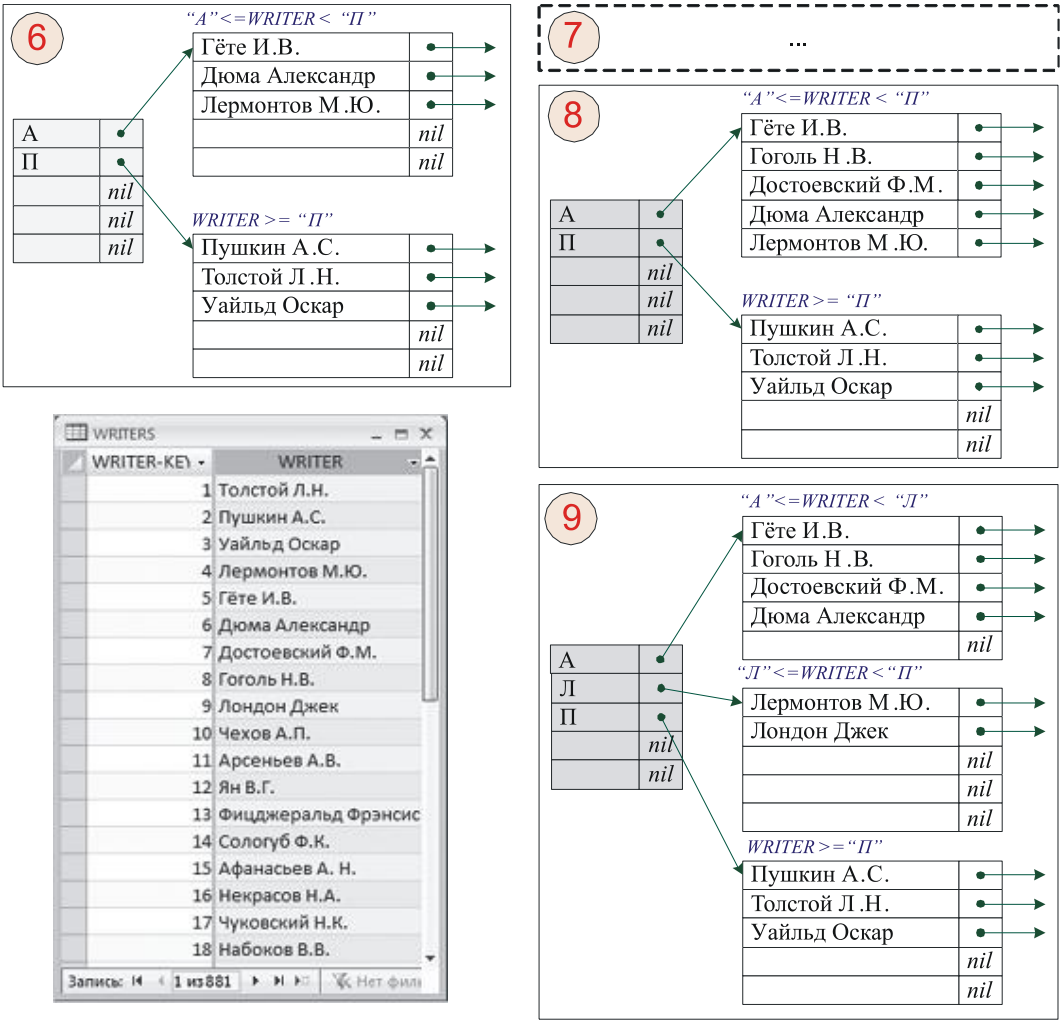


Рис. 7.6. Расщепление узлов B-дерева

Вскоре в дереве появится четвертый, а затем пятый лист. Но после того как в индекс попадут первые 25 записей из таблицы авторов, все листья дерева окажутся полностью заполненными. Теперь, прежде чем создать шестой лист, системе придется решить более сложную задачу. Корневая страница способна управлять только пятью дочерними узлами и не в состоянии удержать шестой, поэтому СУБД между корнем и листьями создает новый уровень иерархии. На этом уровне размещаются две новых страницы и указатели на них передаются на корневой лист. В свою очередь под управление вновь созданным страницам передаются все листья дерева.

Процесс расщепления и создания новых листов продолжится до тех пор, пока не проиндексируются все данные. Во время индексирования будут появляться все

новые и новые страницы и листы, а дерево станет представлять собой многоуровневую иерархическую структуру. Страницы верхних уровней B-дерева будут выступать в роли индекса, а листья самого нижнего уровня — хранить данные и указатели на записи в таблице (рис. 7.7).

Как только возникает задача поиска какой-нибудь записи, мы считываем индекс корневой страницы и по указателю переходим к нужной дочерней странице, затем читаем индекс у дочерней страницы и переходим к следующему узлу. Процесс продолжается до тех пор, пока мы не доберемся до требуемого листа с данными. Благодаря тому, что B-дерево сбалансировано, поиск в индексе осуществляется за одинаковое количество переходов по индексным страницам, независимо от того, какие данные мы ищем. Такая особенность B-деревьев весьма благоприятно сказывается на производительности СУБД.

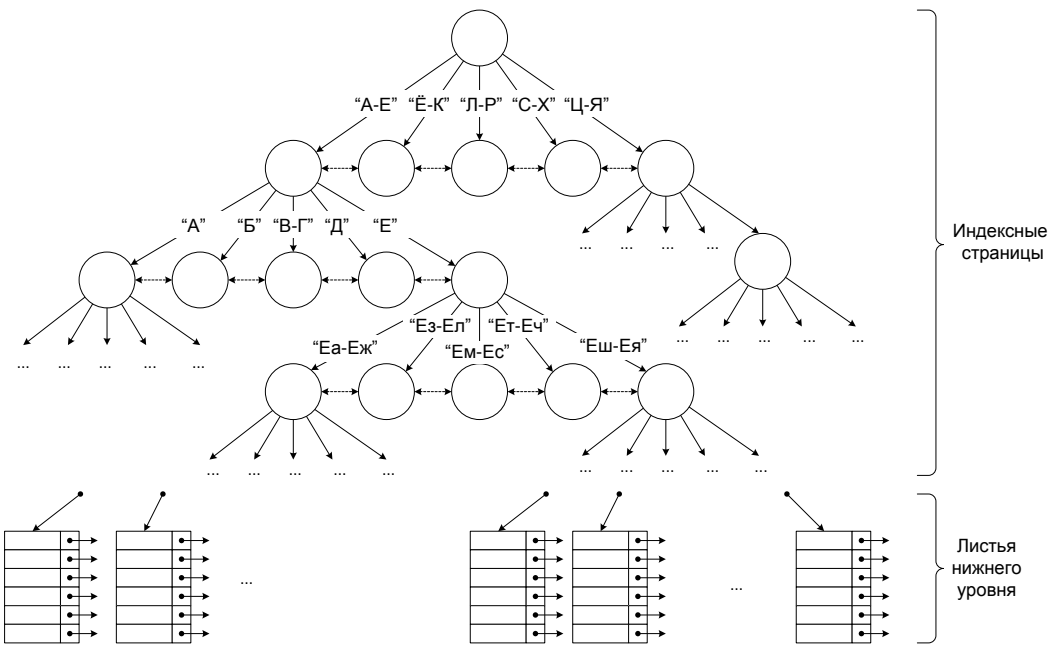


Рис. 7.7. Фрагмент индекса на основе B-дерева

Зачастую для ускорения переходов между узлами B-дерева, лежащими на одном уровне, дочерние страницы индекса связывают друг с другом указателями. То же самое может происходить и с листьями дерева.

Индексы на основе B-деревьев не лишены недостатков. Один из основных — существенные затраты на перестройку индекса, в особенности, если было изменено, добавлено или удалено много записей. Это приводит к расщеплению большого количества переполненных страниц или наоборот — сложению пустых или полупустых страниц и листьев.

Замечание

В-деревья — основное средство построения индексов в больших многопользовательских СУБД, например таких как MS SQL SERVER, InterBase и ORACLE. Если индекс кластерный, то даже физическое хранение реляционной таблицы осуществляется в виде В-дерева, в этом случае строки данных реально хранятся в листьях нижнего уровня дерева.

Битовые индексы

Основное назначение *битовых индексов* (bitmap indexes) — индексирование столбцов с небольшим количеством различных значений. Классический пример — пол человека, с точки зрения БД его можно задать тремя значениями: "М", "Ж" и "NULL". В подобных индексах для представления существования определенного значения столбца целесообразно использовать битовую последовательность (строку из нулей и единиц).

Если индекс на основе В-деревьев хранит однозначное соответствие между строкой таблицы и записью в индексе, то битовый индекс обеспечивает ссылку на большое количество строк одновременно.

Индексы на основе битовых карт занимают малый размер (он прямо пропорционален объему индексируемых данных) и поэтому обеспечивают гораздо более высокую скорость выборки.

Правила назначения вторичных индексов

После краткого знакомства с физическими основами построения индексов рассмотрим правила индексирования таблиц. Речь пойдет исключительно о порядке назначения вторичных индексов. Поля первичного и вторичного ключей нас не интересуют, т. к. автоматически индексируются всеми СУБД. Принудительная индексация ключевых полей может понадобиться только в том случае, когда они входят в состав составного индекса.

В первую очередь следует индексировать столбцы отношений, которые наиболее часто используются в запросах на выборку данных. Последовательность полей в составном индексе должна соответствовать последовательности полей, следующих после директивы `ORDER BY` или `GROUP BY`. При этом далеко не обязательно плодить отдельные индексы для каждого варианта запроса, т. к. оптимизаторы запросов большинства современных СУБД перед выполнением инструкции SQL сумеют подобрать наиболее подходящий индекс.

Наиболее эффективны индексы, накладываемые на столбцы, в которых не хранятся повторяющиеся значения.

Замечание

Основным средством управления индексами являются инструкции структурированного языка запросов: `CREATE INDEX`, `ALTER INDEX` и `DROP INDEX`. Эти и другие команды мы рассмотрим в главах, посвященных изучению SQL.

Можно смело индексировать уже полностью заполненные справочные таблицы, этот же совет в равной степени подходит ко всем таблицам, содержимое которых или не изменяется вовсе, или слабо подвержено изменениям. В качестве примера приведем справочник авторов в базе книжного магазина или таблицы базы телефонных номеров города. Впрочем, необходима одна оговорка — чем больше индексов, тем больше заполняется дисковое пространство. Если для вас это критично, то следует вовремя остановиться.

Никогда не создавайте индекс только на основе предположения, что он когда-нибудь пригодится. Более того, не индексируйте все столбцы подряд. Будет ли использоваться такой индекс, неизвестно, но хранить еще одну копию таблицы вам все равно придется.

Кроме того, не стоит индексировать столбцы большого размера (например, большие текстовые поля и поля, предназначенные для хранения больших бинарных объектов, BLOB), а также поля, допускающие неопределенные значения `NULL`.

Логические столбцы и столбцы с небольшим количеством многократно повторяющиеся значений следует индексировать битовым индексом.

Внимание!

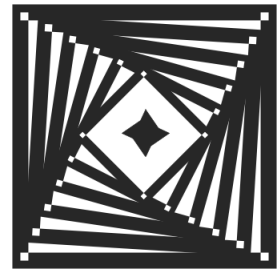
Если вы администрируете большую, активно эксплуатируемую многопользовательскую базу данных, то периодически необходимо заставлять СУБД перестроить все индексы. Подобные операции следует проводить в период наименьшей загрузки БД (в выходные и праздничные дни или в ночное время), предварительно не забыв создать полную копию БД и журналов транзакций.

Резюме

Современные СУБД позволяют создавать индексы трех типов: первичный, кластерный и вторичный. Основной структурой, используемой программистами для построения индексов, выступает сбалансированное B-дерево, но это не единственное решение. Существуют индексы на основе хеширования, битовых карт и R-деревьев.

При работе с индексами разработчику БД всегда приходится балансировать между двумя противоречиями: если индексов слишком много, снизится производительность СУБД, а если индексов недостаточно, упадет скорость выполнения запросов.

К индексам мы еще вернемся в главах, посвященных структурированному языку запросов.



Глава 8

Транзакции и параллельная обработка данных

Практически ни одна сторона современной деятельности человека не обходится без информационных систем. Базы данных встречают нас в магазинах и банках, в туристических фирмах и библиотеках, на производстве и в быту. Содержащиеся в БД данные активно читаются, обновляются, удаляются. В ходе осуществления операций чтения/записи не исключено возникновение различных исключительных ситуаций, их могут порождать ошибки в программном обеспечении, сбои в работе аппаратной части системы, защита от нарушений целостности данных, ограничения бизнес-логики приложений, отсутствие прав доступа и просто некорректные действия пользователей. Последствия от сбоев могут быть весьма печальными — уничтожение или искажение обрабатываемых в БД данных, поэтому нам необходимо добиться того, чтобы ни один из сбоев в работе БД не привел к фатальным результатам.

Борьба с исключительными ситуациями — далеко не единственная задача, над которой ломают копья разработчики БД. Масла в огонь подливает еще одна проблема. К содержащейся в БД информации в одно и то же время получают доступ сотни пользователей. Одним хочется ознакомиться с прайс-листом доступных товаров и услуг, другим необходимо занести эти данные в таблицы, третьи проводят анализ продаж, четвертые готовят квартальный отчет… Список можно продолжить, но уже сейчас ясно, что вероятность одновременного обращения пользователей к одним и тем же записям таблиц БД весьма высока, а это чревато не меньшими неприятностями, чем при возникновении исключительной ситуации, скажем, от сбоя в работе оборудования ЛВС. Вспоминайте свое золотое детство, в те времена вы наверняка сталкивались с проблемой совместного доступа с другими детьми к разделяемому ресурсу в виде машинки, куклы или песочницы. И если все стороны сразу не находили компромиссного решения, то попытка одновременной игры с машинкой заканчивалась либо уничтожением игрушки, либо слезами какой-то из конфликтующих сторон.

Понятие транзакции

Наиболее эффективный способ обработки исключительных ситуаций в БД и решение проблемы параллельного доступа к строкам таблиц — организация обращений пользователей к данным только в рамках особой логической единицы работы СУБД, именуемой *транзакцией* (transaction).

Замечание

Транзакцией называется действие или набор действий, направленные на получение или обновление данных и выполняемые как единая процедура. Транзакция не может быть выполнена частично, она либо выполняется полностью, либо полностью отменяется.

Различают три основных типа транзакций:

- *Транзакции извлечения* — применяются для выборки данных для их последующего отображения или печати.
- *Транзакции обновления* — работают во время вставки, обновления или удаления записей из БД.
- *Смешанные транзакции* — допускают проведение операций чтения и редактирования записей.

После старта транзакции промежуточные результаты всех ее действий записываются не непосредственно в таблицы БД, а в *журнал транзакций*. Такой подход позволяет в любой момент времени вернуть БД в исходное состояние (состояние до старта транзакции). *Откат транзакции* осуществляется в том случае, если хотя бы один из операторов, входящих в состав транзакции, выполняется с ошибкой. Во время отката в обратном порядке отменяются все операции транзакции. Если же все операторы выполняются верно, то результаты транзакции *фиксируются* — данные из журнала переносятся в БД.

Для того чтобы СУБД смогла гарантировать корректное завершение или успешный откат транзакции, в журнале транзакций приходится хранить существенный объем данных:

- сведения о старте транзакции и ее идентификатор;
- перечень объектов БД, на которые повлияла транзакция;
- описание каждой из входящей в состав транзакции операций обновления данных (вставка, обновление, удаление);
- значения всех полей, изменившихся в результате выполнения транзакции, причем хранению подлежат как исходное, так и новое значение;
- сведения о завершении транзакции.

Как видите, по своей сути журнал транзакций представляет собой самостоятельную базу данных, автоматически управляемую СУБД. На ведение журнала затрачивается достаточно большой вычислительный и дисковый ресурс сервера СУБД, но с такими издержками приходится мириться, ведь на другой чаше весов лежат требования по обеспечению надежного хранения основных данных.

Замечание

Для управления транзакциями в язык SQL введены инструкции `COMMIT` и `ROLLBACK`. Первая инструкция вызывается при успешном завершении транзакции, она дает указания СУБД зафиксировать все изменения. Вторая команда, напротив, требует от СУБД отменить все изменения и вернуть БД к состоянию на момент вызова транзакции.

Любая транзакция обязана соответствовать ряду требований, большинство специалистов по базам данных рекомендуют список из четырех взаимозависимых элементов:

- *Атомарность* (atomicity) — транзакция должна представлять собой единую и неделимую логическую единицу работы. Транзакция может быть выполнена полностью или не выполнена совсем. Представьте себе, что транзакция снимает некую сумму денег со счета и переводит ее в другой банк. Было бы обидно, если бы транзакция выполнилась частично: только сняла деньги со счета, а потом, неважно по какой причине, на другой счет их не положила.
- *Согласованность (правильность)* (consistency) — транзакция должна переводить базу данных из одного согласованного состояния в другое согласованное состояние. Однако в процессе выполнения транзакции допускается появление несогласованных промежуточных результатов. Говоря о согласованности результатов, мы имеем в виду, что после выполнения транзакции данные будут соответствовать заложенной в базе данных логике и правилам поддержания целостности данных.
- *Изолированность* (isolation) — транзакции должны выполняться независимо одна от другой и ни одна из транзакций не должна быть допущена к промежуточным результатам другой транзакции.
- *Устойчивость (долговечность) (durability*) — результаты выполненной транзакции должны немедленно сохраняться в БД, чтобы не быть утерянными в результате вероятных последующих сбоев.

Читателю, хорошо владеющему английским языком, перечисленные требования легко запомнить, для этого достаточно сохранить в своей памяти аббревиатуру из первых букв английских слов: atomicity, consistency, isolation, durability — ACID.

В некоторых работах, посвященных базам данных [17], согласованность замещают требованием *сериализуемости* (serializability). Под сериализуемостью понимают возможность одновременного выполнения нескольких транзакций без их взаимного влияния, что очень важно для многопользовательских систем. Но если еще раз внимательно прочитать требования ACID, вы заметите, что они поглощают в себя и понятие сериализуемости. Справедливости ради отметим, что общепризнанный "гуру" в области БД Крис Дейт также критикует подход, основанный на принципах ACID [7]. Но это только подчеркивает тот факт, что современные информационные технологии находятся в состоянии постоянного совершенствования.

Внимание!

В составе транзакции **категорически запрещается** применять операторы, которые прямо или косвенно могут изменить структуру БД (таблиц, индексов, хранимых процедур, просмотров, триггеров, и т. п.) или повлиять на права пользователей. Кроме того, внутри транзакции не следует выполнять другие транзакции.

Проблемы параллельного доступа к данным

При проектировании многопользовательской БД настоящей головной болью разработчика может стать организация совместного доступа двух и более пользователей к одним и тем же данным. В этом случае СУБД должна не только уметь корректно восстанавливать состояние БД при сбоях в транзакциях, но и обеспечивать бесконфликтную работу пользователей с совместно используемым ресурсом.

Для иллюстрации сложности организации многопользовательского доступа рассмотрим ключевые проблемы, с которыми могут столкнуться пользователи при совместном обращении к БД:

- Потерянные обновления.
- Неактуальные чтения (зависимость от незафиксированных результатов).
- Несогласованная обработка (чтение мусора).
- Чтение строк-фантомов.

Потерянные обновления

Разговор о проблемах параллельного доступа к данным начнем с наиболее значимой из них — проблемы потерянных обновлений. Потеря обновлений происходит в случаях, когда одна, вполне успешная транзакция переписывает изменения, осуществленные другой не менее корректной транзакцией.

Рассмотрим элементарный пример. Допустим, что в некоторой таблице в поле `Field1` хранятся данные о количестве товара на складе. Пусть изначально это значение равно 10. Рассмотрим ситуацию, когда к этому полю обращается две транзакции. Первая изымает 5 единиц товара с целью перевести их в торговый зал. Одновременно с первой вторая транзакция приходует на складе только что полученные 10 единиц этого же товара. Простейший расчет показывает, что в результате последовательного выполнения указанных транзакций в поле `Field1` должно оказаться значение 10 – 5 + 10 = 15. А теперь рассмотрим ситуацию, в которой наши транзакции начинают выполняться совместно (табл. 8.1). В некоторый момент времени t_2 транзакция 1 читает содержимое поля `Field1` и помещает его в локальную переменную `X`. То же самое делает и транзакция 2, но в момент времени t_3. В момент времени t_4 первая транзакция возвращает в поле результаты своих расчетов 10 – 5 = 5. Несколько позже и вторая транзакция записывает в поле результат своих расчетов 10 + 10 = 20. Так как транзакция 2 не знает, что во время ее выполнения значение поля `Field1` было обновлено другой транзакцией, она не учитывает эти изменения. В итоге мы получили ошибочный результат, а правильный потеряли, отсюда и название проблемы — потерянные обновления.

Таблица 8.1. *Пример потерянных обновлений*

Время	Транзакция 1	Транзакция 2	Field1
t_1	Старт		10
t_2	Чтение из поля `X=10`	Старт	10
t_3	Расчет `X=X-5=5`	Чтение из поля `X=10`	10
t_4	Запись в поле `Field1=X`	Расчет `X=X+10=20`	5
t_5	Фиксация транзакции	Запись в поле `Field1=X`	20
t_6		Фиксация транзакции	20

Неактуальные чтения ("грязное чтение")

Проблема неактуальных чтений имеет место, когда незафиксированные изменения, осуществленные одной транзакцией, читаются (или обновляются) другой. В случае перезаписи этих промежуточных значений или откате первой транзакции незафиксированные изменения могут быть отменены, а прочитавшая их транзакция с этого момента будет работать с неверными данными. Поэтому проблему неактуальных чтений метко назвали "грязным чтением" (*dirty read*) или "чтением мусора".

Рассмотрим уже знакомый нам склад. Теперь первая транзакция приходует 10 единиц товара, но потом, по какой-то причине отказывается от своей работы и осуществляет откат. Напротив, вторая транзакция завершается корректно и передает в торговый зал 5 единиц этого же товара. В случае корректного совместного выполнения этих транзакций в поле `Field1` должно оказаться значение 5. А теперь посмотрим, что может случиться в аналогичной ситуации, если СУБД не сумеет обеспечить нормальный многопользовательский доступ (табл. 8.2). Допустим, транзакция 1 уже отправила в поле значение 20. Стартующая к этому моменту времени вторая транзакция читает из поля `Field1` незафиксированные изменения первой транзакции. И хотя в результате отката первой транзакции в момент времени t_6 в поле было возвращено изначально правильное значение 10, вторая транзакция уже начала работу с "мусором" (неверным промежуточным значением работы своей визави) и, в конце концов, вернет в базу ошибочное число 15.

Таблица 8.2. *Пример неактуальных чтений*

Время	Транзакция 1	Транзакция 2	Field1
t_1	Старт		10
t_2	Чтение из поля `X=10`		10
t_3	Расчет `X=X+10=20`		10

Таблица 8.2 *(окончание)*

Время	Транзакция 1	Транзакция 2	Field1
t_4	Запись в поле `Field1=X`	Старт	20
t_5	…	Чтение из поля `X=20`	20
t_6	Откат транзакции	Расчет `X=X–5=15`	10
t_7		Запись в поле `Field1=X`	15
t_8		Фиксация	15

ЗАМЕЧАНИЕ

На сленге программистов проблема неактуальных чтений называется "грязным чтением". Этот термин фактически узаконен, т. к. широко используется в таком серьезном документе, как стандарт SQL:2003.

Несогласованная обработка

В двух предыдущих примерах в результате неграмотно организованного совместного доступа транзакций к разделяемому ресурсу в БД появлялись ошибочные данные. Теперь рассмотрим ситуацию, когда все транзакции завершаются корректно и данные в БД находятся в непротиворечивом состоянии, но мы все равно получаем ложные результаты.

Проблема несогласованной обработки возникает тогда, когда транзакция считывает из базы значение, после чего вторая транзакция обновляет это значение[1]. Если в этот момент времени первая транзакция продолжает выполняться, то имеющиеся в ее распоряжении данные становятся неактуальными.

Допустим, что два оператора из разных магазинов практически одновременно обратились к серверу нашего склада с запросом о количестве находящихся на складе холодильников определенной марки. Обе транзакции получили ответ — 10 шт. Затем оператор, инициализировавший транзакцию 1, заказывает с нашего склада все 10 холодильников. Холодильники благополучно списываются (теперь на складе их 0) и первая транзакция на вполне законных основаниях фиксируется. В это время, второй, менее расторопный оператор (который может быть потратил пару секунд на подтверждение заказа у своего руководства или на подсчет ворон, сейчас это уже не важно) также затребовал с нашего склада несколько холодильников, допустим 5. Но на складе уже пусто! Если система поддержки целостности данных спроектирована грамотно, то она не допустит занесения в поле `Field1` отрицательного значения и известит об этом транзакцию 2. Но для оператора 2 от

[1] Несогласованную обработку в англоязычной литературе часто называют *неповторяемым чтением* (non-repeatable read).

этого это не легче, он полагает, что в БД произошел сбой, хотя на самом деле БД находится в согласованном состоянии (табл. 8.3).

Таблица 8.3. *Пример несогласованной обработки*

Время	Транзакция 1	Транзакция 2	Field1
t_1	Старт	Старт	10
t_2	Чтение из поля `X=10`	Чтение из поля `X=10`	10
t_3	Заказ 10	Ожидание	10
t_4	Расчет `X=X–10`	Ожидание	10
t_5	Запись в поле `Field1=X=0`	Ожидание	0
t_6	Фиксация	Заказ 5	0
t_7		…	

Чтение строк-фантомов

Четвертая проблема параллельной обработки данных получила название проблемы чтения строк-фантомов (row-phantom). Своим появлением на свет записи-призраки обязаны не потусторонним силам, а вполне объяснимому взаимному влиянию двух несогласованных транзакций.

Допустим, что в рамках первой транзакции мы создаем огромную инвентаризационную ведомость, в которой будет отражена информация о категориях товаров, хранящихся на складе, количестве единиц и суммарной стоимости каждой из категорий, а в конце отчета выводится общая сумма всех товаров на складе. Для удобства пользователя (получателя отчета) все остатки группируются по наименованию, в отчет входят данные о номерах приходных ордеров, датах поступления, названиях фирм-поставщиков и т. п. Одним словом, это не просто отчет, а целый многостраничный фолиант, на построение которого уходит достаточно много времени. Теперь, допустим, что где-то в середине работы первой транзакции на сцену выходит транзакция 2. Своим появлением на свет она обязана тому факту, что на склад был отгружен мешок сахара (или партия ноутбуков, если вы не сторонник сладкого). После поступления товара на склад транзакция прекращает свою работу и на вполне законных основаниях фиксируется. В это время транзакция 1 подходит к своему счастливому завершению и отправляет в адрес БД последний запрос: "На какую сумму хранится товаров на складе?" На языке SQL это будет всего одна строка с примерно таким текстом:

```
SELECT SUM(WARE_SUMPRICE) FROM WAREHOUSE WHERE WARE_COUNT>0
```

Приводя эту последнюю строку транзакции, я не преследую цель начать изложение основ языка SQL (этим мы займемся немного позднее), а просто хочу про-

демонстрировать тот факт, что для получения денежной суммы остатка товаров на складе нет необходимости перебирать все строки таблицы SCLAD, а достаточно лишь воспользоваться агрегирующей инструкцией SUM. По этой команде СУБД самостоятельно просуммирует остатки, лишь бы выполнялось условие WARE_COUNT>0. В результате СУБД возвратит какое-то значение X. Кажется все верно, но в значение X уже включена денежная сумма с учетом стоимости вновь появившегося мешка сахара. Но этот мешок не попал на первые листы инвентаризационной ведомости. В итоге в ведомости не сходится баланс, бухгалтерия в шоке, начальник склада лишен премии, а к нашей базе данных не придраться — она находится в идеальном согласованном состоянии…

Управление параллельными транзакциями

Появление перечисленных проблем многопользовательского доступа к данным объясняется одной-единственной причиной — неверным управлением параллельным доступом. Последствия некорректного доступа к данным могут быть самыми негативными. Так, потерянные обновления и неактуальные чтения переводят БД в несогласованное состояние, а неповторяемые чтения и чтения фантомов грозят выдачей пользователю некорректных результатов.

Как найти выход из сложившейся ситуации? На первый взгляд может показаться, что решение очевидно. Достаточно сделать так, чтобы в каждый момент времени выполнялась только одна транзакция, а команда на запуск очередной транзакции выдавалась только после фиксации изменений, сделанных предыдущей транзакцией. Но переход к строго последовательному выполнению транзакций практически исключает реальный многопользовательский доступ к БД, что нас не устраивает.

На сегодняшний день наиболее рациональным решением, позволяющим минимизировать последствия от конфликтов многопользовательского доступа, считается разработка специального графика выполнения транзакций. Этот процесс называется планированием транзакций. Цель планирования — отыскание такой упорядоченной последовательности выполнения операторов транзакций, чтобы транзакции могли выполняться совместно, не влияя друг на друга.

ВНИМАНИЕ!

Основное правило совместного выполнения транзакций заключается в том, что в ходе обращения к БД транзакция должна видеть только согласованные данные, доступ к промежуточным или используемым другими данным должен быть заблокирован!

Особенности планирования параллельных транзакций во многом зависят от степени оптимизма разработчика БД. Пессимисты не исключают вероятности столкновения интересов транзакций и поэтому стремятся на 100% исключить кон-

фликты между ним. Оптимисты исходят из предположения, что БД не будет использоваться очень активно, поэтому можно не уделять особого внимания методам противодействия столкновениям.

Независимо от степени оптимизма нужно учитывать, что если транзакции обслуживают разные части базы данных или только читают данные из одного и того же ресурса, то в организации их параллельного выполнении нет ничего сложного. Но если две транзакции обращаются к одним и тем же данным, и хотя бы одна из них намерена их изменить, то это и есть потенциальный источник конфликта. На сегодня существует несколько подходов, позволяющих предотвратить споры транзакций за один и тот же ресурс, рассмотрим наиболее показательные из них.

Метод блокировок

Метод блокировок[1] (lock) основывается на "пессимистическом" подходе, который предполагает, что вероятность одновременного обращения нескольких транзакций к одному и тому же ресурсу весьма высока. Метод откладывает выполнение транзакций, потенциально готовых войти в конфликт с транзакциями, выполняющимися в текущий момент времени. Идея механизма блокировок прозрачна — после доступа к определенному ресурсу базы данных некой транзакции попытка обратиться к занятому ресурсу другими транзакциями отвергается, и опоздавшие транзакции переходят в режим ожидания освобождения ресурса.

Различают два вида блокировок:

- *Блокировка для записи.* Осуществившая такую блокировку транзакция может как читать, так и редактировать захваченный ею ресурс. Все остальные транзакции лишаются всех прав доступа к этим данным.
- *Блокировка для чтения*. Заблокировавшая ресурс транзакция намерена только прочитать данные, поэтому к ресурсу могут получить доступ и другие транзакции при условии, что они не станут его обновлять.

Замечание

Блокировка для чтения допускает совместную работу с заблокированным ресурсом всех других транзакций с одним условием — никто не имеет права обновлять заблокированные данные. Блокировка для записи забирает ресурс в эксклюзивное пользование этой транзакцией.

Метод блокировок обладает одним существенным недостатком — он может стать источником тупиковой ситуации, когда две (или более) транзакции попадают в состояние взаимного ожидания освобождения блокировок, удерживаемых каждой из них. Для иллюстрации такого события рассмотрим рис. 8.1. Допустим, что две транзакции работают с таблицей `WRITERS`. Транзакция 1 обрабатывает записи таблицы в порядке их физического хранения в таблице (от первой к последней), а

[1] В ряде книг такой способ сериализации называется "синхронизационным захватом".

транзакция 2 — в обратном. Каждая из транзакций последовательно блокирует обслуживаемые строки и снимает блокировку, когда необходимость в ней отпадает. Где-то в середине таблицы транзакции встречаются друг с другом. Транзакция 1 удерживает блокировку на строке "Лондон Джек" и ожидает освобождения строки "Чехов А.П.". У транзакции 2 дела обстоят ровным счетом наоборот — она не в состоянии продолжить работу, пока не получит доступ к строке удерживаемой первой транзакцией, но при этом не может снять блокировку со строки так нужной транзакции 1. Если бы транзакции были "настоящими джентльменами", то они моментально уступили друг другу дорогу. Но хорошим манерам их не учили, поэтому обе транзакции попадают в состояние *взаимной блокировки* (deadlock) и не могут из него выйти.

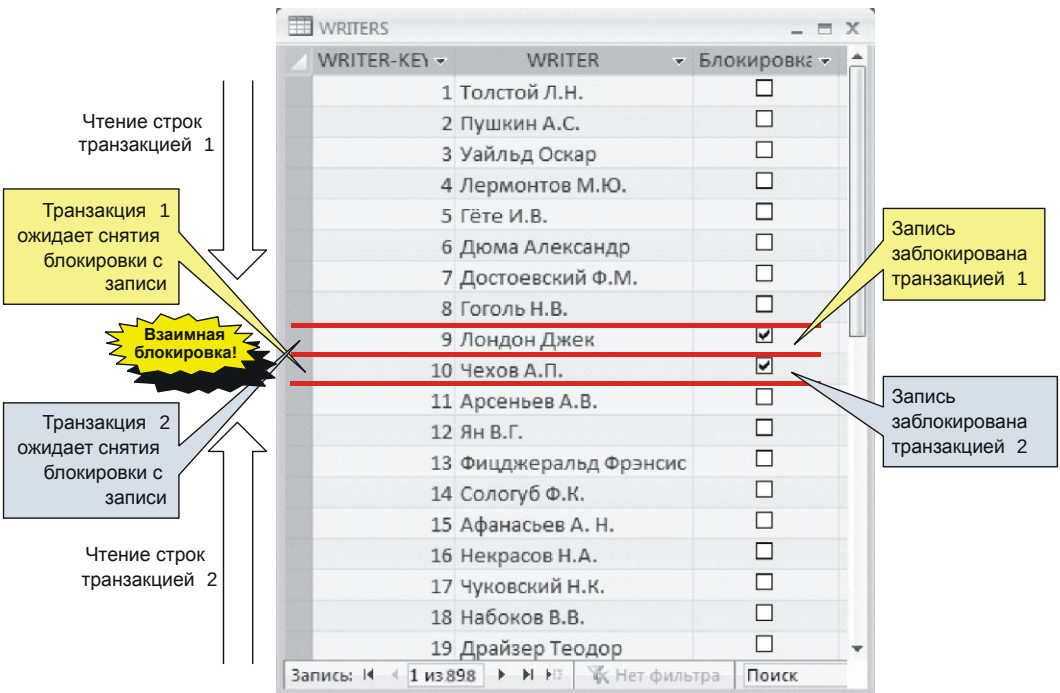


Рис. 8.1. Иллюстрация ситуации взаимной блокировки

Для того чтобы нарушить состояние взаимной блокировки СУБД, приходится идти на радикальную меру — отменять выполнение одной или нескольких транзакций, чтобы хотя бы одна из транзакций завершилась успешно.

Внимание!

Механизм блокировок может послужить источником взаимных блокировок. Это случай, когда две или более транзакций находятся в состоянии ожидания взаимного освобождения захваченного друг другом ресурса.

В нашем примере мы рассмотрели модель, когда система способна блокировать отдельную строку таблицы, однако таким потенциалом обладают далеко не все СУБД. У различных систем имеются разные возможности по уровню *детализации блокировок* (lock granularity). Блокировке могут быть подвергнуты:

- вся база данных;
- отдельный файл, в котором хранится таблица или даже несколько таблиц БД;
- отдельная страница данных с частью таблицы;
- отдельная запись в таблице;
- отдельное поле в записи.

Степень детализации блокировки оказывает существенное влияние на производительность СУБД и на возможности работы протокола управления параллельными транзакциями, причем производительность и параллельность состоят в явной конфронтации друг с другом. Судите сами, вряд ли можно назвать многопользовательской СУБД, у которой блокируется вся БД. С другой стороны, при блокировке на уровне отдельных полей системе приходится обрабатывать огромный пласт служебной информации, что замедляет ее работу, но действительно облегчает параллельную работу транзакций.

Обычно за наложение блокировок отвечает система, создаваемые ею блокировки называются *внутренними* (implicit locks). Вместе с тем СУБД позволяет вмешиваться в процесс и программисту, тогда ему разрешается вручную накладывать свои собственные блокировки на элементы данных, такие блокировки называются *явными* (explicit locks).

Из-за сложности разрешения взаимных блокировок рассмотренный метод простых блокировок в явном виде не реализован ни в одной из современных СУБД. Однако заложенные в методе идеи, послужили появлению ряда усовершенствованных методов. В частности метода временных меток и метода двухфазных блокировок.

Метод временных меток

Метод временных меток также относится к классу пессимистических блокировок. Он представляет собой усовершенствованный протокол управления параллельностью, позволяющий устанавливать очередность выполнения транзакций, при которой более старые транзакции (транзакции с более поздним значением временной отметки) имеют более высокий приоритет при разрешении возникающих конфликтов. Для этих целей каждой транзакции в момент старта присваивается уникальная *временная метка* (time stamp).

Метод временных меток не исключает взаимных блокировок, но значительно упрощает процесс принятия решения. Теперь при столкновении интересов транзакций преимущество получает наиболее "древняя", а "молодые" транзакции откатываются в исходное состояние и последовательно запускаются СУБД после того, как конфликтная ситуация будет исчерпана.

Метод двухфазной блокировки

Чтобы снизить вероятность взаимных блокировок и, соответственно исключить необходимость отката транзакций программистами разработан *метод двухфазной блокировки* (two-phase locking). Наш очередной знакомый также относится к разряду пессимистических методов сериализации.

Транзакция, работающая по алгоритму двухфазной блокировки, последовательно проходит два этапа: фазу роста и фазу сжатия. На этапе роста транзакция запрашивает у системы управления транзакциями СУБД все необходимые для ее выполнения блокировки, и до тех пор, пока эти блокировки не будут ей предоставлены, транзакция не осуществляет никаких операций с данными. Только получив все необходимое для работы, транзакция переходит к фазе сжатия — обрабатывает данные в таблицах и постепенно освобождает заблокированные элементы данных.

Благодаря тому, что транзакция не начинает обработку данных до тех пор, пока они не будут освобождены другими транзакциями, попадание в тупик практически исключено. Правда, нахождение транзакции в первой фазе может оказаться достаточно долгим, но эту проблему должны решать такие компоненты СУБД, как планировщик и система управления транзакциями.

Оптимистический подход

Все "оптимистические" стратегии сериализации транзакций исходят из предположения, что вероятность столкновения транзакций ничтожно мала. Поэтому в оптимистических методах обычно транзакции выполняются без особых ограничений. В самом общем виде управление транзакцией в оптимистичном духе сводится к следующим этапам:

- *Чтение* — транзакция получает из таблиц интересующие ее данные и сохраняет их в локальный кэш. Затем с локальной копией данных осуществляются все необходимые операции (вставка, редактирование и удаление).
- *Проверка* — система проводит проверку целостности и непротиворечивости модифицированных данных. При выявлении сбоев и ошибок транзакция отменяется, если же изменения осуществлены корректно, транзакция переходит к заключительному этапу.
- *Сохранение* — перенос данных из локальной копии в БД.

Оптимистические методы сериализации имеют одно весьма важное преимущество над коллегами-пессимистами: минимальное время удержания данных в заблокированном состоянии. Своими скоростными качествами оптимистическая блокировка обязана тому факту, что большую часть времени инструкции транзакции работают с данными, размещенными в локальном буфере. Как следствие, операции с копией данных никак не влияют на остальных пользователей БД. Необходимость в блокировке данных возникает только на заключительной стадии работы транзакции — во время переноса обработанных данных из кэша в БД.

Уровни изоляции SQL-транзакций

В заключение поговорим о требованиях современного стандарта структурированного языка запросов SQL:2003 к независимому выполнению параллельных транзакций. В стандарте используется термин *уровни изоляции SQL транзакций* (Isolation levels of SQL-transactions), поэтому мы далее и будем его придерживаться. Итак, стандарт рекомендует разработчикам СУБД обеспечивать четыре уровня изоляции (табл. 8.4), упорядочим их по степени надежности:

- READ UNCOMMITTED (незафиксированное чтение) — наименее защищенный уровень изоляции транзакций; старайтесь его использовать только в тех случаях, когда вы уверены, что все транзакции работают только в режиме чтения.
- READ COMMITTED (фиксированное чтение) — исключается "грязное чтение", но другим транзакциям позволено изменять заблокированные строки.
- REPEATABLE READ (повторяемое чтение) — накладывает блокировки на обрабатываемые транзакцией строки и не допускает их изменение другими транзакциями, но не запрещает добавление новых записей, что может привести к появлению строк-фантомов.
- SERIALIZABLE (сериализуемость) — самый надежный уровень изоляции, полностью исключающий взаимное влияние транзакций.

Таблица 8.4. *Уровни изоляции в стандартах SQL-92 и SQL:2003*

Уровень изоляции	Проблемы параллельного доступа		
	Грязное чтение	Неповторяемое чтение	Фантомные строки
READ UNCOMMITTED	Допускается	Допускается	Допускается
READ COMMITTED	Исключено	Допускается	Допускается
REPEATABLE READ	Исключено	Исключено	Допускается
SERIALIZABLE	Исключено	Исключено	Исключено

Обратите внимание на то, что в табл. 8.4 не нашлось места для такой важной проблемы, как потерянные обновления. Дело в том, что стандарт SQL требует, чтобы независимо от уровня изоляции вероятность потери обновлений данных была сведена к нулю.

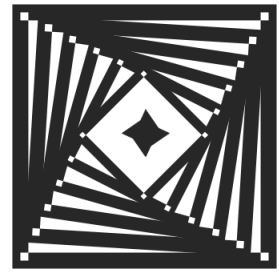
Замечание

Уровни изоляции транзакций назначают с помощью SQL-команды `SET TRANSACTION`. Эта инструкция настраивает поведение только одной очередной транзакции, после выполнения которой система автоматически возвратится к настройкам по умолчанию.

Резюме

Транзакция — очень сильное орудие, неумелое применение которого может существенно снизить производительность БД. Чтобы этого не произошло, при проектировании транзакций следует соблюдать ряд правил:

- транзакции должны затрагивать минимально возможный объем данных;
- проектируйте максимально короткие транзакции;
- остерегайтесь вложенных транзакций;
- там, где это возможно, не применяйте высокие уровни изоляции транзакций.



Глава 9

Проектирование БД

Наконец настал тот час, когда, собрав воедино все знания, полученные на предыдущих страницах книги, мы приступим к обсуждению собственно процесса проектирования БД. У читателя уже есть представление о реляционной модели данных, об объектах этой модели и особенности взаимодействия их друг с другом, мы способны строить ER-модели и осуществлять нормализацию данных, знаем о требованиях к реляционным таблицам и правилах поддержания целостности данных. Одним словом, мы уже достаточно компетентны для того, чтобы начать разговор о процессе проектирования БД в целом. Оговорюсь сразу, что на этих страницах нет четкой инструкции, в которой расписаны все шаги от "А" до "Я", пройдя которые вы получите идеальную БД, к сожалению, такой инструкции в природе не существует и быть не может. Наша цель другая — указать направление движения, отметить обязательные контрольные точки маршрута, проходить мимо которых не стоит, и, по возможности, уберечь начинающего от ряда подводных камней, после встречи с которыми отправится "на дно" любой проект.

Понятие информационной системы

На протяжении всех предыдущих глав мы с вами не раз говорили, что по своей сути любая БД представляет собой скрупулезно построенное хранилище данных. Само по себе хранилище ни для кого особого интереса не представляет до тех пор, пока оно не интегрируется в состав более сложной надсистемы, называемой информационной.

Замечание

Информационная система (Information System, IS) — это система, предназначенная для сбора, корректировки и распространения информации внутри организации и объединяющая в своем составе персонал, оборудование, базы данных и программное обеспечение.

Все элементы информационной системы объединены общей целью — обеспечить поддержку принятия решений по эффективному управлению предприятием, в котором развернута эта система. Почему мы говорим лишь о поддержке принятия

решения, а не о принятии решения в целом? Дело в том, что на сегодняшний день еще рано говорить о системах с развитым искусственным интеллектом, поэтому решение принимает не электронная машина, а человек (называемый экспертом) или группа лиц (экспертная группа). Для принятия того или иного решения эксперт нуждается не просто в связанных друг с другом реляционных таблицах, а в специально подготовленных структурированных данных (графиках, аналитических записках, статистических отчетах, прогнозах и т. п.), именно эти данные информационная система запрашивает у БД и затем предоставляет принимающему решение лицу в удобном для него виде.

Создание информационной системы — многогранный кропотливый процесс, затрагивающий практически все стороны жизнедеятельности организации. В самом общем случае он включает пять этапов:

1. Планирование системы.
2. Анализ задач системы и требований к системе.
3. Проектирование системы.
4. Реализация и ввод в эксплуатацию.
5. Сопровождение.

Каждый из этапов разделяется на десятки составляющих, над которыми трудятся соответствующие специалисты. Мы не станем отнимать их хлеб, но позволим себе одно весьма важное замечание — фундаментом любой информационной системы выступает база данных. Опыт подсказывает, что еще никому не удавалось построить отличный дом на плохом фундаменте, поэтому процедура проектирования БД должна быть неразрывно связана с процессом проектирования информационной системы.

Замечание

База данных, хотя и ключевая, но далеко не единственная составная часть информационной системы. Помимо БД в типичную информационную систему входят такие компоненты, как программное обеспечение поддержки БД, аппаратное обеспечение, прикладное ПО и эксплуатирующий систему персонал.

Прежде чем мы приступим к рассмотрению процесса проектирования БД, зададим себе вопрос, какие показатели служат безусловным признаком того, что предприятию или организации необходимо задуматься о разработке и внедрении своей собственной информационной системы? Необходимость разработки БД обычно наступает, когда существующая на данный момент система (если она вообще есть) перестала удовлетворять требованиям компании. Инициатива обращения за помощью к разработчику БД, безусловно, должна исходить от руководства предприятия, но это не означает, что сотрудники нижних звеньев управления и производства не заинтересованы в совершенствовании системы. Одним словом, на предприятии должна сложиться "революционная ситуация", когда начинает действовать классическая формула "… верхи не могут, а низы не хотят…".

Сравнение начала разработки новой БД с революцией не сильно преувеличено. Как правило, у любой, проработавшей хотя бы пару лет компании уже существует

своя более или менее налаженная система работы. Эта система может быть полностью основана на бумажных документах или частично автоматизирована. Поэтому руководителю предприятия (в большинстве своем консерватору) необходимы очень веские причины для того, чтобы в буквальном смысле слова выкинуть на помойку, пусть плохонькую, но пока еще работающую систему. Большинство этих причин он обнаружит в любом учебнике по организации бизнеса, самая главная из них меркантильная — увеличение прибыли и снижение расходов. Но для большинства людей, привыкших работать по старинке, это не столь наглядно. Для них прибыль и расходы четко ассоциированы с производительностью труда, экономией ресурсов и еще с сотней причин, но не каждый руководитель видит четкую связь между преувеличением собственных доходов и эфемерной информационной системой предприятия. С таким руководителем можно долго дискутировать о повышении эффективности контроля и управления компанией, которое получит предприятие после разработки и внедрения БД. Но, пока не произойдет "революция" в мышлении руководства предприятия, разработчику БД даже не стоит предлагать свои услуги.

Желание руководства компании разработать и внедрить на предприятии информационную систему — необходимое, но не достаточное условие для начала работы. Для создателя БД, особенно на этапе проектирования, очень важна поддержка простых сотрудников предприятия. Дело в том, что руководитель обычно владеет верхней частью информационного "айсберга", а это, образно выражаясь, только внешняя сторона Луны. То, что происходит на местах, руководству по объективным причинам неизвестно, ведь вникнуть во все детали не хватит и жизни. Все мелочи известны рядовым сотрудникам, естественно, каждому на своем рабочем месте. Технолог знает, какие ингредиенты входят в состав выпускаемой продукции, кладовщик умеет принимать готовую продукцию на хранение, водитель знает маршруты доставки продукции заказчикам и т. п. Именно из деталей складывается целостная картина всех бизнес-процессов, протекающих в стенах компании, поэтому игнорировать мелочи не следует.

Наконец, про запас я приберег самый главный вопрос, который просто необходимо задать потенциальному заказчику, готовому заплатить за разработку и внедрение в своей компании новой информационной системы. Дословно он звучит так: "Чем вас не устраивают имеющиеся на рынке программные продукты?" Было бы наивно полагать, что именно мы станем первопроходцами в вопросах разработки БД для отдела кадров, склада или библиотеки. Немного порывшись в Интернете, наверняка можно обнаружить пару-тройку программ по заданной тематике. Ответ на вопрос о программном обеспечении стороннего производителя очень важен. Ведь может статься, что руководство компании просто не знает о положении вещей на рынке ПО. Для нас с вами это скорее минус, чем плюс. Почему? Хотя бы потому, что спустя полгода или год с начала выполнения программистами заказа руководство компании "неожиданно" выяснит о существовании уже готовой БД, написанной одним из конкурентов. В итоге разработчику намекнут, что в его услугах больше нет нужды. Поэтому всегда следует быть в курсе продукции потенциальных конкурентов, что, по крайней мере, сэкономит драгоценное время…

Замечание

Есть существенная разница между универсальным программным продуктом стороннего производителя и программным обеспечением, разработанным специально для компании. Различие ровно такое, как между костюмом, пошитым в массовом порядке на фабрике, и платьем, скроенным портным специально для вас. Первый продукт необходимо долго и упорно адаптировать к условиям конкретной компании, второй — сразу придется впору. Кроме того, наличие в распоряжении компании своего собственного разработчика (да еще и "скованного" контрактом на сопровождение БД) позволит ей оперативно парировать все сбои в работе БД и быстро вносить изменения и доработки.

Этапы жизненного цикла базы данных

База данных не только функционирует внутри информационной системы, но развивается вместе с этой системой, поэтому в начале 1980-х годов был введен термин *жизненный цикл базы данных* (Database Live Cycle, DBLC).

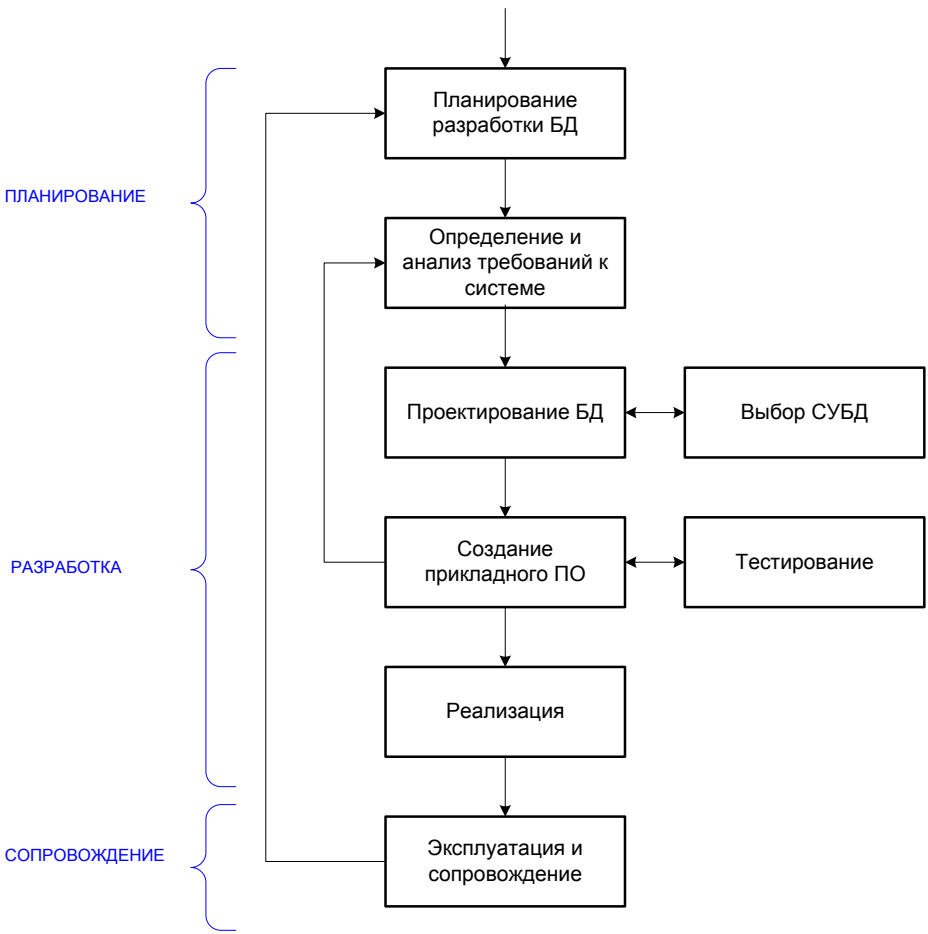


Рис. 9.1. Жизненный цикл разработки БД

На рис. 9.1 представлены наиболее существенные этапы жизненного цикла БД, эта схема и станет путеводителем по излагаемому далее материалу. Вне всякого сомнения, специалист по проектированию БД способен представленную схему расширить, добавив в нее дополнительные элементы и связи, но я уверен, что у него не поднимется рука удалить хотя бы один из блоков, ведь здесь собран тот золотой минимум, обойтись без которого нельзя.

Нашу беседу начнем с самого главного секрета — процесс проектирования БД начинается не с включения компьютера, а с остро заточенного карандаша и пачки листов бумаги. Чем больше вы изведете бумаги на первых этапах проектирования, тем потом меньше придется корпеть за клавиатурой.

Планирование разработки БД

На этапе *планирования разработки* БД проектировщик должен получить ответы на ряд предварительных вопросов:

- Какой объем работы придется выполнить?
- Какой состав сил и средств потребуется для выполнения намеченного объема работ?
- Сколько времени будет затрачено на разработку и реализацию проекта?
- Какова будет стоимость всего проекта?

Безусловно, ответы на перечисленные вопросы окажутся весьма приблизительными, но уже по ним вы получите очень важные ориентиры как для себя, так и для заказчика проекта. Все четыре показателя (объем работ, силы и средства, время и стоимость) должны в равной степени устраивать и проектировщика, и заказчика, причем каждая сторона должна понимать, что это всего лишь предварительные расчеты.

ЗАМЕЧАНИЕ

В идеале этап планирования должен закончиться оформлением юридического документа, представляющего собой договор о намерениях проектировщика и заказчика разработать, реализовать и ввести в эксплуатацию программный продукт.

Определение и анализ требований к системе

Следующий *этап определения и анализа требований к системе* едва ли не самый значимый во всем жизненном цикле БД. На этом этапе должна быть собрана вся информация о компании и протекающих в ней бизнес-процессах, необходимая для проектирования БД. Если у вас это не получится, то расплачиваться за некорректно отработанный этап придется до конца существования проекта. Форма расплаты самая разнообразная — от бесконечных мелких переделок проекта, до полного свертывания работ.

В качестве иллюстрации позволю себе перефразировать пример, приведенный в одной из книг общепризнанного "гуру" в области объектно-ориентированного программирования — Гради Буча. Вряд ли заказчику уже возведенного 100-этажного здания придет в голову мысль попросить строителя перестроить фундамент. Но почему-то заказчику программного продукта, вдруг неожиданно вспомнившему о какой-то дополнительной неучтенной детали проекта, кажется вполне логичным потребовать от программиста переделать фундамент уже готовой программы. Если читатель имеет опыт разработки программного обеспечения, то он наверняка знает, что зачастую легче написать программу заново, чем пытаться перевернуть все с ног на голову в уже готовом проекте.

С запугиванием покончено. Теперь я уверен, что у читателя возросло чувство ответственности за этап определения и анализ требований к системе. Переходим к построению списка вопросов, ответы на которые нужно получить на втором этапе жизненного цикла. Независимо от масштабов нашей БД (быть может, вы автоматизируете билетную кассу провинциального кинотеатра, а может, вам в руки попалась трансконтинентальная торговая сеть) было бы здорово сразу получить следующую информацию:

1. Цели и задачи компании.
2. Организационно-штатную структуру компании.
3. Модель бизнес-процесса компании.

Это еще далеко не весь перечень, но пока переведем дух и обсудим первые три пункта списка. Цели и задачи компании необходимы, чтобы понять направления ее основной деятельности. Зная целевую функцию, можно, пусть даже пока на упрощенном уровне, предсказать, что именно захочет получить от будущей БД заказчик. Самое печальное, что даже заказчик зачастую неспособен кратко и точно описать сферу интересов своей компании… Вспомните книгу английского математика Льюиса Кэрролла "Алиса в стране чудес". На одной из ее страниц состоялась поучительная беседа Алисы и Чеширского кота:

— Скажите, пожалуйста, куда мне отсюда идти? — спросила Алиса.

— А куда ты хочешь попасть? — ответил Кот.

— Мне все равно... — сказала Алиса.

— Тогда все равно, куда и идти, — заметил Кот.

Перенесите этот разговор на сегодняшний день. Пока заказчик базы данных не разберется с направлением движения, разработчику целесообразно оставаться на месте.

Обсудим второй пункт нашего "шпионского" списка. Знание организационной структуры предприятия просто необходимо при определении информационных потоков внутри компании. Кому-то нужны отчеты, кому-то ведомости, кто-то хочет владеть всеми данными, кому-то достаточно знать лишь детали. Более того, если компания не может похвастаться грамотно спроектированной компьютерной сетью, то рассматривайте построение (модернизацию) сети как одну из частных задач проекта БД. Тогда вам придется доказать руководству, что размещение хостов, коммутаторов, маршрутизаторов и прочего оборудования увязывается не только с географическим расположением объектов компании, но и с ее логическими ин-

формационными потоками. Иначе вы рискуете тем, что идеально спроектированная БД окажется развернутой не в узловой точке схождения потоков с информацией, а где-нибудь на задворках системы… После этого вы столкнетесь с необъяснимым снижением производительности сервера и с разрывами транзакций в самых неожиданных местах.

В идеале к организационной структуре компании неплохо добавить и штатную. Она не окажется помехой, даже если вы всего-навсего автоматизируете небольшой участок предприятия, например, склад фабрики. Опыт подсказывает, что к концу проекта руководство неожиданно захочет знать, какой из кладовщиков получал товар по накладной № X, кто из начальников цехов не вернул на склад остатки строительных материалов и кого, в связи со всем вышесказанным, следует отправить на пенсию в ближайшее время…

Третий пункт списка самый сложный во всем этапе определения и анализа требований. Все, о чем мы говорили до сего момента, вполне укладывается в модель знаний специалиста по информационным технологиям. Здесь задействованы либо родные для программиста специальные знания либо понятные каждому общедоступные сведения. Но моделирование бизнес-процесса предприятия даже для суперспециалиста по IT — это действительно проблема. Причина тому — слабые знания в той предметной области, для обслуживания которой предназначена будущая БД. Недостаток знаний — объективная реальность, и с ней ничего не поделаешь. Вы можете быть превосходным программистом и высекать искры из клавиш компьютера ночи напролет, но как только вы столкнетесь с задачей автоматизации реального производства, торгового предприятия, медицинского учреждения или пожарного депо, не стоит завышать свою самооценку. Чем быстрее придет понимание, что ваши знания в этой области стремятся к нулю, тем будет лучше для вас! Если вам так не кажется, то, пожалуйста, перечитайте этот абзац заново. Ведь излишняя самонадеянность погубила уже немало проектов.

Проблему некомпетентности разработчика БД в предметной области компании можно решить несколькими способами. Лучший, но практически не реализуемый подход заключается в срочном поступлении в вуз на специальность экономиста (инженера-строителя, провизора, пожарного и т. п.). Правда, к тому моменту времени, когда вы приобретете достаточный уровень знаний в области бизнес-процессов компании, ваши услуги вряд ли окажутся востребованными. Поэтому рекомендую пойти другим путем — вовлечь в процесс проектирования БД максимальное число специалистов компании.

Вовлечение сотрудников предприятия в проект совсем не означает, что теперь вместо выполнения своих прямых должностных обязанностей они в рабочее время станут рисовать модель будущей БД. Подход будет несколько иной. Вы получите всю требуемую информацию путем:

- опроса основных специалистов компании;
- анализа обязанностей сотрудников;
- изучения документов на рабочих местах, в отделах и службах компании, в особенности документов, претендующих на роль отчетных (ведомости, накладные, заказы, заявки и т. д.);

- наблюдения за процессом функционирования компании, особенно в области документооборота;
- анкетирования сотрудников компании.

Конечно же, собранные подобным образом данные окажутся абсолютно не структурированными, но это все равно лучше, чем отсутствие информации вообще. Кстати, будьте внимательны, в своей совокупности полученная информация может считаться конфиденциальной, что налагает на разработчика БД дополнительную ответственность.

Опытный, разбирающийся в специфике работы компании, разработчик БД может сразу заметить изъяны в требованиях руководства и пользователей к проекту и попытаться немедленно усовершенствовать полученную информацию. Однако для начинающего проектировщика более логичным решением станет внесение правок в (некорректные на его взгляд) требования в присутствии заказчика.

Пока мы обсудили только первые три пункта этапа *определения и анализа требований к системе*, а теперь переходим к оставшимся:

1. Выделение границ и возможностей проекта.
2. Анализ смежных бизнес-процессов с целью выявления перспектив дальнейшего совершенствования БД.

Качество выполнения четвертого и пятого пунктов этапа определения и анализа требований к системе находится в прямой зависимости от опыта и интуиции разработчика БД. Вы только что построили модель бизнес-процесса компании, а теперь необходимо найти компромисс между излишней детализацией процесса, с одной стороны, и недопустимым упрощенчеством — с другой. Умению балансировать ровно посередине научить невозможно, здесь разработчик напоминает канатоходца — мастерство приходит со временем (вместе с шишками и синяками).

Замечание

Очерчивая границы проекта, обязательно задумайтесь о перспективах его дальнейшего развития. Всегда рассматривайте текущий проект как элемент конструктора, к которому завтра наверняка захочется присоединить очередной элемент.

Мы привели всего лишь упрощенное описание этапа определения и анализа требований к системе. На сегодняшний день создана достаточно эффективная методология по разработке требований к программному обеспечению и управлению этими требованиями. К сожалению, в рамках настоящей книги мы не сможем даже поверхностно рассмотреть этот важный вопрос.

Внимание!

Логическим завершением этапа определения и анализа требований к системе должен стать проект технического задания на разработку БД, который согласовывается с участвующими в проекте руководителями основных отделов и служб компании и утверждается руководством компании (заказчиком БД). В окончательное техническое задание проект превратится в середине этапа проектирования БД.

Проектирование БД

На рис. 9.1 этап проектирования БД представляет собой всего лишь один прямоугольник, но на самом деле он не столь прост. Специалисты разделяют этот этап на три фазы (рис. 9.2):

1. Концептуальное проектирование.
2. Логическое проектирование.
3. Физическое проектирование.

Во время концептуального проектирования окончательно формируется замысел будущей базы данных, но без учета любых физических аспектов ее реализации. На этой ступени проектирования разработчика пока не интересует ни конкретная СУБД, на которой позднее развернется БД, ни используемый для создания приложений язык программирования, ни особенности аппаратной платформы. Пока наш основной интерес направлен на создание общей модели, отражающей представления будущих пользователей БД об автоматизируемом участке компании (складе, бухгалтерии, отделе кадров, производственных цехах и т. п.). Все необходимая для этого информация уже должна быть собрана на предыдущем этапе жизненного цикла БД.

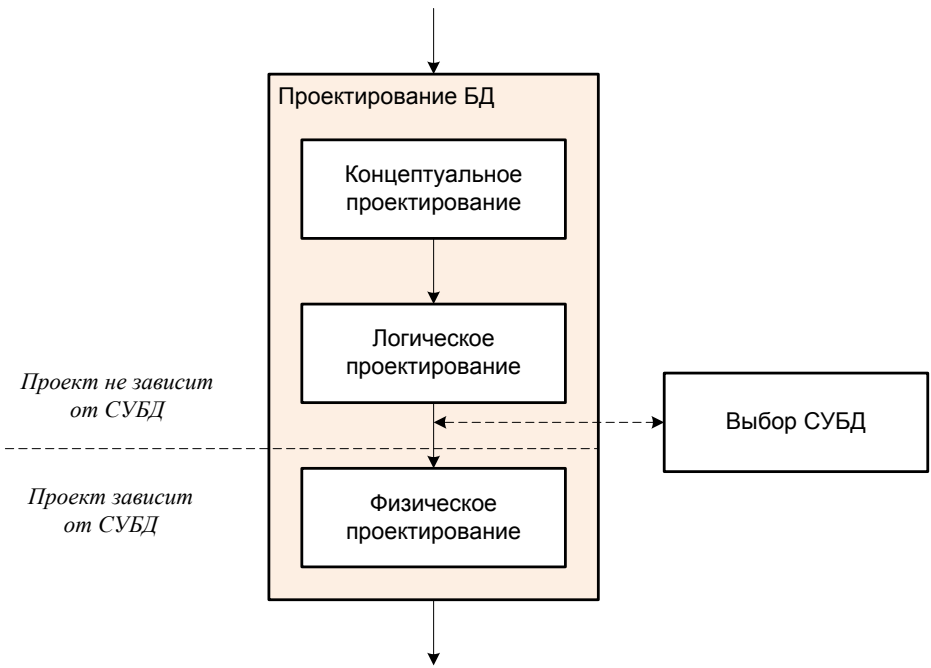


Рис. 9.2. Фазы проектирования БД

Основным средством построения концептуальной модели БД выступает уже хорошо знакомая нам по *главе 5* модель "сущность-связь" (ER-модель) или родст-

венные модели. В основе таких моделей лежит простая и одновременно эффективная методика, позволяющая наглядно представлять смысл подлежащих хранению в БД данных.

На завершающей стадии концептуального проектирования разработчик БД обязательно должен проверить адекватность полученной модели, для этого он обсуждает полученные результаты (элементы ER-диаграмм) с сотрудниками компании. При обнаружении несоответствий в модель немедленно вносят исправления. Процесс сверки прекращается только после того, когда все пользователи подтвердят корректность концептуальной модели.

Таким образом, результатом фазы концептуального проектирования станет ER-модель будущей БД, включающая в себя описание:

- типов сущностей;
- связей между типами сущностей;
- атрибутов (желательно с их предварительным описанием доменов и ограничений);
- первичных ключей.

Фаза логического проектирования предназначена для преобразования обобщенной концептуальной модели в завершенную логическую. Девизом логической фазы может стать фраза: "Анализ и рационализация". Разработчик уточняет все требования, выявленные на концептуальной стадии проектирования, и стремится несколько упростить решение (не снижая его функциональные возможности). Для этого ER-модель проверяют с помощью правил нормализации (см. *главу 6*). В результате мы получаем неизбыточные реляционные таблицы, свободные от присущих ненормализованным данным аномалиям вставки, редактирования и удаления.

Помимо нормализации, на логическом этапе осуществляют следующие действия:

- уточняют ограничения на данные;
- определяют домены данных;
- вводят бизнес-правила и корпоративные ограничения целостности;
- определяют местоположение будущих таблиц (в случае если речь идет о распределенной БД).

Представленную логическую модель вновь сверяют с будущими пользователями БД и заказчиком проекта. Вносят последние уточнения и исправления, и все члены проектной группы и заказчик проекта приходят к единому мнению относительно создаваемой БД.

На этапах концептуального и логического проектирования разработчики обычно используют одну из двух стратегий проектирования БД: *восходящее проектирование* (bottom-up design) или *нисходящее проектирование* (top-down design).

Восходящий подход обычно применяется для сравнительно небольших проектов. Суть метода заключается в том, что проектировщик совместно с заказчиком БД строят полный список атрибутов (полей таблиц) подлежащих хранению. Позднее атрибуты группируют в типы сущностей, которые попадают в модель. Если вы ненадолго вернетесь к *главе 6*, посвященной нормализации таблиц, то увидите, что процесс нормализации основан на восходящем подходе.

Обратная восходящей, нисходящая стратегия лучше подходит для средних и больших проектов. Здесь проектирование начинается с выявления основных типов сущностей и только затем сущности "обрастают" атрибутами. Классический пример нисходящего метода — модель "сущность-связь" (ER-модель).

Какой из стратегий следует отдавать предпочтение? Прав окажется тот разработчик, который станет совместно применять и восходящую и нисходящую методологии (в нашем случае нормализацию таблиц и ER-модель).

Внимание!

По завершении фазы логического проектирования совместно с заказчиком создается окончательная версия технического задания на БД. В техническом задании закрепляют требования к проекту, утверждают логическую модель БД, описывают необходимые отчеты и т. п. После разработки технического задания разработчик БД и заказчик заключают юридический договор на разработку БД, в котором помимо всего прочего окончательно утверждают стоимость и сроки выполнения проекта.

К следующей (физической) фазе проектирования БД переходят после выбора целевой СУБД, именно она определяет особенности будущего программного продукта. О правилах отдания предпочтений той или иной технологии при выборе СУБД вы прочитаете страницей позднее, а пока отметим, что с этого момента все остальные фазы проектирования и этапы жизненного цикла БД приобретают зависимость от СУБД.

Замечание

Ряд специалистов полагает, что целевую СУБД корректнее выбирать не после логического проектирования, а несколько ранее — вслед за этапом концептуального проектирования [11]. Мы не будем вставать ни на ту, ни на другую сторону. Главное, чтобы разработчик определился с СУБД до начала этапа физической разработки БД.

Физическое проектирование — это уточнение решения с учетом имеющихся в наличии разработчика технологий, возможности реализации и требуемой производительности. Только на заключительной фазе проектирования БД на смену так нелюбимой программистами бумажной деятельности приходит реальная работа на компьютере. Во время физического проектирования задачей проектировщика становится перенос логической модели на платформу целевой СУБД. С этой целью разработчик делает следующее:

- создает таблицы и связи между ними;
- назначает вторичные индексы таблиц;
- реализует бизнес-логику БД (в первую очередь, с помощью триггеров и хранимых процедур);
- определяет функциональные характеристики транзакций;
- разрабатывает представления;
- внедряет механизмы защиты (как минимум предусматривает авторизацию пользователей и назначает правила доступа к данным).

Полученная БД тщательным образом документируется. Особенно важно определить пользовательские типы данных, описать таблицы и связи между ними, за-

дать порядок поддержки бизнес-логики, установить назначение порядок вызова хранимых процедур и триггеров.

Несмотря на то, что на рис. 9.2 фазы проектирования нарисованы в виде последовательных операций, вполне допускается их взаимное *перекрытие* (параллельное развитие) и *итеративность* (каждая фаза из-за многократных проверок уточнений осуществляется в несколько циклов).

Выбор СУБД

Успех функционирования БД теснейшим образом связан с СУБД, на основе которой будет реализован проект. Поэтому выбор подходящей платформы во многом определит достоинства и недостатки БД. Существует достаточно широкий набор критериев оценки СУБД, сопоставив которые проектировщик сможет остановить свой выбор на конкретном программном продукте.

На самом первом шаге разработчик должен определить тип модели данных. Напомню, что помимо реляционной в природе существуют сетевая, иерархическая и объектно-ориентированная модели. Наша книга посвящена наиболее популярной реляционной модели, поэтому будем полагать, что с выбором модели у нас вопросов не возникает.

Второй шаг связан с выбором особенностей доступа к БД. Если разрабатываемая БД будет эксплуатироваться всего на одной рабочей станции или в режиме файл-сервер очень небольшой группой пользователей, то имеет смысл остановить свой выбор на настольных СУБД: Microsoft Access, Microsoft FoxPro, БД в формате Blackfish SQL из состава программного пакета Embarcadero RAD Studio XE или Delphi Prism и т. п. Если же изначально предполагаются клиент-серверные решения, то интерес должны представлять более серьезные программные продукты: Oracle, InterBase, Firebird, MySQL, Informix, DB2, Microsoft SQL Server, PostgreSQL и т. п.

После сужения спектра ПО, претендующего стать основой информационной системы компании, нам предстоит проанализировать и сравнить множество показателей исследуемых СУБД:

1. Стоимость программного продукта.
2. Совместимость с действующим стандартом SQL.
3. Переносимость системы (независимость от платформы, ОС и языка).
4. Масштабируемость системы (возможность увеличения числа пользователей без потери производительности).
5. Открытость системы (способность взаимодействовать с ПО сторонних производителей).
6. Требования к аппаратной части компьютера.
7. Производительность системы.
8. Поддержка транзакций и параллельной обработки данных.
9. Наличие удобных средств проектирования.

10. Наличие системы безопасности БД.
11. Утилиты резервного копирования и восстановления.
12. Возможность репликации данных.
13. Развитые инструменты администрирования.
14. Способность к модернизации.
15. Поддержка и обновление СУБД производителем.
16. Способность работать в современных локальных и глобальных сетях.
17. Способность функционировать в распределенных системах.
18. Простота обучения персонала.

Представленный список показателей ни в коем случае не полон, кроме того он не упорядочен по важности. Например, вполне вероятно, что для состоятельной компании более существенным показателем окажется наличие развитой системы защиты баз данных, чем стоимость СУБД.

Результаты анализа программных продуктов различных производителей обсуждают с заказчиком базы данных и только затем принимают решение о выборе целевой СУБД.

ВНИМАНИЕ!

При выборе СУБД не забывайте о существовании бесплатных и условно бесплатных программных продуктов: MySQL, Firebird и PostgreSQL.

Создание прикладного программного обеспечения

Завершив работу над формированием физической структуры БД, проектировщик переходит к разработке приложений, предназначенных для работы с БД. Пользователь не должен вникать в особенности реляционных БД, ему не стоит учить язык запросов SQL, пользователь просто должен обладать необходимым инструментом, благодаря которому он без особых затруднений сможет воспользоваться услугами БД. Поэтому, с одной стороны, задача разрабатываемого нами прикладного ПО — предоставление интуитивно понятного интерфейса конечному пользователю, а с другой — обеспечение прозрачного для пользователя взаимодействия с БД.

В самом общем случае приложение баз данных должно уметь делать следующее:

- получать доступ к БД;
- читать, добавлять, редактировать и удалять данные;
- представлять полученные данные в требуемом пользователем виде (формы, отчеты, многомерное представление и т. д.);
- поддерживать целостность данных, определять дополнительную бизнес-логику и ограничения на данные;
- обеспечивать требуемую безопасность данных.

В качестве инструментальных средств, применяемых для написания прикладного ПО, сегодня используют языки программирования 4-го поколения (*4 genera-*

tion level). Программам попроще обычно хватает возможностей генераторов форм, имеющихся в распоряжении настольных СУБД класса Microsoft Access. Профессиональные клиент-серверные проекты преимущественно разрабатывают на платформах серьезных сред программирования, таких как Embarcadero RAD Studio XE.

Создание пользовательских приложений — сложный и кропотливый процесс, требующий от программиста глубоких знаний не только СУБД и языка программирования, но и операционной системы, а также методов программирования (эффективных алгоритмов).

Как правило, создание коммерческого ПО предваряется выпуском прототипов будущих приложений. Прототип представляет собой работоспособную модель приложения, но с несколько ограниченным функционалом. Основное назначение прототипа — предоставить будущим пользователям возможность опробовать приложение в действии и выяснить, в каких улучшениях они заинтересованы.

Тестирование

Без тестирования мы с вами не сможем гарантировать заказчику правильную работу созданного продукта. Поэтому, независимо от степени сложности БД, для каждого проекта необходимо разработать исчерпывающий план тестирования, распространяющийся на все основные функции БД и прикладного ПО.

По своей сути процесс тестирования ПО представляет собой эксплуатацию приложения в контролируемых условиях с последующим анализом полученных результатов. При этом проверяют работу приложения не только с нормальными, но и с ошибочными данными. Также во время тестирования следует изучить поведение БД и ПО в нештатных ситуациях.

ЗАМЕЧАНИЕ

Во время тестирования ищут и устраняют ошибки в ПО. Для гарантированного исключения повторного появления одной и той же ошибки приложение тестируют повторно.

Различают три уровня тестирования ПО:

- *Модульное тестирование* — тестируют минимальные компоненты ПО, например запрос, процедуру, триггер.
- *Интеграционное тестирование* — модули ПО объединяют и проверяют на корректность взаимодействия между интегрируемыми компонентами.
- *Системное тестирование* — тесту на соответствие техническому заданию подлежит система в целом.

Системное тестирование обычно включает в себя два этапа: альфа-тестирование и бета-тестирование. На первом этапе имитируют реальную работу БД и прикладного ПО, причем в качестве пользователей выступают штатные разработчики проекта. При бета-тестировании программное обеспечение передают заказчику (или третьему лицу) для апробации работы и выявления не замеченных ранее ошибок.

Реализация

На этапе реализации выпускают окончательную версию программного продукта. Прежде чем проектировщик переходит к выпуску БД, необходимо убедиться в выполнении следующих условий:

- база данных и пользовательские приложения обладают всеми указанными в техническом задании функциональными возможностями;
- программное обеспечение успешно прошло альфа-тестирование и все выявленные критические и серьезные ошибки устранены;
- заказчик и проектировщик приняли совместное решение о том, что реализация всех элементов БД в целом завершена.

На завершающей стадии этапа реализации разработчик ПО готовит материалы, необходимые для обучения и технической поддержки пользователей и для сопровождения приложения.

Эксплуатация и сопровождение

Эксплуатация и сопровождение — заключительный этап жизненного цикла БД. Вначале развертывают базы данных и прикладное ПО на сервере и клиентских станциях заказчика ПО. При необходимости администратор СУБД создает учетные записи пользователей базы данных. Затем проект полностью переходит под контроль заказчика.

Внимание!

Развертывание БД и пользовательского ПО возможно только после выполнения всех тестов, доказывающих их стабильность.

Основную часть обязанностей по сопровождению БД возлагают на администратора СУБД или (если на платформе СУБД развернуто несколько больших БД) на администратора БД. Администратор осуществляет профилактическое обслуживание БД, восстанавливает систему после сбоев, управляет учетными записями пользователей, следит за производительностью системы, поддерживает безопасность БД, ведет системный аудит. Одним словом, администратор поддерживает БД в работоспособном состоянии.

Хотя эксплуатация и сопровождение — это заключительный этап жизненного цикла, совершенствование введенного в эксплуатацию проекта может продолжаться. У заказчика могут возникнуть дополнительные пожелания по развитию проекта (добавление новых форм ввода и просмотра данных, запросов, отчетностей). Благодаря гибкости реляционной модели хорошо спроектированная БД даже будет допускать незначительные изменения в структуре таблиц и в бизнес-логике проекта.

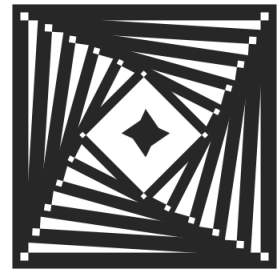
Но однажды наступит день, когда приспособить БД к новым реалиям окажется невозможно. Что следует предпринять? Направление дальнейшего движения под-

скажет стрелка, исходящая из этапа эксплуатации и сопровождения (см. рис. 9.1), мы вновь оказываемся на первой ступеньке лестницы жизненного цикла БД и начинаем планировать разработку новой БД.

Резюме

Процесс проектирования и реализации жизнеспособной БД содержит серию этапов, включающих планирование разработки, определение и анализ требований к системе, проектирование БД, выбор целевой СУБД, создание прикладного ПО, тестирование, реализацию, эксплуатацию и сопровождение.

Перечисленные этапы обладают итеративными чертами и предусматривают возможность возврата к предыдущему или даже к начальному этапу проектирования при невозможности реализовать пожелания заказчика к проекту в заданном объеме и с требуемым качеством.



Глава 10

Защита БД

Наш мир устроен так, что в нем недостаточно просто посадить дерево, спроектировать дом или разработать компьютерную программу. Никто не даст вам гарантии, что завтра дерево не срубят, дом не снесут, а программу не взломают. Окружающий нас мир не всегда доброжелателен, поэтому нужно уметь защищать свое творение, а в идеале — научить свой продукт защищаться самому.

Актуальность защиты компьютерной информации вряд ли вызывает сомнение. Мы живем в такое время, когда практически ежедневно приходят вести о том или ином киберпреступлении. Утрата или хищение данных из БД особенно болезненны, т. к. одним махом затрагивает тысячи, а иногда и миллионы людей. Ценность информации, которую эти люди доверили разработанной вами программе, зачастую имеет не просто денежное измерение. Задумайтесь, сколько будет стоить репутация программиста, если спроектированная им БД рухнет от первой же атаки.

Под защитой информации понимается система мер, направленная на предотвращение доступа к данным несанкционированных пользователей, с целью их чтения, изменения или разрушения. Вместе с тем, лицу, отвечающему за безопасность БД, не следует забывать, что санкционированный пользователь, без всякого на то злого умысла также способен принести массу бед.

Если говорить строго, то безопасность данных — это состояние защищенности, при котором обеспечивается *конфиденциальность*, *доступность* и *целостность данных*. Каждая из составляющих безопасности несет "персональную" ответственность за свой участок обороны:

- Конфиденциальность — отвечает за обеспечение доступа к данным только санкционированным пользователям.
- Целостность — исключает несанкционированное изменение структуры и содержания данных.
- Доступность — позволяет обеспечить доступ к данным санкционированным пользователям по их первому требованию.

Конфиденциальность направлена на сохранение в тайне данных, наиболее важных для организации, и на обеспечение неприкосновенности личных данных. Пробить брешь в защите конфиденциальных данных хотят очень многие, список любопытных "гостей" определяется важностью информации, находящейся в БД. Здесь и представители иностранных разведок, организации, занимающиеся промышлен-

ным шпионажем, конкурирующие предприятия, различные спецслужбы, спамеры, хакеры, преступные элементы и обычные частные лица, желающие покопаться в ваших данных просто из спортивного интереса.

Потеря целостности грозит искажением или даже разрушением хранимых в БД данных. И то и другое, как минимум, приводит к остановке работы БД на период восстановления данных. Вынужденный простой компании влечет за собой не только денежные потери от незаключенных сделок, но и наносит удар по престижу. Иногда потери от искажения данных приносят больше вреда, чем их частичное или даже полное разрушение. Так происходит, когда персонал вовремя не заметил факта искажения хранимой в БД информации и, например, продал пассажиру билеты на давно ушедший поезд.

Утрата доступности приводит к тому, что пользователь лишается возможности обращаться к хранимым в системе данным. Последствия от такого происшествия могут быть не просто печальными, но и самыми трагическими, в особенности, если речь идет об информационной системе диспетчера аэропорта, системе управления вооружением, программном обеспечении опасного производства и т. п.

Конфиденциальность, целостность и доступность составляют минимальный набор требований к безопасности данных. Специалисты по защите информации могут добавить еще несколько аспектов, например *аутентичность* (проверка подлинности данных), *апеллируемость* (подтверждение авторства), *достоверность* и т. д.

ЗАМЕЧАНИЕ

Защищенная база данных — это БД, которая обеспечивает конфиденциальность, доступность и целостность данных пльзователя.

Откуда исходят угрозы

Как бы это ни было обидно, но с точки зрения защиты информации наша база данных всегда будет выступать в незавидной роли жертвы, причем за добычей зачастую следит не один, а целая стая "хищников". Поэтому нападения приходится ожидать в любой момент времени и с любой стороны. Как видите, игра с самого начала неравная. Чтобы сгустить краски, отметим еще одну маленькую деталь "охоты". Жертва всегда обязана играть по правилам (правовым, моральным, этическим), а нападающий соблюдать их обычно не намерен. В такой игре мы никогда не победим, слишком не равны силы противоборствующих сторон и условия сражения. Поэтому целью защиты баз данных может выступать только одно — минимизация потерь от неминуемого нападения. Но откуда ждать беды?

Разобравшись с базовыми требованиями к безопасности данных, выявим источники угроз для защищаемой нами БД. Существует множество вариантов классификации угроз информационной системе: по природе возникновения, по источнику угроз, по способу доступа к защищаемому ресурсу, по степени воздействия на

систему, по расположению источника. Нас устроит классификация по способу осуществления угроз.

Явные угрозы:

- некорректная реализация механизма защиты;
- неправильная настройка механизма защиты;
- неполное покрытие каналов доступа к информации средствами защиты.

Скрытые угрозы:

- нерегламентированные действия пользователя;
- ошибки и закладки в программном обеспечении.

На рис. 10.1 представлена типичная многопользовательская компьютерная система небольшой компании. Пользователи объединены в локальную вычислительную сеть, которая включает в себя сервер БД и предоставляет возможность выхода в глобальную паутину Интернет. Подобные системы развернуты в десятках тысяч офисов, отделов и служб. И практически везде вы без труда найдете все перечисленные угрозы.

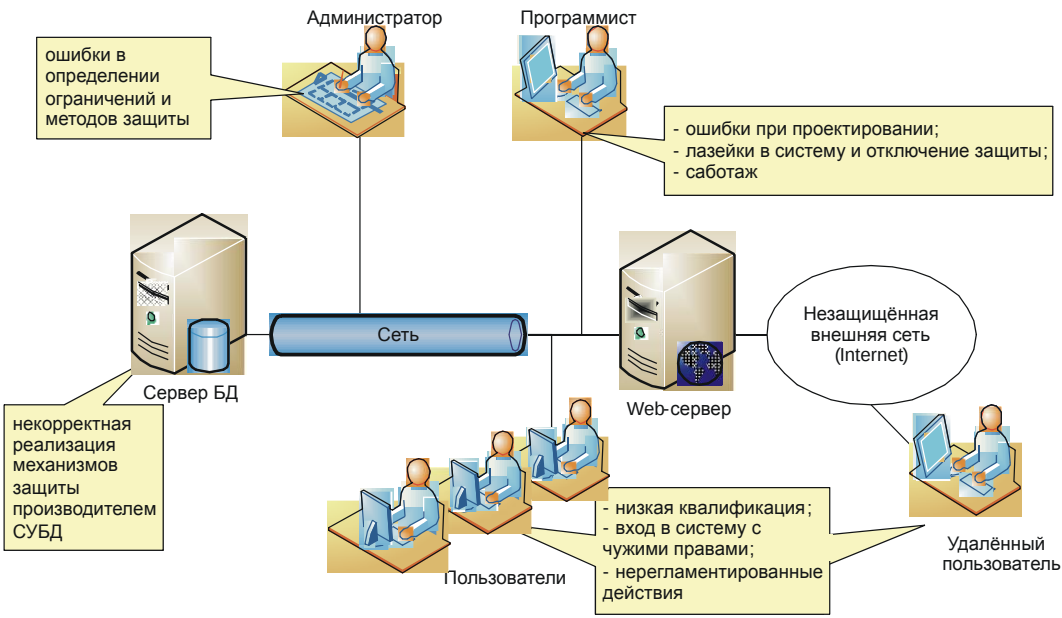


Рис. 10.1. Типичная многопользовательская компьютерная система

Сразу обратим внимание на ведущего специалиста информационной системы — администратора. В нашем примере это лицо собирательное, объединяющее в себе администратора сети, администратора СУБД и администратора БД. Ко всему прочему, администратор отвечает за защиту компьютерной информации, другими словами, пытается противодействовать всем явным угрозам. Но администратор сам выступает в качестве источника угроз, ведь все его промахи в настройке механизма защиты компьютерной системы снижают уровень безопасности.

Программист, даже самый подготовленный, также является источником скрытых угроз для системы, что уж говорить о специалисте среднего уровня. Любой человек склонен ошибаться, и программист — не исключение из правил. Ошибки в исходном коде программ не так страшны, в основном они вылавливаются во время тестирования и в начальный период эксплуатации. В период проектирования разработчик зачастую вынужден идти на сознательное нарушение правил разработки безопасного ПО и отключать или обходить элементы системы защиты. Иногда такие действия продиктованы объективными причинами. Представьте себе, что вы пишете какое-то клиентское приложение для работы с БД, и каждый раз во время запуска программы она требует ввести имя и пароль пользователя для регистрации на сервере. Окно, требующее ввести логин, выскакивает по сто раз в день на протяжении всего периода разработки. В подобных случаях даже самый правильный программист найдет способ обойти утомительную процедуру авторизации. Но ошибка не в создании лазейки, а в другом — программист не должен забыть удалить эту лазейку в систему из кода передаваемой в эксплуатацию окончательной версии приложения. Как вы догадываетесь, так происходит далеко не всегда. Если же программист создает программные закладки сознательно, с целью навредить компании, то выявить их крайне сложно.

В любой компьютерной системе основным действующим лицом является не администратор или программист, а обычный пользователь. Если он обладает высокой квалификацией, то окажется первым помощником разработчика БД. Если высокая квалификация сочетается с не менее высокой ответственностью, то пользователь не станет совершать нерегламентированные действия: устанавливать на своем рабочем компьютере постороннее ПО, посещать не связанные с профессиональными обязанностями сайты, получать почту со спамом и вирусами, не забудет заблокировать компьютер во время обеденного перерыва. Одним словом, подготовленный пользователь постарается не причинить ущерб своей компании, ведь в ней он зарабатывает на жизнь. На другом полюсе находится низкоквалифицированный оператор, обладающий правами доступа к БД. Даже самый отъявленный хакер вряд ли сможет нанести столько урона компьютерной системе, сколько он. Среди остроумных законов Мерфи для программистов есть один, полностью раскрывающий суть проблемы: "Невозможно создать программу с полной защитой от дураков, ибо они крайне изобретательны".

На СУБД воздействуют и весьма неординарными методами. Одна из специфичных для СУБД угроз называется *SQL-инъекцией* (SQL injection). Подобным атакам могут подвергаться СУБД, поддерживающие работу различных интернет-проектов, в первую очередь связанных с электронной коммерцией. Идея инъекции заключается во внедрении в текст запроса вредоносного кода. Если СУБД (а точнее СУБД и администрирующий ее персонал) не в состоянии противодействовать SQL-инъекциям, то злоумышленник относительно легко сможет направить базе данных запрос любого содержания (от чтения и правки данных, до удаления объектов БД).

Мы рассмотрели далеко не все опасности. Можно еще упомянуть кражу данных, разрушение данных в результате физического повреждения оборудования, стихийные бедствия, диверсии, заражения вирусами, отказ в доступе в результате обрывов соединительных кабелей, перебои электропитания и сетевые атаки.

Правила защиты БД

База данных, как составной элемент информационной системы предприятия, нуждается в комплексной системе защиты. В набор средств защиты обязательно должны входить как организационные, так и компьютерные мероприятия. В противном случае мы рискуем создать однобокую систему противодействия угрозам, вооруженную до зубов против определенного набора опасностей (допустим вирусной или сетевой атаки), но абсолютно беспомощную против элементарной кражи.

О компьютерных мерах мы с вами поговорим позже, а пока позволим себе, не останавливаясь на деталях, упомянуть элементы организационной составляющей безопасности.

- *Подбор и расстановка кадров*. Поставить на ответственные посты компании лиц, не прошедших должную проверку, — все равно, что поселить лису в курятнике. Поэтому кадровые органы представляют собой самую первую линию обороны любой компании. Указанное замечание в особенности актуально относительно подбора лиц, имеющих доступ к любой информации или способных получить доступ в силу специфики выполняемых обязанностей. А это не кто иной, как различного рода администраторы, особенно администратор СУБД и системный администратор.
- *Контроль за персоналом*. Люди — это источник основного риска для предприятия, в особенности персонал, имеющий беспрепятственный доступ к данным, представляющим государственную или коммерческую тайну. Ни одна внешняя угроза не сможет принести столько ущерба, сколько внутренняя. Поэтому даже сам факт того, что потенциальный "крот" будет знать, что все его действия проверяются, может если не свести к нулю, то, по крайней мере, существенно снизить риск прорыва системы безопасности.
- *Защита служебных помещений и оборудования*. Грош цена всем стараниям всевозможных специалистов по безопасности, если можно просто войти в офис и изъять жесткий диск с данными из неохраняемого сервера компании. Поэтому во всех солидных организациях вводят пропускной режим, устанавливают системы сигнализации и видеонаблюдения. Одним словом, осуществляют контроль за физическим доступом к оборудованию и к носителям информации.
- *Планирование действий сотрудников в чрезвычайных ситуациях*. Никто не застрахован от чрезвычайных ситуаций природного или техногенного характера. Пожары, наводнения, землетрясения и другие катаклизмы могут причинить невосполнимый ущерб любому из нас. Спасти жизни персонала и минимизировать экономические последствия от катастрофы поможет наличие четких и внятных инструкций по поведению сотрудников в тех или иных ситуациях. Одна-единственная строка в инструкции по пожарной безопасности, в которой будет написано, что при эвакуации системный администратор (или другое лицо) должен изъять архивы на электронных носителях информации, поможет сократить издержки компании по восстановлению своих баз данных.

Наша книга посвящена разработке реляционных баз данных, а не вопросам подбора персонала или определению должностных обязанностей шефа безопасности компании. Поэтому читатель должен понимать, что здесь мы рассмотрели организационные меры защиты информационных систем весьма кратко.

Остановимся теперь на компьютерных мерах защиты. База данных функционирует в рамках информационной системы и подвергается тем же опасностям, что и вся вычислительная сеть, и компьютеры компании. Исходя из этого для защиты информационной системы предприятия нужно предпринять ряд мер, не всегда имеющих прямое отношение к БД:

- сетевая защита;
- защита от вирусов;
- настройка безопасности операционных систем;
- аутентификация и авторизация пользователя;
- криптографическая защита БД;
- резервное копирование данных;
- аудит событий безопасности БД;
- модернизация системного и прикладного ПО;
- доступ к данным только при посредничестве представлений и хранимых процедур.

Все перечисленные меры относятся к ведению системного администратора, причем последние шесть решаются во взаимодействии с администратором СУБД.

Аутентификация и авторизация

Одним из основных способов обеспечения конфиденциальности и целостности информации в БД выступают механизм аутентификации. *Аутентификация* (authentication) — это процедура проверки подлинности пользователя. Обычно пользователь подтверждает то, что он именно тот, за кого себя выдает, путем ввода в систему уникальной (неизвестную другим) информации о себе. В простейшем случае это символьный пароль, но возможен и более творческий подход подтверждения подлинности. Все чаще используется биометрическая аутентификация (дактилоскопия, узор радужной оболочки и сетчатки глаз, анализ голоса) или электронные способы аутентификации (контактные и бесконтактные смарт-карты, радиочастотные идентификаторы, USB-ключи). Часто программисты изобретают нестандартные способы аутентификации, например, приложение может попросить пользователя сообщить о своем домашнем адресе или расписаться в электронной форме, а затем сравнить полученные данные с имеющимися в системе образцами.

Если пользователь работает с базой данных в течение продолжительного периода времени, то желательно осуществлять периодическую аутентификацию, чтобы убедиться, что за компьютером находится санкционированный оператор, а не случайный прохожий. Кроме того, имеет смысл связывать наиболее критичные

действия оператора (допустим удаление наиболее важных данных или уничтожение таблицы) с повторной процедурой аутентификации.

Если пользователь успешно прошел процедуру аутентификации, СУБД осуществляет его авторизацию. *Авторизация* (authorization) — это процедура предоставления пользователю определенных ресурсов и прав на их использование.

ЗАМЕЧАНИЕ

В языке SQL предусмотрены специальные команды, предназначенные для определения прав пользователя: инструкции `GRANT` и `REVOKE`.

В современных СУБД используется два метода определения ограничений на доступ к данным: *избирательный* (discretionary) и *мандатный* (mandatory).

При избирательном доступе каждому пользователю, стремящемуся получить доступ к системе, предоставляются определенные полномочия по обращению с объектами БД. Диапазон полномочий достаточно широк от ограниченного доступа к отдельному атрибуту или строке таблицы до всеобъемлющих прав относительно всей базы данных. Набор прав назначает администратор СУБД. Обычно пользователь вводится в группу с заранее предопределенными ролями (администратор сервера, администратор БД, пользователь БД, гость). Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенной ролью. Пользователи могут обладать разными правами на один и тот же объект, одним разрешено лишь просматривать данные, другим — только добавлять, третьим — осуществлять чтение, вставку и запись.

Если система безопасности СУБД основана на мандатном доступе, то каждому из объектов БД назначается определенный уровень допуска, например "особая важность", "совершенно секретно", "секретно", "для служебного пользования", "несекретно". Доступ к объекту получит только пользователь с полномочиями не ниже чем уровень допуска к объекту.

Криптографическая защита

Криптографическая защита — одно из самых сильных средств противодействия несанкционированному просмотру данных. В основе подавляющего большинства криптографических систем защиты выступает шифрование. Шифрование — это процесс преобразования открытых данных по специальному алгоритму, после чего эти данные нельзя восстановить в исходном виде без ключа дешифрования.

Схематично процесс шифрования и дешифрования данных представлен на рис. 10.2. Исходные открытые данные *S* подвергают криптографическому преобразованию, для этого их обрабатывают шифрующей функцией *E()*, в качестве параметра которой выступает назначаемый пользователем ключ шифрования *k1*. В результате мы получаем зашифрованные данные *R*, которые для стороннего наблюдателя выглядят как последовательность случайных символов. Восстановление открытых данных *S* из зашифрованных *R* возможно только в том случае, если мы обладаем функцией дешифрования *D* и знаем ключ дешифрования *k2*.

Различают два вида криптосистем: на основе *симметричного* и *асимметричного* шифрования. В симметричной криптосистеме для шифрования и расшифровки данных применяется один и тот же ключ. В таких системах для исключения несанкционированного доступа к информации ключ следует держать в секрете. В асимметричной криптосистеме предусмотрены два ключа. Шифрование данных осуществляется несекретным ключом (для вскрытия данных он не подходит), а расшифровка проводится с помощью второго (секретного) ключа.

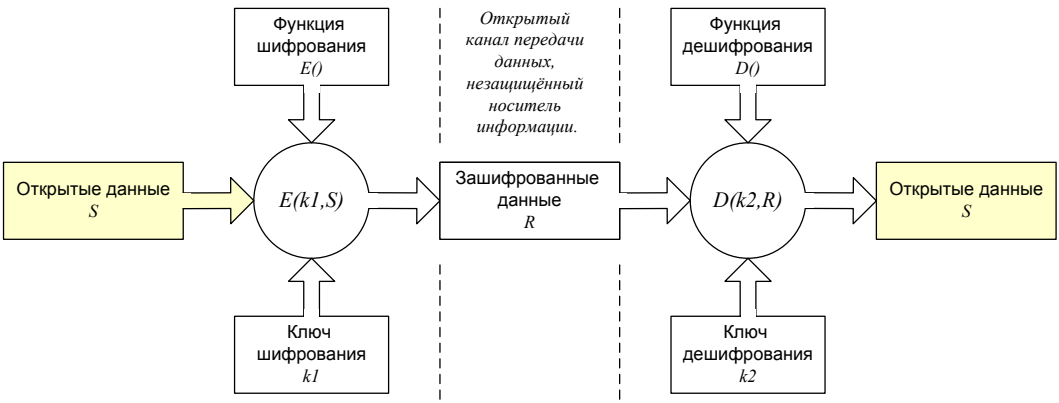


Рис. 10.2. Схема шифрования данных

На сегодня в мире широко распространены несколько стандартов шифрования. Наиболее старый из них, появившийся на свет в 1977 году симметричный алгоритм DES (Data Encryption Standard), он шифрует 64-битовые блоки данных с помощью 64-разрядного ключа (56 значащих и 8 проверочных разрядов). Таким образом, ключ шифра DES имеет 2^{56} комбинаций. Для взлома криптосистемы DES с помощью современного компьютера обычному хакеру потребуется не один десяток лет, если же системой заинтересуются профессионалы, обладающие соответствующими техническими средствами, то DES сможет защитить ваши данные всего за несколько минут [26].

Европейский стандарт симметричного криптоалгоритма называется IDEA (International Data Encryption Algorithm), он действует с 1990 года. Этот алгоритм весьма быстр и более стоек, чем DES. Длина ключа IDEA составляет 128 бит. В США с 2000 года в качестве стандарта принята симметричная криптосистема AES (Advanced Encryption Standard). Основным показателем криптостойкости AES выступает поддерживаемая длина ключа — 128, 192 и 256 бит. Как видите, алгоритм AES даже на основе самого короткого ключа превосходит 56-битовый алгоритм DES. В России в качестве криптографического преобразования используется симметричный алгоритм, определяемый ГОСТ 29147-89. Это достаточно стойкий алгоритм с длиной ключа 256 бит.

Асимметричные алгоритмы применяются реже, чем симметричные. В первую очередь, это объясняется их низкой скоростью и сложностью реализации. Наибо-

лее известный асимметричный алгоритм шифрования называется RSA (первые буквы фамилий авторов идеи Rivest, Shamir и Adleman). Стойкость алгоритма с открытым ключом сопоставима с DES при условии применения длинного ключа (1024 бита и выше).

Замечание

Защищенная СУБД должна уметь шифровать собственно хранящиеся в ней данные (включая служебную информацию), исходный код запросов, хранимых процедур и триггеров, данные, передаваемые к другим компьютерам по незащищенным каналам.

Помимо шифрования находящихся в таблицах данных некоторые СУБД позволяют разработчику шифровать и тексты программ, что исключает несанкционированную модификацию кода. Например, при проектировании хранимых процедур в Microsoft SQL Server ключ ENCRYPTION позволит скрыть исходный текст процедур для пользователей, не имеющих доступа к системным таблицам или файлам базы данных.

Резервное копирование

Для минимизации ущерба от вероятной гибели данных необходимо регулярно создавать копию БД. В такую копию входят не только собственно данные, но и служебная информация. Если вы эксплуатируете простейшею настольную СУБД, то для резервирования достаточно скопировать папку с файлами БД. В продвинутых системах предусмотрены специальные утилиты, позволяющие не просто копировать данные, но и определять график, порядок сохранения данных и создавать инкрементные копии.

Создание резервной копии — необходимое, но не достаточное условие для восстановления данных. Следует разобраться и с местом хранения копий. Самое распространенное (и одновременно самое нелепое) решение сводится к размещению резервных копий на том же самом жестком диске, на котором находится сама БД. Как следствие, при физическом повреждении диска, копии уходят в небытие на таких же правах, что и сама база данных. Поэтому архивы следует содержать на альтернативных носителях данных, например на записываемых CD- или DVD-дисках. Указанные диски желательно не складировать в ящике письменного стола, а направлять в ячейку несгораемого сейфа. В свою очередь, сейф следует располагать в другом помещении. Помещение должно быть защищенным (охраняемым, пожаробезопасным, не подверженным затоплению и т. п.). Напомню, что большим специалистом по выбору мест хранения бесценных вещей считался герой русских народных сказок Кощей Бессмертный. Свою жизнь он спрятал на конце иглы, иглу упаковал в яйцо, яйцо находилось в утке и т. д. Тем не менее, несчастный Кощей пал жертвой Ивана-царевича. В чем причина? Да только в том, что он просто не догадался сделать несколько резервных копий иглы...

Замечание

В качестве девиза для хранения резервных копий предлагаю использовать пословицу: "Дальше положишь, ближе возьмешь".

Аудит событий безопасности

Аудит событий безопасности БД представляет собой процесс получения и анализа данных о происходящих в системе событиях и степени их соответствия требованиям к защите данных. В результате аудита мы получаем оценку состояния защищенности базы данных, выявляем бреши в защите БД и вырабатываем рекомендации по совершенствованию действующих механизмов безопасности.

В идеале сбор информации о состоянии системы безопасности БД должен осуществляться непрерывно, для этого очень многие СУБД автоматически ведут журнал аудита. Журнал аудита обеспечивает индивидуальную ответственность пользователя за все его действия, т. к. все события аудита однозначно связываются с учетной записью пользователя в СУБД. В журнале содержится:

- описание стандартного набора событий (авторизации пользователя, доступа к тем или иным данным и операций с ними; создания, модификации и уничтожения объектов БД; выполнение нештатных SQL-команд и т. д.);
- настраиваемый перечень атрибутов в отдельной записи журнала аудита (дата и время события, идентификатор пользователя, имя и сетевой адрес компьютера, описание события, связанные с событием объекты, признак успешного или неудачного завершения события).

Для удобства анализа данных журнал обязан позволять фильтровать и сортировать свои записи. Нужно отметить, что журнал аудита сам по себе должен быть защищен от несанкционированного доступа.

Перечислим основные события аудита:

- предоставление прав доступа к БД;
- создание, изменение и удаление объектов БД;
- вставка, редактирование и удаление данных;
- изменение настроек сервера и прикладного ПО;
- любые отказы в обслуживании пользователя;
- попытки осуществить потенциально опасные операции без наличия соответствующих прав;
- различного рода исключительные ситуации и ошибки в работе ПО.

Собранная информация передается на анализ независимому эксперту в области защиты информации. Заметьте, что слово "независимый" здесь ключевое. Многие компании даже ни на йоту не сомневаются в лояльности своего "специалиста" и безоговорочно доверяют ему проведение аудита. Такой подход хорош только с точки зрения экономии денежных средств. Даже если местный специалист кристально чист (что к счастью еще часто встречается), он далеко не обязательно является высоким профессионалом в области защиты информации и не всегда скло-

нен замечать свои собственные ошибки. Поэтому работа независимого специалиста в любом случае принесет больше пользы, чем поверхностный аудит своих собственных сотрудников.

Замечание

Журнал аудита — очень важный, но далеко не единственный источник информации для эксперта. Полноценная аудиторская проверка обязательно проведет исследование всей информационной системы компании, изучит техническую и распорядительную документацию, побеседует с ключевыми сотрудниками, одним словом, получит много дополнительной информации об организации, которая позволит рассмотреть проблему безопасности в комплексе.

Модернизация системного и прикладного ПО

Подавляющее большинство производителей программного обеспечения регулярно вносят доработки (улучшают алгоритмы, исправляют выявленные ошибки, добавляют "заплатки" для противодействия недавно появившимся угрозам безопасности). Обычно изменения выкладываются на сервер производителя, а если вы находитесь на гарантийном обслуживании, то производитель лично извещает своего клиента о необходимости модернизировать ПО.

Администратору СУБД следует всегда находиться в курсе всех изменений и нововведений в эксплуатируемом им программном обеспечении и регулярно обновлять пакет своих программ. Если компания эксплуатирует эксклюзивное (разработанное специально для нее) ПО, то администратору СУБД стоит обязательно напомнить руководству компании о необходимости заключения (или продлении) договора с производителем на сопровождение ПО, развернутого на предприятии.

Внимание!

Перед любыми операциями, связанными с модернизацией программного обеспечения, следует создать резервную копию БД.

Доступ к данным только при посредничестве представлений и хранимых процедур

Рассмотренные способы защиты информации с помощью аутентификации, криптозащиты и резервного копирования в том или ином виде имеются во всех компьютерных системах. Наряду с ними СУБД обладает собственными механизмами самообороны, основной из них базируется на применении *представлений* (view).

Специалисты по безопасности БД рекомендуют разработчикам воздерживаться от предоставления доступа пользователям непосредственно к таблицам. Вместо этого общение с данными должно осуществляться только через специальных по-

средников — представления. Представление формируется при выполнении последовательности реляционных операций над одной или несколькими таблицами. В результате получается новое отношение, которое с точки зрения пользователя выглядит как вполне полноценная таблица, хотя на самом деле эта таблица виртуальная. Разработчик может исключить из нее какую-то часть атрибутов и записей, к которым пользователь не имеет прав доступа.

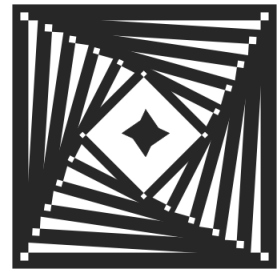
ЗАМЕЧАНИЕ

В языке SQL имеются ряд команд, предназначенных для работы с представлениями: `CREATE VIEW`, `ALTER VIEW` и `DROP VIEW`.

Для защиты от SQL-инъекций доступ к хранимым в сетевых БД данным должен осуществляться не через динамический SQL, а только с помощью *хранимых процедур* (stored procedure). Введя такое ограничение, мы резко снижаем вероятность прорыва наших оборонительных линий, т. к. код хранимой процедуры написан разработчиком БД и (конечно же, при условии достаточной квалификации программиста) не угрожает данным. Если без динамического SQL обойтись никак не возможно, то хорошей идеей станет проверка всех значений передаваемых в параметры запроса, поскольку нельзя исключить вероятность того, что злоумышленник попытается встроить сюда свой вредоносный код.

Резюме

Безопасность хранимой в БД информации определяется не только возможностями системы управления базами данных. Кроме того, на безопасность влияют многочисленные факторы: возможности ОС и установленного прикладного ПО, применяемые аппаратные и программные средства защиты, особенности решения организационных вопросов в компании, физические условия эксплуатации БД, степень подготовленности персонала, особенности законодательства страны и др. Раскрыть все перечисленные аспекты в одной главе невозможно, но эта цель и не преследовалась. Задача была другая. Во-первых, убедить читателя в необходимости учета вопросов безопасности, начиная с первых шагов работы над проектом БД. Во-вторых, доказать, что защищенная БД создается только за счет комплексного использования всех имеющихся в нашем распоряжении возможностей.



Глава 11

Знакомство с SQL

Вполне естественно, что для управления данными в реляционных БД потребовался принципиально новый язык. Среди множества пожеланий, предъявляемых в начале 1970-х годов к делающему первые шаги языку баз данных, мы выделим самые ключевые. Перспективный язык реляционных баз данных должен позволять:

- создавать базы данных и таблицы;
- выполнять основные операции редактирования данных в таблицах (вставку, модификацию и удаление);
- выполнять запросы, преобразующие хранящиеся в таблицах данные в необходимую информацию.

Ко всему прочему разрабатываемый язык должен был в принципе отличаться от высокоуровневых языков программирования тех лет. Во-первых, базы данных работают в трехзначной логике. У них наряду с классическими для любого языка понятиями истина/ложь (`FALSE`/`TRUE`) предусмотрено третье значение неопределенности `UNKNOWN`. Появление третьего компонента, с одной стороны, положительно, т. к. расширяет функциональные возможности языка, но с другой стороны, усложняет язык. Во-вторых, новый язык создавался не только для программистов, но и в интересах пользователей, поэтому в идеале он должен быть не процедурным, а декларативным.[1] В соответствии с этим пользователь лишь ставит БД задачу (указывает, что ему нужно от БД), а каким образом БД станет решать поставленную задачу, пользователя не интересует.

Впервые о языке SQL заговорили в 1974 году благодаря ученому Д. Чамберлине (кстати, работавшему вместе с Коддом в лаборатории IBM в городе Сан-Хосе), правда, первоначально язык назывался несколько иначе — SEQUEL (Structured English Query Language), поэтому от программистов, стоявших у истоков SQL, и сейчас можно услышать сокращение "СИКвЕЛ" вместо "ЭсКюЭль". Переименование SEQUEL в SQL осуществилось по юридическим соображениям, каким именно история умалчивает. Предложенная модель языка стала настолько удачной, что практически сразу была взята на вооружение не только владельцем лаборатории доктора Кодда корпорацией IBM (знаменитый прототип СУБД "System R", разработанный в 1976 году), но и другими компаниями. К самой первой успешной ком-

[1] Современный SQL сочетает черты как декларативного, так и процедурного языка.

мерческой реализации реляционной СУБД, базирующейся на языке запросов SQL, относят СУБД "ORACLE" (конец 70-х годов) и "SQL/DS" фирмы IBM (1981 г.). Однако стандартом SQL стал позднее — в 1986 году, благодаря Американскому национальному институту стандартов (American National Standards Institute, ANSI) и международной организации стандартизации (International Organization for Standardization, ISO). Кстати, узаконенный SQL иногда называют по имени принявшей его организации — ANSI SQL. Следующий шаг был сделан в 1989 году, с появлением SQL/811. В дополнение к уже принятому стандарту были определены функции поддержки целостности данных. В 1990-х годах официально действующим и общепризнанным считается стандарт SQL/92, принятый, как вы уже догадались, в 1992 году. Практически любая серьезная компания, разрабатывающая СУБД, старается поддерживать требования SQL/92. К перечню наиболее известных клиент-серверных СУБД можно отнести Oracle, INGRES, Informix, Sybase, SQLbase, Microsoft SQL Server, DB2 и InterBase. Развитие структурированного языка запросов ни в коем случае не остановилось в 1992 году. В 1995 году стандарт был дополнен новой спецификацией CLI-95 (Call-Level Interface — интерфейс уровня вызовов). Это CLI не что иное, как элемент разработанного корпорацией Microsoft интерфейса открытого доступа к базам данных ODBC (Open DataBase Connectivity) и JDBC (Java DataBase Connectivity). В 1996 приняли новое дополнение к уже существующему стандарту SQL-92, оно в первую очередь коснулось поддержки хранимых процедур (Persistent Stored Modules). После этого в печати стала применяться еще одна аббревиатура — SQL:92/PSM. В 1999 году заговорили об очередном (третьем по счету) стандарте SQL и было опубликовано пять частей стандарта SQL-3 (SQL:99):

- SQL/Framework — концептуальная структура стандарта;
- SQL/Foundation — базисное описание SQL;
- SQL/CLI — уточнения к интерфейсу уровня вызовов;
- SQL/PCM — уточнение описания хранимых процедур;
- SQL/Bindings — определение правил взаимодействия SQL и ряда стандартных языков программирования (в т. ч. Pascal).

Спустя некоторое время появилось еще три части стандарта.

- SQL/MED — управление внешними данными;
- SQL/OLB — правила взаимодействия с объектно-ориентированными языками;
- SQL/Schemata — информационная схема.

Однако многие специалисты вновь скептически отнеслись SQL-3, обвинив его в незавершенности. Поэтому в 2003 году к вопросу модернизации SQL вернулись вновь. В обновленный стандарт с необходимыми изменениями вошли все части прежнего SQL:99 (правда, часть SQL/Bindings в самостоятельном виде существовать перестала и была включена во вторую часть стандарта SQL/Foundation). В дополнение к перечисленным частям SQL:2003 приобрел еще несколько документов:

- SQL/JRT — взаимодействие с языком Java;
- SQL/XML — работа с XML-документами.

Совершенствование SQL отчасти подтверждает один из неписаных законов программирования — лучшее враг хорошего. С каждым очередным этапом разви-

тия стандарта все меньше и меньше производителей программного обеспечения могут его поддерживать в полном объеме. У Криса Дейта на этот счет есть хорошее высказывание: "… в наши дни ни один программный продукт не поддерживает полностью даже SQL:92; вместо этого такие продукты, как правило, поддерживают то, что можно было бы назвать "надмножеством подмножества" стандарта…" [7]. Как следствие стандарт не поспевает за производителями, а это неминуемо ведет к появлению различных ветвей языка, что с каждым годом все более и более уменьшает вероятность появления новой, общепринятой редакции SQL, однозначно поддерживаемой всеми разработчиками ПО.

ЗАМЕЧАНИЕ

Полную спецификацию стандарта SQL вы можете найти в Интернете по адресу **http://www.wiscorp.com/SQLStandards.html**.

Назначение SQL

Прежде чем мы приступим к рассмотрению синтаксических конструкций языка, отметим, что SQL хотя и весьма мощный, но далеко не всемогущий язык. И если вы начинающий разработчик, то стоит сразу уяснить спектр возможностей нашего нового знакомого.

В сферу интересов SQL не попали задачи, стоящие перед прикладным и, тем более, системным программистом (операции ввода-вывода, пользовательский интерфейс, работа с периферийными устройствами и т. п.). Одним словом, на SQL не напишешь ни одного даже самого элементарного приложения для Windows или для любой другой ОС.

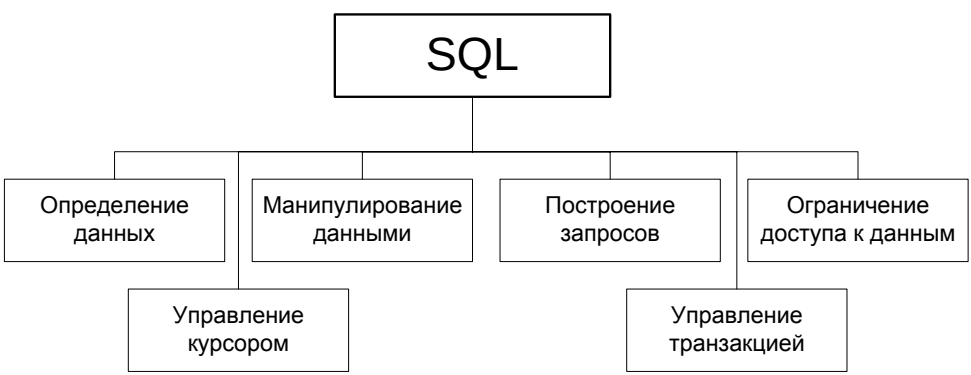


Рис. 11.1. Основные задачи языка SQL

Язык SQL выступает неотъемлемой частью СУБД и применяется только в интересах обработки данных (рис. 11.1):

- *Определение данных* — реализуется средствами подъязыка определения данными (Data Definition Language, DDL). Язык нацелен на создание и удаление базы данных и ее объектов. Перечень объектов БД достаточно велик (таблицы, представления, индексы, курсоры, определения доменов). Визитной карточкой DDL выступают операторы `CREATE`, `ALTER` и `DROP`.
- *Манипулирование данными* — подъязык манипулирования данными (Data Manipulation Language, DML) обеспечивает вставку, редактирование и удаление данных из таблиц БД. В распоряжении DML имеется три команды: `INSERT`, `UPDATE` и `DELETE`.
- *Построение запросов* — подъязык запросов данных (Data Query Language, DQL) позволяет извлекать данные из одной или нескольких таблиц. Подъязык DQL построен всего на одной команде, начинающейся с ключевого слова `SELECT`.
- *Ограничение доступа к данным* — определяет ограничения на права пользователей при работе с объектами БД. В основу подъязыка положены две команды: `GRANT` и `REVOKE`.
- *Управление курсором* — подъязык позволяет обрабатывать данные построчно. Он опирается на квартет команд: `DECLARE CURSOR`, `OPEN CURSOR`, `FETCH CURSOR`, `CLOSE CURSOR`.
- *Управление транзакцией* — включает инструкции `SET TRANSACTION`, `BEGIN TRANSACTION`, `COMMIT` и `ROLLBACK`. Язык позволяет определять уровень изоляции транзакции, стартовать, фиксировать или возвращать транзакцию в исходное состояние.

Далее мы узнаем, каким образом с помощью SQL решаются все шесть перечисленных задач. А сейчас мы познакомимся со стандартизированными типами данных.

Типы данных SQL

Изучение любого языка программирования начинается с описания базовых структур данных, предоставляемых в распоряжение разработчика БД. Мы не станем отклоняться от этого правила и начнем знакомство с SQL с рассмотрения поддерживаемых им типов данных.

На рис. 11.2 представлен перечень стандартных типов данных SQL. В распоряжении СУБД, поддерживающей SQL:92, имеется богатый набор предопределенных типов данных:

- точные числовые типы;
- приближенные числовые типы;
- типы для работы с датой и временем;
- временные интервалы;

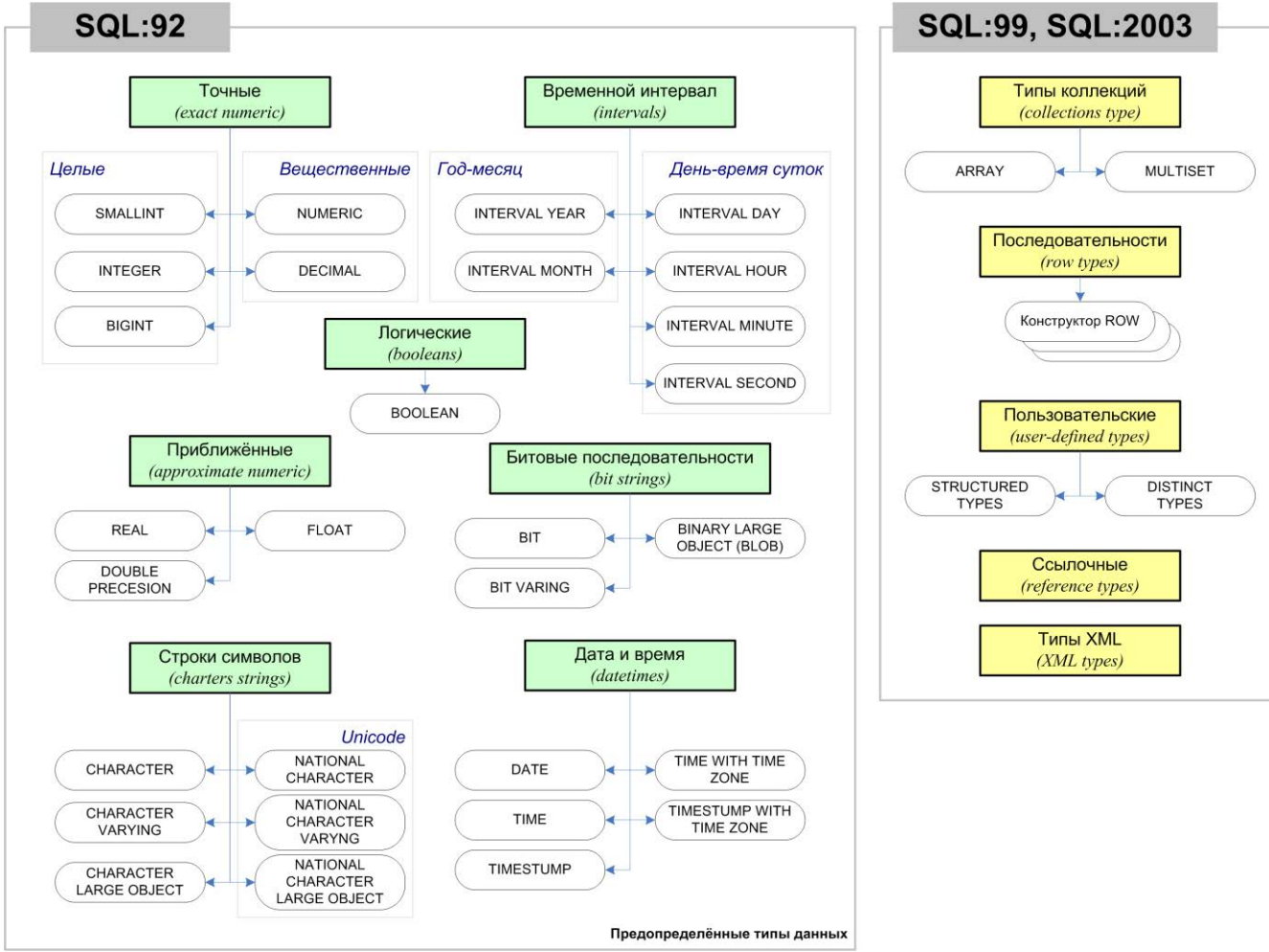


Рис. 11.2. Типы данных SQL

- логические типы данных;
- строки символов;
- битовые строки.

Если СУБД совместима с SQL:99, то к перечню добавляется еще несколько типов. Появившихся в 1999 году новичков нельзя отнести к разряду заранее предопределенных, в стандарте SQL:2003 их называют непредопределенными (non-predefined) и даже более жестко — типами данных, не относящимися к SQL (non-SQL types). С некоторой степенью допущения их можно причислить к структурным типам данных:

- коллекции;
- последовательности;
- типы данных, определяемые пользователем;
- ссылочные типы.

Где используются перечисленные типы данных? Во-первых, при определении столбцов таблиц. Во-вторых, при объявлении переменных в хранимых процедурах, функциях определяемых пользователем и триггерах. В-третьих, при организации обмена данными между БД и клиентским приложением.

Предопределенные типы данных

Точные числовые типы (exact numeric) предназначены для обслуживания целочисленных значений и значений, имеющих дробную часть, без потерь точности. При задании такого типа данных необходимо указать два аргумента: точность (`n`) и масштаб (`m`). Точность задает общее число значащих цифр, используемых при отображении числа. Масштаб определяет число значащих цифр справа от десятичной точки. Обязательно должно соблюдаться условие: точность больше масштаба (`n>m`). Масштаб — необязательный аргумент, если его не указывать, то он считается равным нулю. Например, тип данных `NUMERIC(5,2)` определяет число, состоящее не более чем из пяти цифр, включая две цифры после запятой.

В табл. 11.1 приведены основные типы точных чисел. Единственным дополнением к набору точных числовых данных, существовавших с первых версий стандарта SQL, стал введенный в 1999 году тип данных `BIGINT`.

Таблица 11.1. *Точные числовые типы*

Спецификация	Описание
`NUMERIC [(n[,m])]`	Точное число, описываемое аргументами `n` и `m`
`DECIMAL [(n[,m])]` или `DEC [(n[,m])]`	В отличие от `NUMERIC` способно хранить число с большей точностью, чем определено в аргументе `m`. Поэтому говорят, что `NUMERIC` задает реальное значение точности, а `DECIMAL` — минимальное значение точности

***Таблица 11.1** (окончание)*

Спецификация	Описание
BIGINT	Тип данных, предназначен для хранения больших целых чисел (обычно 64 бита). Типы данных BIGINT, INTEGER и SMALLINT являются частным случаем типа данных NUMERIC, у которого масштаб установлен в 0, а точность определена возможностями СУБД
INTEGER или INT	Тип данных, предназначен для хранения целых чисел (обычно 32 бита)
SMALLINT	Тип данных предназначен для хранения малых целых чисел (обычно 16 битов)

Количество байтов, необходимых для хранения значений NUMERIC и DECIMAL, напрямую зависит от размерности аргумента n. Например, если вы намерены хранить число из 15–16 сохраняемых знаков, то вам потребуется 8 байтов, 34–36 знаков — 16 байтов.

Замечание

Практически во всех СУБД для работы с денежными величинами на базе NUMERIC реализован свой собственный специализированный тип данных. Такие типы данных хранят действительные числа с точностью до 4-го знака после запятой.

Приближенные числовые типы (approximate numeric) представляют собой тип данных для определения чисел с плавающей точкой, осуществляющий хранение числа в научном формате (мантисса плюс порядок). Аргумент n имеет точность, но в отличие от типа NUMERIC не обладает масштабом. Термин "приближенный" вовсе не означает, что внеся в столбец таблицы число, допустим, 4 целых 5 десятых, то на следующий день вы на этом месте найдете 5. Просто тип данных предназначен для обслуживания значений, не требующих высокой точности (табл. 11.2).

***Таблица 11.2.** Приближенные числовые типы данных*

Спецификация	Описание
REAL	Точность и предел значений зависит от СУБД. Как правило, занимает в памяти 6 байтов и в состоянии хранить число в интервале от $1{,}5 \cdot 10^{-45}$ до $3{,}4 \cdot 10^{38}$ с точностью до 7 цифр после запятой
FLOAT [(n)]	Аргумент n определяет минимальное значение точности. Если значение n не указано, то потребует 8 байтов памяти. Тип данных способен хранить число в интервале от $5{,}0 \cdot 10^{-324}$ до $1{,}7 \cdot 10^{308}$ с точностью до 15 знаков
DOUBLE PRECISION	Точность определяется версией СУБД, превышает точность REAL

Логический тип данных (boolean type) языка SQL весьма неординарен. Особенность заключается в том, что два классических элемента (`True`/`False`) булевой логики здесь дополнены третьим значением — неопределенностью `Unknown`. Как следствие логика становится более сложной — трехзначной. В табл. 11.3 приведены результаты основных операций логического И (`AND`) и ИЛИ (`OR`).

Таблица 11.3. *Таблица основных логических операций*

Операция AND	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Операция OR	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Есть особенность и у операции отрицания `NOT`. Если оператор `NOT`, примененный к истине, возвращает ложь — `NOT (TRUE) IS FALSE`, а ко лжи наоборот — `NOT (FALSE) IS TRUE`, то операция отрицания неизвестности вернет неизвестность `NOT (UNKNOWN) IS UNKNOWN`.

Тип данных *строки символов* (characters strings) специализируется на обслуживании текстовых данных (табл. 11.4). При работе с текстовыми данным (объявлении строковой переменной, описании столбца таблицы, параметра хранимой процедуры и т. п.) следует указывать размерность строки `n`.

Таблица 11.4. *Строки символов*

Спецификация	**Описание**

`CHARACTER [n]`
или сокращенно
`CHAR[n]`
	Предназначен для создания текстовой строки фиксированной длины. Число символов в строке определяется в квадратных скобках после указания типа данных. Если в поле типа `CHAR` помещается текстовое значение меньшего размера, чем размерность поля, то оставшиеся позиции символов заполнятся пробелами

`CHARACTER VARYNG[n]`
или сокращенно
`VARCHAR[n]`
	Текстовая строка переменной длины. Максимальный размер строки определяется в квадратных скобках. Преимущество такого типа данных над типом `CHAR` заключается в том, что здесь пустые позиции не заполняются пробелами и соответственно таблица требует меньшего размера оперативной и дисковой памяти

Таблица 11.4 *(окончание)*

Спецификация	Описание
NATIONAL CHARACTER [n] или сокращенно NCHAR [n]	Строка национального символьного набора фиксированной длины
NATIONAL CHARACTER VARYING [n] или сокращенно NCHAR VARYING [n]	Строка национального символьного набора переменной длины
CHARACTER LARGE OBJECT [n] или сокращенно CLOB[n]	Предназначен для определения столбцов таблиц, хранящих большие группы символов
NATIONAL CHARACTER LARGE OBJECT [n]	Предназначен для определения столбцов таблиц, хранящих большие группы национальных символов

Данные типа CHAR и NCHAR в операторах SQL должны выделяться одинарными кавычками, кроме того, при использовании символов национальных алфавитов следует указать спецификацию набора символов, воспользовавшись командой CHARACTER SET:

```
STR = 'Символьные переменные берутся в кавычки'
```

Для данных символьного типа допускается операция сложения (конкатенация):

```
STR = 'Сложение ' + 'строк'
```

Битовая последовательность (bit strings) предназначена для хранения любой двоичной информации (табл. 11.5). Тип данных универсален и позволяет описывать как простейшие логические данные, так и сложные объекты, например файлы мультимедиа.

Таблица 11.5. *Битовые последовательности*

Спецификация	Описание
BIT [n]	Битовая последовательность фиксированной длины. Аргумент n устанавливает длину последовательности в битах. Если аргумент отсутствует (или установлен в 1), то тип данных используется для создания полей логического типа (Да/Нет). Особенность типа данных фиксированной длины в том, что попытка записать в поле этого типа значения меньшей длины, чем указано в аргументе n, приведет к ошибке
BIT VARYNG [n]	Битовая последовательность переменной длины. Максимальное значение битовой последовательности указывается в аргументе n

Таблица 11.5 *(окончание)*

Спецификация	Описание

`BINARY LARGE OBJECT [n]`
или сокращенно
`BLOB [n]`
	Предназначен для хранения больших объектов, например, файлов мультимедиа и изображений

За описание значений *даты и времени* (datetimes) в SQL:2003 отвечает пять типов данных (табл. 11.6). Дата представляется в формате общепринятого в большинстве стран мира Григорианского календаря.

Таблица 11.6. *Тип данных дата и время*

Спецификация	Описание
`DATE`	Тип данных включает три поля: `YEAR` (год) — от 0001 до 9999 `MONTH` (месяц) — от 01 до 12 `DAY` (день) — от 01 до 31 Формат записи: `yyyy-mm-dd`. Полное число позиций, требуемых для отображения даты (вместе с разделителями) равно 11
`TIME [(n)]`	Включает три поля: `HOUR` (часы), `MINUTE` (минуты), `SECOND` (секунды). Если аргумент точности (`n`) не определен, то полное число позиций (вместе с разделителями) равно 8, а формат записи: `hh:mm:ss`. Если вы укажите аргумент точности, то получите возможность работать с долями секунд
`TIMESTAMP [(n)]`	Метка даты-времени, представляющая собой комбинацию типов данных `DATE` и `TIME`. Число позиций — 111
`TIME WITH TIME ZONE`	Тип данных аналогичен `TIME` с двумя дополнительными значениями, характеризующими смещение от Гринвичского меридиана в часах `TIMEZONE_HOUR` и минутах `TIMEZONE_MINUTE`. Из-за учета временной зоны число позиций увеличивается с 8 до 14
`TIMESTAMP WITH TIME ZONE`	Метка даты-времени плюс смещение от Гринвича. Число позиций для отображения даты, времени и временной зоны максимальное — не менее 25

Значения даты и времени допускается задавать литералами в одинарных кавычках, перед значением следует указать название типа, например:

```
TIMESTUMP '2011-11-21 12:00:00'
```

Интервал (interval) представляет собой производную от типа данных "дата-время" и предназначен для описания промежутка времени между двумя временными отсчетами. Стандарт SQL различает две категории интервалов: *год-месяц* (year-

month) и *день-время* (day-time). Как можно догадаться по названию, первая разновидность интервалов оперирует сравнительно большими значениями год YEAR и месяц MONTH. Вторая категория интервалов обладает меньшим диапазоном, но в качестве компенсации может похвастаться точностью до долей секунды. В диапазон интервала "день-время" входят такие значения, как день DAY, час HOUR, минута MINUTE и секунда SECOND. Синтаксических конструкций для определения интервала несколько, в самом общем виде достаточно рассмотреть следующий вариант:

```
INTERVAL start (p) [TO end (q)]
```

Здесь start и end — YEAR, MONTH, HOUR, MINUTE и SECOND. Явного определения параметров p и q обычно не требуется, в этом случае в них передается значение по умолчанию — 2. Двойка является минимальным значением для p, а верхняя ограничивающая планка зависит от конкретной реализации СУБД. Например, мы планируем задать столбец таблицы, предназначенный для хранения временного интервала до 10 000 лет, тогда в команде SQL появится следующая строка:

```
INTERVAL YEAR (4)
```

Цифра 4 укажет SQL, что в столбце таблицы могут храниться значения временного интервала до четырех значащих цифр (от 0 до 99 911).

Параметр q определяет точность до долей секунды и используется только в тех случаях, когда точность интервала задается в секундах. По умолчанию q=2, это означает, что мы учитываем только две значащих цифры перед запятой. Если мы намерены хранить значение времени с максимальной точностью (до четырех знаков после запятой), присвоим q значение 6:

```
INTERVAL MINUTE TO SECOND(6)
```

Для определения конкретного значения в формате типа данных "интервал" нужно воспользоваться следующим синтаксисом:

```
INTERVAL '12:54' HOUR TO MINUTE
```

Как вы догадались — это интервал, равный 12 часам 54 минутам.

С интервалами можно проводить операции сложения и вычитания, кроме того интервал можно умножить или разделить на вещественное число. В операциях с интервалами могут использоваться типы данных "дата-время", например:

```
DATE '2010-01-01' + INTERVAL '0005-1'
```

Операция прибавляет к дате интервал в пять лет и один месяц. В результате сложения мы получим новую дату: DATE '2015-02-01'.

Непредопределенные типы данных

Большинство непредопределенных (non-predefined) типов данных вошли в состав SQL сравнительно недавно. В SQL:1999 появилась спецификация коллекции (collection type) и массива (array), а в SQL:2003 к стандарту добавилось мультимножество (multiset). Кроме того, стандарт признал право на существование таких типов как последовательности, пользовательский тип, тип данных XML и ссылочный тип (см. рис. 11.2).

Основная особенность непредопределенных типов в том, что, даже в действующем стандарте SQL:2003, их называют типами данных, не соответствующими SQL (non-predefined and non-SQL types). Несоответствий много, но первое, что бросается в глаза — нарушение требования атомарности данных (хранение единственного неделимого значения в одной ячейке таблицы). И действительно, массив, мультимножество и последовательность нельзя рассматривать как атомарный тип, и если такой тип данных определяет колонку таблицы то, мы сразу сталкиваемся с проблемами нормализации данных.

Несмотря на указанное противоречие, стандарт есть стандарт, поэтому уделим время рассмотрению основных непредопределенных типов данных.

Массив

Синтаксическая конструкция по определению массива выглядит следующим образом:

```
тип_данных ARRAY [n]
```

В качестве типа данных может выступать любой допустимый в стандарте тип данных, параметр `n` описывает число элементов в массиве.

Пример:

```
INT ARRAY[10]
```

Как видите, стандартный SQL предусматривает лишь задание одномерного массива, определение массивов большей размерности не предусмотрено.

Внимание!

Программистам, привыкшим работать с Delphi, стоит запомнить, что отсчет элементов в массиве SQL всегда начинается с единицы, а не с нуля.

Мультимножество

Для задания мультимножества следует указать тип данных, подлежащих хранению:

```
тип_данных MULTISET
```

Последовательность строится средствами конструктора типа `ROW` и представляет собой пары `<имя_поля> <тип_данных>`. В сконструированную последовательность может входить несколько пар. Для демонстрации возможностей последовательностей воспользуемся следующим примером, допустим, что в таблице `DEMOTABLE` мы собираемся хранить фамилию, имя и отчество человека в одном поле `FIO`. В таком случае последовательность `ROW` позволит нам разделить поле на три части (листинг 11.1).

Листинг 11.1. Применение инструкции `ROW` при создании таблицы

```
CREATE TABLE DEMOTABLE
   (DEMOTABLE_KEY INTEGER PRIMARY KEY,
```

```
    FIO ROW (SNAME VARCHAR(20),
             FNAME VARCHAR(15),
             LNAME VARCHAR(15)),
    BDAY DATETIME)
```

Пользовательский тип

Заранее предопределенные типы данных далеко не всесильны, и зачастую не в состоянии охватить все потребности проектируемой БД. В подобных случаях стандарт предусматривает возможность проектирования пользовательских типов данных (user-defined types), описание которых должно сохраняться в системном каталоге. При создании нового типа данных следует опираться на уже существующие типы.

Объявленная стандартом SQL:2003 синтаксическая конструкция определения пользовательского типа весьма громоздкая, мы остановимся на несколько сокращенном варианте:

```
CREATE TYPE имя типа
[UNDER имя супертипа]
AS тип данных
[DEFAULT] значение
[[NOT] FINALL]
```

В простейшем случае, для создания пользовательского типа "короткая строка" можно воспользоваться следующей конструкцией:

```
CREATE TYPE SHORT_STRING_TYPE AS CHAR(10)
```

Стандарт не ограничивает сложности пользовательского типа данных, что теоретически позволяет нам объявлять достаточно замысловатые структуры (листинг 11.2).

Листинг 11.2. Создание типа данных для хранения адреса

```
CREATE TYPE ADDRESS_TYPE AS
 (ZIPCODE CHAR(6),
  CITYNAME VARCHAR(20) NOT NULL,
  STREET VARCHAR(20) NOT NULL,
  HOME VARCHR(3) NOT NULL)
NOT FINALL
```

Необязательное ключевое слово FINALL указывает, что пользовательский тип не может иметь подтипов, соответственно NOT FINALL предполагает, что мы можем создавать дочерние типы данных. В листинге 11.2 был приведен пример родительского типа данных, специализирующегося на хранении адреса (почтовый индекс, город, улица и дом). А теперь подумаем, как добавить к нему номер телефона (листинг 11.3).

Листинг 11.3. Создание подтипа данных

```
CREATE TYPE ADDRESSEX_TYPE
UNDER ADDRESS_TYPE
AS
(PHONENUM CHAR(11))
FINALL
```

Пример демонстрирует, что дочерний подтип данных в состоянии не только унаследовать родовые характеристики родительского типа данных, но и дополнить их своими.

Вполне естественно, что возможности пользовательских типов данных определяются особенностями диалекта SQL. В некоторых СУБД пользовательские типы пока вообще не поддерживаются, в других отличаются от стандарта, в третьих сильно упрощены.

Другие типы

Ссылочный тип (reference types, REF) может применяться при определении переменных и параметров. Значение ссылки `REF` указывает на строку в типизированной таблице (таблице описанной на основе какого-то структурированного типа данных).

Тип данных XML сам по себе заслуживает нескольких глав книги, поэтому мы пока ограничимся тем, что XML (Extensible Markup Language) представляет собой язык наращиваемой разметки, позволяющий описывать структурированные данные. У XML есть два качества, благодаря которым приобрел популярность у разработчиков. Во-первых, он поддерживается большинством платформ. Во-вторых, указанный язык позволяет разрабатывать Web-приложения.

Подводя итоги, следует отметить, что в рамках стандарта появились типы данных, противоречащие ряду фундаментальных требований к реляционной модели. С одной стороны, это указывает на стремление совершенствовать концепцию реляционных баз данных (что само по себе неплохо), а с другой стороны, закладывает мину замедленного действия под фундамент всей модели.

ЗАМЕЧАНИЕ

Практически в каждой из СУБД есть специфичные для нее типы данных, не имеющие аналогов в стандарте. В качестве характерного примера можно привести диалект PostgreSQL, в котором реализованы экзотические типы, предназначенные для хранения пространственных и геометрических данных (`BOX`, `CIRCLE`, `LINE` и т. д.).

Определение констант

Для числовых типов данных определены константы в виде последовательности цифровых символов с необязательным заданием знака числа и десятичной точкой:

```
- 1000.5
```

Для определения строковой константы следует воспользоваться одинарными кавычками:

```
'Иван Иванович Иванов'
```

При назначении даты и времени лучше всего руководствоваться требованиями ISO. Дата описывается в формате `yyyy-mm-dd`, время `hh.mm.ss`.

Преобразование данных

Существование многочисленных типов данных подразумевает возможность взаимного преобразования значений из одного формата в другой. В основу этой операции положено выражение `CAST`:

```
CAST <исходные данные> AS <целевой тип данных>
```

Таблица 11.7. *Допустимые преобразования типов данных*

Исходный тип данных	Целевой тип данных															
	EN	AN	VC	C	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
Exact Numeric	+	+	+	+	–	–	–	?	?	–	?	+	–	?	–	–
Approximate Numeric	+	+	+	+	–	–	–	–	–	–	?	+	–	?	–	–
Character	+	+	+	+	+	+	+	+	+	+	?	+		?		
Date	–	–	+	+	+		+	–	–	–	?	+	–	?	–	–
Time	–	–	+	+		+	+	–	–	–	?	+	–	?	–	–
Timestamp	–	–	+	+	+	+	+	–	–	–	?	+	–	?	–	–
Year-Month Interval	?	–	+	+	–	–	–	+	–	–	?	+	–	?	–	–
Day-Time Interval	?	–	+	+	–	–	–	–	+	–	?	+	–	?	–	–
Boolean	–	–	+	+	–	–	–			+	?	+	–	?	–	–
User Defined Type	?	?	?	?	?	?	?	?	?	?	?	?	?	?	–	–
Binary Large Object	–	–	–	–	–	–	–	–	–	–	?	–	+	?	–	–

Таблица 11.7 *(окончание)*

Исходный тип данных	Целевой тип данных															
	EN	AN	VC	C	D	T	TS	YM	DT	BO	UDT	CL	BL	RT	CT	RW
Reference type	?	?	?	?	?	?	?	?	?	?	?	?	?	?	–	–
Collection type	–	–	–	–	–	–	–	–	–	–	–	–	–	–	?	–
Row types	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Примечание

Для обозначения типов данных в табл. 11.7 использованы следующие аббревиатуры: EN = Exact Numeric, AN = Approximate Numeric, C = Character (Fixed- или Variable-length, или Character Large Object), VC = строка Character переменной длины, CL = Character Large Object, D = Date, T = Time, TS = Timestamp, YM = Year-Month Interval, DT = Day-Time Interval, BO = Boolean, UDT = пользовательский тип, BL = Binary Large Object, RT = ссылочный тип, CT = коллекция, RW = последовательность.

Само собой разумеется, что далеко не все типы данных допускают взаимную конвертацию своих значений. Например, метку времени `TIMESTUMP` не представить в виде интервала `INTERVAL`, а такой экзотический тип данных, как последовательность `ROW`, не всегда можно преобразовать даже в другую последовательность. Поэтому в описании стандарта SQL:2003 определен перечень допустимых преобразований типов данных (табл. 11.7). Символом "+" в таблице обозначен тот случай, когда преобразование реально и в результате его осуществления мы не теряем точности данных, если преобразование возможно при стечении определенных условий, то оно отмечено символом "?". Символ "–" свидетельствует о невозможности корректной трансформации.

Операторы

В SQL применяются стандартные для любого языка программирования математические операторы сложения (`+`), вычитания (`–`), умножения (`*`) и деления (`/`). Порядок операций точно такой, к какому мы привыкли на уроках арифметики в школе — наивысшим приоритетом пользуются операции умножения и деления, операции в скобках выполняются в первую очередь. Перечисленные операторы применимы к типам данных, оперирующих числами (exact и approximate numeric). Кроме того, математические операторы могут использоваться при работе с такими типами данных, как дата, время и интервал (табл. 11.8).

Таблица 11.8. *Операторы, применимые с датой, временем и интервалами*

Операнд	Оператор	Операнд	Результат
Datetime	-	Datetime	Interval
Datetime	+ или -	Interval	Datetime
Interval	+	Datetime	Datetime
Interval	+ или -	Interval	Interval
Interval	* или /	Numeric	Interval
Numeric	*	Interval	Interval

Для присвоения значения применяют оператор присваивания (=).

Вслед за математическими операторами и оператором присваивания следуют операторы сравнения: больше (>), меньше (<), равно (=), не равно (<>), больше и равно (>=), меньше и равно (<=).

Если все ранее рассмотренные операторы нам уже встречались в обычных языках программирования, то наши очередные "герои" — логические операторы значительно усовершенствованы. Начнем с классических операторов: логическое "И" — AND, логическое "ИЛИ" — OR и логическое отрицание — NOT. Кроме того, перечислю ряд логических операторов, пояснение действий которых мы получим в следующих главах книги: ALL, ANY, BETWEEN, EXISTS, IN, LIKE и SOME.

Встроенные функции

Классический SQL имеет сравнительно небольшой набор встроенных функций (табл. 11.9).

Таблица 11.9. *Основные функции SQL:2003*

Функция	Описание
BIT_LENGTH(битовая строка)	Позволяет выяснить длину строки в битах
CAST(значение AS тип данных)	Преобразует исходное значение к новому типу данных. Допустимые варианты конвертации типов данных приведены в табл. 11.7
CHAR_LENGTH(символьная строка)	Возвращает длину строки в символах
CURRENT_DATE	Возвращает текущую дату
CURRENT_TIME(точность)	Возвращает текущее время с определенной точностью

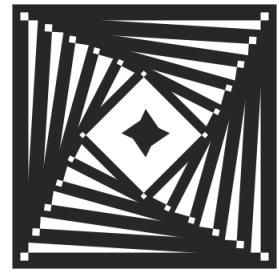
Таблица 11.9 *(окончание)*

Функция	Описание
`CURRENT_TIMESTAMP(точность)`	Информирует о дате и времени с указанной точностью
`LOWER(строка)`	Преобразует текстовую строку к верхнему регистру
`POSITION(первая строка IN вторая строка)`	Позволяет выяснить позицию, с которой начинается вхождение первой строки во вторую
`SUBSTRING(строка FROM n FOR длина)`	Возвращает часть строки, начиная с n-го символа с указанной длиной
`TRANSLATE(строка USING функция)`	Преобразует строку с использованием указанной функции
`TRIM(LEADING | TRAILING | BOTH символ FROM строка)`	Удаляет из строки все первые (`LEADING`), последние (`TRAILING`) или первые и последние (`BOTH`) символы. Пример: `TRIM(LEADING '#' BOTH ' ###Москва')` удалит все символы `'#'` из строки `'###Москва'`
`UPPER(строка)`	Преобразует все символы строки к верхнему регистру

Резюме

Структурированный язык запросов SQL появился на свет вместе с реляционными базами данных в конце 1970-х годов. Это один из немногих языков, которому за очень короткий промежуток времени удалось добиться высокого статуса стандарта. На сегодняшний день действует стандарт языка реляционных баз SQL:2003, но, к сожалению, ни одна из имеющихся на рынке коммерческих СУБД не в состоянии похвастаться абсолютным соответствием не только SQL:2003, но и его предшественникам. Большинство компаний, работающих на рынке программного обеспечения, уже дано идут своим путем, развивая свои собственные диалекты SQL.

Неформально язык SQL можно разбить на шесть подъязыков: подъязык определения данных; манипулирования данными; построения запросов; ограничения доступа к данным; управления курсором и управления транзакциями. Следующая глава будет посвящена наиболее востребованному элементу SQL — подъязыку построения запросов DQL (Data Query Language).



Глава 12

Построение запросов

Эта глава посвящена наиболее важному элементу языка SQL — инструкции SELECT, которая составляет фундамент подъязыка построения запросов DQL (Data Query Language). И это не удивительно, в переводе с английского "select" означает "выбрать", именно для выбора данных изначально и создавался язык запросов. Исходным материалом для операций выборки выступают одна или несколько таблиц и представлений. Результатом выполнения SQL запроса основанного на инструкции SELECT всегда будет таблица. В интерактивном режиме она просто выводится на экран компьютера, но при желании содержимое таблицы можно отредактировать и сохранить.

Содержимое таблицы, полученной в результате запроса, определяет программист, в простейшем случае она может полностью повторять исходные данные, может представлять собой объединение нескольких таблиц или определенную выборку из исходных данных. Ко всему прочему, запросы SELECT активно используются при создании ряда ключевых объектов БД (представлений, хранимых процедур и триггеров).

Базовая синтаксическая конструкция запроса выглядит следующим образом:

```
SELECT [DISTINCT|ALL]
    { имя поля [AS псевдоним] [,…]
     |функция_агрегирования [AS псевдоним] [,…]
     |выражение для вычисления значения [AS псевдоним] [,…]
     |спецификатор.* }
FROM
   {имя_таблицы [AS псевдоним] [,…]
   |имя_представления [AS псевдоним][,…]}
[WHERE условия отбора]
   [GROUP BY имя поля [,…]] [HAVING условие]
   [ORDER BY имя поля [ASC | DESC] [,…]]
```

При описании этой и последующих синтаксических конструкций языка SQL мы станем придерживаться следующих соглашений:

- фигурные скобки определяют обязательный элемент;
- вертикальная черта (|) тождественна условию "ИЛИ" и указывает на необходимость выбора одного из нескольких приведенных значений;

- квадратные скобки определяют необязательный элемент;
- многоточие (...) указывает необязательную возможность повторения конструкции.

Замечание

Практически у всех производителей СУБД синтаксис инструкции SELECT очень близок к требованию стандарта SQL:92.

Запросы на выборку данных всегда начинаются с ключевого слова SELECT, затем следует перечень полей, которые мы планируем увидеть в результате выполнения запроса. В перечень полей кроме названий столбцов таблиц могут входить агрегирующие функции, выражения и просто константы. За списком полей следует еще одно обязательное слово FROM, а за ним — имена таблиц или представлений, из которых будут отбираться данные.

Большинство примеров запросов в этой главе построены на основе таблиц из нашей БД "Книжный магазин", подробно описанной в *главе 6*. Для того чтобы освежить в памяти этот материал, достаточно ненадолго вернуться к ER-модели, представленной на рис. 6.7.

В простейшем случае запрос должен включать в себя операторы SELECT и FROM, имя таблицы и спецификатор *:

```
SELECT * FROM WRITERS
```

Символ * говорит о том, что в результирующее отношение попадут все столбцы из таблицы авторов WRITERS. Это не всегда желательный способ организации запроса, т. к. выборка всех столбцов значительно повышает сетевой трафик и замедляет выполнение запроса. Поэтому рекомендуется всегда указывать минимально-необходимый набор столбцов, допустим:

```
SELECT WRITER FROM WRITERS
```

Эта строка кода потребует, чтобы в результирующем наборе данных присутствовал только один столбец с именем автора книги.

Если необходимо выбрать несколько столбцов, то их перечисляют через запятую.

```
SELECT CONTRACT, CONTRACTDATE FROM CONTRACTS
```

Если таблица содержит повторяющиеся строки, которые вы не хотите видеть в результате выполнения запроса, то после сразу инструкции SELECT, но перед перечнем полей следует вставить предикат DISTINCT:

```
SELECT DISTINCT TITLE FROM BOOKS
```

Обратный предикат ALL указывает на то, что мы намерены получить все строки данных, включая и дубликаты. Впрочем, в явном применении ALL необходимости нет, т. к. это режим по умолчанию.

В тех случаях, когда запрос достаточно сложен и содержит длинные названия полей и таблиц, или при использовании агрегирующих функций или полей, появившихся в результате выполнения выражений, программисты могут ввести псевдонимы полей и таблиц. Для этого применяется ключевое слово AS:

```
SELECT ('№ '+ORDERNUM+' от '+CAST(ORDERDATE AS VARCHAR)) AS ORDERINFO
FROM ORDERS
```

В представленной строке запроса мы описываем выражение, объединяющее номер и дату накладной, и присваиваем этому полю псевдоним ORDERINFO. Заметьте, что для преобразования формата данных столбца ORDERDATE из даты в текст мы применили функцию CAST().

Замечание

Столбцы, полученные в результате проведения каких-либо математических операций, сложения строк или применения агрегирующих функций часто называют вычисляемыми.

Запрос SQL может быть адресован сразу нескольким таблицам и даже таблицам, физически расположенным в разных базах данных. В таких случаях при перечислении столбцов таблиц следует указывать имя БД и имя таблицы:

```
SELECT MYDB.MYTABLE.FIELD1, MYSECONDDB.MYSECONDTABLE.FIELD2, …
```

Порядок сортировки — *ORDER BY*

Полученный в результате выполнения запроса набор строк можно упорядочить. Порядок сортировки данных определяется при помощи команды ORDER BY и перечня полей сортировки. Нижеприведенный пример осуществит сортировку таблицы контрактов по дате их заключения:

```
SELECT CONTRACT, CONTRACTDATE FROM CONTRACTS ORDER BY CONTRACTDATE
```

В порядке сортировки допускается указать сразу несколько столбцов, разделяя их запятыми.

По умолчанию записи упорядочиваются по возрастанию (от А до Я). Если требуется сортировка в порядке убывания (от Я до А), то после имени столбца поставьте оператор DESC:

```
SELECT * FROM WRITERS ORDER BY WRITER DESC
```

Независимо от используемых в запросе предикатов и инструкций, фраза ORDER BY всегда должна быть последним элементом в предложении SELECT.

Некоторые диалекты SQL при упорядочении позволяют вместо наименований столбцов указывать их номера (имеется в виду порядок определения столбцов в предложении SELECT):

```
SELECT CONTRACT, CONTRACTDATE FROM CONTRACTS ORDER BY 2 ASC, 1 DESC
```

Условие отбора данных — *WHERE*

В результате выполнения запросов пользователь обычно планирует получить не все данные, а какую-то их выборку. Например, книги, написанные в жанре детектива, книги, полученные в прошлом году, или произведения, пользовавшиеся наибольшим спросом в предыдущем квартале. Для того чтобы ограничить набор выводимых данных, необходимо в конструкцию SELECT добавить предложение WHERE.

Совместно с WHERE применяются пять типов ограничений:

1. *Сравнение* — результаты вычисления одного выражения сравниваются с другим. Сравнения осуществляется с помощью операторов: =, <>, <, >, <= и >=. Результат сравнения может принимать значения: True, False и Unknown.
2. *Попадание в диапазон* — проверяется, попадает ли результат вычисления выражения в определенный диапазон значений.
3. *Соответствие шаблону* — проверяется, соответствует ли некоторое строковое значение определенному шаблону.
4. *Неопределенное значение NULL* — содержит ли поле неопределенное значение NULL.
5. *Принадлежность множеству значений* — входит ли результат во множество значений.

Все перечисленные ограничения описываются сразу после инструкции WHERE. В одном запросе с помощью операторов AND, OR и NOT допускается комбинировать несколько ограничений.

Сравнение

Сравнение — наиболее распространенное условие ограничения результирующего набора данных:

```
SELECT * FROM CONTRACTS WHERE CONTRACTDATE>'01/01/2011'
```

Приведенное выражение выберет информацию о контрактах, заключенных позднее 1-го января 2011 года.

Другой вариант более сложного запроса возвратит данные о поставках книг, осуществленных до 1 января 2010 года с номерами контрактов от 10 до 20:

```
SELECT * FROM ORDERS
WHERE ORDERSDATE<'01/01/2010' AND ORDERNUM>=10 AND ORDERNUM<=20
```

Для объединения трех условий мы использовали ключевое слово AND (логическое "И"), кроме этого, в тексте SQL-запроса можно использовать условие OR (логическое "ИЛИ"). Например, покупатель хочет узнать, есть ли у нас книги "Три мушкетера" или "20 лет спустя":

```
SELECT * FROM BOOKS
WHERE TITLE= 'Три мушкетера' OR TITLE= '20 лет спустя'
```

Благодаря тому, что все текстовые символы идентифицируются их числовым кодом, вполне допустимо в условия отбора вводить сравнения с текстовыми данными:

```
SELECT WRITER FROM WRITERS
WHERE WRITER <'P'
```

Если необходимо построить запрос на основе комбинации нескольких условий AND и OR, то следует пользоваться круглыми скобками. Условия, заключенные в круглые скобки, должны выполняться одновременно.

Попадание в диапазон — *BETWEEN*

Предикат BETWEEN осуществляет контроль за тем, чтобы интересующий нас результат входил в определенный диапазон значений:

```
Имя поля BETWEEN Начало диапазона AND Окончание диапазона
```

Или наоборот, чтобы результат не принадлежал определенному диапазону:

```
Имя поля NOT BETWEEN Начало диапазона AND Окончание диапазона
```

Например запрос

```
SELECT * FROM CONTRACTS WHERE CONTRACTDATE
                        BETWEEN '01/01/2010' AND '31/12/2010'
```

вернет все контракты, заключенные в 2010 году начиная с первого января и заканчивая последним днем декабря.

Соответствие шаблону — *LIKE*

Предикат LIKE определяет условия поиска с указанием шаблонов:

```
LIKE::= шаблон подстроки
[NOT] LIKE шаблон подстроки
[ESCAPE служебный символ подстановки]
```

Поиск указанной подстроки в строке. Например, для выбора всех авторов, в имени которых есть символ "о", мы подготовим такой запрос:

```
SELECT * FROM WRITERS WHERE WRITER LIKE '%о%'
```

Запрос вернет данные об авторах, в фамилии которых имеется символ "о". В шаблоне подстроки помимо набора интересующих нас символов допускается применять два трафаретных символа:

- подчеркивание "_" используется вместо любого одиночного символа в выражении;
- символ процента "%" заменяет последовательность любых символов.

Замечание

Если столбец входит в состав индекса, то для ускорения процесса поиска записей желательно, чтобы шаблон подстроки не начинался с трафаретных символов.

Например, для отбора всех авторов, у которых во второй позиции фамилии есть символ "о" и фамилия заканчивается на "ой", выражение должно выглядеть так: '_о%ой'.

Если искомая строка должна включать служебный символ, используемый в качестве символа подстановки, то перед указанием служебного символа ставится ключевое слово ESCAPE:

```
SELECT * FROM WRITERS WHERE WRITER LIKE 'П#%' ESCAPE '#'
```

Проверка неопределенного значения — *IS NULL*

Предикат IS NULL предназначен для проверки факта заполнения поля данными:

```
NULL ::=конструктор значений строки IS [NOT] NULL
```

Следующий пример вернет все записи, у которых поле AMOUNT не определено:

```
SELECT * FROM DELIVERIES WHERE AMOUNT IS NULL
```

Если вы, наоборот, намерены выбрать непустые строки, то запрос должен выглядеть иначе:

```
SELECT * FROM DELIVERIES WHERE AMOUNT IS NOT NULL
```

Принадлежность множеству — *IN*, *ALL*, *ANY*, *SOME*

Оператор подзапроса IN проверяет, входит ли результат вычисления в заданное множество. Множество может описываться как поэлементно, так и быть результатом другого, вложенного подзапроса:

```
SELECT …
WHERE значения строки [NOT] IN табличный подзапрос|элементы множества
```

Например:

```
SELECT * FROM WRITERS WHERE WRITER IN ('ПУШКИН  А.С. ', 'ДОСТОЕВСКИЙ Ф.М' )
```

В основной запрос разрешено вкладывать подзапрос, он следует за инструкцией WHERE. Подзапрос описывается благодаря уже знакомому нам оператору SELECT. Допустим, что нам требуется узнать, имеются ли в таблице WRITERS писатели, с которыми в таблице WRITERS_BOOKS не сопоставлено ни одной записи:

```
SELECT * FROM WRITERS
WHERE WRITER_KEY NOT IN (SELECT WRITER_KEY FROM WRITERS_BOOKS)
```

Предикат IN определяет, будут ли значения строки обнаружены в наборе значений, который либо определен, либо получен по результатам табличного подзапроса. Обратная задача решается при удалении ключевого слова NOT.

Приведенный пример принадлежности к множеству одновременно демонстрирует порядок создания вложенных подзапросов. Язык SQL не лимитирует количество вложений, поэтому механизм вложенных подзапросов является весьма мощным оружием при построении сложных отчетов.

Многие диалекты SQL поддерживают дополнительные операторы проверки принадлежности множеству, используемые совместно с вложенными подзапросами: ANY, ALL и SOME:

```
SELECT …
WHERE значения строки {ALL | ANY | SOME} {подзапрос}
```

Оператор ALL в своем роде эквивалентен логической операции AND ("И"), он выдаст значение TRUE, если подзапрос не возвратит ни одной записи или, наоборот, когда все строки в наборе выдерживают сравнение. Соответственно результат FALSE мы получим в случае, когда сравнение не выдержит хотя бы одна строка в наборе. Например, если мы захотим узнать названия фирм поставщиков, с кем у

нас пока не заключено договоров на поставку, то нужно написать следующие строки кода на языке SQL:

```
SELECT SUPPLIER
FROM SUPPLIERS
WHERE SUPPLIER_KEY <> ALL (SELECT SUPPLIER_KEY FROM CONTRACTS)
```

Операторы ANY/SOME возвратят TRUE, если хотя бы одна из записей подзапроса выдержала сравнение, обратный результат FALSE мы получим, если хотя бы одна из записей не выдерживает сравнения. Например, теперь мы хотим узнать, с кем из поставщиков у нас заключен хотя бы один контракт:

```
SELECT SUPPLIER
FROM SUPPLIERS
WHERE SUPPLIER_KEY = ANY (SELECT SUPPLIER_KEY FROM CONTRACTS);
```

Предикат существования — *EXISTS*

Задача предиката проверить факт существования записей, удовлетворяющих определенному критерию. Предикат может использоваться только совместно с подзапросами:

```
EXISTS::= EXISTS (подзапрос)
```

Результатом выполнения предиката будет булево значение True или False. Логическая истина возвращается в том случае, если в возвращаемом подзапросом наборе данных присутствует хотя бы одна запись. Ключевое слово NOT EXISTS задает обратные правила обработки.

Основное достоинство предиката существования — высокая скорость выполнения подзапроса. Если вы хотите распознать запрос, основанный на существовании каких-то данных, то не стоит писать следующее:

```
IF (SELECT COUNT(*) FROM BOOKS WHERE условие)>0 …
```

При исполнении этого кода (прежде чем оценить условие IF) СУБД будет вынуждена прочитать все строки в исследуемом наборе данных. Чем больше записей в таблице BOOKS, тем дольше станет выполняться запрос. Напротив запрос, построенный на основе инструкции EXISTS, моментально вернет значение TRUE, как только СУБД столкнется с первой же строкой, удовлетворяющей контролируемому условию:

```
IF EXISTS (SELECT * FROM BOOKS WHERE условие) …
```

Многотабличные запросы

Вывод информации из одной таблицы представляет собой частный случай. Значительно чаще возникает необходимость объединения информационных потоков из различных таблиц. Для этого предусмотрен ряд синтаксических структур, позволяющих соединять таблицы. Простейший способ объединения двух и более

таблиц — использование инструкции WHERE с определением полей, применяемых для соединения таблиц. Кроме того, для выполнения объединения после ключевого слова FROM необходимо через запятую перечислить имена объединяемых таблиц. Например, мы хотим соединить таблицы поставщиков (SUPPLIERS) и таблицу контрактов (CONTRACTS). Указанные таблицы объединяются по ключевым полям SUPPLIER-KEY (в таблице поставщиков это первичный ключ, в таблице контрактов — внешний):

```
SELECT SUPPLIER, CONTRACT, CONTRACTDATE FROM SUPPLIERS, CONTRACTS
WHERE SUPPLIERS.SUPPLIER-KEY = CONTRACTS.SUPPLIER-KEY
```

Подобные объединения называют внутренними. Это один из простейших способов объединения в реляционных базах данных. Он реализован во всех существующих сегодня диалектах SQL.

В синтаксисе ANSI имеется еще несколько способов объединения отношений (табл. 12.1).

***Таблица 12.1.** Основные способы соединения таблиц*

Способ соединения	Синтаксис
Перекрестное CROSS	таблица1 CROSS JOIN таблица2 Это прямое декартово произведение, обеспечивающее все возможные объединения строк из каждой таблиц. Соединение такого типа на практике не применяется
Естественное NATURAL	таблица1 [NATURAL] [тип соединения] JOIN таблица2 Соединение внешнего ключа со связанным с ним родительским ключом
Слияние UNION	таблица1 UNION JOIN таблица2 Соединение, в которое входят те строки каждой из таблиц, для которых совпадения не были установлены. Используется для объединения результатов запросов в одну таблицу
Объединение посредством предиката ON	таблица1 [тип соединения] JOIN таблица2 ON предикат где тип соединения {INNER|LEFT|RIGHT|FULL|OUTER}
Объединение посредством имен столбцов USING	таблица1 [тип соединения] JOIN таблица2 USING (имя столбца,…) где тип соединения {INNER|LEFT|RIGHT|FULL|OUTER}

Стандарт SQL:2003 в одном запросе предусматривает применение не более чем одной операции объединения (CROSS, NATURAL, UNION, ON или USING). Перекрестное соединение CROSS в реальных БД обычно не применяют, что проку от всевозмож-

ных комбинаций соединения всех со всеми. Естественное соединение означает автоматическое объединение таблиц по всем столбцам с одинаковыми именами.

Слияние *UNION*

Слияние необходимо в случае, когда требуется объединить результаты нескольких запросов в одну результирующую таблицу. Допустим, что у нас функционируют две не связанные в единую сеть базы данных двух книжных магазинов. Чтобы получить суммарную информацию, необходимо воспользоваться оператором UNION:

```
SELECT * FROM DB1.CONTRACTS
UNION
SELECT * FROM DB2.CONTRACTS
```

В данном примере мы соединили две идентичные таблицы, и на практике у нас не возникнет никаких проблем, если в момент разработки таблиц будут грамотно назначены первичные ключи, что исключит конфликты слияния.

Можно объединять и разные отношения, лишь бы совпало количество столбцов и тип данных объединяемых столбцов.

По умолчанию запрос UNION удаляет повторяющиеся строки из результирующего отношения, однако если их следует сохранить, то нужно позвать на помощь предикат ALL:

```
SELECT * FROM DB1.CONTRACTS
UNION ALL
SELECT * FROM DB2.CONTRACTS
```

Объединение *ON*

Наиболее распространенный способ объединения таблиц посредством предиката рассмотрим на примере таблиц поставщиков и контрактов. Допустим, таблица поставщиков содержит 7 записей, но, т. к. наш магазин только начал работу, пока контракты заключены лишь с четырьмя поставщиками: "Заря", ЗАО "Яшма" и ОАО "Рассвет" (рис. 12.1).

Осуществим внутреннее объединение таблиц SUPPLIERS и CONTRACTS с помощью оператора INNER JOIN:

```
SELECT *
FROM SUPPLIERS
INNER JOIN CONTRACTS ON SUPPLIERS.SUPPLIER_KEY=CONTRACTS.SUPPLIER_KEY
```

В итоге мы получим отношение из пяти строк, в котором вы не найдете поставщиков "Заря", ЗАО "Яшма" и ОАО "Рассвет" (рис. 12.2). Причина в том, что в поле внешнего ключа SUPPLIER_KEY таблицы контрактов CONTRACTS отсутствуют значения их первичных ключей. Сразу разберемся в терминологии. При объединении таблиц зачастую оперируют терминами внутренняя и внешняя таблица. Табли-

ца, описанная слева от знака равенства, называется внешней, справа — внутренней. Эти таблицы еще называют левой и правой.

Замечание

Результат внутреннего объединения на основе конструкции `INNER JOIN` будет точно такой же, как и результат объединения таблиц с помощью `WHERE`. Одно из отличий синтаксической конструкции с оператором `JOIN` от конструкции с оператором `WHERE` в том, что при работе с `JOIN` после слова `FROM` нет необходимости перечислять все объединяемые таблицы.

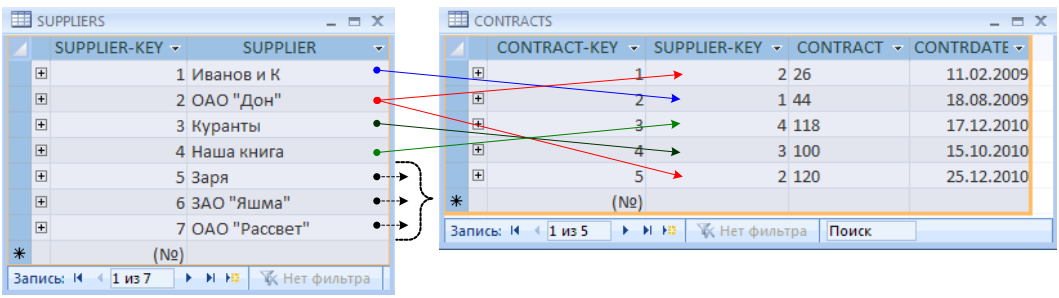


Рис. 12.1. Данные таблиц `SUPPLIERS` и `CONTRACTS`

Рис. 12.2. Внутреннее объединение (`INNER JOIN`) таблиц `SUPPLIERS` и `CONTRACTS`

А что делать, если нам требуется увидеть в результирующей таблице все записи из таблицы поставщиков? Для этого необходимо осуществить левое объединение таблиц (другими словами, заставить вывести все записи из левой таблицы независимо от того, есть ли у них подчиненные записи в правой таблице). Запрос незначительно изменится — заменим `INNER` на `LEFT`:

```
SELECT *
FROM SUPPLIERS

LEFT JOIN CONTRACTS ON SUPPLIERS.SUPPLIER_KEY=CONTRACTS.SUPPLIER_KEY
```

На этот раз в результирующем отношении мы увидим всех поставщиков (рис. 12.3) несмотря на то, что с ними пока не заключались контракты.

LEFT JOIN Suppliers-Contracts

SUPPLIERS.SUPPLIER-KEY	SUPPLIER	CONTRACT-KEY	CONTRACTS.SUPPLIER-KEY	CONTRACT	CONTRDATE
1	Иванов и К	2	1	44	18.08.2009
2	ОАО "Дон"	1	2	26	11.02.2009
2	ОАО "Дон"	5	2	120	25.12.2010
3	Куранты	4	3	100	15.10.2010
4	Наша книга	3	4	118	17.12.2010
5	Заря				
6	ЗАО "Яшма"				
7	ОАО "Рассвет"				
(№)		(№)			

Запись: 1 из 8 Нет фильтра Поиск

Рис. 12.3. Левое объединение (`LEFT JOIN`) таблиц `SUPPLIERS` и `CONTRACTS`

Для создания правого объединения замените в предыдущем примере ключевое слово `LEFT` на `RIGHT`. Теперь запрос возвратит все записи из правой таблицы и соответствующие ей записи левой таблицы. Полное объединение требует использования ключевого слова `FULL`. Такое объединение позволит вывести все записи как из левой, так и из правой таблиц.

Редко встречающийся тип перекрестного объединения `CROSS` заставит запрос возвратить все возможные комбинации строк из объединяемых таблиц. Ценность сформированного таким образом результирующего набора строк весьма условна, но, раз такая возможность существует, то о ней нужно знать.

При необходимости можно соединять практически бесконечное число таблиц. В листинге 12.1 приведен пример варианта запроса SQL, формирующий единое отношение из всех десяти таблиц БД книжного магазина (см. рис. 6.5).

Листинг 12.1. Пример объединения нескольких таблиц

```
SELECT *
FROM SUPPLIERS
LEFT JOIN  CONTRACTS  ON SUPPLIERS.SUPPLIER_KEY=CONTRACTS.SUPPLIER_KEY
LEFT JOIN  ORDERS     ON CONTRACTS.CONTRACT_KEY=ORDERS.CONTRACT_KEY
LEFT JOIN  DELIVERIES ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
RIGHT JOIN PUBLISHERS ON DELIVERIES.PUBLISHER_KEY
                                    =PUBLISHERS.PUBLISHER_KEY
INNER JOIN BOOKS      ON DELIVERIES.BOOK_KEY=BOOKS.BOOK_KEY
INNER JOIN GENRES_BOOKS  ON GENRES_BOOKS.BOOK_KEY=BOOKS.BOOK_KEY
INNER JOIN GENRES        ON GENRES_BOOKS.GENRE_KEY=GENRES_BOOKS.GENRE_KEY
INNER JOIN WRITERS_BOOKS ON WRITERS_BOOKS.BOOK_KEY=BOOKS.BOOK_KEY
INNER JOIN WRITERS       ON WRITERS_BOOKS.WRITER_KEY=WRITERS.WRITER_KEY
```

Внимание!

Порядок объединения таблиц имеет критическое значение. Если таблицы соединяются в правильном порядке, то общее число обрабатываемых строк будет меньше. Описывая

запрос, следует выполнять сначала максимально ограничивающий поиск, чтобы отфильтровать как можно большее число строк на ранних фазах выполнения запроса с соединениями.

Объединение *USING*

Альтернативой объединению посредством ON выступает объединение, построенное на основе USING. В этом случае требуется указать имена столбцов, существующих в обеих таблицах (при условии, что имена столбцов совпадают):

```
SELECT *
FROM SUPPLIERS
LEFT JOIN CONTRACTS USING SUPPLIER-KEY
```

Объединение разрешается проводить и по нескольким столбцам, разделяя их запятыми.

Агрегирующие функции

Агрегирующие (обобщающие) функции в качестве исходных параметров принимают значения, указанные в запросе (после слова SELECT), и вычисляют результат. Как правило, эти функции применяются в запросах группировки с помощью команды GROUP BY. Наиболее распространенные функции приведены в табл. 12.2.

***Таблица 12.2.** Агрегирующие функции*

Функция	Описание
COUNT	Возвращает количество строк: **SELECT** COUNT(*) **FROM** ORDERS Тип возвращаемого значения — INTEGER
AVG	Вычисляет среднее арифметическое для указанных элементов: **SELECT** AVG(имя_столбца) **FROM** имя_таблицы Используется только для полей цифровых типов данных. Тип возвращаемого значения FLOAT
SUM	Вычисляет сумму значений: **SELECT** SUM(имя_столбца) FROM имя_таблицы Применяется только для цифровых типов данных. Будьте внимательны. Если в результате вычисления суммы итоговое число превысит максимальное значение используемого типа данных, то возникнет ошибка
MAX	Возвращает наибольшее из всех значений: **SELECT** MAX(имя_столбца) **FROM** имя_таблицы

Таблица 12.2 *(окончание)*

Функция	Описание
MIN	Возвращает наименьшее из всех значений: **SELECT** MIN(имя_столбца) **FROM** имя_таблицы

ЗАМЕЧАНИЕ

Все агрегатные функции работают с единственным полем таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к текстовым полям. Функции SUM и AVG сохраняют работоспособность только при обслуживании числовых полей. Совместно с перечисленными функциями разрешено применять команду DISTINCT.

Группировка данных — *GROUP BY*

Предложение GROUP BY предназначено для осуществления группировки выходных строк по какому-либо признаку, например по равенству значений каких-либо столбцов. В группирующих запросах обычно присутствуют рассмотренные чуть ранее в табл. 12.2 агрегирующие функции.

Допустим, что мы хотим узнать, сколько всего книг поступило по каждому из приходных ордеров, и уточнить, сколько наименований книг было в каждой партии. Для этого нам следует объединить таблицы ORDERS и DELIVERIES и, используя агрегирующие функции SUM() и COUNT(), построить группирующий запрос (листинг 12.2).

Листинг 12.2. Пример группирующего запроса

```
SELECT SUM(DELIVERIES.AMOUNT) AS SUMAMOUNT,
       COUNT(DELIVERIES.AMOUNT) AS COUNTAMOUNT,
       ORDERS.ORDERNUM, ORDERS.ORDERDATE
FROM ORDERS
INNER JOIN DELIVERIES ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
GROUP BY   ORDERS.ORDERDATE, ORDERS.ORDER
```

В результате выполнения запроса мы получим представленное на рис. 12.4 отношение. Обратите внимание на то, каким образом мы присвоили имена результатам, возвращенным агрегирующими функциями — для этого мы дважды воспользовались предикатом AS. Еще одно замечание, касающееся особенностей построения группирующих запросов, — все имена полей, приведенные в списке группирующего запроса SELECT, должны присутствовать и во фразе GROUP BY. Единственное исключение делается для полей, обрабатываемых агрегирующей функцией.

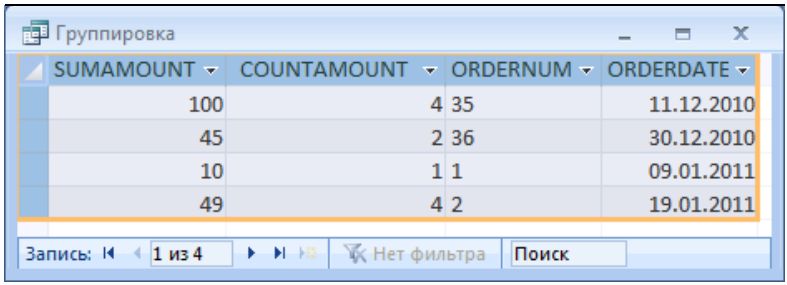


Рис. 12.4. Результаты группирующего запроса

Замечание

В результате выполнения группирующего запроса для каждой отдельной группы создается одна-единственная группирующая строка.

Дополнительная фильтрация группы строк — *HAVING*

Предикат `HEAVING` используется после группировки (предложения `GROUP BY`) и предназначен для дополнительной фильтрации уже сформированных групп строк. Поведение элемента `HEAVING` подобно ключевому слову `WHERE`, но он работает не со всеми строками таблиц, а со строками результирующего набора. Фильтрация может осуществляться с помощью агрегирующих функций (`COUNT`, `SUM`, `AVG`, `MAX` и `MIN`). Для примера, несколько модифицируем текст предыдущего запроса и научим его выводить информацию только о поставках, состоящих из более десяти экземпляров книг (листинг 12.3).

Листинг 12.3. Ограничение группирующего запроса с помощью предиката `HAVING`

```
SELECT SUM(DELIVERIES.AMOUNT) AS SUMAMOUNT,
       COUNT(DELIVERIES.AMOUNT) AS COUNTAMOUNT,
       ORDERS.ORDERNUM, ORDERS.ORDERDATE
FROM ORDERS
INNER JOIN DELIVERIES ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
GROUP BY ORDERS.ORDER, ORDERS.ORDERDATE
HAVING SUM(DELIVERIES.AMOUNT) >10
```

Оконные функции

В стандарте SQL:2003 появилась еще одна разновидность функций, называемых *оконными* (window). Видимо, название "оконная функция" связано с тем, что

результирующий набор строк разбивается на секции (окна). Но даже, исходя из такого предположения, название не вполне точно раскрывает назначение этих функций, на мой взгляд, было бы корректнее назвать их аналитическими, но стандарт есть стандарт. Оставим споры вокруг названия и обсудим суть нововведения.

Оконные функции призваны развить возможности, заложенные в рассмотренных несколько ранее агрегирующих функциях. Существующие со времен SQL:92 функции `COUNT()`, `SUM()` и т. д. (см. табл. 12.1) возвращают по одному значению на группу выходных строк запроса, а предлагаемые разработчиками нового стандарта функции должны приобрести большую гибкость и смогут возвращать данные с большей степенью детализации.

Стандарт SQL:2003 предусматривает появление ряда новых инструкций:

- Функции ранжирования.
- Дистрибутивные функции (`PERCENT_RANK`, `CUME_DIST`).
- Функция, возвращающая номер строки в секции.

Предложено множество оконных агрегирующих функций (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, `VAR_POP`, `VAR_SAMP`, `STDDEV_POP`, `STDDEV_SAMP`, `REGR_COUNT`, `COVAR_POP`, `PERCENTILE_CONT` и т. д.).

Функции ранжирования присваивают строкам результирующего отношения определенный вес (ранг). Если две и более строки претендуют на один вес, то все они получат одинаковое весовое значение. Если следующая строка набора претендует на более высокий вес, то она получит более высокий ранг.

Стандартом предусмотрено существование двух функций ранжирования. Функция `RANK()` возвращает ранг строки в секции набора строк по следующему правилу: единица плюс количество рангов, находящихся до текущей строки. Такой подход может привести к появлению разрывов между весовыми коэффициентами. Например, равные по весу две первые строки получают ранг, равный единице, а третья строка получает вес, равный трем. Функция `DENSE_RANK()` работает несколько иначе. Она возвращает ранг строк в секции результирующего набора без промежутков в ранжировании. Ранг строки равен количеству различных значений рангов, предшествующих строке, увеличенному на единицу.

Продемонстрировать процесс ранжирования можно на примере базы книжного магазина. Если в качестве весового коэффициента мы выберем количество поставленных книг `AMOUNT`, тогда инструкция SQL станет выглядеть так, как в листинге 12.4.

Листинг 12.4. Пример ранжирующего запроса

```
SELECT ORDERNUM, ORDERDATE, PUBLISHER, TITLE, AMOUNT,
       RANK() OVER (ORDER BY AMOUNT) AS RANK
FROM BOOKS
INNER JOIN DELIVERIES ON BOOKS.BOOK_KEY=DELIVERIES.BOOK_KEY
INNER JOIN PUBLISHERS ON PUBLISHERS.PUBLISHER_KEY=DELIVERIES.PUBLISHER_KEY
INNER JOIN ORDERS ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
```

Результаты выполнения запроса иллюстрирует экранный снимок (рис. 12.5).

	ORDERNUM	ORDERDATE	PUBLISHER	TITLE	AMOUNT	RANK
1	2	2011-01-19 ...	Мир	Война и мир	5	1
2	1	2011-01-09 ...	Мир	Фауст	10	2
3	2	2011-01-19 ...	Детская литература	Три мушкетёра	12	3
4	2	2011-01-19 ...	Кавказ	Двадцать лет спустя	12	3
5	35	2010-12-11 ...	Мир	Севастопольские рассказы	15	5
6	35	2010-12-11 ...	Мир	Война и мир	20	6
7	36	2010-12-30 ...	Кавказ	Поэтический сборник	20	6
8	2	2011-01-19 ...	Детская литература	Анна Каренина	20	6
9	36	2010-12-30 ...	Гослитиздат	Идеальный муж	25	9
10	35	2010-12-11 ...	Художественная литература	Сказка о царе Салтане	25	9
11	35	2010-12-11 ...	Мир	Евгений Онегин	40	11

Рис. 12.5. Ранжирование строк с помощью функции `RANK()`

Внимание!

При работе с оконными функциями большинство диалектов SQL требует применения ключевого слова `OVER()`. Наличие круглых скобок указывает на то, что `OVER()` может принимать дополнительные операторы.

Дистрибутивные функции так же, как и функции ранжирования, решают задачу присвоения весового значения строкам секции отношения. Но теперь, в отличие от обычных функций ранжирования, вес нормируется к единице, другими словами, может принимать любое значение в диапазоне от 0,0 до 1,0.

Дистрибутивных функций также две. Функция `PERCENT_RANK()` определяет вес строки *R* по формуле

$$R = RK - 1/NR - 1. \quad (12.1)$$

Здесь *RK* — весовое значение, полученное с помощью рассмотренной ранее функции `RANK()`, *N* — общее число строк в секции.

Функция `CUME_DIST()` рассчитывает вес строки *R*, пользуясь выражением

$$R = NP/NR. \quad (12.2)$$

Числитель *NP* хранит число строк до *R*, а знаменатель *NR* содержит число строк в секции.

Как всегда, любые нововведения в стандарт приводят к неоднозначной реакции в стане разработчиков СУБД. Например, функция `ROW_NUMBER` была поддержана в Oracle, MS SQL Server, DB2, MySQL, а популярные СУБД PostgreSQL и InterBase пока "находятся в раздумье". Следующие строки кода (листинг 12.5), написанные на диалекте Transact SQL, позволят сформировать секцию первых 20 записей сведений о контрактах, упорядоченных по дате их заключения.

Листинг 12.5. Разделение результата запроса на секции

```
WITH ContractRows AS
(SELECT Contract, ContrDate,
ROW_NUMBER() OVER (ORDER BY ContrDate)AS RowNum
FROM Contracts)

SELECT *
FROM ContractRows
WHERE RowNum BETWEEN 1 AND 20
```

Заметьте, что задача решается в два этапа. На первом этапе мы создаем искусственное отношение `ContractRows`, в которое с помощью `Row_Number()` вводим вычисляемый столбец `RowNum`, где и будут содержаться номера строк. На втором этапе мы отбираем из отношения `ContractRows` требуемый диапазон строк.

Еще одно замечание относится к инструкции `OVER`. Это ключевое слово, идентифицирующее функцию как аналитическую. Иначе анализатор кода не смог бы отличить обычную агрегирующую функцию `COUNT()` от ее аналитической оконной "однофамилицы" `COUNT()`.

Основное достоинство оконных аналитических функций заключается в возможности разделения результатов запроса на сегменты, которые и подвергаются анализу. Допустим, что в рамках единого запроса нам требуется получить аналитическую информацию о поступлениях книг в магазин за определенный период времени. В результирующем наборе строк мы хотим увидеть:

- название издательства;
- число всех экземпляров книг, пришедших из издательства;
- сколько наименований произведений было получено;
- количество конкретных произведений с их названиями.

Средствами группировки это сделать невозможно, а с помощью оконных функций без проблем (листинг 12.6).

Листинг 12.6. Пример аналитического запроса с использованием инструкции `OVER()`

```
SELECT PUBLISHER,
       SUM(AMOUNT) OVER (PARTITION BY PUBLISHER) AS SUM_BY_PUBLISHER,
       COUNT(*) OVER (PARTITION BY PUBLISHER) AS COUNT_DELEVERIES,
       AMOUNT,
       TITLE
FROM PUBLISHERS
INNER JOIN DELIVERIES ON PUBLISHERS.PUBLISHER_KEY
                                          =DELIVERIES.PUBLISHER_KEY
INNER JOIN BOOKS ON BOOKS.BOOK_KEY=DELIVERIES.BOOK_KEY
```

```
INNER JOIN ORDERS ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
WHERE ORDERDATE BETWEEN '2010-01-01' AND '2012-12-31'
ORDER BY PUBLISHER
```

Нам понадобилась помощь оконных функций `SUM()` и `COUNT()`. Для того чтобы анализатор SQL понял, что это аналитические функции, мы вновь воспользовались ключевым словом `OVER()`. Для разрезки результирующего множества на сегменты применялся оператор `PARTITION BY`, который разделил весь результирующий набор строк на окна по названиям издательств. Результаты выполнения иллюстрирует рис. 12.6. По этому экранному снимку можно сделать вывод о том, что из издательства "Гослитиздат" было всего одно поступление. Из издательства "Детская литература" мы получили всего 32 экземпляра книг (столбец `SUM_BY_PUBLISHER`), они включали два наименования (столбец `COUNT_DELEVERIES`): книга "Анна Каренина" (20 экземпляров) и "Три мушкетера" (12 экземпляров).

	PUBLISHER	SUM_BY_PUBLISHER	COUNT_DELEVERIES	AMOUNT	TITLE
1	Гослитиздат	25	1	25	Идеальный муж
2	Детская литература	32	2	20	Анна Каренина
3	Детская литература	32	2	12	Три мушкетёра
4	Кавказ	32	2	12	Двадцать лет спустя
5	Кавказ	32	2	20	Поэтический сборник
6	Мир	90	5	10	Фауст
7	Мир	90	5	5	Война и мир
8	Мир	90	5	20	Война и мир
9	Мир	90	5	15	Севастопольские рассказы
10	Мир	90	5	40	Евгений Онегин
11	Художественная литература	25	1	25	Сказка о царе Салтане

Рис. 12.6. Результаты выполнения аналитического запроса

К сожалению, далеко не все СУБД поддерживают стандартизированные в SQL:2003 оконные функции. Когда задумка будет реализована во всех современных диалектах SQL в полном объеме? Пока этот вопрос остается риторическим. Дело в том, что к моменту опубликования последнего стандарта, большинство разработчиков уже ввело в разрабатываемые СУБД свои собственные решения, обычно называемые *функциями оперативного анализа данных* (OnLine Analytic Processing, OLAP).

Рекурсивные запросы

Язык SQL позволяет строить рекурсивные запросы, другими словами, объединять таблицу саму с собой. Допустим, что в нашем распоряжении имеется таблица `DEPARTMENTS`, содержащая перечень отделов и служб компании и фамилии сотруд-

ников этих отделов, кроме того, имеется столбец логического типа, в котором ставится отметка, если строка содержит фамилию руководителя отдела (рис. 12.7).

DEPARTMENTS

DEP_KEY	DEPARTMENT	PPLNAME	MANAGER
1	Бухгалтерия	Алфёрова И.В.	☑
2	Бухгалтерия	Кузьмин А.С.	☐
3	Бухгалтерия	Фёдоров С.В.	☐
4	Отдел сбыта	Яковлев В.П.	☑
5	Отдел сбыта	Гусев В.А.	☐
6	Отдел сбыта	Сулименко С.В.	☐
7	Отдел сбыта	Данилов В.А.	☐
8	Склад	Клоков А.В.	☑
9	Склад	Шуверов В.П.	☐
10	Склад	Журавлёва Е.Ф.	☐

Рис. 12.7. Содержимое таблицы DEPARTMENTS

Предположим, что нам поступила заявка сформировать список сотрудников всех отделов с фамилией их руководителя. Для этого как раз и следует объединить таблицу саму с собой. Для построения рекурсивного запроса нам потребуется пойти на хитрость и объявить в секции FROM одну и ту же таблицу дважды. Язык SQL не станет возражать против этого при одном условии — если мы присвоим таблице два разных псевдонима, в моем примере это A и B:

```
SELECT A.DEPARTMENT, A.PPLNAME, B.PPLNAME AS MANAGER
FROM DEPARTMENTS AS A, DEPARTMENTS AS B
WHERE A.DEPARTMENT=B.DEPARTMENT AND B.MANAGER=TRUE
ORDER BY A.DEPARTMENT, A.PPLNAME
```

Рекурсивный запрос

DEPARTMENT	PPLNAME	MANAGER
Бухгалтерия	Алфёрова И.В.	Алфёрова И.В.
Бухгалтерия	Кузьмин А.С.	Алфёрова И.В.
Бухгалтерия	Фёдоров С.В.	Алфёрова И.В.
Отдел сбыта	Гусев В.А.	Яковлев В.П.
Отдел сбыта	Данилов В.А.	Яковлев В.П.
Отдел сбыта	Сулименко С.В.	Яковлев В.П.
Отдел сбыта	Яковлев В.П.	Яковлев В.П.
Склад	Журавлёва Е.Ф.	Клоков А.В.
Склад	Клоков А.В.	Клоков А.В.
Склад	Шуверов В.П.	Клоков А.В.

Рис. 12.8. Результат рекурсивного запроса

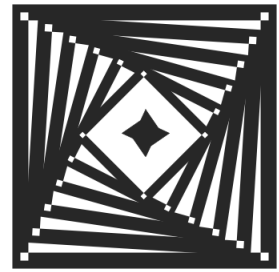
Все остальное — дело техники. Мы описываем условие отбора данных WHERE, объединяя виртуальные таблицы A и B по полю, содержащему название отдела, и

ставим дополнительное условие — из таблицы `B` отбирать только записи менеджеров. Результат выполнения такого запроса представлен на рис. 12.8.

Резюме

Запросы на выборку данных из таблиц и представлений строятся на основе инструкции `SELECT`. Инструкция `SELECT` позволяет отбирать только часть данных в соответствии с правилами, задаваемыми пользователем, сортировать и группировать полученный набор данных. Условия отбора данных строятся с помощью сравнения, попадания в диапазон, соответствия шаблону, принадлежности множеству и другими способами. Условия отбора можно усложнять за счет применения логических операций. В результате выполнения запроса пользователь получает отношение, содержащее ноль и более строк.

В следующей главе мы изучим еще две составных части SQL: подъязык манипулирования данными (DML, Data Manipulation Language) и подъязык управления транзакциями.



Глава 13

Манипулирование данными и управление транзакциями

В этой главе мы познакомимся с одной из основных функций языка SQL — возможностью манипулировать данными, другими словами, добавлять, редактировать и удалять записи таблиц. Для обеспечения согласованности и надежности данных, находящихся в БД, в современных СУБД все операции манипулирования данными осуществляются в рамках транзакций, гарантирующих корректное выполнение команд SQL.

Язык манипулирования данными DML

Для редактирования данных в состав SQL введен *подъязык манипулирования данными* (Data Manipulation Language, DML). Его базис составляют три оператора:

- `INSERT` — предназначен для вставки в таблицу новых записей.
- `UPDATE` — позволяет редактировать записи в таблице.
- `DELETE` — удаляет записи.

Перечисленные инструкции могут применяться различными способами (которые, в первую очередь, определяются структурой таблицы) и охватывать одну, несколько или все строки таблицы одновременно.

Вставка, инструкция *INSERT*

Назначение инструкции `INSERT` — вставка в таблицу одной или нескольких строк. Практически все диалекты SQL полностью поддерживают синтаксическую конструкцию, рекомендованную стандартом:

```
INSERT INTO {имя таблицы | имя представления}
{имя столбца[,…]}
{DEFAULT VALUES | VALUES (Значение1[,…]) | инструкция SELECT}
```

Имя пополняемой таблицы указывают после ключевых слов `INSERT INTO`. С помощью `INSERT` допускается вставка новой записи и в представление, но при усло-

вии, что представление обслуживает одну таблицу. Сразу за именем таблицы следует перечень столбцов, в которые мы собираемся внести новую информацию. Если список столбцов окажется неполным, то в оставшиеся поля записывается значение по умолчанию. Если, в свою очередь, значение по умолчанию не было определено заранее (на этапе создания таблицы) и на столбец наложено ограничение NOT NULL, то работа оператора прерывается. Еще одним поводом для приостановки операции вставки новой записи может послужить попытка поместить данные в столбец, который самостоятельно генерирует новое значение (например, в столбцы автоинкрементного типа).

Если требуется добавить значение всего в одно поле, то следует воспользоваться таким примером:

```
INSERT INTO WRITERS
(WRITER)
VALUES
('Пушкин Александр Сергеевич')
```

В результате таблица авторов пополнится новой записью. Если добавляемая запись должна передать значения в несколько столбцов, то имена столбцов и значения необходимо разделять запятыми:

```
INSERT INTO ORDERS
(CONTRACT_KEY, ORDER, ORDERDATE)
VALUES
(2,4,'4/05/2011')
```

Замечание

Если вы осуществляете вставку данных во все столбцы, то в команде на добавление новой строки перечислять столбцы не обязательно. Но в этом случае последовательность передаваемых значений должна четко соответствовать физическому порядку столбцов таблицы.

При реализации команды INSERT некоторые разработчики СУБД расширили стандартные возможности SQL. Например, MySQL и DB2 позволяют за одно обращение к команде INSERT вставлять в таблицу несколько записей:

```
INSERT INTO CONTRACTS
(SUPPLIER_KEY, CONTRACT, CONTRACTDATE)
VALUES
(11, 47, '02/02/2012'), (23, 47, '02/02/2012'), (2, 48, '03/02/2012')
```

Такое решение весьма удобно, особенно, когда требуется добавить все строки в таблицу в рамках единственной транзакции.

Если вместо ключевого слова VALUES в инструкции на вставку новой записи применить выражение DEFAULT VALUES, то в столбцах таблицы окажутся значения, определенные по умолчанию, в оставшихся полях (не имеющих определения DEFAULT) вы обнаружите неопределенные значения NULL.

ВНИМАНИЕ!

Инструкция INSERT совместно с выражением DEFAULT VALUES часто служит источником ошибок, например в ситуации, когда таблица содержит столбцы внешних ключей (FOREGIN KEY) или столбцы, не допускающие неуникальные значения.

Большинство диалектов SQL для вставки значений по умолчанию вообще не требуют от пользователя каких-либо дополнительных указаний. Допустим, что мы создали таблицу, у которой первое поле по умолчанию принимает нулевое значение:

```
CREATE TABLE DEMO (FIELD1 INTEGER DEFAULT 0, FIELD2 INTEGER NOT NULL)
```

Если значение по умолчанию вас устраивает, то для добавления в таблицу новой строки в инструкции INSERT большинства производителей СУБД достаточно описать второе поле:

```
INSERT INTO DEMO
(FIELD2) VALUES (555)
```

Если поле таблицы допускает, чтобы в нем находилось неопределенное значение NULL, то в предложении VALUES команды INSERT следует передавать в это поле NULL:

```
INSERT INTO DEMO
(FIELD1, FIELD2) VALUES (NULL, 1)
```

Стандарт SQL:2003 рекомендует производителям доработать свои диалекты SQL так, чтобы они позволяли вставлять в целевую таблицу строки, извлекаемые инструкцией SELECT из другой таблицы:

```
INSERT INTO SUPPLIERS (SUPPLIER)
VALUES(SELECT NEWSUPPLIER FROM SECOND_SUPPLIER_TABLE)
```

Условием корректности выполнения такой команды должно быть соответствие типов столбцов целевой и донорской таблиц.

Редактирование, инструкция *UPDATE*

Вам никогда не хотелось все исправить? Если да, то сейчас мы поговорим о самой главной команде в нашей с вами жизни — инструкции редактирования данных UPDATE:

```
UPDATE {имя таблицы | имя представления}
SET {{имя_столбца=
      выражение | значение | NULL | DEFAULT | ARRAY | ROW=строка}[,…]}
[WHERE условие отбора | WHERE CURRENT OF имя курсора]
```

Минимальный код, позволяющий за один присест изменить все записи в таблице, выглядит следующим образом:

```
UPDATE WRITERS SET WRITER='Кукушкин'
```

Предложенную строку вряд ли стоит когда-нибудь применять в реальных проектах. Почему? Хотя бы потому, что она одним махом поменяет имена всех авторов из таблицы писателей на фамилию гражданина Кукушкина. Да простят меня все Кукушкины, но в данном случае такой поступок не вполне корректен.

Неоспоримое удобство инструкции UPDATE заключается в возможности практически мгновенного изменения данных во всей таблице. Допустим, что существует некая таблица SALES, в одном из полей которой хранится значение заработной платы сотрудников компании:

```
UPDATE SALES SET PPLMONEY=PPLMONEY*2
```

Рассмотренный пример разработан специально для тех, кто хочет почувствовать себя Робин Гудом. Благодаря единственной строке кода мы умудрились повысить зарплату всем сотрудникам предприятия ровно в два раза. Если после этого вас не уволят, то отметим, что приведенные примеры заставят СУБД внести изменения во все записи таблицы, поэтому старайтесь ограничивать набор редактируемых записей с помощью оператора WHERE. Правила определения ограничений WHERE в команде UPDATE полностью соответствуют правилам, применяемым в инструкции SELECT:

```
UPDATE ORDERS SET ORDER= '46'
WHERE ORDER_KEY=167
```

В последнем примере мы всего-навсего изменили номер приходного ордера у накладной с первичным ключом 167. Так как значение первичного ключа 167 может принадлежать всего лишь одной строке таблицы, то все изменения затронули всего одну запись в таблице.

Замечание

Во многих СУБД редактирование данных осуществляется следующим образом: строка со старой информацией помечается как удаленная, а в таблицу вставляется новая строка, с обновленными данными. А для того, чтобы не нарушить целостность данных, в эту запись переносится старое значение первичного ключа.

Если мы ошиблись очень сильно и нам необходимо одновременно внести изменения в несколько полей, то следует разделять пары "имя поля = значение" запятыми:

```
UPDATE DELIVERIES
SET BOOK_KEY=784, AMOUNT=240
WHERE BOOK_KEY=783 AND ORDER_KEY=987
```

Еще раз обращаю внимание читателя на то, что в данном случае синтаксис WHERE на сто процентов соответствует синтаксису WHERE в операторе SELECT. Вместе с тем, есть и второй способ применения оператора — WHERE CURRENT OF. Такая инструкция используется при работе с курсором: будет отредактирована запись, на которой позиционирован курсор.

Удаление, инструкция *DELETE*

Инструкция DELETE специализируется на удалении строк из таблиц данных:

```
DELETE FROM имя_таблицы
[{WHERE предикат}]
| {WHERE CURRENT OF имя_курсора}
```

Минимальная команда включает инструкцию DELETE FROM и имя таблицы:

```
DELETE FROM GENRES
```

Такая команда полностью очистит таблицу, что в обычных случаях не применяется, т. к. скорее напоминает вредительство. Чтобы осуществить адресное удаление строк, необходимо после предиката WHERE описать условие удаления:

```
DELETE FROM DELIVERIES WHERE ORDER_KEY=286 AND BOOK_KEY=78
```

Условия удаления могут быть достаточно сложными:

```
DELETE FROM CONTRACTS
WHERE NOT EXISTS
(SELECT * FROM ORDERS WHERE ORDERS.CONTRACT_KEY = CONTRACTS.CONTRACT_KEY)
```

В данном примере мы удалили все записи из таблицы контрактов, по которым книги не поступили в таблицу ORDERS.

Благодаря возможностям WHERE можно создавать весьма неординарные конструкции по очистке таблиц от излишних данных. Хорошим подтверждением этого утверждения может стать часто встречающаяся задача удаления повторяющихся значений из таблицы. Смоделируем такую ситуацию в таблице PUBLISHERS. Допустим, что по нашему недосмотру таблица "обогатилась" строками с дублирующимися названиями издательств (рис. 13.1). Это не просто неприятная новость, а в первую очередь потенциальная угроза для бизнес-логики и целостности данных БД, поэтому от дубликатов следует немедленно избавиться.

PUBLISHERS

PUBLISHER-KEY	PUBLISHER
1	Мир
2	Художественная литература
3	Гослитиздат
4	Кавказ
5	Детская литература
6	Издательство "Правда"
7	Алые паруса
8	Северное сияние
9	Мир
10	Мир
11	Художественная литература
12	Алые паруса

Рис. 13.1. Таблица с повторяющимися значениями

Для решения задач подобного рода следует вспомнить о возможностях группировки данных:

```
DELETE FROM PUBLISHERS
WHERE PUBLISHER_KEY NOT IN
(SELECT MIN(PUBLISHER_KEY) FROM PUBLISHERS GROUP BY PUBLISHER)
```

Вся соль этого примера сосредоточена в последней строке кода, содержащей вложенный подзапрос SELECT. Здесь вы ввели условие группировки по полю

PUBLISHER, содержащему повторяющиеся значения с условием вывода только записей с минимальным значением первичного ключа. Как следствие, подзапрос отбросит все дубликаты данных. Остальное — дело техники, во второй строке инструкции SQL мы определяем, что удалению подлежат все записи, не попавшие в список группировки, для этого нам понадобилось выражение NOT IN. Результат работы иллюстрирует рис. 13.2 — дубликатов как и не бывало.

PUBLISHERS

PUBLISHER_KEY	PUBLISHER
1	Мир
2	Художественная литература
3	Гослитиздат
4	Кавказ
5	Детская литература
6	Издательство "Правда"
7	Алые паруса
8	Северное сияние
#Удалено	#Удалено
#Удалено	#Удалено
#Удалено	#Удалено
#Удалено	#Удалено

Рис. 13.2. Таблица после удаления дубликатов

Конструкция WHERE CURRENT OF предназначена для позиционного удаления. Удаляется строка, на которой находится курсор:

```
DELETE FROM ORDERS WHERE CURRENT OF ORDERS_CURSOR
```

Слияние данных, инструкция *MERGE*

Инструкция MERGE развивает возможности базовых команд INSERT, UPDATE и DELETE. Она предоставляет способ выполнять операции вставки, редактирования и удаления данных над целевой таблицей, основываясь на некотором наборе правил, которые определяются в результате сравнения строк целевой и исходной таблицей.

В стандарте SQL:2003 рекомендована простейшая синтаксическая конструкция обращения к инструкции:

```
MERGE INTO <целевая_таблица> [ [ AS ] <псевдоним_слияния> ]
USING <исходная таблица>
ON <условия совпадения>  <спецификация операции слияния>
```

Ключевое слово INTO определяет таблицу (представление), в которую будут вставлены (обновлены или удалены) строки. Необязательный псевдоним результатов слияния следует за словом AS. Имя исходной таблицы (представления или выражения), из которой будут браться строки, сопоставляемые с исходной таблицей,

должно следовать после USING. После ON задаются условия, которые должны применяться для определения факта совпадения строк в целевой и исходной таблицах.

Спецификации операции слияния задаются с помощью выражений:

- WHEN MATCHED THEN <спецификация слияния обновлением> — указывает, что запись целевой таблицы будет изменена, если существует соответствие между строкой целевой и исходной таблиц;
- WHEN NOT MATCHED THEN <спецификация слияния вставкой> — указывает, что запись будет добавлена в целевую таблицу, если сопоставимая строка не найдена;
- WHEN NOT MATCHED BY SOURCE THEN <спецификация> — определяет операцию редактирования или удаления над строками, которые присутствуют в целевой таблице и отсутствуют в исходной.

Транзакции

С точки зрения языка структурированных запросов транзакция представляет собой набор последовательно выполняемых инструкций SQL, рассматриваемых СУБД как единое целое. Поэтому транзакция должна быть выполнена целиком и полностью или не выполнена совсем.

В современных СУБД различают два способа запуска транзакций: неявный и явный. За неявный запуск отвечает система. Она самостоятельно стартует транзакцию при поступлении команд редактирования данных (INSERT, UPDATE и DELETE) и в ряде других случаев, которые определятся спецификой конкретной СУБД. При любом сбое в процессе выполнения указанных инструкций СУБД автоматически, без какого-либо вмешательства со стороны человека, осуществляет операцию отката и возвращает БД к состоянию на момент запуска транзакции. При корректном завершении команд СУБД фиксирует изменения в таблицах БД. Явный способ старта транзакций — зона ответственности программиста. Для этого стандарт SQL:2003 предоставляет ему три команды: START TRANSACTION, ROLLBACK и COMMIT.

Замечание

В любом случае явный способ управления транзакциями предпочтительнее, т. к. дает программисту дополнительные рычаги по управлению бизнес-логикой БД.

Согласно требованиям стандарта SQL:2003 для уведомления системы о начале транзакции в адрес СУБД следует отправить команду START TRANSACTION. Для этих целей стандарт предусматривает следующую синтаксическую конструкцию:

```
START TRANSACTION [READ ONLY| READ WRITE]
[ISOLATION LEVEL
{READ COMMITED| READ UNCOMMITED | REPETABLE READ | SERIALIZABLE}]
[DIAGNOSTIC SIZE INT]
```

Если вы ненадолго вернетесь к рассмотренной ранее команде SET TRANSACTION, то увидите, что команда на запуск транзакции полностью повторяет параметры на-

стройки уровня изоляции. Так что изменить уровень изоляции можно непосредственно в коде старта транзакции.

Относительно порядка явного запуска транзакции в лагере разработчиков СУБД имеются разногласия. Ряд производителей СУБД (MS SQL Server, Postgre) вместо `START TRANSACTION` используют команду `BEGIN TRANSACTION`, в СУБД Interbase (Firebird, Yaffil) транзакция запускается командой `SET TRANSACTION`. В Oracle и DB2 вообще пошли своим путем и стартуют транзакции неявно. Таким образом, рекомендуемая SQL:2003 команда `START TRANSACTION` поддерживается только MySQL.

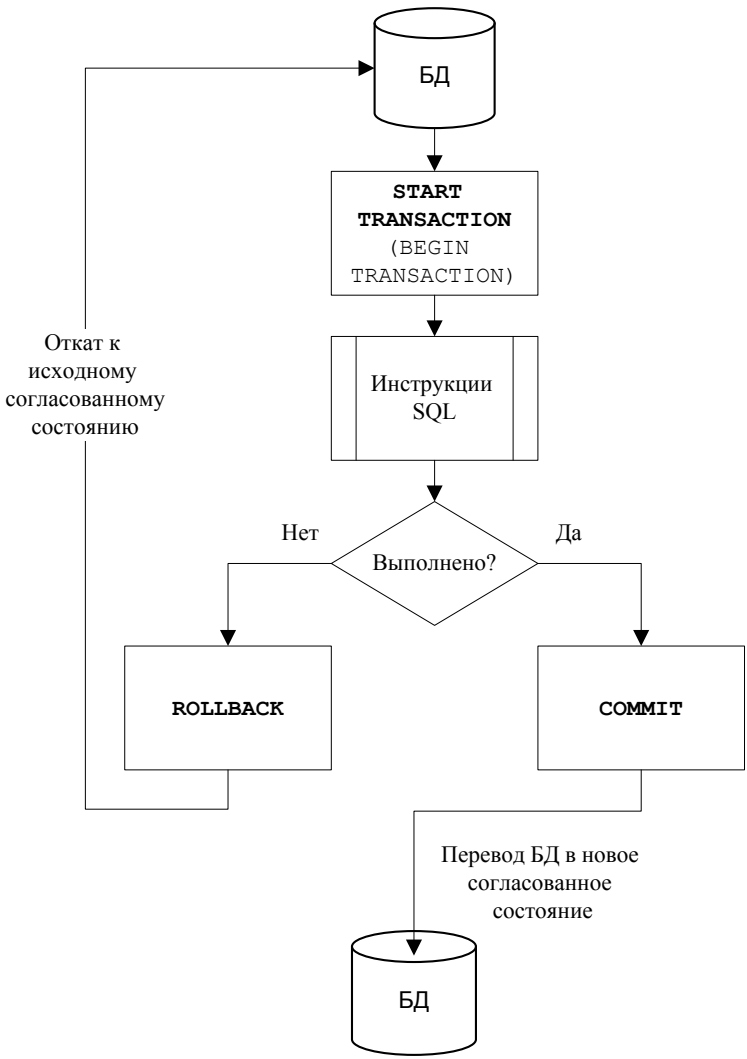


Рис. 13.3. Блок-схема выполнения транзакции

После явного старта транзакции в адрес СУБД передается набор инструкций на языке SQL, на рис. 13.3 они представлены как заранее определенный блок команд. Это может быть хранимая процедура или динамически сгенерированные инструкции SQL. Корректность выполнения всего блока команд обязательно проверяется системой, как минимум на предмет поддержки целостности и непротиворечивости данных. Кроме того, программист также имеет возможность внедрить в блок команд дополнительные управляющие конструкции, в которых он реализует еще один уровень проверки. Если корректность выполнения всего блока SQL не вызывала никакого сомнения, то мы должны подать команду `COMMIT`:

```
COMMIT [AND [NO] CHAIN]
```

Инструкция явным образом завершает транзакцию и фиксирует все изменения в БД, для этого данные переносятся из служебного журнала транзакций в физические файлы таблиц. Команда `AND CHAIN` указывает системе, что со следующей транзакцией следует работать так, будто она является следующим звеном предыдущей. В результате вторая транзакция станет использовать общие настройки первой транзакции (например, уровень изоляции). Команда `COMMIT AND NO CHAIN` (или просто `COMMIT`) задает системе настройки по умолчанию.

Если в ходе выполнения блока команд транзакции выявляются нарушения, то следует воспользоваться командой отката БД к исходному состоянию:

```
ROLLBACK [AND [NO] CHAIN]
[TO SAVEPOINT имя точки сохранения]
```

Получив команду на откат, система отказывается от переноса данных из журнала транзакций в таблицы, таким образом, БД остается в том же состоянии, что в момент запуска этой транзакции. В большинстве систем имеется возможность отката не к началу транзакции, а к одной из промежуточных точек сохранения. Возвратиться к этой точке позволяет команда `TO SAVEPOINT`. Вернувшись к указанной точке, транзакция сможет вновь сделать попытку выполнить оставшийся блок команд.

За определение точки сохранения транзакции отвечает команда:

```
SAVEPOINT имя точки сохранения
```

Указанная инструкция, к сожалению, не поддерживается MySQL и Postgre, но реализована в Oracle, MS SQL Server, Interbase и DB2. В теле транзакции может быть задано несколько точек. Точка принадлежит только одной транзакции и возврат к ней может быть осуществлен только из той транзакции, в которой она была задана.

Механизм хранения точек весьма ресурсоемок, поэтому в SQL предусмотрена команда освобождения ненужных точек:

```
RELEASE SAVEPOINT имя точки сохранения [ONLY]
```

Будьте аккуратны, т. к. согласно стандарту SQL:2003 удалению подлежит не только выбранная точка, но и все точки, созданные после нее. Указанная команда поддерживается только в DB2, Interbase и Firebird. Остальные производители считают, что система должна самостоятельно "расправляться" с точками сохранения.

Из всех упоминаемых в этой книге клиент-серверных СУБД наиболее простым механизмом поддержки транзакций отличается MySQL. Например, возможность частичного отката (с поддержкой точек сохранения) в этой СУБД появилась сравнительно недавно — в версии 4.0.14. Но современный диалект MySQL легко справляется с задачами управления транзакциями. Пример иллюстрирует листинг 13.1.

Листинг 13.1. Сохранение результатов транзакции средствами MySQL

```
BEGIN
SELECT @VAR=SUPPLIER_KEY FROM SUPPLIERS WHERE SUPPLIER='ОАО ПРЕССА'
BEGIN WORK
 INSERT INTO CONTRACTS
 (SUPPLIER_KEY, CONTRACT, CONTRDATE)
 VALUES
 (@VAR, 34, '12/06/2011')
COMMIT
```

Диагностирование ошибок в работе транзакции

В каких случаях система (или программист) принимают решение о необходимости отката транзакции? Ответ на этот вопрос нам уже известен. Во-первых, при возникновении конфликта между параллельно выполняющимися транзакциями. В этом случае транзакции-неудачники должны возвращаться в исходное состояние. Во-вторых, откату подлежат транзакции, которые вводят или удаляют данные, ведущие к нарушению целостности БД (разрушают ключи, связи между таблицами). В-третьих, когда содержащиеся в транзакции пользовательские данные противоречат логике БД (недопустимые значения, отсутствие обязательных значений, нарушение ограничений CHECK). В-четвертых, когда права пользователя, вызвавшего транзакцию, не соответствуют правилам обеспечения безопасности.

Нам осталось разобраться в одном — как программист БД узнает о возникновении исключительной ситуации? Здесь не обойтись без помощи СУБД.

Для диагностирования ошибок в работе транзакции стандарт SQL рекомендует разработчикам СУБД возвращать программисту код исключительной ситуации — SQLSTATE. Физически SQLSTATE представляет собой разделенный на два блока 5-символьный код (рис. 13.4). В первых двух символах содержится код класса ошибки, код начинается с символов '0', '1', '2', '3', '4', 'A', 'B', 'C', 'D', 'E', 'F', 'G' или 'H'. В оставшихся трех — код подкласса, он начинается с символов '5', '6', '7', '8', '9', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'.

Согласно требованиям SQL:2003 программист должен иметь возможность получить код исключительной ситуации после выполнения любой операции с БД. Свидетельство успешного завершения операции — появление нулей во всех разрядах SQLSTATE (класс '00' и подкласс '000'). Это признак того, что откат транзакции

нам не нужен. Все остальные значения говорят о той или иной неприятности. При желании все значения кодов исключительной ситуации вы найдете в документе "Foundation (SQL/Foundation)" стандарта SQL:2003.

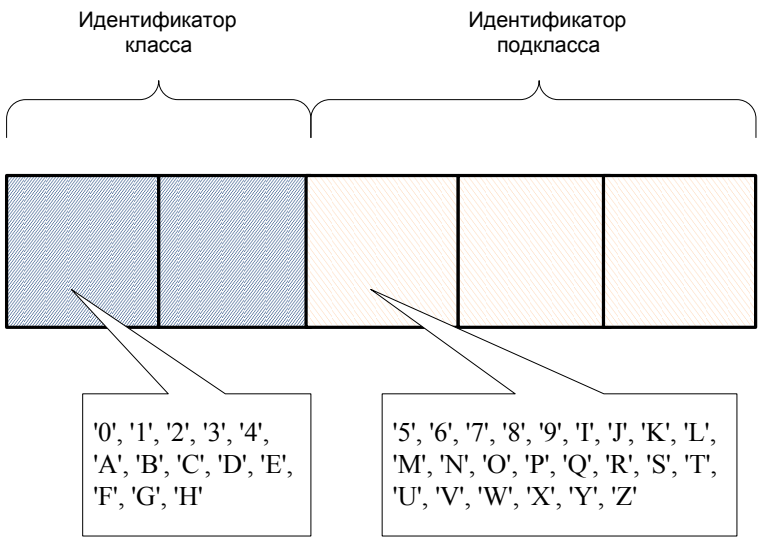


Рис. 13.4. Структура кода `SQLSTATE`

Разработчики СУБД используют свою систему извещения о возникновении ИС, которая зачастую отличается от требований стандарта. В коде PL/SQL СУБД Oracle и в Informix предусмотрена возможность описания раздела `EXCEPTION`, в котором осуществляется вся обработка ошибок. В Interbase и Firebird пока используется код `SQLCODE` (который применялся в устаревшем стандарте SQL:92) и своя дополнительная система информирования `GDSCODE`. В Transact SQL СУБД MS SQL Server для получения информации об ошибке программист обращается к глобальной переменной `@@ERROR`, признаком того, что операция прошла удачно, станет наличие в переменной нулевого значения. Одним словом, сколько СУБД, столько и решений.

В листинге 13.2 приведен пример управления транзакцией в MS SQL Server с помощью системной переменной `@@ERROR`.

Листинг 13.2. Управление транзакцией на диалекте T-SQL

```
DECLARE @ERR INT -- декларируем локальную переменную
SET @ERR=0       -- передаем в @ERR нулевое значение
BEGIN TRANSACTION

DELETE FROM DELIVERIES WHERE DELIVERY_KEY=13
SET @ERR=@ERR+@@ERROR -- проверяем код ИС
DELETE FROM …         -- инструкция удаления
```

```
SET @ERR=@ERR+@@ERROR -- "накапливаем" ошибку
DELETE FROM …          -- инструкция удаления
SET @ERR=@ERR+@@ERROR -- "накапливаем" ошибку

IF @ERR=0
   COMMIT TRANSACTION -- фиксируем транзакцию
ELSE
   ROLLBACK TRANSACTION – откат транзакции
GO
```

В рамках транзакции осуществляется несколько операций удаления. После выполнения каждой инструкции DELETE код ошибки "накапливается" в локальной переменной @ERR. В заключительных строках мы проверяем суммарное значение переменной. Если оно по прежнему равно нулю, то это свидетельствует о корректном выполнении всех инструкций, и мы имеем право зафиксировать транзакцию. В противном случае (при возникновении хотя бы одной ошибки в любом месте кода) осуществляем откат транзакции.

Настройка уровня изоляции

На страницах этой книги мы уже много говорили о роли транзакций в обеспечении восстанавливаемости данных и в организации параллельной обработки данных. Для настройки поведения транзакций в SQL:2003 имеется инструкция SET TRANSACTION. Ее синтаксическую конструкцию вы встретите в следующих строках:

```
SET [LOCAL] TRANSACTION
[READ ONLY| READ WRITE]
[ISOLATION LEVEL
{READ COMMITED| READ UNCOMMITED | REPETABLE READ | SERIALIZABLE}]
[DIAGNOSTIC SIZE INT]
```

Если вы внимательно прочитали *главу 8*, то вам практически не потребуются мои объяснения по данной инструкции. Ключ LOCAL необходим в том случае, если вы хотите указать системе, что настройки транзакции относятся только к локальному серверу, иначе вы будете управлять транзакциями того сервера, к которому в данный момент подключены. Ключ READ ONLY указывает на то, что транзакция намерена осуществлять только чтение данных. Данная настройка действительна лишь для одной транзакции, а затем все вернется к исходному состоянию. Альтернативный ключ READ WRITE разрешает операции чтения и записи. Ключ ISOLATION LEVEL определяет уровень изоляции следующей транзакции, возможные состояния уже были описаны в табл. 8.4, поэтому сейчас мы повторяться не станем. Заключительный ключ DIAGNOSTIC SIZE определяет, сколько сообщений об ошибках в транзакции следует хранить в системе.

Замечание

Инструкция SET TRANSACTION настраивает поведение только одной очередной транзакции, после выполнения которой система автоматически возвратится к настройкам по умолчанию.

Рассмотрим пример настройки транзакции в СУБД Interbase (листинг 13.3). Диалект ISQL позволяет не только определять уровень изоляции, но и конкретизировать особенности доступа к определенным таблицам.

Листинг 13.3. Установка параметров транзакции в Interbase

```
SET TRANSACTION NAME MYTRAN /*имя транзакции MYTRAN*/
ISOLATION LEVEL READ COMMITTED
NO RECORD_VERSION WAIT /*разрешено читать только последнюю версию строк*/
RESERVING SUPPLIERS, CONTRACTS FOR SHARED WRITE /*блокировка для совместной
записи*/
```

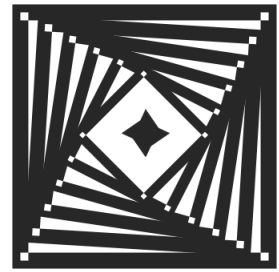
Как видите, частные диалекты SQL зачастую предоставляют бóльший сервис, чем предусмотрено стандартом.

Резюме

В этой главе мы познакомились с основными инструкциями SQL, позволяющими манипулировать данными. Базовые синтаксические конструкции, связанные с операциями вставки INSERT, редактирования UPDATE и удаления DELETE записей поддерживаются всеми диалектами SQL, благодаря чему обеспечивается совместимость СУБД различных производителей.

Для того чтобы обеспечить непротиворечивость и целостность данных, разработчику БД предоставляется возможность управлять процессами запуска, сохранения и отката транзакций. Эти операции осуществляются с помощью инструкций START TRANSACTION, COMMIT и ROLLBACK.

В следующей главе мы заканчиваем изучение основ SQL. Здесь мы встретимся с подъязыками определения данных DDL (Data Definition Language), управления курсором и доступом к данным.



Глава 14

Определение данных в SQL

В этой главе мы изучим порядок создания, изменения и удаления основных объектов базы данных средствами подъязыка определения данных (Data Definition Language, DDL).

Программист, разрабатывающий клиентские приложения, и, тем более, обычный оператор, как правило, не нуждаются в услугах DDL и работают с уже подготовленной БД. Но без знания подъязыка DDL немыслима работа администраторов баз данных, ведь этот подъязык позволяет создавать, редактировать и удалять основные объекты БД, такие как схемы, домены, таблицы.

Весь подъязык DDL опирается на три команды: `CREATE`, `ALTER` и `DROP`, отвечающие соответственно за создание, изменение и удаление объекта БД.

При изучении инструкций определения данных как всегда за основу будет взят стандарт SQL:2003, вместе с тем мы постараемся раскрыть возможности наиболее популярных диалектов языка SQL современных СУБД.

Базы данных (схемы)

В SQL под схемой понимается поименованная коллекция связанных между собой объектов базы данных. Определение схемы приводит к созданию собственно физической базы данных, так что с точки зрения SQL между понятиями "схема" и "база данных" можно поставить знак равенства.

У СУБД различных производителей процесс определения схем существенно отличается. В СУБД INGREES схема формируется средствами утилиты CREATEDB. В Informix Universal Server, MySQL, Microsoft SQL Server, PostgreSQL используется инструкция `CREATE DATABASE`. Изначально в СУБД DB2 и ORACLE схема данных формировалась по умолчанию во время развертывания сервера, и все создаваемые таблицы принадлежали одной схеме. Позднее Oracle отказался от идеи общей схемы и стал позволять создавать отдельные экземпляры БД, для этого в состав диалекта SQL этой СУБД была введена инструкция `CREATE DATABASE`.

Рассмотрим несколько примеров создания новой БД на различных диалектах SQL. Популярные СУБД MySQL и PostgreSQL удовлетворятся одной строкой:

```
CREATE DATABASE BOOKSMARKET
```

При работе с диалектом ISQL СУБД Interbase (Firebird, Yaffil) можно воспользоваться следующей командой:

```
CREATE DATABASE 'C:\Data\IB\BOOKSMARKET'
LENGTH=100 USER IVANOV PASSWORD HELLO-WORLD!
```

В простейшем случае для Microsoft SQL Server также достаточно передать инструкцию `CREATE DATABASE` и имя создаваемой БД, но для создания новой базы данных рассмотренного в *главе 5* книжного магазина с настройками файлов данных и журнала транзакций одной-двух строк кода явно недостаточно, потребуется код, приведенный в листинге 14.1.

Листинг 14.1. Создание БД средствами T-SQL

```
CREATE DATABASE BOOKSMARKET
ON
   (NAME = bookshop_dat,
    FILENAME = 'c:\data\booksmarket.mdf',
    SIZE = 10MB,
    MAXSIZE = 100MB,
    FILEGROWTH = 5MB)
LOG ON
   (NAME = bookshop_log,
    FILENAME = 'c:\data\booksmarket.ldf',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB)
```

В этом примере мы определили не только параметры БД, но и журнала транзакций.

Несмотря на всеобщее признание команды `CREATE DATABASE` производителями СУБД, в SQL:2003 она в явном виде отсутствует, а вместо нее предусмотрена следующая синтаксическая конструкция:

```
CREATE SCHEMA <имя_схемы> [AUTHORIZATION имя владельца]
            [DEFAULT CHARTERSET набор символов по умолчанию]
            [PATH <символьный путь к файлам БД> ]
            [ <дополнительные инструкции>... ]
```

В минимальной нотации достаточно только указать название будущей БД. При необходимости можно определить путь к файлам создаваемой схемы и задать дополнительные опции, связанные с авторизацией и кодировкой символов (если последняя должна отличаться от назначенной по умолчанию):

```
CREATE SCHEMA BOOKSMARKET AUTORIZATION Ivanov
```

В большинстве СУБД предусмотрена возможность перестроения схемы. Такая операция может понадобиться при переносе и изменении размеров БД, пере-

настройке параметров сортировки и в других случаях. Обычно в подобной ситуации следует обращаться к команде ALTER SCHEMA или ALTER DATABASE.

Процесс уничтожения схемы особым изыском не отличается. Практически во всех диалектах SQL имеется инструкция:

```
DROP DATABASE <имя_базы_данных>
```

В свою очередь, стандарт SQL:2003 опирается на команду DROP SCHEMA:

```
DROP SCHEMA <имя_схемы> [CASCADE|RESTRICT]
```

По умолчанию (ключевое слово RESTRICT) инструкция полагает, что удалению подлежит БД, не содержащая никаких объектов, и если это не так, то выполнение команды отменяется. Если следует обеспечить удаление схемы с каскадным уничтожением всех принадлежащих ей таблиц, представлений и других объектов, используем RESTRICT.

Таблицы

После создания базы данных можно переходить к формированию таблиц. Таблица — это основной объект реляционной базы данных, поэтому нам следует уделить внимание средствам подъязыка DDL, нацеленным на создание таблицы. Начнем с рассмотрения синтаксической конструкции SQL:

```
CREATE [ {GLOBAL | LOCAL} TEMPORARY] TABLE имя_таблицы
  ({определение поля | [ограничение таблицы]}., …
  [ON COMMIT {DELETE | PRESERVE} ROWS]):
  определение поля ::= имя_поля {имя домена | тип данных [размер]}
                                [ограничение поля…]
[DEFAULT значение по умолчанию]
[COLLATE имя сравнения]
```

Минимальная конструкция, нацеленная на создание простейшей таблицы, должна включать операторы CREATE TABLE, имя создаваемой таблицы и перечень полей с их размерами и типами. В результате мы получим пустую таблицу. В следующем примере создается таблица PEOPLES, включающая в пять полей. Первое поле целочисленное, второе, третье и четвертое — символьные и последнее поле предназначено для хранения даты (листинг 14.2).

Листинг 14.2. Пример создания таблицы

```
CREATE TABLE PEOPLES
(IndNum  Int,
 Surname Char(30),
 FName   Char(20),
 LName   Char(20),
 BDay    Date)
```

В таблице должно быть одно или несколько определений поля. Определение поля обязательно должно включать имя, тип данных и при необходимости размер поля. В разных диалектах SQL могут различаться названия типов полей и формат хранения данных (например, даты и времени). По умолчанию столбцы таблицы допускают помещение в них значения NULL.

В соответствии со стандартом SQL для таблицы можно задать одно или несколько ограничений (табл. 14.1).

Таблица 14.1. *Ограничения полей таблицы*

Ключ	Описание
NOT NULL	Запрет на вставку в поле неопределенного значения NULL
UNIQUE	Значение поля должно быть уникальным
PRIMARY KEY	Признак первичного ключа. Значение поля должно быть уникальным, оно не может содержать NULL, в таблице это ограничение может использоваться только один раз
CHECK	Ограничение-проверка на допустимое значение поля. В скобках за оператором CHECK указывается предикат, проверяющий допустимость значения
FOREIGN KEY и REFERENCES	Оба ограничения весьма схожи, единственное различие между ними в том, что FOREIGN KEY — ограничение внешнего ключа для таблицы, а REFERENCES — ограничение внешнего ключа для поля. При установке ограничения для таблицы сразу за FOREIGN KEY в скобках необходимо указать перечень столбцов, относящихся к ключу. В остальном синтаксис одинаков. Ограничения внешнего ключа требуют, чтобы все значения, присутствующие во внешнем ключе, соответствовали значениям родительского ключа, т. е. внешний ключ должен иметь ссылочную целостность

Приведенный далее пример создает две таблицы из нашего книжного магазина, рассмотренного в *главе 6* (см. рис. 6.7). Начнем с таблиц поставщиков SUPPLIERS:

```
CREATE TABLE SUPPLIERS
(SUPPLIERS_KEY INT PRIMARY KEY,
 SUPPLIER VARCHAR(40) NOT NULL UNIQUE)
```

В таблице всего два поля: первичный ключ SUPPLER_KEY и имя поставщика SUPPLIER, причем имя должно быть уникальным. Предложенная команда, к сожалению, не определяет порядок присвоения значения первичного ключа новой строке таблицы, поэтому усовершенствуем запрос. Например, диалект T-SQL Microsoft SQL Server позволяет создать автоинкрементный первичный ключ, для этого требуется добавить команду IDENTITY:

```
CREATE TABLE SUPPLIERS
(SUPPLIER_KEY INT IDENTITY (1,1) PRIMARY KEY,
 SUPPLIER VARCHAR(40)  NOT NULL UNIQUE)
```

Согласно ER-модели (см. рис. 6.7) данные о поставщиках должны учитываться в таблице контрактов. Для связи с таблицей поставщиков в контракты включается поле внешнего ключа SUPPLIERS_KEY (листинг 14.3).

Листинг 14.3. Пример создания дочерней таблицы

```
CREATE TABLE CONTRACTS
(CONTRACT_KEY INT IDENTITY (1,1) PRIMARY KEY,
 SUPPLIER_KEY INT NOT NULL,
     FOREIGN KEY (SUPPLIER_KEY) REFERENCES SUPPLIERS(SUPPLIER_KEY),
 CONTRACT VARCHAR(20) NOT NULL,
 CONTRACTDATE DATE NOT NULL)
```

Установка значения по умолчанию производится при помощи оператора DEFAULT. Таким образом, обеспечивается автоматический ввод в столбец некоторого значения, если его явным образом не укажет пользователь в момент вставки новых данных (листинг 14.4).

Листинг 14.4. Определение ограничений на поля создаваемой таблицы

```
CREATE TABLE DELIVERIES
(DELIVERIES_KEY INT IDENTITY (1,1) PRIMARY KEY,
 ORDER_KEY INT NOT NULL,
    FOREIGN KEY (ORDER_KEY) REFERENCES ORDERS(ORDER_KEY),
 BOOK_KEY INT NOT NULL,
    FOREIGN KEY (BOOK_KEY) REFERENCES BOOKS(BOOK_KEY),
 PUBLISHER_KEY INT NOT NULL,
    FOREIGN KEY (PUBLISHER_KEY) REFERENCES PUBLISHERS (PUBLISHER_KEY),
AMOUNT SMALLINT DEFAULT 10, CHECK (AMOUNT>0 AND AMOUNT<=100))
```

В этих же строках вы найдете оператор CHECK, позволяющий наложить определенные ограничения на вводимые в поле данные. В нашем случае мы определили допустимые границы для размера партии книг — теперь поле AMOUNT откажется принимать значения, менее 1 и более 100. Отметим, что порядок применения оператора CHECK отличается в различных диалектах SQL.

Листинг 14.5 демонстрирует еще один аспект применения значений по умолчанию в ISQL СУБД Interbase.

Листинг 14.5. Использование контекстной переменной в Interbase

```
CREATE TABLE ORDERS
  (ORDER_KEY INTEGER PRIMARY KEY,
   CONTRACT_KEY INTEGER NOT NULL,
```

```
  ORDER INTEGER NOT NULL,
  ORDERDATE DATE NOT NULL,
  INSERTEDBY VARCHAR(50) DEFAULT CURRENT_USER)
```

В представленном примере таблица заказов дополнена столбцом INSERTEDBY, в который по умолчанию будет передаваться содержимое контекстной переменной CURRENT_USER. Такое решение позволит нам контролировать, кто именно вносил в таблицу данные об очередном заказе.

Для автоматического ввода значений первичных ключей разработчики СУБД Interbase придумали оригинальное решение — генераторы. Хотя идея генераторов в явном виде отсутствует в стандарте SQL:2003, уделим ей несколько минут.

Для создания генератора нужно воспользоваться следующей инструкцией из листинга 14.6.

Листинг 14.6. Создание генератора значений

```
CREATE GENERATOR ORDER_KEY_GENERATOR;
SET GENERATOR ORDER_KEY_GENERATOR TO 0;
```

В этом примере в первой строчке в базе данных создается генератор с именем ORDER_KEY_GENERATOR, а затем этому генератору присваивается стартовое значение 0. Для обращения к генератору предусмотрена функция GEN_ID, обладающая двумя параметрами, в которые передаются имя генератора и шаг приращения. Для назначения значения для поля первичного ключа таблицы функция GEN_ID вызывается в триггере или хранимой процедуре (листинг 14.7).

Листинг 14.7. Применение генератора в триггере таблицы

```
CREATE TRIGGER ORDERS_BEFOREINSERT
FOR ORDERS
    ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
IF (NEW.ORDER_KEY IS NULL) THEN
    NEW.ORDER_KEY=GEN_ID(ORDER_KEY_GENERATOR, 1);
END
```

Несмотря на некоторые сложности работы с генераторами, у них есть неоспоримое достоинство — они предоставляют программисту новую степень свободы при управлении значениями первичных ключей.

Вернемся к традиционному SQL. Зачастую при обработке данных программисты баз данных нуждаются в создании временных хранилищ данных, в которых будут находиться какие-то промежуточные или сводные результаты. Одно из решений таких задач — использование временных таблиц. Временные таблицы су-

ществуют только на время текущего сеанса SQL и по окончании его автоматически удаляются. При создании временной таблицы требуется воспользоваться ключевым словом TEMPORARY. В зависимости от того, создаем мы глобальные или локальные временные таблицы, можно (но не обязательно) указать ключевые слова GLOBAL или LOCAL соответственно:

```
CREATE GLOBAL TEMPORARY TABLE TmpTable
(IntValue INT, FloatValue FLOAT)
```

Оператор COLLATE предназначен для явного указания порядка сортировки в таблице. Дело в том, что наборы символов, используемые в языке SQL, обычно отсортированы по коду каждого символа, т. е. набор строчных символов следует после прописных символов (как, например, в наборе символов ASCII). Если вас такая ситуация не устраивает, то необходимо самостоятельно создать объект сортировки (CREATE COLLATION) в базе данных и сослаться на него в таблице.

Единожды созданная таблица не является статичным объектом, в SQL предусмотрена возможность модификации структуры уже существующих таблиц. Для этой цели реализована инструкция ALTER TABLE:

```
ALTER TABLE имя_таблицы
{ADD[COLUMN] определение столбца}
| {ALTER [COLUMN] имя изменяющегося столбца}
| {DROP [COLUMN] имя_столбца RESTRICT | CASCADE}
| {ADD определение ограничения для таблицы}
| {DROP CONSTRAINT имя_ограничения RESTRICT | CASCADE}
```

Особенности синтаксической конструкции рассмотрены в табл. 14.2.

Таблица 14.2. *Описание операторов команды* `Alter Table`

Оператор	Описание
ADD [COLUMN]	Добавляет в таблицу новый столбец. Новый столбец определяется так же, как в операторе CREATE TABLE
ALTER [COLUMN]	Используется для создания или отмены значения по умолчанию для столбца
DROP [COLUMN]	Удаляет из таблицы столбец. При использовании параметра RESTRICT перед удалением столбца СУБД проверит наличие ссылок на него из других таблиц и представлений. Если таковые имеются, то столбец не будет удален. Наоборот, при задании параметра CASCADE вместе со столбцом будут удалены все объекты, ссылающиеся на него
ADD	Позволяет добавить к таблице новое ограничение
DROP CONSTRAINT	Удаляет уже существующие ограничения. Если в предложении задан параметр RESTRICT, то в этот момент столбец не должен использоваться как родительский ключ для внешнего ключа другой таблицы. Если передается параметр CASCADE, то внешние ключи, имеющие ссылки или ограничения FOREIGN KEY, уничтожаются

Для добавления к таблице еще одного столбца, воспользуйтесь следующим кодом:

```
ALTER TABLE ORDERS ADD COLUMN ORDERSPRICE INT
```

Для изменения типа уже существующего столбца:

```
ALTER TABLE ORDERS ALTER COLUMN ORDERSPRICE FLOAT
```

И, наконец, для удаления столбца:

```
ALTER TABLE ORDERS DROP COLUMN ORDERSPRICE
```

В заключение подраздела, посвященного инструкции `CREATE TABLE`, приведу еще одну, предусмотренную многими разработчиками СУБД, возможность создания таблицы на основе уже существующей в БД. Практически все современные диалекты SQL позволяют создать такую же таблицу (с тем же набором столбцов), как и исходная. В СУБД Oracle, PostgreSQL, MySQL для этих целей следует написать примерно такую команду:

```
CREATE TABLE DELIVERIES_2
AS
SELECT * FROM DELIVERIES WHERE DELIVERY_KEY=-1
```

Благодаря команде `SELECT` мы получили точно такую же по структуре таблицу, как и таблица поставщиков `DELIVERIES`. Чтобы итоговая таблица оказалась пустой, мы задали заранее невыполнимое условие равенства первичного ключа значению –1.

Положительный эффект указанного способа "кражи" структуры таблицы можно многократно усилить, воспользовавшись огромным потенциалом инструкции `SELECT`. Листинг 14.8 демонстрирует работу T-SQL (MS SQL Server), создающего временную таблицу на основе объединяющего запроса.

Листинг 14.8. Применение временной таблицы в T-SQL

```
SELECT SUPPLIERS.SUPPLIER, CONTRACTS.CONTRACT, CONTRACTS.CONTRDATE
INTO #TEMPTABLE
FROM SUPPLIERS
INNER JOIN CONTRACTS ON SUPPLIERS.SUPPLIER_KEY = CONTRACTS.SUPPLIER_KEY
ORDER BY CONTRACTS.CONTRDATE, CONTRACTS.CONTRACT
```

В результате мы получаем таблицу, содержащую, данные о поставщиках, упорядоченные по дате и номеру контракта.

Индексы

Основное назначение индекса заключается в обеспечении ускоренного доступа к строкам таблицы на основе значений одного или нескольких столбцов. Кроме того индекс позволяет решить еще ряд задач:

- упорядочивание строк таблицы в заданном порядке;

- быстрый поиск данных;
- обеспечение ввода уникальных данных.

Физически индекс представляет собой уникальный набор значений и список указателей на страницы данных, где эти значения находятся. С точки зрения разработчиков СУБД различают несколько базовых способов построения индексов, наиболее популярные из них основаны на идее сбалансированных B-деревьев, чуть реже применяется технология хеширования. У некоторых СУБД есть и эксклюзивные решения, например, у Oracle (начиная с версии 7.3) существует возможность создания индексов на основе битовых карт.

Замечание

Использование индекса требует меньшего числа обращений к жесткому диску, поэтому благодаря индексам существенно возрастает скорость поиска и сортировки данных.

Получив команду на создание индекса, СУБД сканирует таблицу и строит список уникальных значений, которые позже будут задействованы при сортировке и поиске данных.

Система индексирования реализована во всех СУБД без исключения, но в стандартном SQL синтаксическая конструкция создания индексов отсутствует. Вместе с тем диалекты SQL большинства производителей весьма схожи и основаны примерно на следующей упрощенной конструкции:

```
CREATE [UNIQUE] [ASCENDING|DESCENDING] INDEX имя_индекса ON имя_таблицы
(имя_столбца [, … n])
```

В случае создания индекса с опцией `UNIQUE` СУБД прекращает допускать ввод повторяющихся значений в столбец (столбцы), включенные в определение индекса:

```
CREATE UNIQUE INDEX siSUPLIER ON SUPPLIERS (SUPLIER)
```

Довольно часто приходится создавать составные индексы, т. е. состоящие из двух и более столбцов таблицы:

```
CREATE INDEX siOREDR ON OREDRS (OREDR, OREDRDATE)
```

В последнем примере индекс строится по полям с номером и датой заказа.

Некоторые СУБД (DB2, ORACLE, MS SQL) способны создавать особый вид индекса — кластерный (clustered). *Кластерный индекс* — специальный индекс, заставляющий СУБД хранить данные в таблицах именно в том порядке, который задает этот индекс. Применение кластерного индекса способствует значительному сокращению времени доступа к данным, поэтому в таких СУБД основной индекс таблицы всегда нужно определять как `CLUSTERED`:

```
CREATE CLASTERED INDEX siPUBLISHER ON PUBLISHERS (PUBLISHER)
```

В таблице может быть назначен только один кластерный индекс. Кластерный индекс вместо хранения простого ключевого значения, ссылающегося на данные в таблице, хранит целую строку.

Замечание

В кластерный индекс включаются именно те столбцы и именно в таком порядке, который соответствует наиболее часто используемому представлению таблицы.

Допускается создавать индексы, осуществляющие сортировку данных в обратной последовательности при помощи ключа DESCENDING:

```
CREATE DESCENDING INDEX siPUBLISHER_DESC ON PUBLISHERS (PUBLISHER)
```

Перенастройка индекса производится инструкцией ALTER INDEX. Особенности команды зависят от эксплуатируемой СУБД, например MS SQL Server позволяет отключать индекс:

```
ALTER INDEX siPUBLISHER_DESC ON PUBLISHERS DISABLE
```

Такая операция может пригодиться, когда в короткий промежуток времени большей части данных, хранящихся в таблице издательств, грозит удаление или изменение. Мы временно отключаем индекс, чтобы приостановить процесс индексирования и не перегружать сервер баз данных. Если вы предположили, что для включения индекса в MS SQL Server потребуется команда ENABLE — вы ошиблись, в Transact-SQL для этой задачи применяется инструкция ALTER INDEX REBUILD. Однако если вы работаете в Oracle, то ENABLE как раз то, что надо. В Interbase включением/отключением ведают ключевые слова ACTIVE/INACTIVE.

Синтаксис удаления индекса включает в себя ключевые слова DROP INDEX и непосредственно имя таблицы и индекса, разделенные точкой:

```
DROP INDEX имя_индекса
```

Возрастание объема индексируемых данных и увеличение числа полей, входящих в индекс, задействует больше ресурсов компьютера в процессе индексирования таблицы. Однако если таблица не является статичным объектом и хранящиеся в ней данные постоянно изменяются, то СУБД с каждой операцией вставки или редактирования данных будет вынуждена обновлять индексы, т. е. повторно сканировать таблицу. Так что разработчик баз данных должен уметь балансировать на грани разрешения противоречия между скоростью доступа к фрагменту данных (требующего максимального количества индексов) и скоростью обновления данных в таблице (не любящих индексов вообще). Поиском разрешения указанного противоречия разработчики баз данных занимались, занимаются и будут заниматься всегда, т. к. этот процесс творческий, как и любая другая задача поиска золотой середины. Перечислим некоторые рекомендации по определению индексов.

Целесообразно индексировать:

- столбцы, входящие в директивы сортировки (ORDER BY) и группировки данных (GROUP BY). Причем последовательность определения полей в индексе должна соответствовать последовательности полей в директиве;
- если большей частью времени таблица не редактируется, а используется только для выборки данных (например, телефонный справочник города), то индексируйте ее "вдоль и поперек". А при пополнении этой таблицы большой порцией данных целесообразно вообще удалить старые индексы и создать их вновь после внесения новых записей.

Не следует индексировать:

- столбцы первичного и внешнего ключа, т. к. большинство СУБД индексируют их автоматически;
- столбцы типа BLOB и столбцы, построенные на основе данных этого типа;

- символьные поля большого размера;
- поля, допускающие неопределенные значения NULL;
- часто изменяемые столбцы, т. к. это замедлит выполнение операций модификации данных;
- столбцы, которые содержат много повторяющихся значений.

Домены

Если вы видите, что для корректного описания допустимых значений, передаваемых в то или иное поле таблицы, возможностей типа данных не хватает, то это признак того, что пора воспользоваться услугами доменов. Если типизация отвечает за физический формат хранения данных, то домен позволяет накладывать дополнительные логические ограничения на значения, передаваемые в столбец таблицы. Поэтому механизм доменов существенно обогащает наши возможности по определению ограничений.

Для описания домена SQL:2003 предусматривает синтаксическую конструкцию:

```
CREATE DOMAIN <имя_домена> [ AS ] <предопределенный тип данных>
[ <значение по умолчанию> ]
[ <ограничения домена>[,...] ]
[ <правила сравнения> ]
```

Допустим, что нам следует создать домен, ограничивающий ввод значений диапазоном от 1 до 100 и по умолчанию принимающий значение, равное единице, тогда следует написать примерно такие строки:

```
CREATE DOMAIN DOMAIN_SMALLRANGE AS INTEGER
DEFAULT 1
CHECK (VALUE BETWEEN 1 AND 100)
```

Созданный домен сохраняется в системном каталоге БД, поэтому его можно применять при определении полей множества таблиц.

Например, в Interbase услуги домена могут понадобиться при определении первичного ключа инкрементного типа во всех таблицах нашей базы данных (листинг 14.9).

Листинг 14.9. Применение домена при настройке ограничений поля таблицы

```
CREATE DOMAIN DOMAIN_PR_KEY AS INTEGER NOT NULL
CONSTRAINT CHECK(VALUE>0)
    ...
CREATE TABLE GENRES (
             GENRE_KEY DOMAIN_PR_KEY,
              …);
```

Столбцы, основанные на определении домена, наследуют все атрибуты домена: тип данных, состояние неопределенности NULL и ограничения CHECK.

Еще одно преимущество домена в том, что при необходимости изменить ограничения во всех таблицах достаточно перенастроить всего один домен с помощью инструкции ALTER DOMAIN. В завершении рассказа о доменах упомяну, что для удаления домена следует воспользоваться инструкцией DROP DOMAIN.

Представления (виртуальные таблицы)

Представление (view) — это результат одной или нескольких реляционных операций над таблицами БД, в результате которых получается новое отношение. С точки зрения пользователя представление выглядит как вполне обычная таблица, хотя на самом деле является виртуальным объектом, полученным в результате выполнения запроса. Возвращаемые представлением строки, как правило, не подлежат редактированию, за исключением случая, когда представление собирается всего из одной таблицы.

Для создания представления необходимо отработать следующую синтаксическую конструкцию:

```
CREATE [RECURSIVE] VIEW <имя_представления>
AS <запрос на основе SELECT> [WITH [CASCADED|LOCAL] CHECK OPTION]
```

Выражение CRETAE VIEW создает представление таблицы или нескольких таблиц, иногда называемых виртуальной таблицей. Наличие команды RECURSIVE говорит о том, что создается представление, которое будет получать данные из себя самого. Определение представления содержит обычный запрос на базе оператора SELECT, но с некоторыми ограничениями: например нельзя использовать оператор ORDER BY, упорядочивающий набор строк. При объединении двух и более таблиц нужно быть внимательным при совпадении имен полей таблиц. В таких случаях команда SELECT * … (выбор всех полей) недопустима и совпадающее имя следует переименовать при помощи оператора AS. Явный список имен полей после оператора SELECT необходимо описать и в случае, если поля содержат вычисляемые значения или существуют объединенные поля с разными именами в соответствующих им таблицах.

Собственно текст запроса, создающий представление, описывается на базе инструкции SELECT (см. *главу 13*). В листинге 14.10 приведен пример создания простейшего представления, объединяющего все данные из таблиц поставщиков и контрактов.

Листинг 14.10. Пример создания представления

```
CREATE VIEW VW_SUPPLIERS_CONTRACTS
   SELECT SUPPLIERS_KEY, SUPPLIER, CONTRACT_KEY, CONTRACT, CONTRACTDATE
   FROM SUPPLIERS
   INNER JOIN CONTRACTS ON SUPPLIERS.SUPPLIER_KEY =
                                   CONTRACTS.SUPPLIER_KEY
```

К представлению, как и к обычной таблице, можно обратиться через ключевое слово SELECT:

```
SELECT * FROM VW_SUPPLIERS
```

Группа команд [WITH [CASCADED|LOCAL] CHECK OPTION] применяется только в представлениях, допускающих обновление своих данных, точнее данных в соответствующих локальных таблицах. Предложение WITH CHECK OPTION указывает на то, что редактироваться будут только те данные, которые показаны в текущем представлении. Параметры CASCADED и LOCAL уточняют порядок обновления. Если в предложение включен параметр LOCAL, то проверка допустимости редактирования/удаления будет осуществляться только в текущем представлении, параметр CASCADED требует, чтобы проверка проводилась не только в локальном, но и во всех дочерних представлениях.

Внимание!

При описании представления следует соблюдать ряд ограничений, например, запрещено применять операторы UNION, EXCEPT, INTERSECT, GROUP BY, HAVING, DISTINCT.

Представления — весьма полезный инструмент. Пользователь воспринимает представление как реальную таблицу и даже не задумывается о том, что это результат выполнения запроса к одной или нескольким таблицам. Благодаря механизму представлений разработчик баз данных получает возможность, во-первых, организовать доступ к информации в наиболее удобном для пользователя виде, во-вторых, скрыть от пользователя ненужные ему данные и, в-третьих, упростить свою работу, создавая представления для базовых отношений между таблицами и обращаясь к ним по мере необходимости вместо построения одних и тех же запросов.

Нетрудно догадаться, что представление редактируют с помощью инструкции ALTER VIEW, а удаляют — DROP VIEW.

Хранимые процедуры

Хранимая процедура (stored procedure) — это полноценный объект базы данных, нашедший широкое применение в архитектуре "клиент-сервер". Физически процедура представляет собой подпрограмму на языке SQL, сохраняемую в системном каталоге сервера. Благодаря тому, что процедура выполняется на сервере (в роли которого выступают наиболее мощные компьютеры), она не требует ресурсов от клиентских машин, что в значительной степени повышает производительность.

В код хранимой процедуры выносятся как наиболее часто вызываемые, так и максимально "тяжеловесные" и трудоемкие команды: вставка, модификация, удаление и отбор данных из таблиц. Благодаря тому, что процедура может содержать блоки команд на языке SQL, переменные, условные операторы и циклы, она позволяет строить весьма сложные логические конструкции, решающие серьезные задачи.

Независимо от диалекта SQL того или иного сервера баз данных при создании новой хранимой процедуры программист определяет уникальное имя процедуры и, при необходимости перечень входных и возвращаемых параметров.

Хранимую процедуру можно вызывать из другой хранимой процедуры, из триггера базы данных и из клиентского приложения. Для этого передается имя процедуры и при необходимости список ее параметров.

В следующих строках приведена обобщенная синтаксическая конструкция, позволяющая создать хранимую процедуру:

```
CREATE PROCEDURE имя_процедуры
[параметр1 тип_параметра, …, параметрN тип_параметра]
AS
<ТЕЛО ПРОЦЕДУРЫ>
```

А листинг 14.11 иллюстрирует пример реальной хранимой процедуры, редактирующей название поставщика в таблице SUPPLERS для СУБД Oracle.

Листинг 14.11. Создание хранимой процедуры на диалекте PL/SQL СУБД Oracle

```
CREATE PROCEDURE PROC_SUPPLIERS_UPDATE
(KEY INT, SUPPLIER_NAME CHAR(40))
BEGIN
   UPDATE SUPPLIERS SET SUPPLIER=SUPPLIER_NAME
   WHERE SUPPLIER_KEY=KEY
END
```

Для вызова этой процедуры потребуется набрать такие строки кода:

```
CALL PROC_SUPPLIERS_UPDATE(112, 'ОАО Аметист')
```

Для сравнения синтаксиса рассмотрим процедуру на диалекте T-SQL Microsoft SQL Server, на этот раз мы добавляем в таблицу SUPPLERS нового поставщика и возвращаем его первичный ключ (листинг 14.12).

Листинг 14.12. Создание хранимой процедуры средствами T-SQL

```
CREATE PROCEDURE PROC_SUPPLIERS_INSERT
@SUPPLIER_NAME VARCHAR(40), @KEY INT OUT
AS
INSERT INTO SUPPLIERS
   (SUPPLIER)
    VALUES
   (SUPPLIER_NAME)

SET @KEY=@@IDENTITY
GO
```

Еще один пример процедуры, добавляющей новую запись в таблицу издательств, но теперь на диалекте ISQL сервера Interbase, иллюстрирует листинг 14.13.

Листинг 14.13. Хранимая процедура в ISQL СУБД Interbase

```
CREATE PROCEDURE PROC_PUBLISHERS_INSERT
(in_PUBLISHER VARCHAR(40))
AS
BEGIN
  INSERT INTO PUBLISHERS
    (PUBLISHER)
  VALUES
    (:in_PUBLISHER);
END
```

Несмотря на несовместимость диалектов Oracle, MS SQL и Interbase между ними все-таки много общего — и все это благодаря существованию стандарта SQL. Поэтому специалисту, изучившему SQL любой серьезной СУБД, для перехода с одного диалекта на другой особого переучивания не потребуется.

Триггеры

Триггер (trigger) — это особый вид хранимой процедуры сервера, представляющий собой набор инструкций на языке SQL, автоматически выполняющихся в момент событий вставки (INSERT), редактирования (UPDATE) или удаления (DELETE) данных из таблиц. Наличие триггеров позволяет нам построить еще одну "линию обороны" для поддержки целостности и непротиворечивости данных.

Замечание

Существенное отличие триггера от хранимой процедуры в том, что за вызов триггера отвечает только сервер SQL, а клиентское приложение никогда не сможет управлять работой триггеров. Благодаря этой особенности триггер попросту незаменим в вопросах поддержания внутренней логики базы данных.

Стандарт SQL:2003 рекомендует разработчикам СУБД обеспечить возможность описывать триггер, реагирующий не только на факт свершившегося события (AFTER), но и предоставить возможность вызова триггера перед событием (BEFORE). Таким образом, в идеальной реализации языка SQL должна иметься возможность создания трех пар триггеров:

- BEFORE INSERT и AFTER INSERT;
- BEFORE UPDATE и AFTER UPDATE;
- BEFORE DELETE и AFTER DELETE.

Вполне естественно, что для создания триггера на различных платформах SQL применяются разные синтаксические конструкции, но независимо от диалекта все они стартуют с команды CREATE TRIGGER. Стандарт SQL:2003 предлагает производителям придерживаться конструкции:

```
CREATE TRIGGER имя_триггера
{BEFORE | AFTER) (DELETE | UPDATE | INSERT [OF имя_столбца])}
ON имя_таблицы
[FOR EARCH {ROW|STATEMENT}]
[WHEN дополнительные условия выполнения триггера]
BEGIN ATOMIC
 Блок кода
END
```

Как видите, стандарт допускает обработку событий для отдельного столбца, например, строки SQL из листинга 14.14 указывают, что триггер вызывается только при попытке обновить столбец с именем AMOUNT таблицы поставок DELIVERIES.

Листинг 14.14. Создание триггера согласно рекомендациям стандарта SQL:2003

```
CREATE TRIGGER TR-DELIVERIES-UPDATE-AMOUNT
              AFTER UPDATE OF AMOUNT
ON DELIVERIES
BEGIN ATOMIC
…
END
```

Впрочем, в различных диалектах SQL реакция на событие, связанное с отдельным столбцом, сильно отличается от рекомендаций SQL:2003. В Transact-SQL СУБД Microsoft SQL Server для этих целей применяют встроенные функции Update() или UpdatedColumns(). Первая возвращает значения TRUE/FALSE в зависимости от того, был ли изменен столбец, вторая позволяет проверить на обновления сразу несколько столбцов (листинг 14.15).

Листинг 14.15. Создание триггера на диалекте T-SQL

```
CREATE TRIGGER TR-DELIVERIES-UPDATE-AMOUNT
ON DELIVERIES
FOR UPDATE
AS
IF UPDATE(AMOUNT)
BEGIN
…
END
```

Вновь возвращаемся к инструкции CREATE TRIGGER, рекомендованной стандартом. Теперь обратите внимание на строку FOR EARCH. Следующие за командой ключевые слова определяют порядок выполнения блока кода триггера в зависимости от числа строк таблицы, подвергшихся модификации. Допустим, что наш триггер вызывается после операции удаления (AFTER DELETE) строк из таблицы. За один раз из таблицы может быть удалена одна или целая группа записей, и если вы выберете команду ROW, то содержащийся в триггере блок кода будет выполнен для каждой из удаляемых записей. Другими словами, если из таблицы вычеркивается 100 строк, то триггер выполнится 100 раз. Напротив, команда STATEMENT определяет то, что, независимо от числа удаляемых записей, триггер должен быть выполнен всего один раз.

Внимание!

При работе с триггерами все новички, не сговариваясь, допускают одну и ту же классическую ошибку — забывают о том, что за одну операцию триггер может обработать более одной записи!

Практически во всех диалектах языка SQL в момент срабатывания триггера системой управления базами данных создается одна, реже две временные логические таблицы. В любом случае первая из таблиц хранит содержимое обслуживаемой таблицы до возникновения события (например, вставки), вторая — после изменений в таблице. Мы можем сравнивать содержимое этих таблиц и оценивать результаты выполнения операции. У разных производителей указанные таблицы называются по-разному, в Oracle, Interbase и FireBird — OLD и NEW, в DB2 имена таблиц назначает программист.

Приведем пример триггера Interbase который при вводе новой записи в таблицу заказов ORDERS использует временную таблицу NEW для проверки наличия данных о контракте на поставку (листинг 14.16).

Листинг 14.16. Создание триггера в ISQL

```
CREATE TRIGGER TR_ORDERS_INSERT
FOR ORDERS
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
   IF (NEW.CONTRACT_KEY IS NULL) THEN EXCEPTION CONTRACT_ERROR
END
```

Если в новой таблице отсутствуют данные о ключе контракта CONTRACT_KEY — триггер генерирует исключительную ситуацию.

В Microsoft SQL Server дела обстоят несколько иначе. Здесь перед удалением строки создается логическая таблица DELETED, в которую автоматически передаются удаляемые строки. Если же мы добавляем новые или редактируем существую-

щие строки, то новые (модифицированные) строки сначала направляются в логическую таблицу INSERTED. В коде триггера мы имеем возможность проанализировать содержимое логической таблицы и, если оно нас по какой-то причине не устраивает, отказаться от изменений (осуществить откат транзакции).

Допустим, что нам необходимо гарантировать, чтобы дата приходного ордера поставки книг в магазин всегда была позднее, чем дата заключения контракта на эту поставку, в противном случае запись не должна попасть в таблицу ORDERS. Листинг 14.17 иллюстрирует решение данной задачи.

Листинг 14.17. Использование логической таблицы INSERTED в триггере T-SQL

```
CREATE TRIGGER TR_ORDERS_INSERTUPDATE
 ON ORDERS FOR INSERT, UPDATE
AS
BEGIN
  DECLARE @D_ORDER DATETIME, @KEY INT
  SELECT  @D_ORDER =ORDERDATE, @KEY=CONTRACT_KEY FROM INSERTED
  DECLARE @D_CONTRACT DATETIME
  SELECT  @D_CONTRACT=CONTRDATE FROM CONTRACTS WHERE CONTRACT_KEY=@KEY

  IF @D_ORDER<@D_CONTRACT -- проверка корректности дат
    BEGIN
     IF @@TRANCOUNT>0 ROLLBACK TRAN -- откат транзакции
     RAISERROR(100106,16,10) -- несоответствие дат!
    END
END
```

Для решения поставленной задачи мы опрашиваем логическую таблицу INSERTED, чтобы узнать первичный ключ контракта @KEY и дату приходного ордера @D_ORDER. Получив ключ контракта, обращаемся к таблице контрактов и выясняем дату его заключения @D_CONTRACT. Затем сравниваем дату ордера и дату контракта, и если не соблюдается условие @D_ORDER<@D_CONTRACT, то осуществляем откат транзакции. По правилам хорошего тона программирования при откате транзакции следует сгенерировать сообщение об ошибке, в T-SQL эта задача возложена на функцию RAISERROR().

В блоке кода триггера мы имеем право вызывать описанные в базе данных хранимые процедуры. На диалекте PostgreSQL эта операция выглядит следующим образом:

```
CREATE TRIGGER TR_DELIVERIES
BEFORE INSERT OR UPDATE ON DELIVERIES
  FOR EARH ROW
  EXECUTE PROCEDURE PRC_CONTROL_BOOKS()
```

Еще одна область применения триггеров — ведение аудита изменений в наиболее важных таблицах базы данных. Допустим, что намерены отслеживать все операции, связанные с таблицей контрактов, в таком случае в СУБД ORACLE мы смогли бы создать триггер, приведенный в листинге 14.18.

Листинг 14.18. Триггер на диалекте PL/SQL

```
CREATE TRIGGER TR_CONTRACTS_AUDIT
   AFTER DELETE OR INSERT OR UPDATE ON CONTRACTS
   FOR EARCH ROW
BEGIN
  INSERT INTO CONTRACTS_LOG
    (USERNAME, EVENTDATE, OLDCONTRACT, NEWCONTRACT)
  VALUES
    (USER, SYSDATE, :OLD.CONTRACT, :NEW.CONTRACT)
END TR_CONTRACTS_AUDIT;
```

Помимо названия контракта в таблице CONTRACTS_LOG мы сохраняем имя пользователя, изменявшего данные, и дату изменений.

В базах данных Interbase триггеры зачастую применяются для определения значений для столбцов первичного ключа (листинг 14.19).

Листинг 14.19. Определение значения для первичного ключа таблицы Interbase

```
CREATE TRIGGER TR_GENRES_BF-INSERT FOR GENRES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
IF (NEW.GENRE_KEY IS NULL) THEN
    NEW.GENRE_KEY =GEN_ID(SYSNUMBER,1);
END
```

Триггер вызывается перед событием вставки новой записи в таблицу GENRES, значение первичного ключа возвращает функция GEN_ID, полученное значение вставляется в поле GENRE_KEY.

Внимание!

Триггер — весьма сильное оружие, злоупотребление которым может навредить базе данных. Достоинство триггера — способность определения логики работы таблиц, недостаток — скрытость этой логики от конечного пользователя, последний может просто не понять, что от него хочет таблица. Еще один повод для беспокойства состоит в том, что триггеры способны заметно снизить производительность БД, особенно в случае, когда в таблицах редактируется или удаляется большое количество записей.

При написании кода триггера следует учитывать два ограничения:

1. В теле триггера запрещается создавать, изменять или удалять объекты БД. Другими словами, нельзя использовать команды CREATE, ALTER и DROP.
2. Запрещено управлять правами доступа к объектам БД, т. е. выполнять команду GRAND или REVOKE.

Завершая тему о возможностях SQL по созданию объектов БД, упомянем инструкции по изменению и удалению триггеров. Для редактирования блока кода, содержащегося в триггерах, следует воспользоваться инструкцией ALTER TRIGGER, а вопросами удаления ведает инструкция DROP TRIGGER.

Курсоры

Обычный, построенный на основе инструкции SELECT, запрос сразу возвращает нам некоторый результирующий набор записей. Однако зачастую логика разрабатываемой БД требует дополнительной обработки отдельной строки в результирующем наборе, на которую не способна команда SELECT. В таких случаях на помощь приходит *курсор* (cursor), представляющий собой указатель на отдельную запись в отношении. С помощью курсора мы сможем последовательно перебирать строки полученного отношения, осуществляя с ними дополнительные операции, например редактирования.

Замечание

Курсоры применяются в программных конструкциях SQL, в которых требуется обеспечить поочередный доступ к строкам отношения. Преимущество построчного доступа в том, что он позволяет программисту работать с данными, опираясь не только на SQL, но и на процедурные элементы программирования.

Для работы с курсорами последовательно применяют четыре команды SQL:

1. Объявить курсор (DECLARE CURSOR);
2. Открыть курсор (OPEN);
3. Считать данные из курсора (FETCH);
4. Закрыть курсор (CLOSE).

Все операции начинаются с объявления курсора. Процесс объявления весьма схож с организацией запроса данных на основе инструкции SELECT. Синтаксическая конструкция объявления курсора в SQL:2003 выглядит следующим образом:

```
DECLARE имя_курсора [SENSETIVE|INSENSITIVE|ASENSETIVE]
                    [[NO] SCROLL] CURSOR
FOR инструкция на основе SELECT
[FOR {READ ONLY | UPDATE [OF имя_столбца, … ]}]
```

Следующее за именем курсора предложение SQL [SENSETIVE|INSENSITIVE|ASENSETIVE] определяет порядок доступа курсора к данным. Параметр SENSITIVE устанавливает такой режим работы, когда чтение дан-

ных курсором осуществляется непосредственно из таблицы (представления), благодаря этому курсор становится чувствительным ко всем вносимым другими пользователями изменениям в результирующем наборе данных. Значение INSENSITIVE напротив максимально снизит чувствительность курсора, с этой целью результирующий набор помещается в отдельную временную таблицу, недоступную другим пользователям. Наконец, режим ASENSETIVE указывает на то, что порядок взаимодействия курсора с данными определяется СУБД самостоятельно.

Замечание

Курсоры большинства СУБД по умолчанию работают в режиме INSENSETIVE. В этом случае курсор запоминает состояние БД на момент своего открытия. Для того чтобы увидеть изменения, сделанные другими пользователями, следует закрыть и повторно открыть курсор.

Параметр SCROLL говорит о том, что курсор способен извлекать строки в любом направлении (от первой к последней записи или, наоборот, от последней к первой), причем направление перемещения курсора допускается изменить в любой момент времени. Обратный параметр NO SCROLL допускает только последовательный доступ к строкам от первой к последней. По возможности разработчику БД следует отдавать предпочтение однонаправленным курсорам, т. к. они менее ресурсоемкие.

Как всегда у различных диалектов SQL имеются свои нюансы по работе с курсорами. Так, DB2, Interbase и Oracle нацелены на работу с однонаправленным курсором, пробегающим строки таблиц от первой к последней. Несмотря на большую схожесть с Oracle, СУБД PostgreSQL способна прокручивать курсоры во всех направлениях, то же самое можно сказать и о курсорах MS SQL Server.

Параметр READ ONLY запрещает любую модификацию данных. Использование ключевого слова UPDATE [OF имя_столбца, ...] напротив, позволяет явно определять столбцы, которые могут быть обновлены.

Для объявления курсора, способного поочередно просмотреть все строки отношения из таблиц поставок и заказов (OREDRS и DELIVERIES), достаточно написать строки кода, приведенные в листинге 14.20.

Листинг 14.20. Объявление курсора в T-SQL

```
DECLARE CUR_DELIVERIES NO SCROLL CURSOR
FOR SELECT DELIVERY_KEY FROM DELIVERIES
        INNER JOIN ORDERS ON ORDERS.ORDER_KEY=DELIVERIES.ORDER_KEY
        WHERE ORDERS.ORDERDATE>'01.01.2010'
FOR READ ONLY
```

Для получения права на считывание данных из курсора его следует открыть командой OPEN:

```
OPEN имя_курсора
```

Для извлечения данных из открытого курсора применяется предложение FETCH. Вот синтаксис инструкции:

```
FETCH
[{NEXT | PRIOR | FIRST | LAST {ABSOLUTE n | RELATIVE n}}
FROM
имя_курсора
INTO имя_переменной1[,…]
```

Если при объявлении курсора опущено ключевое слово SCROLL, то единственным способом считывания является последовательное прохождение полученного набора данных. Для нормального порядка считывания (от первой записи к последней) применяют ключевое слово NEXT. Если же во время декларации курсора вы указали, что считывание может осуществляться в любом порядке, то для обращения к предыдущей записи разрешается применять ключевое слово PRIOR. Инструкции FIRST и LAST соответственно перемещают курсор к самой первой и к самой последней записи в результирующем наборе. Ключевое слово ABSOLUTE n определяет, что считывание начнется с *n*-й строки результирующего набора, RELATIVE n — вызывает считывание *n*-й строки относительно текущей строки. Если аргумент положителен, то осуществляется перемещение вперед, при отрицательном значении — назад. Необязательное ключевое слово FROM, стоящее перед именем курсора, делает код более читабельным. За командой INTO следуют имена переменных, в которые передаются извлеченные из строки данные. Количество и тип переменных должны четко соответствовать количеству и типу столбцов, описанных в инструкции SELECT во время объявления курсора.

Команда CLOSE закрывает открытый в данный момент курсор. Синтаксис инструкции требует указать имя открытого курсора:

```
CLOSE имя_курсора
```

Закрытие курсора освобождает все связанные с ним ресурсы, а также отключает все ранее установленные блокировки. Работа с закрытым курсором невозможна.

Пример перебора строк курсора с именем CUR_DELIVERIES в MS SQL Server иллюстрирует листинг 14.21.

Листинг 14.21. Перебор строк отношения с помощью курсора

```
DECLARE @DELIVERY_KEY INTEGER --объявили переменную @DELIVERY_KEY
OPEN CUR_DELIVERIES

FETCH FIRST FROM CUR_DELIVERIES INTO @DELIVERY_KEY
WHILE @@FETCH_STATUS=0 ––перебор строк, пока не достигли последней
BEGIN
   --операции со строками, например заполнение другой таблицы или отчета
   --на основе значений первичного ключа @DELIVERY_KEY
      FETCH NEXT FROM CUR_DELIVERIES INTO CUR_DELIVERIES
END
CLOSE CUR_DELIVERIES
DEALLOCATE CUR_DELIVERIES --освобождение ресурса
```

В T-SQL о статусе операций чтения данных курсором можно судить по глобальной переменной `@@FETCH_STATUS`. Пока операции чтения выполняются успешно, в указанной переменной мы будем обнаруживать нулевое значение. Любое другое значение указывает на сбой чтения, например, при достижении конца набора. В Oracle аналогичную задачу решают атрибуты курсора `%FOUND` (чтение успешно) и `%NOTFOUND` (сбой чтения). Цикл чтения данных из аналогичного курсора на языке PL/SQL СУБД Oracle приведен в листинге 14.22.

Листинг 14.22. Чтение данных из курсора на диалекте PL/SQL

```
BEGIN
OPEN CUR_DELIVERIES
LOOP
    FETCH CUR_DELIVERIES INTO DELIVERY_KEY;
    EXIT WHEN CUR_DELIVERIES %NOTFOUND;
    -- другие операции
    -- с курсором
END LOOP;
CLOSE CUR_DELIVERIES;
END;
```

ВНИМАНИЕ!

Старайтесь писать такой код, чтобы все операции с курсорами выполнялись в максимально сжатые сроки. Никогда не следует долго держать курсор в открытом состоянии — в SQL-программировании это признак дурного тона.

Управление доступом к данным

Стандарт SQL не предусматривает никаких команд, предназначенных для создания учетной записи пользователя. Поэтому практически во всех СУБД для этих целей придуманы свои собственные методы. Обычно пользователя можно создать интерактивно или через системные хранимые процедуры. Есть и индивидуальные решения. Разворачиваемая на базе операционной системы Microsoft Windows Server (2000/2003/2008) СУБД Microsoft SQL Server в состоянии оперировать учетными записями пользователей, зарегистрированных на контроллере домена, таким образом, эта СУБД встраивается в целостную политику безопасности домена. Примерно так же поступают СУБД Ingres, Informix и DB2, они умеют извлекать учетную запись пользователя из недр операционной системы.

Хотя стандарт SQL не определяет порядок создания учетных записей, но в нем предусмотрен механизм, ограничивающий права пользователей по доступу к объектам СУБД и данным, расположенным в таблицах и представлениях.

Управление наборами привилегий

Системы безопасности подавляющего большинства современных СУБД для определения прав пользователей используют заранее сформированные списки привилегий. Администратор системы обладает самым широким перечнем полномочий, разработчик БД имеет меньше прав, полномочия обычного пользователя еще более сужены. В терминах SQL перечень заранее сформированных полномочий называют *ролью* (role). Функциональная нагрузка, возлагаемая на роли СУБД, во многом схожа с задачами групп пользователей в Windows. Роль значительно упрощает процесс администрирования подсистемы авторизации СУБД — вновь появившийся пользователь просто наделяется правами из той или иной роли. Это гораздо удобнее, чем явное описание перечня полномочий для каждого пользователя в отдельности (рис. 14.1 и 14.2).

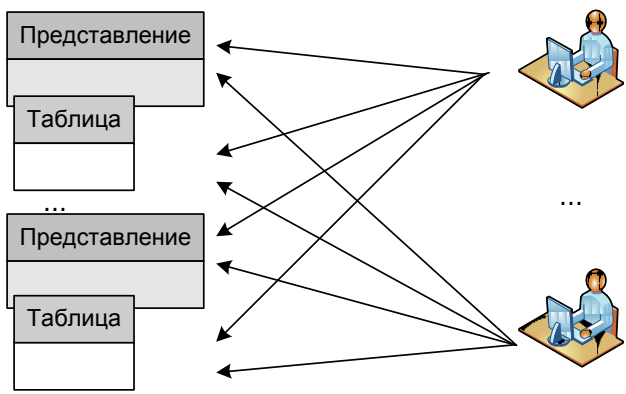


Рис. 14.1. Прямое определение полномочий пользователя

Например, администратор БД может создать роль Guest, которой будет позволено лишь просматривать некоторые таблицы, а роль Supervisor, наоборот, получит абсолютные права над БД. Впрочем, наличие механизма ролей не запрещает администратору определять права индивидуально для каждого из пользователей. Все зависит от конкретных условий обстановки.

Стандарт требует от создателей СУБД, чтобы в их диалектах SQL присутствовала инструкция `CREATE ROLE`, позволяющая создавать именованный набор полномочий, который позднее можно передавать пользователям БД. Синтаксис такой инструкции выглядит следующим образом:

```
CREATE ROLE имя роли [WITH ADMIN {CURRENT_USER | CURRENT_ROLE}]
```

В простейшем случае команда на создание новой роли выглядит так:

```
CREATE ROLE DB_OWNER
```

По умолчанию роль присваивается активному в данный момент пользователю, таким образом, продолжать инструкцию словами `WITH ADMIN CURRENT_USER` не обязательно. Если мы дополним инструкцию строкой `WITH ADMIN CURRENT_ROLE`, то

создаваемая роль приобретет точно такие же полномочия, как и у роли программиста, вызвавшего инструкцию.

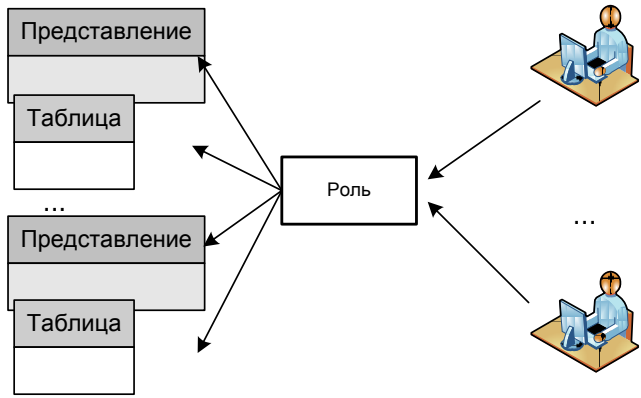


Рис. 14.2. Определение полномочий с помощью роли

Определение роли точно в соответствии с пожеланиями стандарта не реализовано ни в одной из упоминаемых в книге СУБД. Одноименные решения имеются в ORACLE (нужно отметить, что здесь конструкция значительно усовершенствована) и в MS SQL Server. В PostgreSQL аналогом считается команда `CREATE GROUP`. В остальных системах эта задача реализуется косвенно, например, через вызов системных процедур либо не поддерживается вообще.

У всех команд SQL, начинающихся со слова `CREATE`, имеются коллеги `DROP` и `ALTER`. Для роли не сделали исключений, поэтому удаление роли осуществляется командой `DROP ROLE`, редактирование — `ALTER ROLE`.

Предоставление привилегий

Предоставление прав на защищаемый объект санкционированному пользователю или роли осуществляется при посредничестве инструкции `GRANT`. Использование указанной команды гарантирует, что пользователь или роль имеет право выполнить определенные операции с объектом. Так как в СУБД полными правами на объект всегда владеет его создатель, то и право вызова `GRANT` обычно предоставляется владельцу объекта.

Команда `GRANT` вызывается при задании привилегий на объект базы данных или для определения заранее описанной роли. Синтаксис команды следующий:

```
GRANT {привилегия на объект [, …]| имя роли [, …]}
 ON имя объекта
 TO {получатель привилегии [, …] }
 [WITH GRANT OPTION] [WITH HIERARHY OPTION] [WITH ADMIN OPTION]
 [FROM {CURRENT_USER | CURRENT_ROLE}]
```

Если мы планируем воспользоваться инструкцией для определения прав пользователя на объект, то сразу после ключевого слова GRANT следует указать возможные значения привилегий (табл. 14.3).

Таблица 14.3. *Привилегии инструкции* GRANT

Привилегия	Описание
ALL PRIVILEGES	Назначить все привилегии
SELECT | INSERT | UPDATE | DELETE	Право на просмотр, вставку, редактирование и удаление
REFERENCES	Право управления ограничением внешнего ключа (FOREIGIN KEY), право использовать столбцы в любом ограничении
USAGE	Дает право использовать данный объект для определения другого объекта
UNDER	Право на создание подтипов или объектных таблиц
TRIGGER	Право на создание триггера
EXECUTE	Запуск на выполнение

Допустим, что мы намерены разрешить просмотр, вставку и редактирование данных из таблицы заказов некому пользователю USER1. Тогда необходимо написать следующую команду:

```
GRANT SELECT, INSERT, UPDATE
 ON ORDERS
 TO USER1
```

Привилегия может распространяться и на отдельный столбец или несколько столбцов таблицы, в таком случае имена столбцов перечисляют в круглых скобках сразу за именем таблицы:

```
GRANT UPDATE
 ON DELIVERIES (ORDER_KEY, AMOUNT)
 TO USER2
```

Так как предоставление привилегий для работы таблицей является наиболее распространенным вариантом обращения к команде GRANT, то нам достаточно только указать имя таблицы. Если же следует обратиться к объекту другого типа, то стандарт требует уточнить тип объекта (табл. 14.4).

Таблица 14.4. *Тип объекта инструкции* GRANT

Объект	Описание
[TABLE] имя объекта	Таблица
DOMAIN имя объекта	Домен

Таблица 14.4 *(окончание)*

Объект	Описание
`COLLATION` имя объекта	Сравнение
`CHARACTER SET` имя объекта	Набор данных
`TRANSLATION` имя объекта	Трансляция
`SPECIFIC ROUTINE` имя объекта	Подпрограмма

Следующие строки кода дают право на запуск процедуры для двух пользователей:

```
GRANT EXECUTE
 ON SPECIFIC ROUTINE MYPROC
 TO USER1, USER2
```

Для повышения читабельности кода в инструкцию SQL можно добавлять предложение `FROM CURRENT_USER` или `FROM CURRENT_ROLE`. В первом случае мы говорим, что привилегия была предоставлена от имени текущего пользователя, а во втором — от имени текущей роли. Если вы по каким-то причинам не завершили инструкцию `GRANT` этим предложением, то система станет полагать, что привилегия предоставлена от лица пользователя.

В SQL принято, что полноправным владельцем объекта всегда выступает его создатель. Что делать, если вы намерены передать свои полномочия другому лицу? Оператор `GRANT` позволяет наследовать привилегии одного пользователя другим. Право передачи привилегий определяет предложение `WITH GRANT OPTION`:

```
GRANT USAGE ON DOMAIN MY_DOMAIN TO USER1 WITH GRANT OPTION
```

Второй вариант опций `WITH HIERARHY OPTION` позволит осуществлять операции выбора данных (инструкция `SELECT`) не только над объектом, но и над всеми подчиненными ему объектами.

Привилегии можно присвоить не только пользователям, но и ролям:

```
GRANT ALL PRIVILEGES
ON SUPPLIERS, CONTRACTS
TO ADMINDB_ROLE
```

Допускается передавать привилегии от одной роли к другой:

```
GRANT ADMINDB_ROLE
TO USERDB_ROLE
WITH ADMIN OPTION
```

Обратите внимание, что при присвоении ролей нам потребовалась опция `WITH ADMIN OPTION`, именно она позволяет присваивать роли.

Лишение привилегий

Наряду с предоставлением прав пользователю или роли стандарт SQL предусматривает и обратную возможность — лишение привилегий.

Синтаксис инструкции выглядит следующим образом:

```
REVOKE {[опции] | привилегия на объект [, …]| имя роли [, …]}
ON объект
FROM имя получателя [, …]
[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]
{CASCADE | RESTRICT}
```

Инструкция REVOKE также позволяет лишить пользователя или роль определенных прав доступа к объекту БД. Применение команд REVOKE и GRANT очень похоже. Если вы больше не хотите, чтобы пользователь BADUSER мог работать с таблицей заказов ORDERS, то можно выполнить следующую инструкцию:

```
REVOKE ALL PRIVILEGES
ON ORDERS
FROM BADUSER CASCADE
```

Параметры CASCADE и RESTRICT уточняют, отменяется только определенная привилегия (RESTRICT) или же последует каскадное удаление всех связанных привилегий (CASCADE).

Внимание!

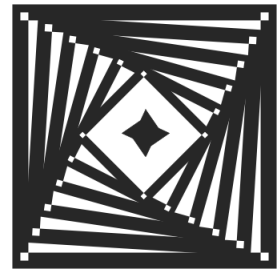
Существует вероятность, что одна и та же привилегия будет предоставлена пользователю от двух других независимых владельцев объекта. Допустим, что спустя некоторое время один из предоставивших привилегию передумает и решит отобрать данные права. При таком развитии событий SQL рекомендует, чтобы у попавшего в немилость пользователя по-прежнему сохранялись права на объект, до тех пор, пока инструкцию REVOKE не вызовет второй владелец объекта.

Если требуется одновременно отобрать права у нескольких пользователей, то в инструкции REVOKE (GRANT) можно просто перечислить имена пользователей (или ролей) через запятую:

```
REVOKE DELETE
ON WRITERS
TO BADUSER1, BADUSER2 CASCADE
```

Резюме

Исходя из факта, что все современные реляционные СУБД вооружены своей собственной версией языка SQL, только частично соответствующей современному стандарту, вывод об общем состоянии дел со стандартом SQL:2003 получается неоднозначным. Отрицательная сторона в том, что полноценной межплатформенной совместимости на базе SQL:2003 между СУБД разных производителей сегодня не существует и в обозримом будущем вряд ли появится. С другой стороны, положительной новостью является то, что большинство диалектов структурированного языка запросов наиболее близки к SQL:92, что позволяет знающему этот стандарт программисту очень быстро адаптироваться к любой версии СУБД.



Глава 15

Основы XML

Расширяемый язык разметки (eXtensible Markup Language, XML) был разработан специально для создания структурированных документов — он как никто другой позволяет представлять элементы данных и манипулировать ими. Несмотря на то, что последняя буква аббревиатуры XML содержит слово language (язык), корректнее говорить о XML не столько, как о языке, сколько как о стандарте.

Стандарт XML появился на свет в 1998 году, именно тогда он был опубликован консорциумом W3C (World Wide Web Consortium). Стандарт XML возник не на пустом месте, он основан на уже давно существовавшем *стандартном обобщенном языке разметки* (Standard Generalized Markup Language, SGML). SGML знаком не столь широкому кругу разработчиков программного обеспечения. Корни SGML уходят в компанию IBM. Создателями языка (в ту пору он назывался GML) считаются три сотрудника фирмы: Чарльз Голдфарб (Charles Goldfarb), Эдвард Мошер (Edward Mosher) и Раймонд Лори (Raymond Lorie). Придуманный ими в 1970-х годах обобщенный язык разметки предназначался для представления текстовых данных. Заложенные в GML идеи оказались настолько хорошими, что язык был взят за основу международного стандарта, специализирующегося на публикации и распространении сложных технических документов. Как только GML был стандартизирован, его название дополнилось еще одной буквой "S" (Standard). Возможности SGML существенно превысили своего предшественника GML, основное достижение в том, что SGML стал представлять собой метаязык, в котором пользователь мог создавать свои собственные языки разметки. Достаточно сказать, что всеобщий любимец — язык гипертекстовой разметки HTML — определен на основе метаязыка SGML.

Раз мы упомянули факт существования языка разметки гипертекста (Hypertext Markup Language, HTML), то напомним, что он предназначен для создания документов с одновременным включением в них инструкций, описывающих порядок отображения этих документов. Гипертекстовые документы мы видим каждый раз, открывая программы доступа (браузеры) к ресурсам Интернета. Язык HTML весьма хорош, но у него есть существенный недостаток — он представляет собой набор статических правил по форматированию текста и мало подходит для хранения данных, в особенности, если пользователь хочет самостоятельно определять их структуру.

Язык XML занимает свою собственную нишу среди наследников SGML. Основное его достоинство в том, что он позволяет пользователям создавать свой собственный набор тегов. Благодаря такой возможности XML и назван *расширяемым*. SGML и HTML не вполне подходят для описания структурированных документов. Если говорить точнее — SGML способен справиться и с более сложной задачей, но из-за высокой сложности языка освоение стандартного обобщенного языка разметки под силу далеко не каждому. Поэтому в своем первозданном виде SGML не получил широкого распространения. Язык гипертекстовой разметки HTML создан абсолютно для других задач. Он специализируется на оформлении, а не управлении структурными элементами документа.

Любой документ XML может рассматриваться как символьная строка, состоящая из символов разметки и данных. Благодаря тому, что двоичные данные в XML отсутствуют, документ на основе расширяемого языка разметки можно открыть и отредактировать в любом текстовом редакторе, от элементарного блокнота до текстового процессора класса Microsoft Word. Однако полноценная работа с документом XML невозможна без помощи специализированных программ-анализаторов (parsers).

Анализаторов XML достаточно много. Поклонникам операционной системы Windows наиболее известен процессор MSXML. Кроме того, свои анализаторы имеются у IBM, Apache, Oracle и у ряда других производителей программного обеспечения.

Анализаторы несут ответственность за проверку правильности XML-документа и, только убедившись в этом, предоставляют всю необходимую информацию приложению, ориентированному на работу с данными XML. Зачастую, XML-документ использует дополнительное описание, благодаря которому разработчик может наложить более строгие ограничения на хранимые в документе данные. Такое описание мы далее станем называть *определением документа* (рис. 15.1).

Столкнувшись с XML-документом, анализатор разбирает его содержимое на две составляющие: разметку и данные. Анализатор перерабатывает содержимое XML-документа и предоставляет приложению не в виде текстовых данных, а в соответствии с *объектной моделью документа* (Document Object Model, DOM) или в виде *простого интерфейса программирования для XML* (Simple API for XML, SAX). Каждый из конкурирующих анализаторов имеют как положительные, так и отрицательные черты. Особенность анализатора DOM в том, что он просматривает весь документ, строит его древовидную структуру и предоставляет приложению все запрашиваемые им элементы фактически за один прием. Анализатор DOM эффективен при работе с малыми по объему документами, а при наличии больших документов, которые сложно целиком загрузить в память, следует переключить внимание на SAX. Отличительная особенность анализатора SAX в том, что он обеспечивает последовательный доступ к элементам документа по мере его чтения, правда, начав работу, анализатор не в состоянии прерваться, например, чтобы вернуться к уже прочитанной части документа. Зато в сравнении с DOM анализатор SAX практически не потребляет память. С точки зрения практического применения DOM значительно проще SAX.

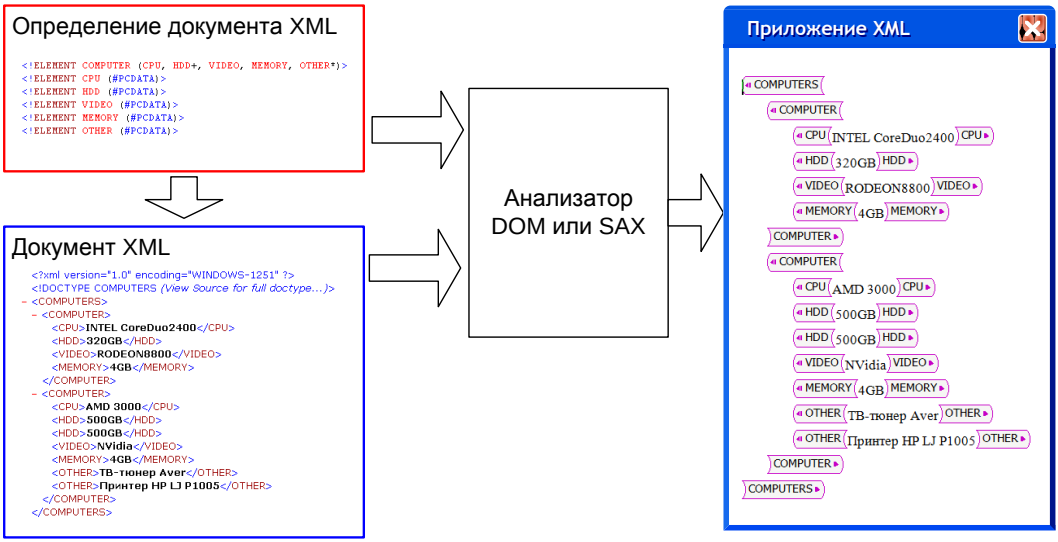


Рис. 15.1. Организация работы с XML-документом

Благодаря усилиям консорциума W3C объектная модель документа DOM сделана по-настоящему кроссплатформенной. Переносимость между платформами обеспечивается тем, что DOM описана не на каком-то конкретном языке программирования, а на стандартизированном языке описания интерфейсов IDL. Особенность IDL в том, что он отвечает лишь за корректное описание интерфейсной части. Вопросы реализации интерфейсов (другими словами, забота о наполнении интерфейсов работающим кодом) отдаются на откуп разработчикам. Это весьма удобное решение. С одной стороны, любой разработчик может испытать свои силы в области создании своего собственного анализатора XML-документа, а с другой — интерфейсная часть результирующего программного продукта у всех разработчиков окажется идентичной, и как следствие понятной всем заинтересованным в эксплуатации анализатора сторонам. На момент написания этих строк книги последняя спецификация DOM была размещена на странице **http://www.w3.org/TR/DOM-Level-3-Core/**. Это третий уровень объектной модели документа, он датирован 7 апреля 2004 года.

Если модель DOM активно поддерживается консорциумом W3C и как следствие стандартизирована, то с SAX все наоборот — он до сих пор не стал стандартом. Несмотря на это, анализаторы SAX разрабатывают не только отдельные энтузиасты, но и серьезные компании класса IBM и Oracle. Еще одна особенность SAX в том, что он изначально был определен в виде интерфейсов Java. Как следствие для работы с классическим SAX следует научиться программировать на Java как минимум с использованием JDK 1.1. У SAX есть логическое развитие — SAX2. Корпорация Microsoft реализовала SAX2 на платформе COM и сделала его доступным для программных продуктов из пакета Microsoft Visual Studio (Visual C++ и Visual Basic).

ЗАМЕЧАНИЕ

Для получения последней информации о расширяемом языке разметки следует обращаться на сайт консорциума W3C, в частности по вопросам, связанным с XML, на страницу **http://www.w3.org/standards/xml/**.

Правильность и допустимость документа

При разборе документа анализатор оценивает его корректность по двум критериям:

- Правильность оформления (well-formed) документа.
- Допустимость (valid) документа.

Контроль правильности относительно прост и включает лишь проверку синтаксиса и нескольких простых правил определения элементов XML, о которых мы поговорим немного позднее. Правильно оформленный документ станет доступным приложению сразу после проверки документа анализатором.

Проверка допустимости проводится значительно строже. Во-первых, допустимый документ должен быть правильным. Во-вторых, допустимый документ обязан соответствовать своему определению (или, говоря с некоторой степенью упрощения — шаблону), обычно представленному в формате DTD- или XML-схемы.

Второй уровень проверки не обязателен для всех XML-документов, более того, далеко не каждый документ обладает определением в виде DTD- или XML-схемы. Но с точки зрения приложений баз данных это весьма желательная черта XML-документа.

Построение простейшего документа XML

Правила создания рядового документа XML столь просты, что для написания демонстрационного проекта в формате XML уйдет не более минуты (листинг 15.1).

Листинг 15.1. Простейший XML-документ

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<!--Это комментарий-->
<example>
     <line1>Простой пример работы</line1>
     <line2>с языком XML</line2>
</example>
```

Первая строка выступает визитной карточкой частью XML-документа. Здесь указано, что мы работаем с документом XML первой версии и документ предна-

значен для работы в кодировке Windows-1251, что автоматически разрешает использование символов кириллицы.

Еще одна характерная черта первой строки документа — применение в ней особенных символов разметки — `<? … ?>`. Эти символы обрамляют инструкции, которые подскажут анализатору XML порядок работы с содержащимися в документе данными.

Вторая строка содержит комментарий создателя документа. Текст комментария заключается в символы `<!-- … -->`, при разборе документа анализатор игнорирует комментарий.

Судя по примеру, едва ли не самыми популярными символами в нашем документе выступают левая (<) и правая (>) угловые скобки. Это зарезервированные символы, предназначенные для описания тегов, информирующих о начале и завершении элемента документа.

Элементы

Каждому открывающему тегу должен сопоставляться закрывающий (за исключением первой строки документа). В нашем вводном примере существует связь между тегами `<example>` и `</example>`, `<line1>` и `</line1>`, `<line2>` и `</line2>`. Признак закрывающего тега — наклонная черта `/`.

Совокупность открывающего, закрывающего тега и данных внутри них называется элементом документа:

`<line3>`*данные*`</line3>`

Внимание!

Язык XML чувствителен к регистру! В первую очередь, про этот факт следует помнить при определении названий тегов, например, в XML тег `<LINE>` не эквивалентен `<line>`.

Язык XML заслужил звание расширяемого языка разметки потому, что он позволят нам определять новые теги. При остром желании в названиях тегов допустимо использовать символы национальных алфавитов, но мы этого делать не станем.

Еще одно обязательное условие построения корректного XML-документа — наличие корневого элемента, в нашем случае это `example`. Внутри корневого располагаются дочерние элементы, они, в свою очередь, могут быть родительскими для других элементов. Поэтому XML-документ как нельзя лучше подходит для хранения иерархических данных.

Формат XML не предусматривает возможностей по оформлению документов, ведь он разрабатывался для описания структур и данных. Внешний вид разработанного нами документа зависит от решения производителя программного обеспечения, например на рис. 15.2 представлен документ XML в окне Internet Explorer корпорации Microsoft.

Замечание

Задача оформления XML-документа обычно решается с помощью расширяемого языка стилей (*eXstended Stylesheet Language, XSL*) и его модификации языка XSLT (*XSL Transformations*).

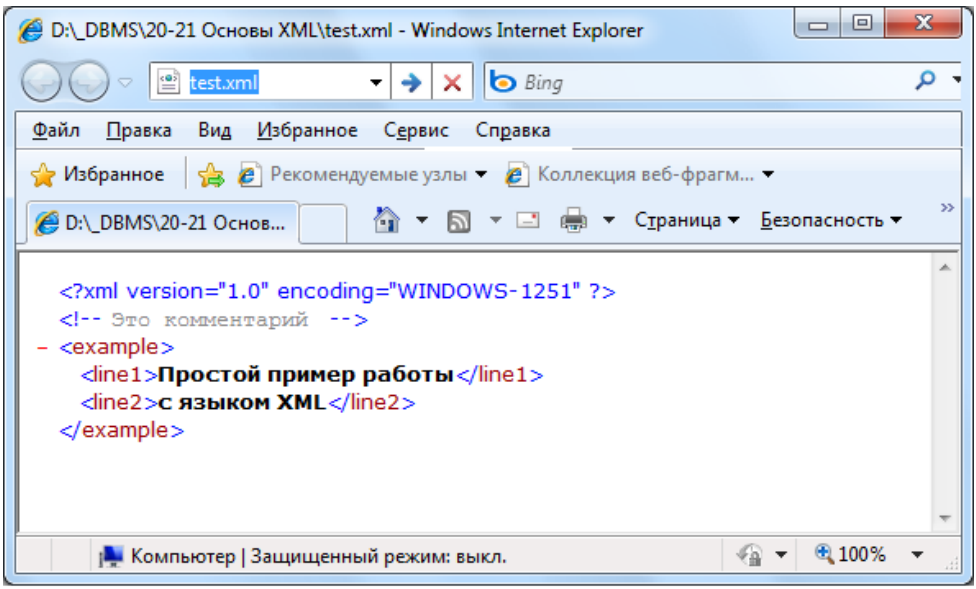


Рис. 15.2. Внешний вид XML-документа в браузере Windows Internet Explorer

После того как мы справились с первым примером простейшего документа XML (см. листинг 15.1), настал черед свести воедино базовые правила определения элементов XML:

- Каждому открывающему тегу должен соответствовать закрывающий тег.
- Документ XML позволяет использовать вложенные друг в друга теги при условии, что они не перекрываются.
- В документе XML должен быть всего один корневой элемент.
- Имена тегов должны начинаться с буквы. После первого символа разрешается применять цифровые и буквенные символы. Имя не должно содержать символы пробела и двоеточия ":". В именах запрещено применять аббревиатуры "XML", "xml", "Xml" и т. п.
- Имена чувствительны к регистру.
- В отличие от HTML, стандарт XML допускает пробелы, символ табуляции и символ абзаца. Указанные специальные символы (в терминах XML они называются пробельными (whitespace)) могут быть полноценными элементами данных.

Как бы это не показалось удивительным, но столь простые правила позволяют нам обмениваться данными с помощью XML.

Специальные символы

Кроме угловых скобок к специальным символам относятся амперсант (&), двойная (") и одинарная кавычки ('). Нельзя исключить вероятность, что перечисленные символы могут встречаться внутри документа, допустим в названиях атрибутов. В этом случае их заменяют на комбинацию символов из табл. 15.1.

Таблица 15.1. *Специальные символы XML*

Зарезервированный символ XML	Комбинация для замены
<	<
>	>
&	&
"	"
'	'

Атрибуты

Предположим, что мы через Интернет скачиваем прайс-лист одного из своих поставщиков (листинг 15.2). В нем содержатся данные, систематизированные по жанру произведения, а внутри элементов жанра <GENRE> .. </GENRE> хранятся элементы с описанием книг <BOOK>..</BOOK>.

Листинг 15.2. XML-документ с атрибутами

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<pricelist>
<GENRE>Роман
   <BOOK WRITER="Толстой Л.Н." PUBLISHER="Мир" PRICE="100.00">
    Война и мир</BOOK>
   <BOOK WRITER="Толстой Л.Н." PUBLISHER="Детская литература"
                                                  PRICE="57.00">
    Анна Каренина</BOOK>
   <BOOK WRITER="Достоевский Ф.М." PUBLISHER="Кавказ" PRICE="210.00">
    Братья Карамазовы</BOOK>
  </GENRE>
  <GENRE>Поэма
   <BOOK WRITER="Гете И.В." PUBLISHER="Худ. литература" PRICE="45.5">
    Фауст</BOOK>
  </GENRE>
</pricelist>
```

Самое главное в новом примере то, что открывающий тег описания книг содержит пока не встречавшиеся нам блоки — атрибуты. Атрибут состоит из названия и значения. Порядок именования атрибутов ничем не отличается от правил задания имен для элементов документа. Значения атрибутов заключаются в двойные или одинарные кавычки, в нашем примере в них содержится дополнительная информация, описывающая ту или иную книгу.

Благодаря атрибутам в документах XML начинают просматриваться некоторые родовые черты таблиц классических баз данных, ведь атрибуты можно расценивать, как поля реляционных таблиц. Схожесть возрастет еще больше после того, как мы обсудим возможности наложения ограничений на передаваемые через атрибуты данные (см. табл. 15.2).

Пространство имен

До тех пор, пока создаваемые нами XML-документы не находят массового применения и используются только в домашнем компьютере или в пределах небольшой компании, можно особенно не задумываться над выбором названий для элементов документа. Если мы полагаем, что для определения книги наиболее подходит имя элемента `BOOK`, а для издательства `PUBLISHER`, то нам ничто не помешает использовать именно эти названия. Однако, как только разработчик сталкивается с XML-документами, спроектированными для профессионального программного обеспечения, эксплуатируемого значительным числом пользователей, ситуация с подбором названий несколько изменяется. Ведь нельзя исключить вероятность, что наши имена уже задействованы сторонними разработчиками в других целях.

На сегодняшний день существует значительное число стандартизированных или ставших стандартом де-факто названий элементов. Вы их встретите в масштабируемых векторных рисунках формата SVG (Scalable Vector Graphics), в словаре для описания математических выражений MathML, в языке управления химической информацией (Chemical Markup Language, CML) и во многих других документах, производных от XML. Кто поручится за то, что применяемые вами названия не станут конфликтовать с именами других разработчиков? Выход из положения заключается в определении в наших документах имен, гарантированно отличающихся от имен сторонних производителей. Конечно же, можно проявить немного выдумки и выбрать такие названия элементов и атрибутов, которые вряд ли встретятся у других разработчиков. Для этого достаточно дополнить свои имена чем-нибудь вроде глобальных универсальных идентификаторов GUID или, на худой конец — прозвищем любимой кошки. Но в результате подобного подхода наш документ утратит одно из важнейших достоинств XML — наглядность. Поэтому для снятия проблемы совпадения имен стандарт XML предусматривает более рациональное решение — введение пространства имен XML (XML Namespaces).

Пространство имен XML — это соглашение об именовании, в котором элементы XML документа помимо собственно имени снабжены особым префиксом, создаваемым по тем же правилам, по которым назначаются *универсальные идентификаторы ресурса* URI (Uniform Resource Identifier). Разработчики стандарта XML учли, что благодаря URI в Интернете обеспечивается однозначность доменных имен, а эта особенность как нельзя лучше подходит и для обеспечения уникальности названий элементов в документах XML.

Замечание

URI (*Uniform Resource Identifier*) не что иное, как *унифицированный идентификатор ресурса*. URI — включает в себя адрес с ресурсом в Интернет (URL, *Uniform Resource Locator*) и *унифицированные имена ресурсов* (URN, *Uniform Resource Names*).

Переработаем прайс-лист книжного магазина так, чтобы имена элементов стали соответствовать рекомендациям пространства имен XML и в результате приобрели право называться квалифицированными именами. В таких целях нам следует вставить в документ строку примерно следующего содержания:

```
<pl:pricelist xmlns:pl="http://www.myurl.ru/pricelist">
```

В начале строки мы объявляем префикс `pl` и связываем его с элементом `pricelist`. Заметьте, что имя префикса не стандартизировано и нам разрешается заменить его любой приемлемой аббревиатурой. Далее мы информируем анализатор о том, что имя соответствует пространству имен XML. Для этого мы применяем атрибут `xmlns:pl` (xmlns — сокращение от XML Namespace). В заключение мы воспользуемся любым вымышленным URI, допустим **http://www.myurl.ru/pricelist**. Особо отмечу, что для создания квалифицированного имени нет необходимости указывать реальную страничку в глобальной паутине, просто рекомендация по пространству имен XML требует пользоваться соглашением URI для префиксной части названий элементов.

Замечание

Имя элемента, построенного в соответствии с рекомендациями пространства имен XML, называется *квалифицированным именем* (qualified names).

После создания префикса мы получаем право использовать его при именовании всех дочерних по отношению к элементу `<pricelist>` элементов и атрибутов. С этой целью мы добавляем префикс к начальному и конечному тегам элементов и перед названиями атрибутов так, как предложено в очередном примере прайс-листа (листинг 15.3).

Листинг 15.3. Документ XML с поддержкой уникальности имен

```
<?xml version="1.0" encoding="WINDOWS-1251"?>
<pl:pricelist xmlns:pl="http://www.myurl.ru/pricelist">
```

```
  <pl:GENRE> Роман
    <pl:BOOK WRITER="Толстой Л.Н." PUBLISHER="Мир"
     PRICE="100.00"> Война и мир
    </pl:BOOK>
  </pl:GENRE>
  <pl:GENRE> Поэма
    <pl:BOOK WRITER="Гете И.В." PUBLISHER="Художественная литература"
     PRICE="45.5"> Фауст
    </pl:BOOK>
  </pl:GENRE>
</pl:pricelist>
```

При обработке подобного документа анализатор автоматически заменит префиксную часть имени на соответствующий префиксу универсальный идентификатор ресурса URI, что гарантирует уникальность имен элементов и атрибутов.

Внимание!

Тот факт, что порядок именования префикса пространства имен XML опирается на правила URI, ни в коем случае не означает, что в глобальной сети вы обнаружите интернет-ресурс с указанным в XML-документе адресом.

Если в плане создания квалифицированных имен для элементов документа все понятно, то с именованием атрибутов в спецификации XML имеется некоторая неоднозначность. По сути, атрибуты принадлежат своим элементам, поэтому большинству анализаторов XML будет безразлично — вставляем ли мы перед именами атрибутов префиксы или обделяем атрибуты своим вниманием. В результате анализатор будет полагать, что строка

```
<pl:BOOK WRITER="Гете И.В." PUBLISHER="Художественная литература"
     PRICE="45.5"> Фауст</pl:BOOK>
```

идентична строке

```
<pl:BOOK pl:WRITER="Гете И.В." pl:PUBLISHER="Художественная литература"
     pl:PRICE="45.5"> Фауст</pl:BOOK>
```

в которой префиксы расставлены перед каждым атрибутом. Поэтому порядок именования атрибутов отдается на откуп разработчику документа.

Подытожим рассказ о пространстве имен XML. Введение пространства имен в XML необходимо для решения двух задач:

1. Предоставления возможности программному обеспечению отличать одноименный элемент разных XML-документов.
2. Обеспечения удобства группировки всех связанных элементов и атрибутов одного XML-приложения, что облегчает приложению их распознавание.

Определение документа

Нельзя исключить вероятность того, что заложенная разработчиком смысловая нагрузка в элементы документа окажется не столь очевидной как для пользователя, так и для анализатора XML. Примеры неоднозначностей в толковании разметки документа можно приводить долго. Допустим, создавая элемент `<PHONENUM>`, мы с вами рассчитываем, что в нем будет храниться 11-значный номер телефона, но другой читатель предположит, что достаточно ограничиться сокращенным 7-значным городским номером абонента. Или другой пример: создавая элемент `<INN>`, разработчик надеется, что в нем обязательно окажется уникальный индивидуальный номер налогоплательщика, но пользователь может посчитать, что это поле вообще необязательно для заполнения. Для решения подобных проблем в базах данных разработан весьма эффективный аппарат поддержки целостности и непротиворечивости данных, а как обстоят дела в XML?

В ситуации неоднозначного толкования разметки следует воспользоваться еще одним козырем XML — способностью подключения к документу дополнительного описания, называемого *определением типа документа* (Document Type Definition, DTD). DTD конкретизирует, что именно должно содержаться в элементах и атрибутах документа. Не нравится DTD, тогда воспользуйтесь альтернативным решением — XML-схемой.

В DTD и XML-схеме содержатся дополнительные пояснения о составе элементов и атрибутов документа, признаки обязательности для заполнения и описание применяемых типов данных.

DTD

Несмотря на то, что описание документа XML на основе DTD несколько утрачивает популярность (сегодня более распространено описание в формате XML-схемы), DTD по-прежнему часто встречается во многих XML-документах, поскольку хорошо справляется с определением простых структур данных.

Описание DTD позволяет решить двуединую задачу. Во-первых, документ XML, помимо данных содержащий и специальное описание, позволяет анализатору провести более глубокую проверку корректности содержимого файла. Во-вторых, благодаря DTD документ становится понятным специалисту, в нашем случае разработчику базы данных, поддерживающей XML.

Описание структуры документа XML с помощью DTD начинается сразу после обязательного заголовка документа и включает слово `DOCTYPE`. Признаком завершения секции описания выступает закрывающая квадратная скобка и закрывающий тег `]>`. Простейшее описание будет выглядеть следующим образом:

```
<!DOCTYPE имя_элемента [ ]>
```

Внутри секции содержится некоторый набор тегов `<!` и `>`, в которые заключаются имена элементов структуры или списки атрибутов. Восклицательный знак

указывает на то, что это элемент объявления (declaration element). Сразу за восклицательным знаком (без пробела) следует слово `DOCTYPE` и имя элемента структуры.

Замечание

Синтаксис DTD существенно отличается от правил построения документа XML. Это объясняется тем, что DTD фактически является самостоятельной подветвью языка SGML. Кстати, это одна из причин отказа от DTD и предпочтения XML-схем.

При желании определение DTD можно включить в документ XML или хранить во внешнем файле. В первом случае несколько изменяется первая строка файла с данными:

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>
```

Если документ содержит в себе описание DTD, то в строку включается признак автономности `standalone` со значением `"yes"`. Если же определение документа располагается во внешнем файле, то объявление автономности отменяется, об этом анализатор уведомляется значением `"no"`.

В простейшем случае описание DTD заносится непосредственно в документ XML, сразу после заголовка с версией (листинг 15.4).

Листинг 15.4. Определение DTD в составе документа XML

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>
<!DOCTYPE pricelist [
     <!ELEMENT GENRE (#PCDATA)>
     <!ELEMENT BOOK (#PCDATA)>
        <!ATTLIST BOOK
              WRITER CDATA    #REQUIRED
                PUBLISHER CDATA #IMPLIED
                PRICE CDATA     #REQUIRED>]>
<pricelist>
  <GENRE>Роман
     <BOOK
       WRITER="Толстой Л.Н."
       PUBLISHER="Мир"
       PRICE="100.00">
       Война и мир</BOOK>
  </GENRE>
</pricelist>
```

В листинге 15.4 мы создаем блок определения типа документа, информирующего нас о том, что:

- каждый информационный блок документа `pricelist` состоит из двух элементов (`GENRE` и `BOOK`);

- оба элемента представляют собой *разобранные символьные данные* (Parsed Character Data, PCDATA) или, проще говоря, элементы хранят обычный текст;
- элемент BOOK содержит список атрибутов (тег <!ATTLIST). Из них два атрибута обязательны для заполнения (#REQUIRED) и один необязательный (#IMPLIED).

Столкнувшись с объявлением DTD, анализатор сохраняет его в памяти. Затем, в процессе чтения блоков данных XML, анализатор проверяет их на соответствие требованиям DTD. Если выявляются отсутствующие (или наоборот лишние) элементы и атрибуты, или элемент данных находится не в ожидаемом месте, или содержимое элементов и атрибутов не соответствует объявлению DTD, то анализатор делает вывод о том, что документ некорректен.

К сожалению, описание DTD несколько ограничивает нас при определении типа данных, подлежащих хранению в том или ином элементе. В предложенном примере мы воспользовались типом CDATA — это наиболее универсальное описание, представляющее собой обычные символьные данные.

В нашем примере элемент BOOK содержит объявления атрибутов. Список атрибутов начинается с тега <!ATTLIST, далее следует имя элемента, которому принадлежит список, затем идет описание атрибутов и закрывающая угловая скобка. Ключевое достоинство атрибута в том, что тип хранимых в нем данных не ограничивается банальным CDATA (табл. 15.2). Конкретизируя тип атрибута, мы явным образом указываем анализатору порядок обработки символьных данных.

Таблица 15.2. *Типы атрибутов DTD*

Тип	Описание
CDATA	В качестве значения атрибута могут использоваться символьные данные
ID	Признак, что значение атрибута должно быть уникальным. Весьма полезное качество при работе с базами данных, так как позволяет однозначно идентифицировать элемент, которому принадлежит уникальный атрибут
IDREF	В атрибуте хранится ссылка на идентификатор (ID) элемента
IDREFS	В атрибуте находится разделенная пробелами последовательность значений IDREF
ENTITY	Ссылка на внешний объект (например, файл мультимедиа). При проверке корректности документа подобный объект пропускается анализатором
ENTITIES	Разделенные пробелами ссылки на внешние объекты
NMTOKEN	Строка символьных данных, которая будет рассматриваться анализатором как корректное значение
NMTOKENS	Строка, содержащая несколько разделенных пробелами NMTOKEN

Таблица 15.2 *(окончание)*

Тип	Описание
Перечислимый атрибут	Атрибут, содержащий список допустимых значений. Пример: `<!ATTLIST video format (MPEG1 | MPEG2 | MPEG4) "MPEG2">` В скобках мы перечислили все допустимые для атрибута значения видеоформата. По умолчанию предлагается применять формат видеозаписи MPEG2

Описание DTD позволяет формировать достаточно сложные модели XML-данных. Допустим, нам необходимо построить иерархическую структуру описания состава системного блока персонального компьютера (листинг 15.5).

Листинг 15.5. Определение DTD системного блока персонального компьютера

```
<!DOCTYPE COMPUTERS [
 <!ELEMENT COMPUTER (CPU, HDD+, VIDEO, RAM, OTHER*)>
 <!ELEMENT CPU (#PCDATA)>
 <!ELEMENT HDD (#PCDATA)>
 <!ELEMENT VIDEO (#PCDATA)>
 <!ELEMENT RAM (#PCDATA)>
 <!ELEMENT OTHER (#PCDATA)>
]>
```

Строки примера указывают на то, что элемент `COMPUTER` должен содержать информацию о процессоре `CPU`, жестком диске `HDD`, видеокарте `VIDEO`, памяти `RAM` и другом оборудовании `OTHER`. Названия вложенных элементов перечисляют в круглых скобках после слова `COMPUTER`. Подобное перечисление требует, чтобы при вводе XML-данных строго соблюдался порядок следования элементов, на первом месте процессор, на втором — винчестер и т. д. Если обозначенный порядок по каким-то причинам нарушится, то анализатор немедленно известит нас об ошибке в документе.

Обратите внимание еще на одну особенность описания DTD для рассмотренного примера с компьютером. Во второй строке листинга после названий элементов `HDD` и `OTHER` появились ранее не встречавшиеся нам символы `+` и `*`. Благодаря ним мы приобретаем дополнительные возможности (табл. 15.3) по введению ограничений на количество элементов данных и установке признака обязательности и необязательности элемента.

Таблица 15.3. *Ограничение количества элементов*

Определитель	Описание
?	Признак необязательного элемента, он может вообще отсутствовать в XML-данных либо появляться один раз
+	Элемент может появиться один или несколько раз
*	Элемент может не появляться или появляться несколько раз
Пустой определитель	Значение по умолчанию, элемент обязателен и может появляться только один раз

С помощью символов + и * мы сумели создать определение DTD для XML-документа с переменным числом элементов. Теперь при описании состава компьютера нам позволено подключать один или несколько жестких дисков и неограниченное число дополнительного оборудования OTHER, при этом документ XML остается корректным.

Хранение DTD во внешнем файле

Зачастую описание DTD хранят отдельно от документа XML. В этом случае содержание файла описания структуры документа для системного блока компьютера будет выглядеть так, как в листинге 15.6.

Листинг 15.6. Определение DTD в отдельном файле

```
<!ELEMENT COMPUTER (CPU, HDD+, VIDEO, RAM, OTHER*)>
 <!ELEMENT CPU (#PCDATA)>
 <!ELEMENT HDD (#PCDATA)>
 <!ELEMENT VIDEO (#PCDATA)>
 <!ELEMENT RAM (#PCDATA)>
 <!ELEMENT OTHER (#PCDATA)>
```

Файл описания сохраняется с расширением dtd, например computers.dtd. Чтобы анализатор XML понял, что документ XML нуждается в подключении файла computers.dtd, следует включить дополнительную строку с указанием имени файла описания. В простейшем случае, когда файлы описания и документ находятся в одном и том же каталоге локального компьютера, достаточно указать лишь имя файла dtd (см. вторую строку листинга 15.7).

Листинг 15.7. Документ XML, основанный на определении computers.dtd

```
<?xml version="1.0" encoding="WINDOWS-1251" standalone="no"?>
<!DOCTYPE COMPUTERS SYSTEM "computers.dtd">
```

```
<COMPUTERS>
     <COMPUTER>
          <CPU>INTEL CoreDuo2400</CPU>
          <HDD>320GB</HDD>
          <VIDEO>RODEON8800</VIDEO>
          <RAM>2GB</RAM>
     </COMPUTER>
     <COMPUTER>
          <CPU>AMD 3000</CPU>
          <HDD>500GB</HDD>
               <HDD>500GB</HDD>
          <VIDEO>NVidia</VIDEO>
          <RAM>4GB</RAM>
          <OTHER>ТВ-тюнер Aver</OTHER>
          <OTHER>Принтер HP LJ P1005</OTHER>
     </COMPUTER>
</COMPUTERS>
```

Файл описания разрешается располагать в отдельном каталоге на компьютере или даже в Интернете. Например:

```
<!DOCTYPE computers SYSTEM "file:///c:/dtdcash/computers.dtd">
<!DOCTYPE computers SYSTEM "http://www.BHV.ru/examples/computers.dtd">
```

В первом случае мы уведомляем анализатор, что файл описания можно обнаружить в папке `c:/dtdcash/`, из второго примера анализатор поймет, что доступ к файлу осуществляется через Интернет с помощью URI:

http:// www.BHV.ru/examples/computers.dtd.

До этого момента при указании места расположения описания DTD мы пользовались системными идентификаторами `SYSTEM`. Наряду с системным (правда несколько реже) встречается и открытое (`PUBLIC`) определение ресурса DTD.

```
<!DOCTYPE computers PUBLIC "-//DBMS//Chapter15//RU" >
```

Заметьте, что вместо явного указания на имя файла открытое определение применяется для ссылки на запись в каталоге. Обычно открытый идентификатор описывается в формате FRI (Formal Public Identifiers):

```
-//Владелец//Описание класса//Язык/Версия
```

Для того чтобы анализатору было проще найти копию документа, допускается объединять открытое и системное описание, причем в последнем случае слово `SYSTEM` опускают:

```
<!DOCTYPE computers PUBLIC "-//DBMS//RU" "computers.dtd">
```

Замечание

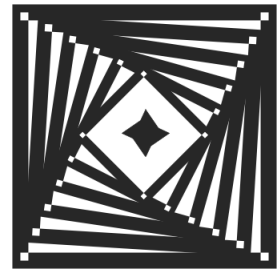
DTD обладает достаточно сложным синтаксисом, подробное описание которого, к сожалению, невозможно изложить в одной главе нашей книги. Всю возможную информацию по особенностям DTD вы найдете по адресу **http://www.w3.org/TR/REC-xml#dt-doctype**.

Резюме

Стандарт XML предназначен для создания иерархически структурированных документов, позволяющих хранить и обмениваться данными. XML молниеносно захватил огромный кусок рынка программного обеспечения. Достаточно сказать, что стандарт XML применяется в текстовых редакторах, базах данных, электронной коммерции, Интернете, графических приложениях.

Для повышения качественного уровня структуризации документа и придания большей строгости данным, хранимым в документе XML, стандартом предусмотрена возможность создания дополнительного описания данных. Рассмотренное описание документов в формате DTD позволяет определить связи между элементами документа (отношение "главный-дочерний"), поставить признак обязательности, назначить последовательность следования элементов, предоставить варианты выбора значений, ограничить число элементов.

Несколько ограниченные возможности DTD очень скоро перестали устраивать разработчиков, создающих проекты средней и высокой сложности. Например, для корректного обмена данными между БД, как минимум, требуется уметь проверять тип данных, но описание DTD на это неспособно. Именно поэтому в 2001 году консорциум W3C принял новый, обладающий более широкими семантическими возможностями стандарт, в котором содержание документа определено на основе понятия XML-схемы (XML schemas).



Глава 16

XML Schemas

Рассмотренное в предыдущей главе описание Document Type Definitions (DTD) вполне справляется с определением структуры XML-документов небольшой степени сложности, поэтому DTD до сих пор находит применение во многих проектах XML. Вместе с тем, компьютерные технологии никогда не стоят на месте, и очень скоро после выхода стандарта XML консорциум W3C пришел к выводу, что настал черед разработки более высокоуровневого формата описания XML-документов, тем более, что к тому моменту времени у консорциума имелся ряд интересных наработок. В результате в 2001 году W3C представил разработчикам новую хорошо продуманную технологию XML Schemas (Схема XML), позволяющую создавать определение XML-документа с более высокой степенью детализации.

Технология XML Schemas освободилась от недостатков DTD и приобрела ряд преимуществ, вот только ключевые из них:

- поддержка широкого спектра стандартных типов данных;
- возможность создания пользовательских типов данных;
- простое определение сложных структур данных с возможностью многократного использования этих структур;
- возможность проверять не только структуру, но и содержимое данных;
- полная и безусловная поддержка пространств имен.

Еще одно преимущество XML Schemas в том, что в отличие от описания DTD, целиком и полностью построенного на SGML, XML-схема основана на относительно простом XML. Благодаря опоре на доступный синтаксис определение структуры документа на базе XML-схемы начало постепенно вытеснять DTD. Лучшее тому подтверждение — небольшой пример. Допустим, что в нашем распоряжении есть простейший документ XML "Hello, World!" — классическое приветствие (листинг 16.1).

Листинг 16.1. Документ XML "Hello, World!"

```
<?xml version="1.0"?>
<message>
Hello, World!
</message>
```

Как будет выглядеть соответствующая представленному в листинге 16.1 документу схема? Примерно как в листинге 16.2.

Листинг 16.2. Схема для документа "Hello, World!"

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="message" type="xs:string"/>
</xs:schema>
```

Первая строка схемы ничем не отличается от строк, применяемых при создании обычного документа XML. Вторая строка открывает описание самого главного элемента схемы — `<schema>` с одновременным определением пространства имен. Третья строка указывает на то, что в результирующем документе должен существовать единственный элемент `"message"` строкового типа. Четвертая строка завершает описание схемы, закрывая тег `<schema>`. На первый взгляд все не так уж и сложно, не правда ли?

Замечание

Файл XML-схемы должен иметь расширение имени xsd. Это первые буквы слов XML Schema Definition — определение схемы XML.

Несмотря на свою простоту, наш пример схемы способен научить нас первым трем правилам построения XML-схем:

- корневым узлом схемы всегда должен выступать элемент `<schema>`;
- схема должна ссылаться на пространство имен **http://www.w3.org/2001/XMLSchema**;
- в минимальной нотации определение элементов схемы должно содержать имя и тип данных элемента.

Основная причина успеха описания документов XML на базе XML-схем — не простота, а более серьезные возможности по определению данных, что позволяет шире применять XML-документы в базах данных. Благодаря XML Schemas программист получил возможность объявлять тип данных элементов и атрибутов, диапазон допустимых значений, перечисления, значения по умолчанию и многое другое.

Многие разработчики сред программирования очень быстро откликнулись на появление нового формата определения документов и научили свои редакторы автоматически конвертировать документы из описания DTD в более совершенный формат XML Schema Definition (XSD). В их число входят редактор XML Mapping Tool из состава Embarcadero RAD Studio. На рис. 16.1 вы увидите результаты преобразования рассмотренного в *главе 15* описания файла computers.dtd (см. листинг 15.5) к формату файла xsd.

По сравнению с упрощенным примером схемы, на рис. 16.1 вы найдете значительно больше родовых черт XML-схем. По аналогии со всеми рассмотренными

ранее XML-документами схема начинается с указания версии XML и конкретизации используемой в ней кодировки символов. Вторая строка схемы открывается обязательным корневым элементом `<schema>`. Сразу за ним следует объявление пространства имен XML Schema и целевого пространства имен.

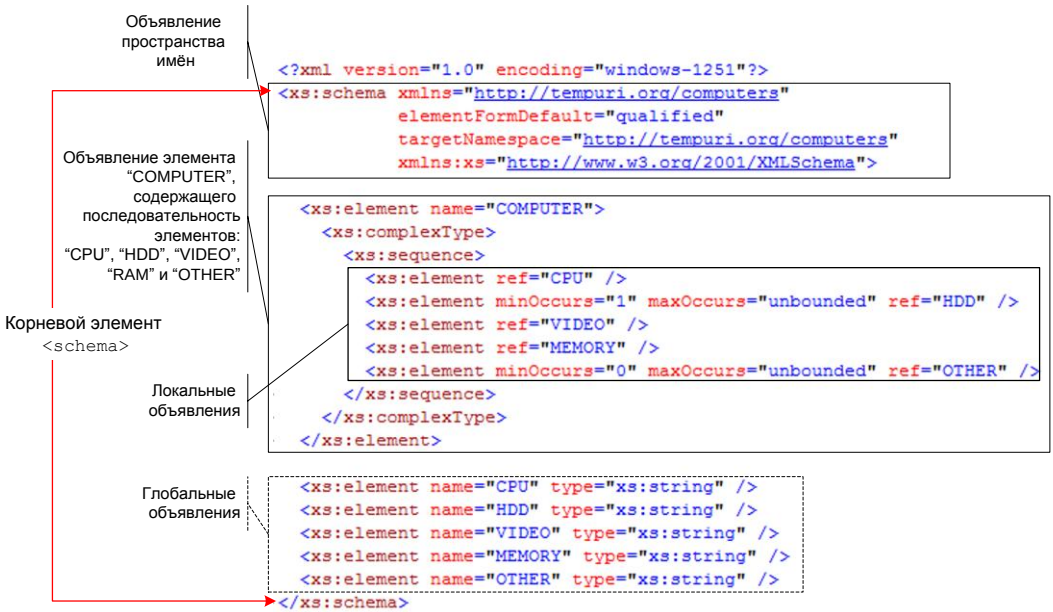


Рис. 16.1. Файл описания computers.xsd

По умолчанию определение любой схемы всегда должно ссылаться на пространство имен **http://www.w3.org/2001/XMLSchema**. Это говорит о том, что элемент `<schema>` является частью пространства имен XML Schema. Обозначенный URI рекомендован стандартом, поэтому его следует включать во все проекты, не забывая строго соблюдать регистр символов. В противном случае мы рискуем поссориться с XML-анализаторами, которые забракуют схему даже при несоответствии регистра хотя бы в одном из символов адреса.

При формировании XML-схемы разработчик имеет возможность на основе стандартизированных типов данных описывать свои собственные глобальные типы данных, которые позднее послужат для определения локальных элементов схем. Так как создаваемые нами пользовательские типы данных не принадлежат пространству имен XML Schema, мы вводим еще одно альтернативное пространство — целевое пространство имен. Для этого предназначен атрибут `targetNamespace`, в нашем примере он указывает на то, что данная схема составлена для целевого пространства имен **http://tempuri.org/computers**. В результате мы достигаем задуманного эффекта — наши объявления элементов и атрибутов становятся квалифицированными.

В некоторых случаях от гарантированной уникальности имен можно отказаться. Для этого в XML-объявлении предусмотрен еще один атрибут — `elementFormDefault`. Указанное в нашем примере значение `qualified` не противоречит здравому смыслу и уведомляет анализатор, что все элементы документа квалифицированы. Но выбрав обратное значение — `unqualified`, мы уведомляем анализатор, что какая-то часть элементов схемы может быть и неквалифицированна. Подобное решение нужно принимать с осторожностью, так что начинающему разработчику лучше его избегать.

Если за включение (отключение) квалифицированных имен элементов отвечает атрибут `elementFormDefault`, то за аналогичное действие с именами атрибутов несет ответственность `attributeFormDefault`. Еще раз отметим, что в подавляющем большинстве случаев для элементов предпочтительнее квалифицированные имена. Для атрибутов схемы требование по квалификации имен не столь строго и по умолчанию значение атрибута `attributeFormDefault` устанавливается в `unqualified`.

В качестве префикса пространства имен в соответствии с пожеланиями консорциума W3C следует применять символы `xs`, однако если префикс вам не по душе, то можете подобрать любой другой.

Определение элемента <*element*>

Корневому элементу `<schema>`, в свою очередь, принадлежит иерархия дочерних элементов схемы, описание которых начинается с тега `<element>`. Вот синтаксическая конструкция объявления элемента:

```
<element
 name="имя элемента"
 type="тип данных"
 ref="ссылка на глобальное объявление типа элемента"
 form="квалифицирован (qualified) или неквалифицирован (unqualified) "
 minOccurs="минимальное количество элементов"
 maxOccurs="максимальное количество элементов или без ограничений
           (unbounded) "
 default="значение по умолчанию"
 fixed="фиксированное значение">
```

При определении элемента схемы разработчик обязан сделать две вещи: назначить имя элементу и описать ограничения на принимаемые элементом данные. В минимальной нотации достаточно задать имя и назначить тип данных элемента, например:

```
<element name="HDD" type="string"/>
```

Строки, подобные этой, вы обнаружите в области глобальных объявлений нашего примера (см. рис. 16.1).

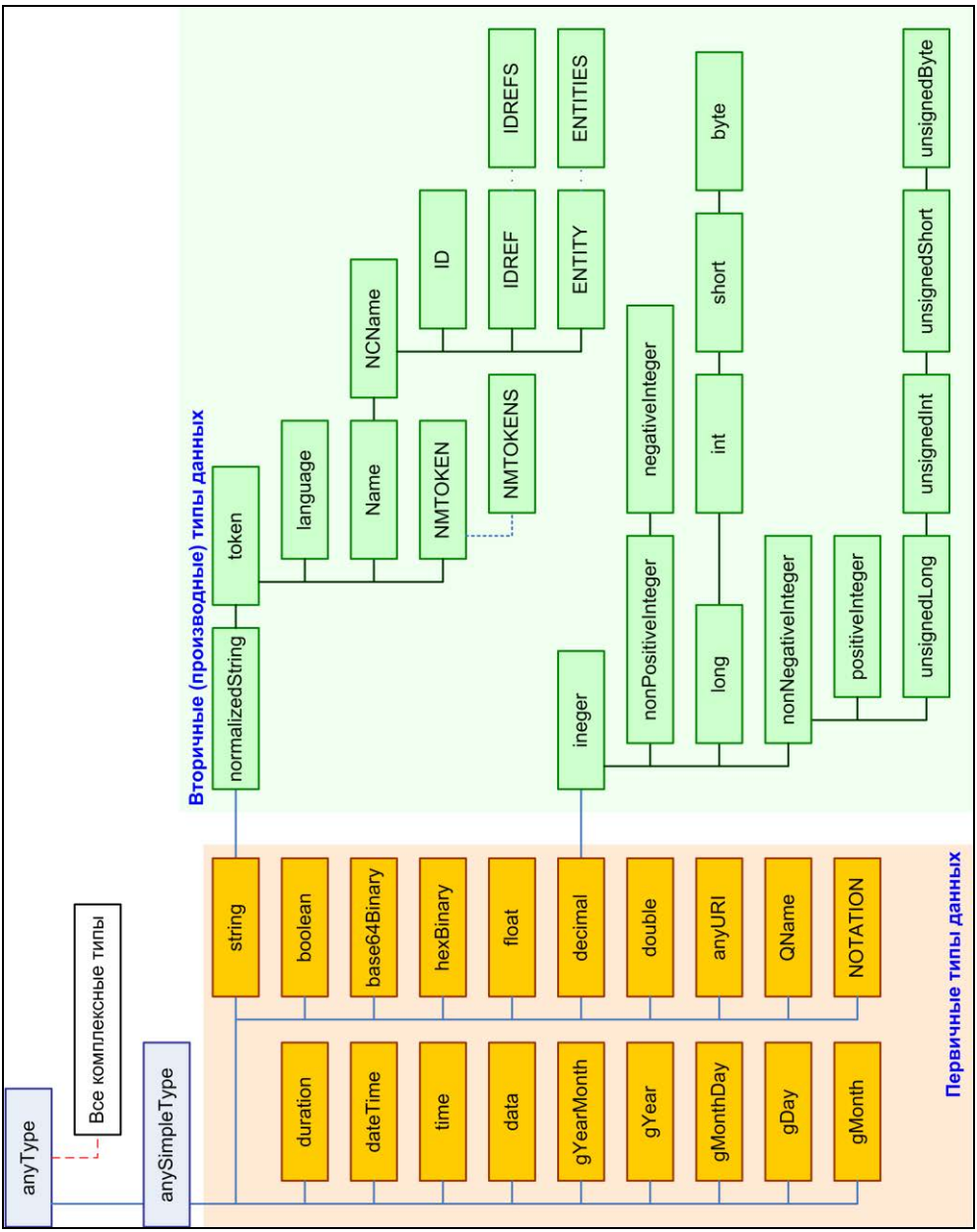


Рис. 16.2. Классификация типов данных XML

При определении типа данных допускается использовать не только стандартные, но и реализованные на их основе производные типы данных. Это очень полезное качество XML-схем позволяет вводить ограничения, сходные с применяемыми в базах данных доменными ограничениями.

Тип данных

Консорциум W3C стандартизировал типы данных, применяемых в XML-схемах. Согласно спецификации существует два типа: *встроенные первичные типы* (built-in primitive data types) и *вторичные типы* (derived data types). Перечень первичных и вторичных типов данных приведен на рис. 16.2. Представленные типы в том или ином виде встречаются в большинстве языков программирования и, в частности, в рассмотренном ранее структурированном языке запросов SQL, поэтому мы уклонимся от их углубленного изучения. Отметим лишь то, что вторичные типы данных делятся на две большие подгруппы: производные от строкового типа данных `string` и производные от десятичного типа `decimal`.

Еще одно важное замечание, касающееся типов данных XML: на данные того или иного типа могут накладываться дополнительные ограничения, например, по допустимому диапазону значений или по числу символов. Как это осуществить, мы поговорим немного позднее (см. табл. 16.1), а сейчас отмечу лишь, что для описания ограничений необходимо воспользоваться услугами тега `<restriction>`.

Производные типы *<simpleType>*

У XML Schemas много достоинств, но с точки зрения применения XML-схем в БД, ключевое из них — гибкие возможности в определении пользовательских типов данных. Стандарт позволяет нам, опираясь на уже заложенные в него типы данных, создавать свои собственные производные типы и применять их для определения элементов и атрибутов схемы.

Главным помощником в создании производных типов данных станет тег `<simpleType>`. Вот несколько упрощенная синтаксическая конструкция создания производного типа:

```
<simpleType name = "имя"
  final = (list или union или restriction))
  {другие атрибуты, не принадлежащие пространству имен}...>
```

В рамках производных типов нам разрешается накладывать ограничения, создавать перечисления и даже объединения типов. Наши требования к подлежащим хранению данным задаются с помощью атрибута `final`, он способен принимать одно из трех значений:

`<restriction>` — ограничение на значения;

`<union>` — объединение типов;

`<list>` — структура специализируется на хранении перечисления.

Ограничения на допустимые значения, передаваемые в поля документа XML, назначают с помощью определения <restriction> и при активном посредничестве предусмотренных в стандарте специальных атрибутов (табл. 16.1).

Таблица 16.1. *Атрибуты определения ограничений* <restriction>

Фасет	Описание
minExclusive	Определяет минимальное значение диапазона, исключая указанное значение
minInclusive	Определяет минимальное значение диапазона, включая указанное значение
maxExclusive	Определяет максимальное значение диапазона, исключая указанное значение
maxInclusive	Определяет минимальное значение диапазона, исключая указанное значение
totalDigits	Назначает количество значащих цифр в цифровых типах данных
fractionDigits	Назначает количество знаков после запятой
Length	Указывает число элементов в списках или количество символов в текстовых типах данных
minLength	Минимальное число элементов в списках или в текстовых типах данных
maxLength	Максимальное число элементов в списках или в текстовых типах данных
enumeration	Позволяет определять допустимые значения в перечислениях
whiteSpace	Определяет реакцию на пробелы во вводимых значениях. Может принимать три значения: preserve — документ не реагирует на пробелы; replace — несколько повторяющихся пробелов трансформируются в один пробел; collapse — несколько повторяющихся пробелов трансформируются в один пробел и удаляются все пробелы перед значением и после него
pattern	Шаблоны и ограничения на текстовые типы

Атрибут <restriction> может находиться не только в структуре <simpleType>, но и в элементах <simpleContent> и <complexContent>.

Продемонстрируем порядок определения ограничений. Предположим, что нам необходимо создать тип данных, допускающий хранение целочисленных значений в диапазоне от 1 до 10000. Решение иллюстрирует листинг 16.3.

Листинг 16.3. Ограничение диапазона целых чисел

```
<xs:simpleType name="HDDSize">
  <xs:restriction base="xs:integer">
```

```
    <xs:minInclusive value="1"/>
    <xs:maxInclusive value="10000"/>
  </xs:restriction>
</xs:simpleType>
```

Определение опорного типа данных — обязательное условие при работе с ограничениями, нам нужно лишь подобрать тип, наиболее подходящий на эту роль (рис. 16.2). В приведенном ранее примере мы остановили свой выбор на типе данных `integer`, об этом мы уведомляем анализатор, написав следующую строку:

```
<xs:restriction base="xs:integer">
```

Затем мы устанавливаем требуемые ограничения, заполнив атрибуты `minInclusive` и `maxInclusive`.

Еще один пример (листинг 16.4) демонстрирует применение атрибута `length`. Названный атрибут определяет число символов, которые могут быть переданы в элемент или атрибут. Подобное ограничение может пригодиться при определении типа данных для хранения 11-значного телефонного номера.

Листинг 16.4. Тип данных для хранения телефонного номера

```
<xs:simpleType name="PhoneNum">
  <xs:restriction base="xs:string">
    <xs:length value="11" fixed="true"/>
  </xs:restriction>
</xs:simpleType>
```

Если число символов в строке может варьироваться в заданных пределах, то вместо `length` применяем атрибуты `minLength` и `maxLength`.

Листинг 16.5 предлагает пример работы с ограничением `restriction` и демонстрирует возможность создания перечислений. Допустим, что мы намерены сформировать некоторый перечень значений, в рамках которых пользователь будет назначать классификационный признак для той или иной книги.

Листинг 16.5. Перечень допустимых значений

```
<xs:simpleType name="genreKind">
  <xs:restriction base="xs:string">
      <xs:enumeration value="Не классифицировано"/>
      <xs:enumeration value="Проза"/>
      <xs:enumeration value="Поэзия"/>
      <xs:enumeration value="Учебная литература"/>
      <xs:enumeration value="Научная литература"/>
  </xs:restriction>
</xs:simpleType>
```

Представленный в листинге 16.5 простой тип данных genreKind позволяет элементу использовать только одно из перечисленных в нем значений. Но это не догма, небольшая помощь объявления <list>, и мы приобретем возможность передавать в элемент несколько значений. Для этого пишем следующие строки (листинг 16.6).

Листинг 16.6. Получение данных из перечисления

```
<xs:simpleType name="genreKindList">
        <xs:listitemType="xs:genreKind">
</xs:simpleType>
```

Теперь элемент типа genreKindList позволит принимать одновременно несколько разделенных пробелами значений из списка genreKind.

Объявление <union> дает возможность комбинировать различные типы данных. Предположим, что мы хотим допустить хранение столь необходимого в базах данных неопределенного значения NULL в элементе, специализирующемся на обслуживании целых чисел. С этой целью мы создадим пользовательский тип данных NULLString, который способен содержать лишь один текстовый элемент NULL (листинг 16.7).

Листинг 16.7. Объявление пользовательского типа данных "NULLString"

```
<xs:simpleType name="NULLString">
     <xs:restriction base="xs:string">
     <xs:enumeration value="NULL">
</xs:restriction>
</xs:simpleType>
```

А теперь настал черед объявления <union>. В листинге 16.8 мы описываем еще один пользовательский тип NULLorInteger и с помощью <union> указываем, что он специализируется на хранении целых чисел или типа данных NULLString.

Листинг 16.8. Объявление комбинированного пользовательского типа данных

```
<xs:simpleType name="NULLorInteger">
     <xs:union memberTypes="integer xs:NULLString">
</xs:simpleType>
```

С этого момента мы имеем возможность объявлять элементы, способные содержать как целые числа, так и текстовое значение NULL (листинг 16.9).

Листинг 16.9. Объявление элемента заданного типа

```
<element name="intKey" type="xs:NULLorInteger"/>
```

Глобальное и локальное объявление

Синтаксическая конструкция XML-схемы позволяет создавать глобальные и локальные объявления элементов. Разница между глобальными и локальными объявлениями лишь в том, что первые выступают прямыми потомками головного элемента `<SCHEMA>`, а вторые — нет. Благодаря близости к головному элементу схемы глобальные объявления приобретают одну полезную черту — они получают право выступать в роли шаблона для определения характеристик производных от них локальных элементов.

Если в нашей XML-схеме уже описан некий глобальный элемент, характеристики которого подходят для определения и текущего элемента, то для ссылки на глобальный тип элемента активируйте атрибут `ref` и укажите имя глобального элемента. Например (см. рис. 16.2), у нас имеется глобальное объявление процессора `CPU`:

```
<xs:element name="CPU" type="xs:string"/>
```

Для указания того, что локальный элемент строится на основе глобального, достаточно написать такую строку:

```
<xs:element ref="CPU"/>
```

С этого момента локальный элемент вберет в себя все родовые черты (включая и имя) глобального элемента.

Если в документе требуется создать локальный тип элемента, мы просто помещаем объявление в качестве дочернего относительно элементов, вложенных в схему `<SCHEMA>`. Листинг 16.10 демонстрирует порядок работы с локальными объявлениями.

Листинг 16.10. Локальное объявление

```
<xs:element name="COMPUTER">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="NOTE" type="xs:string"/>
      <xs:element ref="CPU" />
```

Для этого мы ввели еще один элемент `NOTE` — примечание, предназначенное для хранения дополнительных текстовых данных. Указанный элемент выступает в качестве дочернего по отношению элементу с именем `COMPUTER`, поэтому считается объявленным локально.

Квалифицирование элемента

Атрибут `form` определяет порядок квалификации отдельного элемента. Если атрибут не задействован, то отношение текущего элемента к пространству имен зависит от настроек атрибута `elementFormDefault` (с которым мы познакомились

при обсуждении заголовка XML-схемы). Если же мы намерены явным образом квалифицировать элемент, то присваиваем атрибуту `form` значение `qualified` или `unqualified`.

Ограничения на число элементов

Если вы немного подзабыли порядок работы с определением документа в формате DTD, то напомню, что для определения числа повторов элементов в документе XML с помощью определения DTD мы использовали символы `*`, `+` и `?` (см. табл. 15.3). Подобный подход неудобен, т. к. из-за расплывчатых указаний DTD мы не в состоянии задать четкие ограничения на число элементов в результирующем документе.

В схеме XML предложено более элегантное решение опирающееся на атрибуты `minOccurs` и `maxOccurs`. Первый из них определяет минимальное число элементов, которые должны встретиться в корректном документе XML, второй — максимальное число элементов. Например:

```
<xs:element name="HDD" minOccurs ="1" maxOccurs="8"/>
```

Здесь говорится, что как минимум в системном блоке компьютера должен находиться один жесткий диск, а верхний предел не должен превышать восемь винчестеров. Если число дисков не ограничено, то вместо `"8"` в атрибут `maxOccurs` следует поместить значение `"unbounded"`.

ЗАМЕЧАНИЕ

В XML-схеме допускается и не применять атрибуты `minOccurs` и `maxOccurs`, в данном случае анализатор станет полагать, что они установлены в 1.

Значение по умолчанию и фиксированное значение

Для определения значения по умолчанию предназначен атрибут `default`. Само значение передается после знака присвоения в двойных кавычках:

```
<xs:element name="CPU" type="xs:string" default="Intel CoreDuo"/>
```

С этого момента при встрече неопределенного элемента, отвечающего за описание процессора компьютера, анализатор вставит в пустое место значение по умолчанию.

Несколько реже требуется создавать элементы, значение которых никогда не изменяется. Для определения фиксированного значения в описание элемента следует включить атрибут `fixed`:

```
<xs:element name="DefaultLanguage" type="xs:string" fixed="Russian"/>
```

С этого момента любое отклонение от значения `Russian` при определении языка по умолчанию в содержимом XML-документа будет расцениваться как некорректное.

Создание сложных структур <complexType>

При построении XML-схемы нам предоставлена возможность формировать сложные пользовательские структуры на базе стандартизированных первичных и вторичных типов данных. Для определения структуры стандартом XML предусмотрен элемент <complexType>.

С некоторой степенью приближения объявление <complexType> можно сравнивать с записью языка Delphi или структурой из C++. Комплексная структура специализируется на хранении множества разнотипных элементов и атрибутов из набора стандартизированных в XML типов данных (см. рис. 16.2). Если возможностей стандартных типов данных вам недостаточно, то следует вспомнить о существовании элемента <simpleType> . Он позволяет накладывать на данные дополнительные ограничения, например, определять диапазон допустимых значений или даже объединять необъединяемое — конструировать составные типы, способные комбинировать различные типы данных в рамках одного элемента или атрибута.

Созданную структуру можно объявить как глобально (в качестве прямого наследника <schema>), так и локально в более глубоких уровнях вложения схемы. В первом случае структура обязательно должна быть именованной, чтобы позволить разработчику ссылаться на нее при определении локальных элементов.

Предложенная в начале главы (см. рис. 16.1) XML-схема уже содержит до сих пор незаслуженно обойденное нашим вниманием объявление структуры комплексного типа <complexType>. Это указание анализатору XML на то, что создается сложная структура, в составе которой имеется некоторый перечень элементов.

Рассмотрим пример комплексной структуры для хранения домашнего адреса (листинг 16.11).

Листинг 16.11. Комплексная структура "Address"

```
<xs:complexType name="Address">
   <xs:sequence>
      <xs:element name="zip" type="xs:decimal"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="home" type="xs:string"/>
      <xs:element name="apartment" type="xs:decimal"/>
   </xs:sequence>
</xs:complexType>
```

Структура Address содержит пять разнотипных элементов, отвечающих за хранение почтового индекса, названия города, улицы, номера дома и квартиры. Ком-

плексная структура может содержать одно из объявлений, уточняющее состав структуры, об этом мы поговорим несколько позднее (см. табл. 16.2).

В состав сложной структуры могут входить группы элементов, заданные с помощью объявления <group>. Такое решение весьма полезно, когда одну и ту же модель содержимого следует представить в нескольких элементах. Допустим, что у нас имеется три разнотипных элемента с именами item1, item2 и item3. Все перечисленные элементы объявлены глобально (листинг 16.12), т. е. являются прямыми потомками <schema>.

Листинг 16.12. Группа элементов

```
<xs:element name="item1" type="xs:string"/>
<xs:element name="item2" type="xs:double"/>
<xs:element name="item3" type="xs:decimal"/>
```

Тогда на основе трех глобальных элементов мы способны создать глобальное объявление группы (листинг 16.13).

Листинг 16.13. Глобальное объявление группы

```
<xs:group name="itemGroup">
  <xs:sequence>
    <xs:element ref="item1"/>
    <xs:element ref="item2"/>
    <xs:element ref="item3"/>
  </xs:sequence>
</xs:group>
```

Еще раз напомню, что глобальное определение обязательно должно быть именованным. Для того чтобы наш дочерний элемент смог воспользоваться услугами объявленной глобально группы, нам потребуется примерно такая строка:

```
<group ref="itemGroup">.
```

В табл. 16.2 вы найдете несколько объявлений, позволяющий осуществить тонкую настройку структуры <complexType>.

Таблица 16.2. *Спецификация состава комплексной структуры* <complexType>

Объявление	Описание
<simpleContent>	Результирующая структура может включать только атрибуты и текстовые данные, элементы или смешанное содержимое внутри объявления <simpleContent> находиться не могут
<complexContent>	Результирующая структура может иметь атрибуты, элементы или смешанное содержимое и даже быть пустой

Таблица 16.2 *(окончание)*

Объявление	Описание
<group>	Позволяет создавать ссылки на группу элементов, определенную в глобальном объявлении схемы: <xs:complexType name="demo"> <xs:group ref="itemGroup"/> <!--другие элементы структуры--> </xs:complexType> При указании ссылки требуется использовать префикс имени и имя группы
<sequence>	Элементы структуры должны передаваться строго в заданной последовательности
<choice>	Указание на то, что в экземпляре XML-документа может содержаться только одно из объявлений элементов, перечисленных после объявления <choice>. Например, объявление <choice> <element name="weight" type="float"> <element name="size" type="float"> </choice> говорит, что в документ с данными может быть включен элемент weight или size. Одновременное включение обоих элементов запрещено
<all>	Антипод объявления <sequence>. При работе с XML-документом объявление <all> разрешает нам заполнять элементы структуры в любом порядке

Некоторые из перечисленных в табл. 16.2 объявлений могут включать в свой состав вложенные объявления. Например, последовательность <sequence> вправе включать внутренние элементы <sequence>, <choice> и ссылки на <group>.

Определение атрибута <*attribute*>

Если вы хорошо разобрались с особенностями объявления с элементов схемы, то изучение основ декларирования атрибутов не должно вызвать особых затруднений. Тем более что синтаксические конструкции определения элемента и атрибута очень схожи:

```
<attribute
 name="имя атрибута"
```

```
 type="тип атрибута"
 ref="ссылка на глобальное объявление"
 form="квалифицирован (qualified) или неквалифицирован (unqualified)"
 use="опциональный (optional) или запрещенный (prohibited)
      или обязательный (required)"
 default="значение по умолчанию"
fixed="фиксированное значение">
```

Рассмотрим пример работы с атрибутами. В листинге 16.14 мы вновь вернемся к системному блоку компьютера и усовершенствуем описание процессора.

Листинг 16.14. Применение атрибутов для описания процессора

```
<xs:complexType name="CPU">
  <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
  <xs:attribute name="CoreCount" type="xs:byte" default="1"/>
  <xs:attribute name="Frequency" type="xs:positiveInteger"
                      use="required"/>
  <xs:attribute name="Notes" type="xs:string" use="optional"/>
</xs:complexType>
```

Теперь процессор представлен в виде комплексной структуры, включающей четыре атрибута. Атрибут `Manufacturer` предназначен для описания производителя процессора, `CoreCount` — определит количество ядер, `Frequency` — тактовую частоту и `Notes` — примечание.

Подключение XML-схемы к документу

Присоединение XML-схемы к экземпляру документа обеспечивается с помощью атрибута `schemaLocation`. Порядок объявления следующий: сначала следует пространство имен URI, затем пробел и расположение схемы. Первые строки XML-документа, ссылающегося на файл схемы computers.xsd, могут выглядеть так, как в листинге 16.15.

Листинг 16.15. Подключение схемы computers.xsd к документу XML

```
<?xml version="1.0" encoding="utf-8"?>
<COMPUTERS xmlns="http://www.myurl.ru/computers"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.myurl.ru/computers computers.xsd">
<!--Элементы документа -->
</COMPUTERS>
```

Можно одновременно ссылаться на несколько схем, в этом случае их просто разделяют запятыми:

```
xsi:schemaLocation="http://www.myurl.ru/computers computers.xsd,
                    http://www.myurl.com/second computers2.xsd"
```

Допускается указание расположения файла схемы и без применения пространства имен. Для этого предназначен атрибут `noNamespaceSchemaLocation`. Если файл со схемой размещен в том же каталоге, что и документ, то заключительный ингредиент объявления может содержать только имя файла:

```
xsi:noNamespaceSchemaLocation="computers.xsd"
```

Если же схема "зарыта" достаточно глубоко, то путь к файлу следует описывать по правилам URI:

```
xsi:noNamespaceSchemaLocation=
                "file://C://ProgramData//MyProg//computers.xsd".
```

Пример схемы computers.xsd

Полученые знания позволяют нам создать достаточно серьезную версию схемы для хранения данных о системном блоке компьютера (листинг 16.16).

Листинг 16.16. Схема computers.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns="http://tempuri.org/computers"
           elementFormDefault="qualified"
           targetNamespace="http://tempuri.org/computers"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="COMPUTER">
  <xs:complexType>
       <xs:sequence>
       <xs:element ref="ComputerManufacturer"/>
         <xs:element ref="CPU" />
         <xs:element ref="MOTHEBOARD"/>
         <xs:element minOccurs="1" maxOccurs="8" ref="HDD" />
         <xs:element ref="VIDEO" />
         <xs:element ref="RAM" />
         <xs:element minOccurs="0" maxOccurs="unbounded" ref="OTHER" />
       </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="ComputerManufacturer">
  <xs:simpleType>
```

```
      <xs:restriction base="xs:string">
      <xs:enumeration value="HP"/>
        <xs:enumeration value="IBM"/>
        <xs:enumeration value="Toshiba"/>
        <xs:enumeration value="Другой производитель"/>
      </xs:restriction>
  </xs:simpleType>
</xs:element>

<xs:element name="CPU">
  <xs:complexType>
      <xs:attribute name="Manufacturer" type="xs:string"
                                          use="required"/>
      <xs:attribute name="CoreCount" type="xs:byte" default="1"/>
      <xs:attribute name="Frequency" type="xs:positiveInteger"
                                           use="required"/>
      <xs:attribute name="Socet" type="xs:string" use="required"/>
      <xs:attribute name="Notes" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<xs:element name="MOTHEBOARD">
  <xs:complexType>
      <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
      <xs:attribute name="Model" type="xs:string" use="required"/>
      <xs:attribute name="Socet" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="HDD">
  <xs:complexType>
     <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
      <xs:attribute name="Size" type="xs:positiveInteger" use="required"/>
      <xs:attribute name="Speed" type="xs:positiveInteger"
                                              default="7200"/>
  </xs:complexType>
</xs:element>

<xs:element name="VIDEO">
  <xs:complexType>
  <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
  <xs:attribute name="Model" type="xs:string" use="required"/>
  <xs:attribute name="RAMSize" type="xs:positiveInteger" use="required"/>
```

```
  </xs:complexType>
</xs:element>

<xs:element name="RAM">
  <xs:complexType>
      <xs:attribute name="Manufacturer" type="xs:string" use="required"/>
      <xs:attribute name="Size" type="xs:positiveInteger" use="required"/>
      <xs:attribute name="Speed" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>

<xs:element name="OTHER" type="xs:string" />
</xs:schema>
```

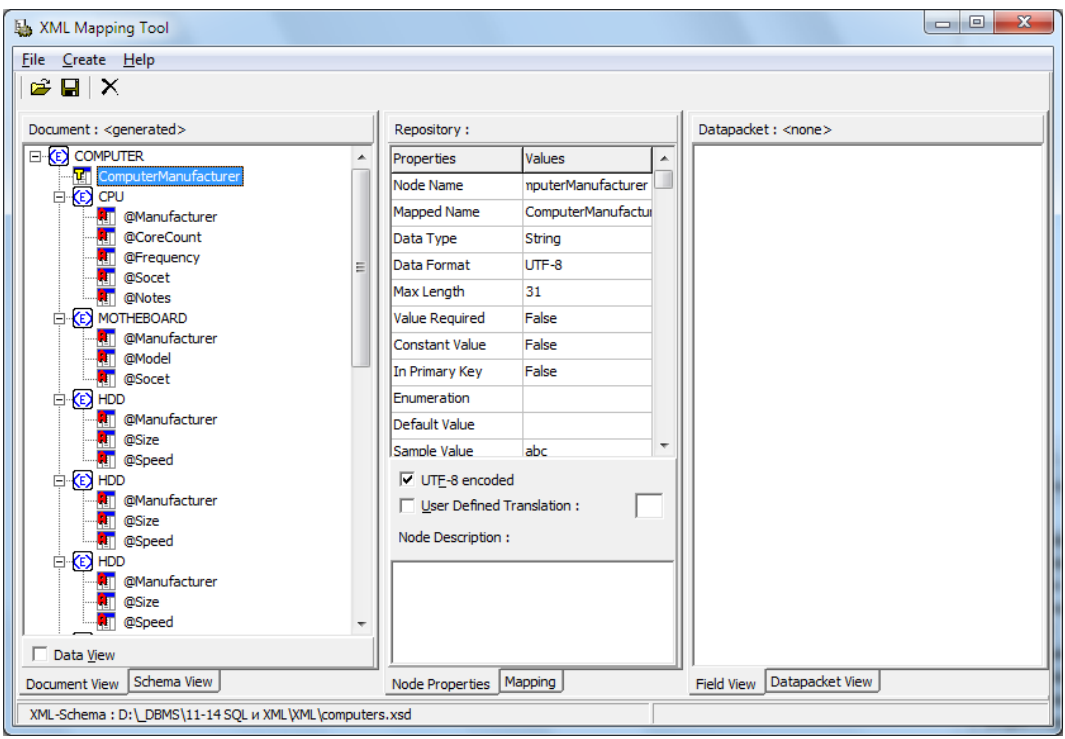


Рис. 16.3. Графическое представление схемы computers.xsd редактором XML Mapping Tool

Современные редакторы XML предоставляют разработчику удобный интерфейс, позволяющий проектировать данные не только по старинке (в текстовом редакторе), но и более современным способом (визуально). Какое из решений подходит вам в большей степени, выбирайте сами, я сторонник комплексного подхода.

В любом случае профессионалу следует уметь создавать XML-документы и схемы в обычном текстовом редакторе, но, вместе с тем даже закоренелому консерватору не стоит отказываться от наглядности, которую дают нам визуальные средства проектирования. Об этом вы можете судить сами по рис. 16.3, на котором представлен экранный снимок XML-схемы нашего примера, полученный с помощью редактора XML Mapping Tool из пакета Embarcadero RAD Studio.

Пример документа computers.xml

Нашу работу со схемами XML мы завершаем листингом 16.17, демонстрирующим порядок заполнения документа XML, опирающегося на схему из листинга 16.16.

Листинг 16.17. Документ computers.xml, использующий схему computers.xsd

```
<?xml version="1.0" encoding="utf-8"?>
<COMPUTERS xmlns="http://www.myurl.ru/computers"
      xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
      xs:noNamespaceSchemaLocation="computers.xsd">
  <COMPUTER>
    <ComputerManufacturer>IBM</ComputerManufacturer>
     <CPU>
       <Manufacturer>Intel</Manufacturer>
       <CoreCount>2</CoreCount>
       <Frequency>2700</Frequency>
       <Socet>P775</Socet>
     </CPU>
    <MOTHEBOARD>
      <Manufacturer>ASUS</Manufacturer>
      <Model>GT400</Model>
      <Socet>P775</Socet>
    </MOTHEBOARD>
    <HDD>
      <Manufacturer>WesternDigital</Manufacturer>
      <Size>320</Size>
      <Speed>7200</Speed>
    </HDD>
    <HDD>
      <Manufacturer>Fujitsu</Manufacturer>
      <Size>500</Size>
      <Speed>7200</Speed>
    </HDD>
```

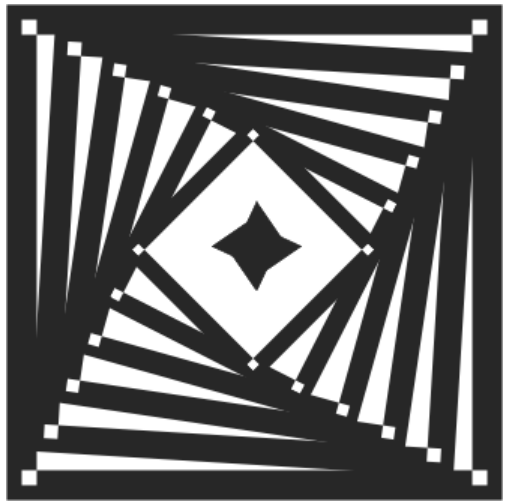
```
    <VIDEO>
      <Manufacturer>NVidea</Manufacturer>
      <Model>Unknown</Model>
      <RAMSize>512</RAMSize>
    </VIDEO>
    <RAM>
      <Manufacturer>Samsung</Manufacturer>
      <Size>4</Size>
      <Speed>800</Speed>
    </RAM>
    <OTHER>TVTuner AVER</OTHER>
    <OTHER>DVD-RW</OTHER>
  </COMPUTER>
</COMPUTERS>
```

Резюме

Стандарт XML представляет собой мощный язык, нашедший сегодня широчайшее применение во многих отраслях компьютерных технологий. Вокруг XML сосредоточено многочисленное семейство стандартов и языков, опирающихся на XML или выполняющих вспомогательные функции. Среди них:

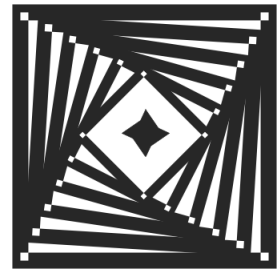
- XSLT (Extensible Stylesheet Language) — расширяемый язык преобразования стилей, предназначенный для преобразования XML-данных в другие форматы, чаще всего в HTML;
- XPath (XML Path Language) — язык, позволяющий обслуживать отдельную часть XML-документа;
- XQUERY — язык запросов XML;
- XLink — опирающийся на атрибуты синтаксис для использования ссылок в XML-документах;
- XForms — технология форм, конкурирующая с формами HTML;
- SVG (Scalable Vector Graphics) — многоцелевой формат масштабируемой векторной двумерной графики.

Описание XML-документа не ограничивается только возможностями DTD и XML-схемой. Существует ряд альтернативных подходов. Наиболее известный из них — RELAX NG. В сравнении с XML-схемой технология RELAX NG более проста в освоении, опирается на строгую математическую модель и легко компонуется. Благодаря перечисленным достоинствам RELAX NG пользуется заслуженным уважением среди многих разработчиков ПО.



ЧАСТЬ II

Проектирование приложений БД в Delphi



Глава 17

Концепция приложения БД Delphi

Эта глава открывает вторую часть книги, посвященную проектированию приложений баз данных на языке программирования Delphi. Излагаемый здесь материал предназначен для читателя, имеющего представление о языке программирования Delphi, и поэтому не затрагивает основы языка. Мы рассмотрим правила разработки настольных приложений баз данных и приложений, работающих по технологии "клиент-сервер", изучим базовые технологии доступа к данным, познакомимся с несколькими десятками компонентов Delphi, приобретем знания разработки многоуровневых проектов DataSnap, научимся создавать отчеты, сможем представлять данные в формате диаграмм.

Пример простейшего приложения БД

Несмотря на то, что эта глава в большей степени нацелена на изложение теории, свою работу мы начнем с небольшого практического примера. Он преследует две цели. Во-первых, продемонстрировать способность Delphi создавать элементарный проект БД без единой строки кода. Во-вторых, дать возможность начинающему разработчику БД "потрогать" своими руками среду разработки и ее компоненты.

На роль первого проекта будет претендовать база данных, состоящая всего из одной таблицы. Она предназначена для обслуживания списка поставщиков из демонстрационной базы данных книжного магазина (см. рис. 6.7). Разобравшись с составом столбцов таблицы, переходим к следующему шагу — выбору СУБД. И здесь нельзя не отметить одну из уникальных черт Delphi — абсолютную всеядность. Среда разработки способна сотрудничать со всеми популярными разновидностями настольных и клиент-серверных СУБД. Наш первый проект не претендует на лавры многопользовательской клиент-серверной БД, поэтому для нас подойдет практически любой формат таблицы для настольной системы, будь то Microsoft Access, FOXPRO, dBase и др. Я остановил свой выбор на Microsoft Access.

Если вы не повторили БД "Книжный магазин", разработанную нами в *главе 6*, то желательно это сделать сейчас.

Для создания файла таблицы нам с вами предстоит выполнить несколько несложных шагов.

1. Запустите программу Microsoft Access.
2. Создайте в программе новую базу данных и сохраните ее под именем books.accdb (или books.mdb, если вы работаете в старой версии программы).
3. На панели быстрого доступа найдите страницу **Создание**, а на этой странице команду **Таблица**.
4. Создайте любую из справочных таблиц, например таблицу поставщиков `SUPPLIERS` (рис. 17.1).

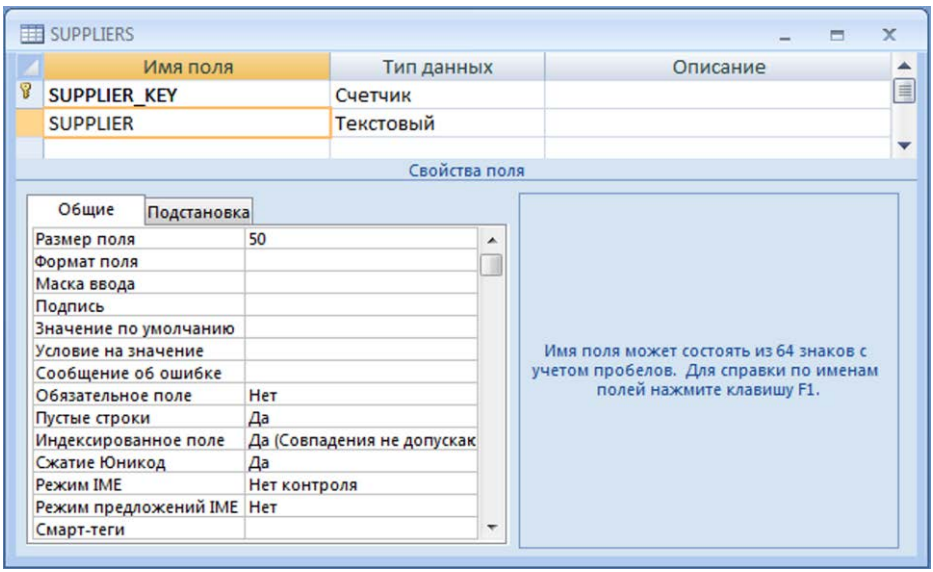


Рис. 17.1. Таблица поставщиков в режиме конструктора

Создаем новый проект в Delphi. Для этого воспользуемся пунктом меню **File | New | VCL Forms Application - Delphi**. Сразу, после появления на экране главной формы, сохраняем новый проект в отдельную папку.

Теперь настал черед визуального проектирования, для нашего приложения понадобится помощь четырех компонентов:

1. Таблица `ADOTable1`, этот компонент вы обнаружите на страничке **dbGo** палитры компонентов.
2. Источник данных `DataSource1` (страничка **Data Access**).
3. Навигатор данных `DBNavigator1` (страничка **Data Controls**).
4. Сетка для отображения данных `DBGrid1` (страничка **Data Controls**).

Разместив все перечисленные компоненты на форме проекта (рис. 17.2), выберите таблицу — компонент `ADOTable1`. Нам предстоит проделать наиболее важную операцию — подключить компонент-таблицу к физической таблице, которая (как вы помните) хранится в файле под именем books.accdb. Все компоненты со страни-

цы dbGO палитры компонентов Delphi для взаимодействия с базами данных применяют разработанную в стенах Microsoft технологию ActiveX Data Objects (ADO). Для настройки подключения нам следует обратиться к свойству `ConnectionString` компонента `ADOTable1`. Щелчок по кнопке с многоточием напротив свойства `ConnectionString` выведет на экран компьютера редактор строки соединения.

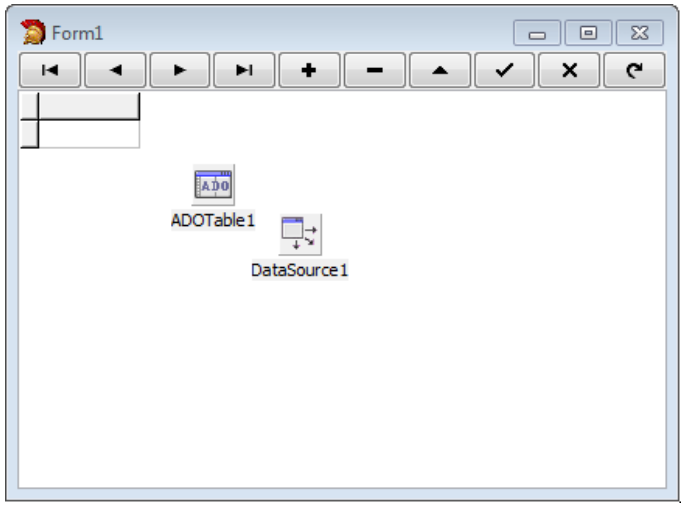


Рис. 17.2. Внешний вид первого проекта БД

Для этого нам необходимо выполнить определенную последовательность действий:

1. Выберите элемент **Use Connection String** (Использовать строку соединения) и нажмите кнопку **Build** (Построить). Это действие вызывает редактор "Свойства канала передачи данных" (рис. 17.3).
2. На странице **Поставщик данных** редактора **Свойства канала передачи данных** выделите строку **Microsoft Office 12 Access Database Engine…** и нажмите кнопку **Далее**.
3. На странице **Соединение** в строке ввода **Источник данных** укажите путь и имя файла базы данных. Щелкните по кнопке **Проверить соединение**. Если вы ни в чем не ошиблись, то на экране окажется сообщение — **Проверка соединения выполнена**.
4. Нажав на кнопку **OK**, запомните настройки строки соединения.

В результате наших действий в свойстве `ConnectionString` компонента `ADOTable1` появится примерно такая строка, содержащая параметры соединения:

```
Provider=Microsoft.ACE.OLEDB.12.0;Data Source=D:\DATA\books.accdb;
Persist Security Info=False
```

Если вы внимательно изучите параметры соединительной строки и их значения, то обязательно обнаружите путь к папке с демонстрационной таблицей, в моем случае это каталог `D:\DATA\`.

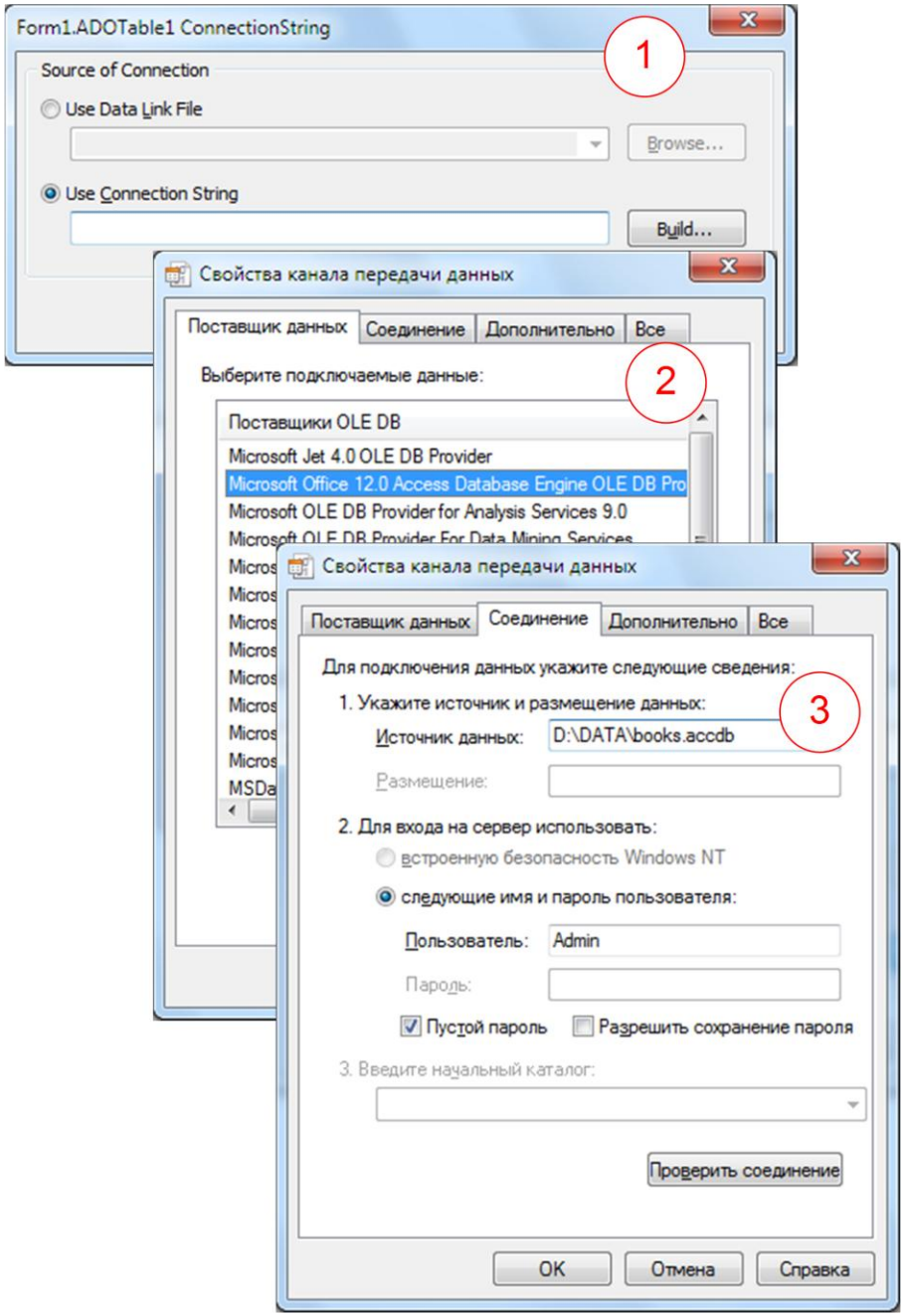


Рис. 17.3. Этапы настройки строки соединения ADO

Закончив подготовку соединительной строки, мы получаем возможность подключить компонент `ADOTable1` к файлу с таблицей. Для этого в инспекторе объектов находим свойство `TableName` и вводим имя таблицы `TableName:=SUPPLIERS`.

Настройка подключения завершена, нам осталось только активировать его, для этого устанавливаем свойство компонента `Active` в состояние `true`.

Можете перевести дух. Нами сделано 90% работы, теперь настроим оставшиеся три компонента.

- В свойство `DataSet` компонента `DataSource1` передаем имя компонента `ADOTable1`.
- В свойство `DataSource` компонента `DBGrid1` передаем имя источника данных `DataSource1`.
- В свойство `DataSource` компонента `DBNavigator1` передаем имя источника данных `DataSource1`.

"Программирование" завершено. Осталось нажать клавишу <F9>. Наш первый проект базы данных готов к работе!

Архитектура приложения БД

Как бы это не показалось парадоксальным, но впечатляющее разнообразие компонентов Delphi и технологий связанных с работой в БД могут сослужить плохую службу, в особенности, когда за дело берется начинающий разработчик. В перечень недочетов, характерных для проектов будущих мастеров программирования, входят две диаметрально противоположные ошибки: попытка задействовать в приложении чуть ли не все имеющиеся в Delphi компоненты или стремление решить все проблемы путем "выворачивания рук" трем-четырем элементам управления. Безусловно, в конечном счете, при подборе компонентов разработчик найдет наиболее рациональное решение, но лучше делать не методом проб и ошибок, а на основе знания концепции построения приложения баз данных Delphi.

Для того чтобы приложение Delphi смогло взаимодействовать с СУБД, ему требуется посредничество драйверов и клиентских библиотек. В операционной системе Windows имеется ряд предустановленных клиентов, в первую очередь, это довольно старые драйверы провайдера открытого подсоединения к БД (*Open Database Connectivity*, ODBC) и технология объектов данных ActiveX (*Microsoft ActiveX Data Objects*, ADO). Именно ADO помогла нам создать первое приложение баз данных. У Embarcadero Technologies имеются свои собственные решения: dbExpress, драйверы для работы с СУБД InterBase и созданная на заре появления Delphi, а на сегодня устаревшая технология Borland DataBase Engine (BDE). Выбор того или иного механизма доступа к БД — весьма ответственное решение, т. к. он существенно влияет на процесс проектирования приложения, в частности определяет набор применяемых компонентов Delphi.

Все компоненты, имеющие прямое отношение к построению настольных и обычных клиент-серверных проектов БД, разделяются на четыре группы (см. рис. 17.4):

1. Компоненты, обеспечивающие соединение приложения с базой данных.
2. Компоненты, реализующие объектно-ориентированное представление реляционных наборов данных (далее мы их просто станем называть наборами данных).

3. Источник данных, связывающий конкретный набор данных, например таблицу, с элементами управления данными.
4. Элементы управления данными, умеющие отображать и редактировать записи в наборе.

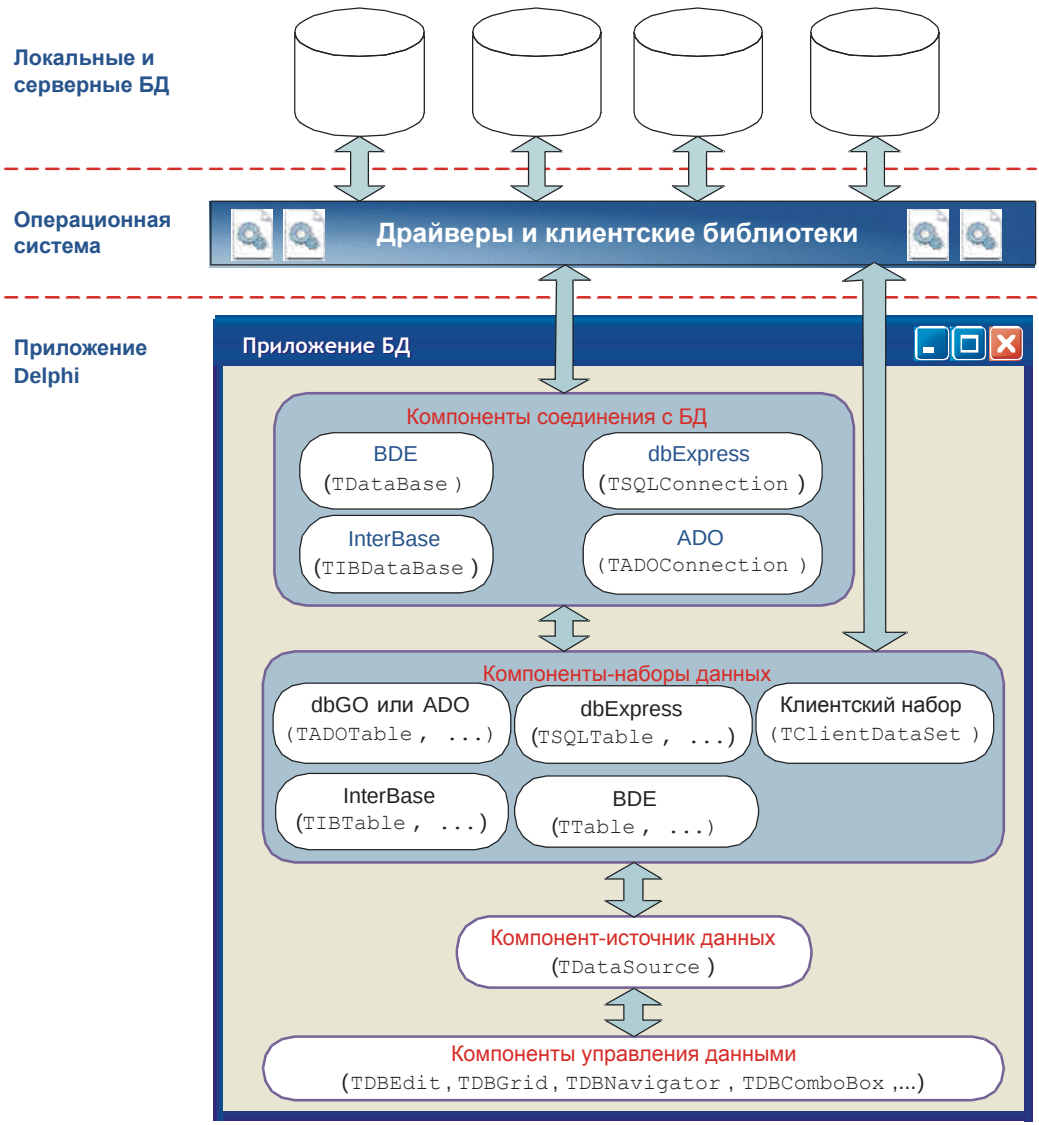


Рис. 17.4. Взаимодействие компонентов Delphi для работы с БД

В предложенном в начале главы примере были задействованы три ингредиента из четырех. Набор данных — таблица `ADOTable1`, источник данных — `DataSource1` и два элемента управления данными: навигатор `DBNavigator1` и сетка `DBGrid1`.

Замечание

Когда мы приступим к обсуждению многоуровневых проектов БД, нам потребуется помощь еще одной группы компонентов — провайдеров. Благодаря провайдерам обеспечивается взаимодействие между сервером приложений и тонким клиентом.

Общая характеристика компонентов соединения с БД

Назначение компонентов, входящих в первую группу, заключается в централизации процесса соединения приложения с сервером баз данных. Кроме того, на этом уровне реализуется механизм регистрации пользователя и процедуры управления транзакциями. Входящие в группу компоненты можно дополнительно классифицировать по принадлежности к технологии доступа к данным. В соответствии с такой классификацией в нашем распоряжении имеются четыре линейки компонентов: Borland Database Engine (BDE), dbGo или (в более ранних версиях Delphi) ActiveX Data Objects (ADO), dbExpress и InterBase. Мы не станем здесь подробно обсуждать особенности того или иного механизма доступа, отметим лишь то, что компоненты InterBase специализируются исключительно на одноименной СУБД и на ее клонах FireBird и Yaffil, все остальные компоненты универсальны и способны работать с широчайшим спектром локальных и клиент-серверных баз данных.

Внимание!

Механизм доступа к данным BDE на сегодняшний день уже устарел и не рекомендуется для использования. Современные версии Delphi поддерживают технологию BDE только ради обеспечения обратной совместимости с приложениями старого парка.

С точки зрения объектно-ориентированной модели все родовые черты компонентов соединения с БД унаследованы от ключевого класса `TCustomConnection`. Указанный абстрактный класс описан в модуле DB, именно он определяет всю идеологию процесса взаимодействия клиентского приложения с удаленным сервером БД.

Общая характеристика компонентов наборов данных

Перейдем к рассмотрению общих черт элементов второй группы — *наборов данных*. Наборам данных делегировано право представлять в наших приложениях отношения (реляционные таблицы, просмотры, запросы и т. д.). Другими словами, наборы данных обеспечивают разработчику удобный интерфейс управления реляционными таблицами и у начинающего программиста складывается впечатление, будто таблицы — это обычные компоненты VCL, а не хранящиеся на сервере СУБД структурированные файлы с данными. В результате программист приобретает возможность управлять расположенными в таблицах данными с помощью

доброжелательных свойств, методов и событий компонентов Delphi. Все операции с данными опираются на инструкции структурированного языка запросов SQL, поэтому в большинстве компонентов нам требуется явным образом описать команду SQL или указать имя и параметры хранимой процедуры.

Замечание

Термин *набор данных* (dataset) станет едва ли не самым частым гостем этих страниц. Давайте сразу договоримся, что под набором данных мы будем понимать компонент Delphi, способный подключаться к таблицам (представлениям, хранимым процедурам) БД и представлять их содержимое в виде коллекции строк и столбцов с данными.

При работе с локальными БД наборы данных умеют получить доступ к строкам физической таблицы в обход компонентов соединения. Поэтому тривиальные однопользовательские проекты баз данных Delphi обходятся без поддержки компонентов соединителей и общаются с драйверами БД напрямую (см. рис. 17.4).

Все наборы данных построены на базе фундаментального абстрактного класса `TDataSet`. Общий вклад `TDataSet` в концепцию наборов данных столь значителен, что ему посвящена отдельная глава этой книги. Многочисленную плеяду "разношерстных" потомков `TDataSet` помимо классификации по технологии доступа к данным можно разделить на четыре направления:

1. *Наборы данных таблицы*. В названии класса таких компонентов присутствует английское слово "Table", например `TIBTable` или `TSQLTable`.
2. *Наборы данных запросы*. В названии вы найдете слово "Query", например `TADOQuery`, `TSQLQuery`.
3. *Наборы данных хранимые процедуры*. В имени таких компонентов есть аббревиатура "StoredProc", например `TIBStoredProc`.
4. Наборы данных, способные взять на себя функционал всех перечисленных в пунктах 1–3 категорий компонентов. Имена универсалов заканчиваются словом "DataSet" — `TADODataSet` и `TSQLDataSet`.

Компоненты-таблицы предназначены для построения простейших настольных приложений баз данных, а в клиент-серверных продуктах они применяются достаточно редко (хотя такая возможность не исключается). Компонент обычно позволяет не только просматривать, но и добавлять, редактировать и удалять данные из физической таблицы. Компонент-таблица настолько глубоко продуман разработчиками Delphi, что в простейшем случае он сможет составить ядро приложения, в котором не будет присутствовать ни одной строки нашего кода. В этом вы убедились, если повторили пример, приведенный в начале главы. Эта примечательная особенность очень нравится начинающим программистам, еще не успевшим освоить структурированный язык запросов. Дело в том, что таблица самостоятельно генерирует требуемые команды SQL. Более того, команды самостоятельно адаптируются к определенному диалекту SQL, который исповедует интересующая нас СУБД.

В сравнении с таблицами компоненты-запросы не столь доброжелательны по отношению к лицам, делающим первые шаги на поприще программирова-

ния, т. к. они требуют безукоризненного знания языка SQL. Текст команды должен соответствовать синтаксису SQL обслуживаемой СУБД и никаких расхождений не допускает. Компонент способен отправить в адрес сервера баз данных текст запроса SQL и при необходимости вернуть результаты его выполнения. В качестве результата может быть получено одно значение, набор строк из одной таблицы или строки, объединяющие несколько таблиц. Для некоторого упрощения работы программиста большинство компонентов запросов располагают встроенным редактором, оказывающим помощь при построении запроса. В перечень неоспоримых достоинств компонента-запроса стоит включить предоставляемую им возможность строить динамические команды SQL. Это весьма удобно, особенно в ситуации, когда нам заранее неизвестно, какие данные потребуются пользователю программного продукта.

В отличие от запросов, хранимые процедуры не требуют от нас описания команды SQL. Их назначение заключается в вызове заранее подготовленных процедур, хранящихся на стороне сервера. Хранимая процедура представляет собой заранее подготовленную и откомпилированную команду или объединение нескольких команд, написанных на языке SQL. Зачастую, кроме имени процедуры компоненту приходится отправить СУБД некоторый перечень параметров. Такая необходимость может возникнуть, например, когда процедура предназначена для осуществления вставки записи в таблицу. По аналогии с компонентами таблицей и запросом хранимая процедура способна возвращать строки из одной или нескольких таблиц. Как вы уже догадались, хранимые процедуры существуют только в приложениях баз данных, поддерживающих модель "клиент-сервер" и построенных на основе современных СУБД Oracle, Microsoft SQL Server, InterBase, Informix и т. п. Простейшие настольные системы баз данных класса Microsoft Access, Paradox, Microsoft FoxPro, dBase и т. д. с хранимыми процедурами "не дружат".

Особое место среди наборов данных отведено компоненту `TClientDataSet`. За счет того, что компонент способен забирать данные в локальный буфер и обрабатывать их в отрыве от БД, он находит широкое применение во всем спектре ориентированного на БД программного обеспечения, особенно при работе с однонаправленными курсорами.

Источник данных

Если перевести название компонента `TDataSource` с английского, то его можно называть источником данных. Это едва ли не самый большой "аскет" из всех компонентов Delphi, в чем вы убедитесь сами, когда приступите к его изучению — перечень свойств, методов и обработчиков событий `TDataSource` едва превышает десяток наименований. Вместе с тем без источника данных не обходится ни одно клиентское приложение БД. Компонент `TDataSource` действительно служит источником данных для компонентов управления данными, реализуя функции моста между ними и разномастными наборами данных.

Общая характеристика компонентов управления данными

Все компоненты управления данными в палитре компонентов Delphi сосредоточены на линейке **Data Controls**. В отличие от всех рассмотренных ранее компонентов, компоненты управления данными являются визуальными и предназначены для построения пользовательского интерфейса приложения. Как мы уже говорили, все элементы управления данными беспомощны без поддержки компонента `TDataSource`, источник данных выступает в роли связующего звена между наборами данных и элементами управления данными (см. рис. 17.4).

Все элементы управления данными разделяются на три ветви (рис. 17.5):

- Компоненты-сетки — позволяют представлять реляционную таблицу или результат запроса SQL в виде двумерной таблицы (компонент `TDBGrid`) или в свободном формате на нескольких панелях (компонент `TDBCtrlGrid`).

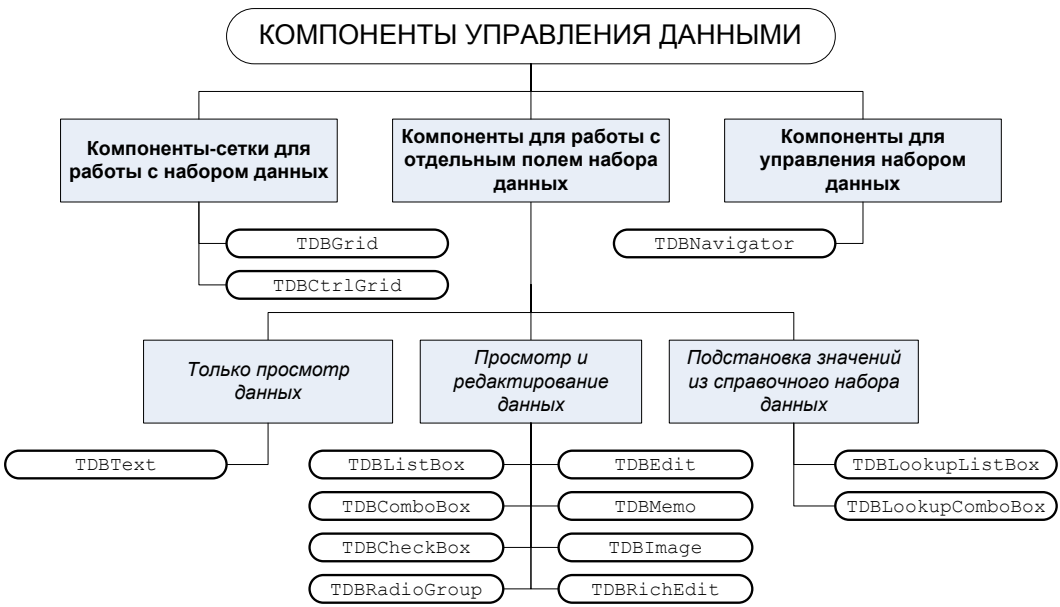


Рис. 17.5. Классификация компонентов управления данными

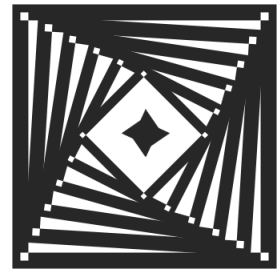
- Компоненты управления данными, предназначенные для обслуживания отдельного поля таблицы — здесь сосредоточены элементы управления, применяемые при создании форм ввода и редактирования данных. Внешний вид и поведение компонентов управления данными идентичны обычным строкам ввода, панелям переключателей, комбинированным спискам, с той лишь разницей, что, в отличие от своих классических коллег, они обладают информационной связью с определенным столбцом реляционной таблицы и способны просматривать и редактировать содержание активной записи.

- Компонент `TDBNavigator` (далее мы станем называть навигатором) — отвечает за перемещение курсора по строкам набора данных и за перевод набора в режимы вставки, редактирования и удаления записи.

Резюме

Для создания приложения баз данных в Delphi реализовано четыре семейства компонентов, отвечающих за соединение с базой данных, представление набора данных, обеспечение взаимодействия между набором данных и элементами управления данными, а также собственно элементы управления данными.

Среда разработки Delphi поддерживает широкий спектр СУБД, начиная от простейших настольных систем и заканчивая сверхсовременными клиент-серверными. Взаимодействие с СУБД можно реализовать с помощью различных технологий: dbExpress, InterBase и dbGo (ADO). Кроме того, Delphi способна работать с таблицами, представленными в формате стандарта XML.



Глава 18

Универсальный набор данных *TDataSet*

Герой этой главы — один из самых незаурядных представителей плеяды классов визуальной библиотеки компонентов Delphi — класс `TDataSet`. Столь пристальное внимание к классу обусловлено тем, что он возглавляет иерархию классов, предназначенных для доступа к реляционным данным, и служит в своем роде первопроходцем в этом направлении (рис. 18.1). Хотя значительная часть объявленных в классе методов не наполнена содержанием (они являются абстрактными), `TDataSet` определяет концепцию построения многочисленных дочерних классов, закладывает в них свои генетические черты и заставляет заполнить программной логикой заранее предопределенные в нем методы. Помимо абстрактной начинки класс обладает списком методов, обеспечивающих ключевой функционал, необходимый для работы с самым общим (независимым от формата хранения данных, будь то Microsoft Access, InterBase, MS SQL Server, Oracle и т. п.) набором данных, именно поэтому в дальнейшем мы станем называть `TDataSet` *универсальным набором данных*.

Каждая из дочерних ветвей класса появлялась на свет вместе с рождением очередной технологии доступа к данным. Первый в перечне — класс `TCustomClientDataSet`, он необходим как в простейших базах данных, зачастую даже не нуждающихся в СУБД, так и в сложных многоуровневых проектах. В частности, на основе `TCustomClientDataSet` реализован весьма остроумный компонент `TClientDataSet`, позволяющий создавать проекты БД практически на пустом месте — на фундаменте обычного бинарного файла или файла в стандарте XML. Для взаимодействия с базами данных с помощью средств, предлагаемых корпорацией Microsoft, предназначены еще два потомка универсального набора данных: `TCustomADODataSet` и `TCustomADONETConnector`. И первый, и второй классы ориентированы на обслуживание наборов записей ActiveX Data Objects (ADO) с той лишь разницей, что `TCustomADODataSet` применяется в классических проектах Delphi, построенных на базе визуальной библиотеки компонентов, а класс `TCustomADONETConnector` — в платформе .NET FrameWork. Для работы с сервером баз данных семейства InterBase предназначен `TIBCustomDataSet`, он дает жизнь семейству компонентов, способных применяться как в проектах VCL, так и в программной модели .NET.

Специально для обслуживания данных в формате Unicode в последних версиях Delphi появился класс `TWideDataSet`, это предок компонентов сравнительно молодой технологии доступа к данным dbExpress. Поэтому наборы данных dbExpress, позволяющие строить как простые клиентские приложения баз данных малой и средней степени сложности, так и продвинутые трехуровневые проекты БД на основе технологии DataSnap также впитали в себя дух класса `TDataSet`.

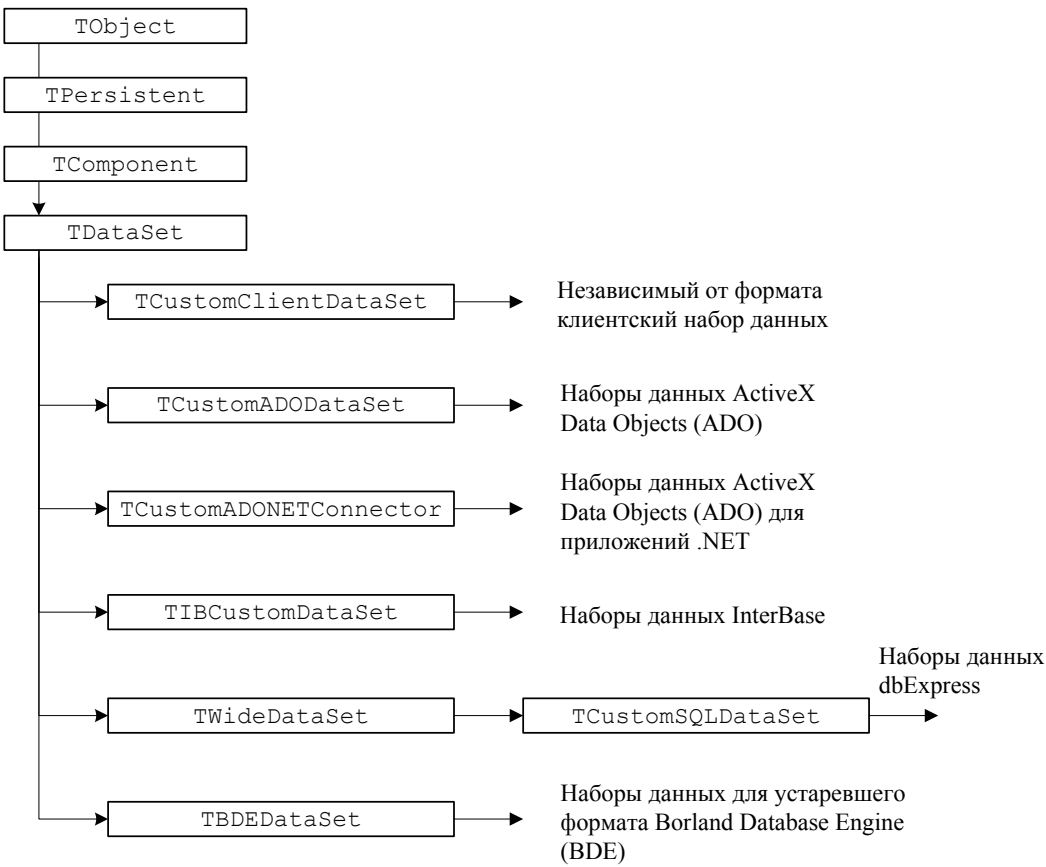


Рис. 18.1. Иерархия наследников `TDataSet`

Замечание

Класс `TWideDataSet` появился на свет в Delphi 2007, до этого момента `TCustomSQLDataSet` брал свое начало непосредственно от универсального набора данных `TDataSet`.

Последнее место отведено старейшине `TBDEDataSet`, указанный абстрактный класс выступает в роли проводника идей уже устаревшей технологии, разработанной Borland (так раньше называлась компания, создавшая Delphi) в начале 1990-х годов — BDE. Развитие BDE остановилось в начале 2000-х годов и сегодня поддерживается в Delphi лишь для обратной совместимости.

Функционал набора данных

Класс `TDataSet` нацелен на решение следующих задач:

- открытие и закрытие набора данных;
- перемещение по записям набора данных;
- редактирование записей в наборе;
- доступ к полям таблиц;
- создание вычисляемых полей и осуществление расчетов;
- фильтрацию набора данных;
- поиск данных;
- определение текущего состояния набора.

ВНИМАНИЕ!

На уровне класса `TDataSet` не определен один очень важный аспект работы с базами данных — подключение к СУБД. Это объясняется существенными различиями во взаимодействии между СУБД и компонентами различных технологий. Подробно порядок присоединения компонента набора данных к СУБД мы рассмотрим в следующих главах.

Открытие и закрытие набора данных

Первая, и едва ли не самая важная черта любого компонента набора данных заключается в способности получить доступ к данным, хранящимся в виде одного или нескольких файлов на жестком диске или в оперативной памяти компьютера. Данные могут быть представлены не только в виде реляционной таблицы, это может быть и отношение, полученное в результате выполнения запроса `SELECT`, и результаты, возвращаемые хранимой процедурой.

Для реального компонента операцию открытия набора данных обязательно должна предварять настройка соединения с физической таблицей БД. В предыдущей главе мы продемонстрировали работу с одним из потомков универсального набора данных `TDataSet` — компонентом-таблицей `TADOTable`. В примере мы собрали строку соединения, в которой указывалось местонахождение файла таблицы и ее имя. Если вы воспользуетесь другим компонентом или компонентом, исповедующим альтернативную технологию доступа к БД, то порядок подключения компонента к физической таблице, безусловно, изменится. Но, несмотря на некоторые различия между потомками `TdataSet`, практически у всех имеются одинаковые свойства и методы, управляющие открытием и закрытием набора данных.

Простейшим рычагом, позволяющим управлять процессом открытия и закрытия большинства наборов данных, выступает свойство

```
property Active: Boolean;
```

Если оно вернет значение `true`, значит, набор данных активен и, как минимум, доступен для просмотра. Ключевое достоинство свойства `Active` — возможность активизации набора данных не только во время выполнения приложения, но и в

процессе разработки проекта. Программист получает возможность буквально "потрогать" данные руками и упрощает процесс проектирования визуального интерфейса.

Возможности свойства `Active` продублированы методами:

```
procedure Open;  //открытие набора
procedure Close; //закрытие набора
```

Перевод в `true` свойства `Active` (или вызов метода `Open`) последовательно вызовет обработчики событий `BeforeOpen()` и `AfterOpen()`.

Замечание

Если набор данных не пуст, то после его открытия курсор устанавливается на первую запись. Соответственно свойство `BOF` набора данных установится в `true`.

Получив доступ к таблице БД, потомок класса `TDataSet` считывает часть, а может быть и все записи (если их не так много) из физической таблицы в свой собственный буфер памяти. С этого момента пользователь получает возможность просматривать и редактировать строки таблицы.

Замечание

Вызов метода `Close()` (присвоение значения `false` свойству `Active`) приведет к цепи из событий `BeforeClose()` и `AfterClose()`.

Обновление набора данных

Если разрабатываемое приложение относится к разряду многопользовательских, то вполне вероятно, что сохраненный в локальном наборе данных окажется в неактуальном состоянии. Почему? Тому несколько причин, и самая простая из них следующая. Ранее уже упоминалось, что после доступа к таблице (представлению, результату запроса) набор данных обычно буферизует полученные строки. Таким образом, для работы пользователю предоставляется не данные из физической таблицы, а их копия, отображенная в локальный кэш. Пока один пользователь неторопливо просматривает строки таблицы из своего буфера памяти, второй пользователь получает доступ к этим данным с другого компьютера, вносит изменения и сохраняет их в таблицу. Но на компьютере первого пользователя внесенные изменения не отражаются, и с этого момента он работает с неактуальной копией.

Для обновления строк в ранее открытом наборе данных предусмотрен метод

```
procedure Refresh;
```

Вызов процедуры гарантирует, что с этого момента приложение получит самые свежие данные. Метод `Refresh()` инициирует события `BeforeRefresh()` и `AfterRefresh()`.

Внимание!

Метод `Refresh()` есть не у всех наследников `TDataSet`. Например, его вызов не имеет смысла для компонентов-запросов `TADOQuery`, `TADOStoredProc`, т. к. эти элементы

управления по определению не работают с "живыми" данными. А для обновления устаревших результатов достаточно отправить запрос повторно — вызвав последовательно пару методов `Close()` и `Open()`.

Еще один способ обновления данных основан на обращении к процедуре

```
procedure Resync(Mode: TResyncMode);
```

В большинстве случаев о вызове этой процедуры позаботится сам компонент, например, для синхронизации набора данных с физическими данными таблицы при создании запросов, которые могут изменить позицию курсора. Параметр `Mode` определяет особенности синхронизации данных. Так, в состоянии `rmExact` процедура инициирует исключительную ситуацию, если в результате обновления данных курсор не смог быть установлен на ту же самую запись. Значение `rmCenter` укажет методу, что после обновления данных курсор должен быть установлен в середину буферизованного набора.

Перемещение по набору данных

Мы уже знаем, что открытый непустой набор данных обеспечивает доступ к своим записям. Текущая запись набора обычно называется активной записью, именно на ней установлен курсор набора. Перемещение курсора приводит к смене активной (текущей) записи. Естественно, в классе `TDataSet` реализован целый ряд методов, обеспечивающих перемещение курсора по набору. К наиболее часто используемым процедурам такого рода относятся следующие:

```
procedure First; //переход на первую запись
procedure Prior; //возврат к предыдущей записи
procedure Next; //переход на следующую запись
procedure Last; //переход на последнюю запись
```

Методы `First()` и `Last()` позиционируют курсор соответственно на первую и последнюю запись. Процедуры `Prior()` и `Next()` осуществляют пошаговый переход соответственно к предыдущей и последующей записям.

Если набор данных работает в режиме фильтрации (заполнено свойство `Filter`), то вместо приведенных ранее процедур целесообразно применять одноименные функции:

```
function FindFirst : Boolean;
function FindPrior : Boolean;
function FindNext : Boolean;
function FindLast : Boolean;
```

Разница между первым и вторым квартетами методов в том, что группа функций `Find...` учтет параметры фильтрации и будет перемещаться по записям, соответствующим критериям фильтрации. Если же фильтр отключен, то поведение функций и процедур ничем не отличается.

Для прыжка сразу на несколько записей понадобится метод

```
function MoveBy(Distance: Integer): Integer;
```

В качестве параметра `Distance` передается дистанция, на которую мы намерены переместиться по набору, от текущей записи. Если значение отрицательное, то перемещение осуществляется к началу набора, положительное — к концу. За отправную точку принимается текущая запись набора. Функция возвращает реальное число записей, на которое удалось переместить курсор.

Вызов методов перемещения имеет смысл только в том случае, если набор не пуст. Узнать об этом поможет метод

```
function IsEmpty: Boolean;
```

Если в наборе нет ни одной строки, то функция возвратит `true`. Если в наборе есть записи, то общее количество доступных строк нам поведает свойство

```
property RecordCount: Integer;
```

Замечание

С процессом перемещения курсора по набору данных связаны два события `BeforeScroll()` и `AfterScroll()`.

Порядковый номер текущей записи хранится в свойстве

```
property RecNo: Integer;
```

Это свойство доступно только в компоненте `TTable`.

Два специальных свойства сигнализируют о местонахождении курсора. Если курсор находится на самой первой записи в наборе, то в `true` установится свойство

```
property Bof: Boolean;
```

Если курсор окажется на самой последней записи, то в состояние `true` установится свойство:

```
property Eof: Boolean;
```

С помощью перечисленных свойств часто реализуют циклы, перебирающие все записи внутри набора (листинг 18.1).

Листинг 18.1. Цикл перебора всех записей набора данных

```
WITH ADOTable1 DO
BEGIN
 DisableControls;
 First; //к первой записи
   WHILE EOF<>TRUE DO //пока не достигнут конец набора
   BEGIN
    {Заполняем список значениями из поля 'SUPPLIER'}
    ListBox1.Items.Add(FieldByName('SUPPLIER').AsString);
    Next; //к следующей записи
   END;
 EnableControls;
END;
```

Обратите внимание на порядок применения в листинге 18.1 пары процедур:

```
Procedure DisableControls;
Procedure EnableControls;
```

Методы отвечают за отключение и включение подключенных к набору данных визуальных элементов управления. Вызывайте эти методы в указанной в примере последовательности. Вызов метода `DisableControls()` значительно ускорит процесс перебора записей, т. к. избавит приложение от необходимости постоянной (с каждой сменой записи) перерисовки содержимого элементов управления.

ВНИМАНИЕ!

Перемещение по набору данных может оказаться затруднительным, если мы работаем с однонаправленным (*unidirectional*) курсором. В этом случае из всех рассмотренных методов работоспособность сохраняют всего два: `First()` и `Next()`. С такой ситуацией мы столкнемся при изучении компонентов dbExpress.

Создание закладок и переход к закладке

При работе с таблицей очень часто возникает необходимость многократного перехода от одной строки к другой (подчас находящейся в другом конце набора данных) с последующим возвратом к первой обрабатываемой записи. Когда в таблице хранится пара-другая тысяч строк, подобные операции занимают очень много времени. Для того чтобы пользователь мог в любой момент времени вернуться к недавно обрабатываемой строке, у потомков `TDataSet` реализован простой, но весьма эффективный способ запоминания местоположения строки в наборе данных — создание *закладки* (*bookmark*). В закладке сохраняется некоторое уникальное значение, однозначно идентифицирующее запись, активную на момент создания закладки. Как только возникает необходимость возврата к помеченной записи, набор данных считывает значение закладки и переводит курсор на нужную строку.

Со сменами версий языка программирования менялось программное решение. В самой первой версии Delphi закладки представляли собой обыкновенные 4-байтовые указатели. Начиная с третьей версии Delphi, от указателей отказались, и закладка стала описываться строковым типом данных. В современных версиях строковый тип данных сменил байтовый массив:

```
type TBookmark = TBytes;
```

Порядок работы с закладкой элементарен. Чтобы запомнить позицию текущей строки набора данных, мы вызываем метод

```
function GetBookmark: TBookmark;
```

Функция возвращает данные в переменную-закладку. Для перехода к закладке следует воспользоваться услугами процедуры

```
procedure GotoBookmark(Bookmark: TBookmark);
```

Вместо функций `GetBookmak()` и `GotoBookmark()` допускается использовать свойство

```
property Bookmark: TBookmark;
```

Когда необходимость в услугах закладки отпадает, распределенный под нее ресурс освободит метод

```
procedure FreeBookmark(Bookmark: TBookmark);
```

При долгой работе с таблицей запись, помеченную закладкой, можно удалить, закладку можно повредить или могут измениться условия отбора записей в результирующий набор данных. В подобной ситуации стоит убедиться, что закладка действительна. Функция

```
function BookmarkValid(Bookmark: TBookmark): Boolean;
```

возвращает значение `true`, если набор данных считает, что закладка корректна.

Внимание!

К сожалению, закладки не поддерживаются однонаправленными наборами данных dbExpress.

Работа с закладками не вызовет никаких сложностей даже у начинающего. Нам необходимо объявить переменную, способную хранить закладку (в нашем примере это `BM:TBookMark`), и две кнопки. Щелчок по кнопке `btnGetBookmark` запоминает местоположение закладки, щелчок по кнопке `btnGoToBookmark` возвращает курсор на ранее сохраненное место и удаляет закладку (листинг 18.2).

Листинг 18.2. Демонстрация работы с закладкой

```
private
    BM:TBookmark; //переменная для запоминания закладки
//…
procedure TForm1.btnGetBookmarkClick(Sender: TObject);
begin
BM:=ADOTable1.GetBookmark; //помечаем запись закладкой
end;

procedure TForm1.btnGoToBookmarkClick(Sender: TObject);
begin
if ADOTable1.BookmarkValid(BM)=true then //проверка наличия закладки
begin
  ADOTable1.GotoBookmark(BM); //возвращаемся к закладке
  ADOTable1.FreeBookmark(BM); //удаляем закладку
  btnGoToBookmark.Enabled:=False;
end;
end;
```

Две закладки можно сравнивать друг с другом. Если они ссылаются на одну и ту же строку таблицы, то метод

```
function CompareBookmarks(Bookmark1: TBookmark;
                          Bookmark2: TBookmark): Integer;
```

возвратит нулевое значение, то же самое произойдет, если хотя бы одна из закладок пуста. Если же закладки не совпадают, то функция `CompareBookmarks()` проинформирует нас об этом, выдав целочисленное значение (разницу между значениями закладок).

Состояние набора данных

В классе `TDataSet` реализовано весьма полезное свойство, информирующее нас о текущем состоянии набора данных:

```
property State: TDataSetState; //только для чтения
```

Возможные значения `TDataSetState` приведены в табл. 18.1.

Таблица 18.1. *Состояния набора данных* `TDataSetState`

Значение	Описание
`dsInactive`	Набор данных закрыт, соответственно его записи недоступны
`dsBrowse`	Набор данных находится в режиме просмотра, причем его записи не претерпевали изменений
`dsEdit`	Текущая запись находится в режиме редактирования и готова к изменениям
`dsInsert`	Текущая запись вставлена в буфер памяти компонента, но пока не сохранена в таблице. Эту запись можно изменить, сохранить или пользователь может вообще отказаться от нее, удалив ее из буфера
`dsSetKey`	В этом состоянии могут находиться только элементы управления `TTable` (страница палитры компонентов BDE) и `TClientDataSet` во время осуществления поиска по ключу (см. методы `SetKey`, `FindKey`, `SetRange`)
`dsCalcFields`	Производится пересчет вычисляемых полей — вызван обработчик события `OnCalcFields`
`dsFilter`	Осуществляется фильтрация данных — вызван обработчик события `OnFilterRecord`
`dsNewValue`	Временное состояние, набор данных переходит в него во время обращения к свойствам `NewValue`, `OldValue` и `CurValue` поля таблицы (класс `TField`). Эти три состояния могут пригодиться, если вы займетесь разработкой своих собственных компонентов на базе класса `TDataSet` или будете разрабатывать многопользовательскую БД
`dsOldValue`	
`dsCurValue`	

Таблица 18.1 *(окончание)*

Значение	Описание
`dsBlockRead`	Осуществляется ускоренное перемещение по набору данных при помощи методов `DisableControls` и `EnableControls` (без отображения факта перехода к следующей записи в элементах управления данными)
`dsInternalCalc`	Состояние набора данных, когда осуществляется пересчет значений во внутренних вычисляемых полях типа `fkInternalCalc`
`dsOpening`	Набор данных открывается, но считывание данных из таблицы еще не окончено

Обратите внимание на рис. 18.2, отображающий последовательность переходов между состояниями набора данных. Перед открытием набор находится в пассивном состоянии `dsInactive`, о процессе открытия недолгое время сигнализирует состояние `dsOpening`. После открытия набор окажется в режиме просмотра данных — `dsBrowse`. Теперь при переводе набора данных в любое другое состояние `dsBrowse` обязательно будет служить промежуточным этапом. Состояние `dsBrowse` похоже на нейтральное положение рычага в ручной коробке переключения скоростей автомобиля. При переключении скорости, рычаг всегда пройдет через нейтральное положение.

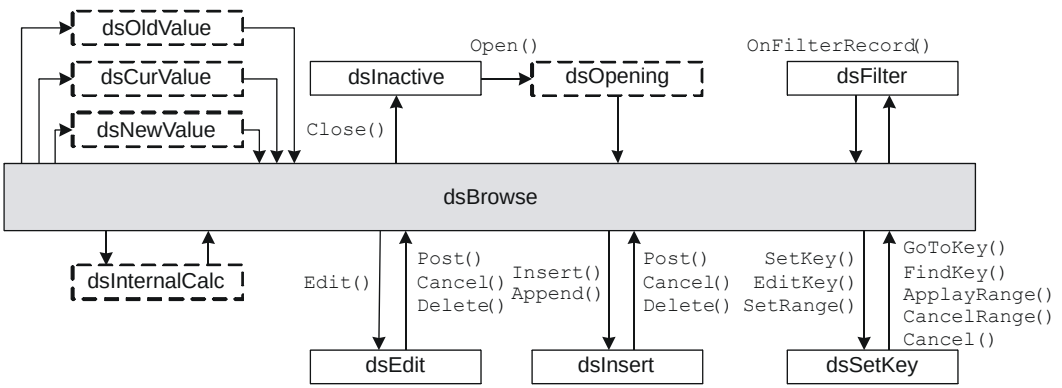


Рис. 18.2. Диаграмма переходов между состояниями набора данных

На рис. 18.2 также указаны основные методы, с помощью которых программист получает возможность "переключать" состояния набора данных. Впрочем, как всегда существуют и исключения… Допустим, что набор данных переведен в режим вставки новой записи (`dsInsert`) или редактирования (`dsBrowse`), пользователь заполняет запись новыми данными и вызывает метод сохранения `Post()`. Согласно диаграмме набор должен перейти в режим `dsBrowse`. Однако если в данные закралась ошибка, например, не заполнено обязательное поле, то вызов метода

Post() инициирует исключительную ситуацию. В этом случае набор данных не перейдет в режим просмотра и останется в исходном состоянии (dsInsert или dsEdit).

Редактирование записей в наборе

Модификация данных в наборе данных возможна только в ситуации, когда свойство

```
property CanModify: Boolean; //только для чтения
```

возвращает значение false. Это признак того, что набор данных "категорически против" редактирования своих строк. Такая "строптивость", в первую очередь, объясняется особенностью доступа к данным. Например, если набор сформирован силами запроса (компонент T…Query) и объединяет в себе данные от нескольких таблиц, то в явном виде он не пригоден к модификациям.

Итак, получив потенциальное согласие набора данных на внесение изменений, можно добавить в конец набора очередную запись. Для этого используйте метод

```
procedure Append;
```

Если вы хотите, чтобы новая запись оказалась в месте положения курсора, то вспомните о процедуре

```
procedure Insert;
```

Вызов методов Append и Insert подготовит набор данных к приему новой записи. Если, например, в качестве средства визуализации выбрана сетка TDBGrid, то после вызова функции на экране своего компьютера пользователь увидит новую пустую строку, готовую к получению данных. Ему останется ввести информацию. Если вы не доверяете пользователю, то одновременно с операцией добавления (или вставки) новой строки можно сразу заполнить запись данными. Для этого применяют методы

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

Особенность этих методов в том, что в качестве параметров направляется массив значений, непосредственно вносимых в таблицу (листинг 18.3). Будьте внимательны — последовательность значений передаваемых в массиве должна совпадать с физической последовательностью полей в таблице.

Листинг 18.3. Вставка новой строки с данными

```
If ADOTable1.CanModify=true then
   ADOTable1.InsertRecord(['Петров', 'Петр', 'Петрович', 8652757575]);
```

Замечание

С процессом вставки новой записи связаны три последовательно возникающие события: BeforeInsert(), OnNewRecord() и AfterInsert().

Для перевода текущей записи в режим редактирования предназначена процедура

```
procedure Edit;
```

После перевода набора данных в режим редактирования можно изменить содержимое всех полей, принадлежащих активной в данный момент записи. Для этого предназначена процедура

```
procedure SetFields(const Values: array of const);
```

ЗАМЕЧАНИЕ

С операцией редактирования набора данных взаимодействуют события `BeforeEdit()` и `AfterEdit()`.

Для быстрой очистки содержимого всех полей текущей записи набора данных предназначена процедура

```
procedure ClearFields;
```

Метод можно вызвать только при условии, что набор данных находится в режиме редактирования.

Чтобы сохранить внесенные изменения, вызывают метод

```
procedure Post;
```

Соответствующие процессу сохранения обработчики событий: `BeforePost()` и `AfterPost()`. Метод `Post()` следует вызывать только в случае, когда в результате редактирования в наборе данных возникли изменения. Узнать об этом поможет свойство

```
property Modified: Boolean;//только для чтения
```

которое возвратит `true`, если набор нуждается в сохранении. И наоборот значение `false` свидетельствует о том, что набор данных не изменялся.

ЗАМЕЧАНИЕ

Вы должны знать, что в `TDataSet` объявлен специальный внутренний метод `CheckBrowseMode()`, который перед сохранением изменений в активной записи набора проверит текущее состояние набора данных — свойство `State`. Если состояние набора `dsEdit` или `dsInsert` и свойство `Modifed` возвратит `true`, то метод сохранит изменения. А если, например, набор неактивен (`dsInactive`), то метод инициирует исключительную ситуацию. Вызов этой процедуры заложен в большинство методов, предназначенных для сохранения данных.

Для удаления текущей записи используйте метод

```
procedure Delete;
```

Перед удалением генерируется событие `BeforeDelete()`, после того как запись будет удалена, компонент вызовет обработчик события `AfterDelete()`.

Для отмены последних изменений в тексте применяйте метод `Cancel()`:

```
procedure Cancel;
```

Соответствующие обработчики событий: `BeforeCancel()` и `AfterCancel()`.

Учтите, что метод может быть полезен только до тех пор, пока изменения не будут сохранены. После этого откат становится невозможным.

Замечание

Методы `Append()`, `Insert()`, `Edit()`, `Post()`, `Delete()` и `Cancel()`, в первую очередь, предназначены для работы с компонентами-таблицами (`TIBTable`, `TADOTable`, `TClientDataSet`). Компоненты-запросы нацелены на работу с SQL-запросами, обычно возвращающими доступные только для чтения данные, поэтому у них вызов указанных методов не всегда возможен.

В классе `TDataSet` заложена виртуальная функция (позже переписанная у потомков), позволяющая выяснять состояние редактируемой активной записи:

```
function UpdateStatus: TUpdateStatus; virtual;
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

Здесь `usUnmodified` — запись не изменялась, `usModified` — запись изменялась, `usInserted` — идет процесс вставки записи и `usDeleted` — процесс удаления.

Организация доступа к отдельному полю

Для описания отдельного поля набора данных в VCL реализован специальный класс `TField`. Об особенностях `TField` мы очень подробно поговорим в следующей главе, а пока познакомимся со способами получения доступа к полям набора данных (экземплярам класса `TField`). В проекте БД различают два способа определения поля: динамический и статический. *Динамические поля* генерируются автоматически потомками `TDataSet` в момент открытия таблицы, связанной с компонентом. Тип данных, размерность, ограничения и другие свойства динамического поля настраиваются в соответствии с характеристиками обслуживаемого им столбца таблицы. Динамическое поле существует до тех пор, пока связанный с ним набор данных открыт. После закрытия набора все динамические поля автоматически уничтожаются. Процессом создания *статических полей* управляет программист, уже во время разработки приложения он получает возможность более тонкой настройки как внешнего вида, так и поведения этого объекта. Созданное статическим способом поле существует независимо от того, открыт или закрыт набор данных.

Для того чтобы выяснить, каким образом создавались поля (динамически или статически), нужно обратиться к свойству

```
property DefaultFields : Boolean; //Только для чтения
```

Если свойство возвращает `true`, то набор данных работает с динамически созданными полями.

Общее количество полей в наборе данных хранится в свойстве

```
property FieldCount: Integer;
```

Отдельному подсчету подлежат поля `BLOB` (Binary Large Object), специализирующиеся на хранении больших бинарных объектов:

```
property BlobFieldCount: Integer;
```

Полный перечень имен полей целевой таблицы возвращают методы

```
procedure GetFieldNames(List: TStrings); virtual; overload;
procedure GetFieldNames(List: TWideStrings); virtual; overload;
```

Единственное различие между двумя версиями перегружаемой процедуры заключается в формате текстовых строк: ANSI или UNICODE. В простейшем случае названия полей можно загрузить в компонент-список `ListBox1` (листинг 18.4).

Листинг 18.4. Получение списка полей набора данных

```
If ADOTable1.Active=true then
   ADOTable1.GetFieldNames(ListBox1.Items);
```

Для доступа к определенному полю, как правило, используют методы

```
function FieldByName(const FieldName: string): TField;
function FieldByNumber(FieldNo: Integer): TField;
```

В первом случае следует указать имя поля, а во втором — его индекс в наборе полей.

Общий список полей содержится в списке типа `TFields` и доступен в свойстве

```
property Fields: TFields;
```

Практически аналогичное свойство, содержащее описание полей в списке, `TFieldList`:

```
property FieldList: TFieldList;
```

Для поиска поля по его имени `FieldName` применяют функцию

```
function FindField(const FieldName: string): TField;
```

В случае успеха функция вернет искомое поле, при неудаче — неопределенный указатель `nil`. Данный метод часто применяется для проверки существования того или иного динамически создаваемого поля.

Иерархический набор параметров (а еще лучше сказать техническое описание) полей набора данных, который может потребоваться, например, в процессе создания таблицы:

```
property FieldDefs : TFieldDefs;
```

Свойство содержит всесторонние данные о структуре таблицы: состав полей и доменные ограничения.

Последовательный список параметров полей набора данных:

```
property FieldDefList: TFieldDefList;
```

Непосредственный доступ к содержимому поля обеспечит свойство

```
property FieldValues[const FieldName: string]: Variant;
```

В качестве параметра передается имя поля.

Замечание

При разработке приложений БД по возможности следует отдавать предпочтение статическим полям. Основное преимущество статических полей `TField` над динамическими в

том, что они хранят основные характеристики поля набора данных прямо в коде приложения, ограждая набор данных от проведения дополнительных операций по сбору метаданных о структуре физической таблицы. К тому же они более надежны, способны генерировать исключительные ситуации, определять ограничения на ввод данных и т. д. Однако у статических полей есть один весьма существенный недостаток — при малейшем изменении структуры целевой таблицы (имени, размера, типа данных столбца) статический объект приводит в негодность весь набор данных, т. к. он не "обучен" обновлять информацию. Другими словами, применяйте статические объекты только тогда, когда вы абсолютно уверены в том, что структура таблиц не будет изменяться.

Вычисляемые поля

Вычисляемое поле (calculated field) — это искусственно создаваемый экземпляр класса `TField`, который возвращает значения, полученные в результате арифметических, логических или других операций с реальными значениями столбцов отношения. Самое главное достоинство вычисляемых полей — экономия дискового пространства, ведь данные получаются в результате расчетов и не нуждаются в хранении в постоянной памяти.

Очень подробный пример создания вычисляемого поля будет предложен вам в следующей главе, поэтому, пока не вникая в подробности создания экземпляра вычисляемого поля, перейдем к методам и свойствам набора данных, имеющим отношение к вычисляемым полям.

Для включения режима автоматического пересчета вычисляемых полей необходимо установить в `true` свойство

```
property AutoCalcFields: Boolean;
```

Алгоритм пересчета описывается в обработчике события:

```
property OnCalcFields: TDataSetNotifyEvent;
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

Данное событие вызывается в четырех случаях: во время открытия набора данных, при переводе набора в режим редактирования, при удалении строки из набора, при переводе фокуса ввода от одного подключенного к `TDataSet` визуального элемента управления к другому (имеются в виду компоненты странички **Data Controls** палитры Delphi).

Агрегатное поле

Среди всего многообразия типов полей выделяются агрегатные поля типа `TADTField`. Особенность этого поля в том, что оно способно объединить в себе несколько скалярных полей. Для того чтобы набор данных был способен автоматически определять поля агрегатного типа, необходимо устанавливать в `true` свойство

```
property ObjectView: Boolean;
```

в противном случае набор данных проигнорирует составные поля.

Фильтрация набора данных

Фильтрация данных позволяет осуществлять отбор части записей по определенному критерию (критериям), описанному в свойстве

```
property Filter: string;
```

Правила описания фильтра достаточно просты и похожи на синтаксис, используемый в языке SQL в предложении `WHERE`. Внутри предложения фильтра возможны операторы, приведенные в табл. 18.2.

Таблица 18.2. *Операторы определения фильтра*

Оператор	Описание
<	Меньше
>	Больше
>=	Больше и равно
<=	Меньше и равно
=	Равно
<>	Не равно
AND	Проверка двух условий на истину
NOT	Проверка условия на ложь
OR	Проверка на выполнение одного из двух условий
+	Добавление чисел, объединение строк
-	Вычитание
*	Умножение двух чисел
/	Деление
*	Подстановочные знаки для частичных сравнений (свойство `FilterOptions` должно включать флаг `foPartialCompare`)

Для включения или отключения фильтра необходимо установить в `true` или `false` свойство

```
property Filtered : Boolean;
```

В листинге 18.5 приведен пример определения фильтра.

Листинг 18.5. Пример определения фильтра

```
With ADOTable1 do
begin
```

```
Filter:='Supplier='+ QuotedStr(Художественная литература);
Filtered:=true;
end;
```

Некоторые особенности в организацию фильтрации данных вносит свойство:

```
type   TFilterOption = (foCaseInsensitive, foNoPartialCompare);
       TFilterOptions = set of TFilterOption;
property FilterOptions: TFilterOptions;
```

Значение `foCaseInsensitive` определяет чувствительность фильтра к регистру символов. Значение `foNoPartialCompare` разрешает отфильтровывать частичные совпадения.

С процессом фильтрации связан специализированный обработчик события:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
                                    var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Используя событие фильтрации (листинг 18.6), программист сможет осуществить более тонкую настройку процесса отбора записей. Благодаря параметру `DataSet` мы получим ссылку на фильтруемый набор данных. При помощи переменной `Accept` осуществляется управление процессом отбора (фильтрации) записи (`true` — включать строку в результирующий набор, `false` — не включать).

Листинг 18.6. Использование события фильтрации

```
procedure TForm1.ADOTable1FilterRecord(DataSet: TDataSet;
                                       var Accept: Boolean);
begin
{настраиваем фильтр на отбор записей на основе значения в поле Amount,
 в результирующий набор будут включены записи >50 и меньше 100}
Accept:=(DataSet['Price']>'50') and (DataSet['Price']<'100');
end;
```

ЗАМЕЧАНИЕ

Обработчик события `OnFilterRecord()` будет работать только в случае, если свойство `Filtered` установлено в состояние `true`.

Организация поиска данных

На уровне класса `TDataSet` реализован наиболее универсальный способ поиска строки с данными. Универсальность заключается в том, что метод `Locate()` способен осуществлять поиск даже в неиндексированных полях:

```
function Locate(const KeyFields: String;
            const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

Здесь KeyFields — имя поля (или полей, разделенных точкой с запятой), в которых будет выполнен поиск. KeyValues — шаблон (или массив шаблонов, когда поиск идет в нескольких полях), по которому будет осуществлен поиск. Опции поиска Options описываются внутри квадратных скобок. Если опции не задействуются, то следует оставлять скобки пустыми:

```
type TLocateOption = (loCaseInsensitive, loPartialKey);
TLocateOptions = set of TLocateOption;
```

Значение loCaseInsensitive указывает на то, что при поиске не учитывается регистр символов, loPartialKey допускает поиск по части ключа.

Если поиск осуществляется в одном столбце таблицы, то за основу следует взять пример из листинга 18.7.

Листинг 18.7. Поиск данных методом Locate() в одном поле

```
if ADOTable1.Locate('SUPPLIER', 'Заря',[])=false
                                    then MessageBeep(MB_ICONEXCLAMATION);
```

Мы пытаемся найти поставщика "Заря" в столбце SUPPLIER. Если поиск завершился безрезультатно, функция MessageBeep() попросит операционную систему уведомить пользователя звуковым сигналом. Допускается задавать условие поиска одновременно по нескольким столбцам набора данных (листинг 18.8).

Листинг 18.8. Поиск данных методом Locate() в нескольких полях

```
with ADOTable1 do
  Locate('SNAME; FNAME',
         VarArrayOf(['Петров', 'Петр']), [loCaseInsensitive]);
```

Встретив строки кода из листинга 18.8, функция осуществит поиск сразу в двух полях таблицы, не обращая внимания на регистр символов. При обнаружении требуемых значений метод позиционирует курсор на этой записи и возвращает true.

Замечание

Хотя в арсенале компонентов наборов данных состоит достаточно богатый список методов поиска данных, разработчики Delphi настоятельно рекомендуют отдать предпочтение методу Locate. Функция старается применить наиболее оптимальный метод поиска, при возможности подключает необходимые для ускорения поиска индексы, а при их отсутствии справляется своими силами.

В классе TDataSet заложен виртуальный метод поиска Lookup(), который перекрыт у его потомков:

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
                const ResultFields: string): Variant; virtual;
```

По аналогии с методом Locate() функция Lookup() обладает тремя параметрами: строковый параметр KeyFields определяет столбец или список столбцов (раз-

деляемый точкой с запятой), в которых производится поиск, параметр `KeyValues` содержит шаблон или массив шаблонов, в соответствии с которыми будет производиться поиск. Последний параметр — `ResultFields` — перечисляет имена столбцов, которым будет присвоен результат функции. Функция не позиционирует курсор на найденной записи, а только возвращает полученные значения в вариантный массив.

Замечание

Если в наборе данных критериям поиска соответствует более одной записи, то методы `Locate()` и `Lookup()` возвратят первую найденную запись.

Обработка событий

В табл. 18.3 воедино сведен перечень обработчиков событий, объявленных на уровне класса `TDataSet`.

Таблица 18.3. *События набора данных* `TDataSet`

Событие	Событие вызывается
property BeforeOpen : TDataSetNotifyEvent;	Перед открытием набора данных
property AfterOpen : TDataSetNotifyEvent;	После открытия набора данных
property BeforeCancel : TDataSetNotifyEvent;	Перед операцией отмены
property AfterCancel : TDataSetNotifyEvent;	После операции отмены
property BeforeDelete : TDataSetNotifyEvent;	Перед удалением записи из набора
property AfterDelete : TDataSetNotifyEvent;	После удаления записи из набора
property BeforeEdit : TDataSetNotifyEvent;	Перед переводом набора в режим редактирования
property AfterEdit : TDataSetNotifyEvent;	По окончанию редактирования набора данных
property BeforeInsert : TDataSetNotifyEvent;	Перед вставкой записи в набор данных
property OnNewRecord : TDataSetNotifyEvent;	Момент вставки (добавления) новой записи
property AfterInsert : TDataSetNotifyEvent;	По завершению вставки записи
property BeforePost : TDataSetNotifyEvent;	Перед сохранением изменений в наборе
property AfterPost : TDataSetNotifyEvent;	После операции сохранения

Таблица 18.3 *(окончание)*

Событие	Событие вызывается
property BeforeRefresh: TDataSetNotifyEvent;	Перед обновлением записей в наборе данных
property AfterRefresh : TDataSetNotifyEvent;	По завершении обновления
property BeforeScroll : TDataSetNotifyEvent;	Перед перемещением к другой записи
property AfterScroll : TDataSetNotifyEvent;	После перемещения по записям
property BeforeClose : TDataSetNotifyEvent;	Перед закрытием набора данных
property AfterClose : TDataSetNotifyEvent;	После закрытия набора данных
property OnCalcFields : TDataSetNotifyEvent;	Процесс пересчета вычисляемого поля
property OnDeleteError: TDataSetErrorEvent;	Ошибка при удалении записи
property OnEditError : TDataSetErrorEvent;	Ошибка при редактировании
property OnPostError : TDataSetErrorEvent;	Ошибка сохранения данных

Большинство из приведенных в таблице событий представляют собой процедуру:

```
TDataSetNotifyEvent = procedure (DataSet:TDataSet) of object;
```

Единственный параметр процедуры позволяет разработчику узнать, какой именно набор данных выступил источником события.

Три последних строки таблицы посвящены обработке исключительных ситуаций, возникающих при осуществлении операций удаления, редактирования и сохранения изменений в наборе данных. Все эти события построены на процедуре:

```
TDataSetErrorEvent = procedure(DataSet: TDataSet;
                               E: EDatabaseError;
                               var Action: TDataAction) of object;
```

Здесь параметр `DataSet` указывает компонент, инициировавший событие; `E` — описывает исключительную ситуацию; формальный параметр `Action` определяет, каким образом набор данных отреагирует на ошибку. Для этого параметру `Action` присваивается одно из трех возможных значений: `daFail` — прекратить операцию и вывести информацию об ошибке, `daAbort` — прекратить операцию без информирования о характере ошибки или `daRetry` — повторить операцию.

Кэширование данных

Большинство потомков `TDataSet` для ускорения доступа к данным, снижения количества физических операций чтения/записи, снижения сетевого обмена и т. п., некоторую часть данных кэшируют в оперативной памяти компьютера. Доступ к

записям, размещенным во внутреннем буфере набора данных, осуществляется при помощи свойства

```
property Buffers[Index: Integer]: PChar;
```

Для этого требуется передать индекс записи в буфере.

ВНИМАНИЕ!

Если набор данных построен на основе однонаправленного курсора, то он не имеет возможности кэшировать записи.

Чтобы узнать объем памяти, отводимой для обслуживания одной строки набора данных, следует познакомиться со свойством

```
property RecordSize: Word;
```

Общее количество записей в буфере хранится в свойстве

```
property BufferCount: Integer;
```

Содержимое активной записи можно получить сразу, при помощи метода

```
function ActiveBuffer: PChar;
```

Индекс текущей записи в буфере находится в свойстве

```
property CurrentRecord: Integer;
```

Индекс активной записи в буфере

```
property ActiveRecord: Integer;
```

Взаимодействие с элементами управления

Как правило, потомки класса `TDataSet` (компоненты `TADOTable`, `TIBTable` и т. д.) еще в период разработки приложения связываются с элементами управления, отвечающими за отображение и редактирование данных. Посредником в этих операциях выступает источник данных `TDataSource`. Для подключения к набору данных источника данных предназначено одноименное свойство:

```
property DataSource: TDataSource;
```

Для проверки факта подключения к набору данных того или иного источника данных вызывают метод

```
function IsLinkedTo(DataSource: TDataSource): Boolean;
```

Если источник данных, указанный в параметре `DataSource`, действительно связан с нашим набором данных, то функция вернет утвердительный ответ `true`.

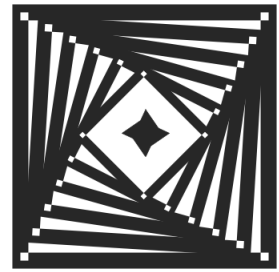
При помощи источника данных `TDataSource` с набором данных связываются внешние элементы управления. В первую очередь, к ним относятся элементы отображения данных — вся страничка `DataControls` палитры компонентов Delphi. Для того чтобы выяснить, активны ли внешние элементы управления, применяют функцию

```
function ControlsDisabled: Boolean;
```

Напомню о существовании методов `DisableControls()` и `EnableControls()`. Первый из них отключает от набора данных, а второй подключает к набору внешние элементы управления. Применение этих методов позволяет значительно ускорить операции, связанные с перебором объемного списка записей, т. к. освобождает систему от обязанности многократной перерисовки информации в компонентах отображения данных.

Резюме

Универсальный набор данных `TDataSet` определяет концепцию построения всех многочисленных дочерних классов, независимо от исповедуемой ими технологии доступа к данным. Объявленные в классе методы и свойства позволяют открывать и закрывать набор данных, перемещаться по записям набора, редактировать данные, обеспечивать доступ к отдельному полю, работать с вычисляемыми полями, фильтровать записи, обеспечивать поиск строк и определять состояние набора данных.



Глава 19

Общая характеристика полей набора данных

При формировании библиотеки VCL программисты Embarcadero Technologies реализовали целый арсенал компонентов, предназначенных для организации доступа к данным. В Delphi XE число этих элементов управления настолько велико, что даже простое их перечисление способно внушить благоговение любому программисту, тем более, начинающему. Но, несмотря на впечатляющий перечень классов, компоненты достаточно просты в изучении, т. к. построены по схожим принципам. А различия между элементами управления, обслуживающими базы данных, в первую очередь, определяются способом доступа к данным. Поэтому даже в названиях компонентов четко прослеживается принадлежность к технологиям dbExpress, dbGO или ADO (ActiveX Data Objects), IB (Interbase), BDE (Borland Database Engine). Если вы предпочитаете работать на основе технологии dbExpress, то в проекте найдут место компоненты `TSQLTable`, `TSQLQuery`, `TSQLStoredProc`. Если вы приверженец dbGO, то понадобятся `TADOTable`, `TADOQuery`, `TADOStoredProc`. При работе с сервером Interbase или FireBird активного применения достойны `TIBTable`, `TIBQuery`, `TIBStoredProc`. Технологию BDE обеспечивают компоненты `TTable`, `TQuery`, `TStoredProc`. Напомню, что на сегодня BDE уже существенно устарела, и поддерживается только для обеспечения обратной совместимости с ранними проектами Delphi, поэтому в новых разработках без особой необходимости BDE не применяется.

Все компоненты доступа к данным обладают общим предком — подробно рассмотренным в предыдущей главе классом `TDataSet`. Напомню, что в этом классе реализованы базовые свойства и методы доступа к данным. А в его потомках они лишь совершенствовались. Материал, излагаемый в настоящей главе, также является общим для всех компонентов доступа к данным. Он посвящен особому классу `TField` — полю набора данных.

Поле таблицы — класс *TField*

Класс `TField` (от англ. field — поле) инкапсулирует физический объект — поле набора данных. Сам по себе класс `TField` абстрактный и в первозданном виде не

используется. А вот богатое "потомство" (рис. 19.1) интегрируется во все компоненты, построенные на основе класса `TDataSet`. В предыдущей главе, посвященной универсальному набору данных `TDataSet`, подробно были рассмотрены способы доступа к отдельному полю набора, так что сейчас я только напомню о существовании у класса `TDataSet` свойства `Fields`, содержащего перечень полей набора данных, и ряда методов, предоставляющих доступ к отдельному полю в наборе по его имени или индексу. В 99 случаях из 100 за создание объекта-поля также отвечает `TDataSet`. При открытии набора данных автоматически рождается ровно столько объектов `TField`, сколько столбцов физически доступно в этом наборе. Созданные таким образом поля (без явного вмешательства программиста) называют *динамическими*. Основная особенность таких полей в том, что они уничтожаются сразу после отключения набора данных от физической таблицы.

Однако в некоторых ситуациях и на долю программиста выпадает счастливая обязанность немного поработать. Засучить рукава приходится для реализации:

- полей подстановки (lookup);
- вычисляемых (calculated) полей;
- агрегатных (aggregate) полей.

Для создания полей такого рода в среде программирования Delphi реализован специальный редактор **Fields Editor**, вызываемый двойным щелчком кнопкой мыши по компоненту-набору данных. Все объекты-поля, созданные руками программиста, именуют *статическими*. В отличие от динамического поля, жизненный цикл его статического собрата начинается практически в момент старта приложения, продолжается до завершения его работы и не зависит от того, открыт или закрыт набор данных. Более того, программист получает возможность обращаться к свойствам поля при помощи обычного инспектора объектов Delphi уже во время разработки приложения, что вносит некоторые дополнительные удобства. Наверное, стоит оговориться, что когда набор данных закрыт, бессмысленно обращаться к значению поля.

Внимание!

Один и тот же набор данных не в состоянии одновременно обладать как динамическими, так и статическими полями. И если в наборе определено хотя бы одно статическое поле, то компонент утратит возможность самостоятельно создавать его динамических коллег. А для того, чтобы компонент "припомнил", как это делается, требуется удалить все созданные вручную статические поля.

Применение статических полей не ограничивается случаями вычисляемых, агрегатных и подстановочных полей. В ситуации, когда программист однозначно уверен, что структура таблицы продумана до тонкостей и в обозримом будущем не поменяется, рекомендуется создавать статические поля для обращения и к обычным столбцам данных таблицы.

Кроме разделения на динамические и статические, поля классифицируют по типу хранящихся в них данных. Даже, пожалуй, правильнее сказать, что тип данных оказывает решающее влияние на появление целой плеяды потомков базового класса `TField` (рис. 19.1). Специализация поля определяется физическим типом

данных, хранящихся в наборе. Если мы подключились к столбцу таблицы, содержащему целые числа, то соответствующий ему объект-поле будет построен на базе класса `TIntegerField`, столбец вещественных чисел — `TFloatField`, дата — `TDateField` или `TDateTimeField` и т. д.

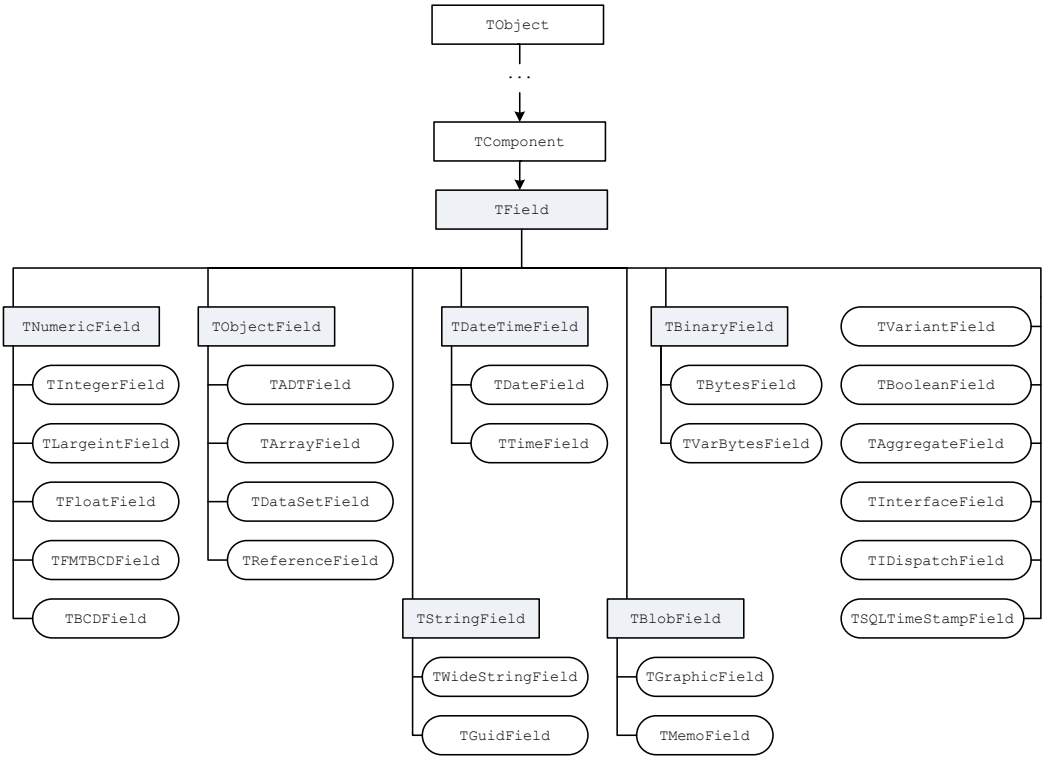


Рис. 19.1. Иерархия наследников класса `TField`

Теперь перечислим основные обязанности класса `TField`:

- чтение данных из текущей записи выбранного столбца таблицы;
- редактирование значений поля текущей записи с одновременной проверкой допустимости вводимых данных;
- указание компонентам отображения данных, как следует представлять содержимое поля на экране;
- обработка вычисляемых полей;
- применение агрегирующих функций;
- работа с подстановочными полями.

Замечание

Особенности работы с агрегатными полями будут рассмотрены в *главе 21*, посвященной изучению клиентского набора данных компонента `TClientDataSet`.

Классификация полей по функциональному назначению

Для того чтобы выяснить назначение поля, необходимо сразу обратиться к его свойству:

```
property FieldKind: TFieldKind;
```

В подавляющем числе случаев обращение производится только для чтения, и редактировать хранящееся в нем значение не только излишне, но даже опасно. Список возможных вариантов значений невелик (табл. 19.1).

Таблица 19.1. *Значения* `TFieldKind`

Значение	Описание
`fkData`	Объект представляет собой поле данных
`fkCalculated`	Это вычисляемое поле, содержимое которого определяется в обработчике события `OnCalcFields()` содержащего его компонента, например `TADOTable`
`fkLookup`	Поле подстановки
`fkInternalCalc`	Внутреннее вычисляемое поле
`fkAggregate`	Агрегатное поле

Самый распространенный тип поля — `fkData`. На физическом уровне это чаще всего поле обычной таблицы, доступ к которому мы получили с помощью компонентов `TIBTable`, `TADOQuery` или их коллег.

Некоторые свойства и методы класса `TField` дублируют возможности свойства `FieldKind`. Например, свойство

```
property Lookup: Boolean;
```

говорит, является ли поле полем синхронного просмотра. Свойство

```
property Calculated: Boolean;
```

возвратит `true`, в случае, когда оно считает себя вычисляемым.

Классификация полей по типу хранимых данных

Ключевой характеристикой поля служит тип данных, который может в нем содержаться. Чтобы выяснить это, прочитайте значение свойства

```
property DataType: TFieldType;
```

В табл. 19.2 представлен список возможных типов данных.

Таблица 19.2. *Типы данных* `TFieldType`

Значение	Описание	Значение	Описание
Целые числа			
`ftSmallint`	16-разрядное целое	`ftInteger`	32-разрядное целое
`ftWord`	16-разрядное беззнаковое целое	`ftLargeint`	Большое целое
`ftAutoInc`	Автоинкрементное 32-разрядное целое		
Вещественные числа			
`ftFloat`	Вещественное число	`ftCurrency`	Денежный тип
Дата/время			
`ftDate`	Дата	`ftTime`	Время
`ftDateTime`	Дата/время	`ftTimeStamp`	Метка дата/время
Символьные			
`ftString`	Символьный тип или строка	`ftWideString`	Строка Unicode
`ftFixedChar`	Фиксированное количество символов	`ftGuid`	Глобальный уникальный идентификатор (GUID)
Универсальные бинарные поля			
`ftBytes`	Битовый массив	`ftVarBytes`	Битовый массив переменной длины
Поля бинарных больших объектов переменной длины (BLOB) и их производные			
`ftBlob`	Большой двоичный объект — BLOB	`ftMemo`	Текстовое многострочное поле
`ftOraBlob`	BLOB-поле СУБД Oracle	`ftFmtMemo`	Текстовое многострочное поле с поддержкой форматирования
`ftGraphic`	Графическое поле для растровой графики	`ftOraClob`	CLOB-поле СУБД Oracle
`ftDBaseOle`	OLE-поле формата dBase	`ftParadoxOle`	OLE-поле формата Paradox
`ftTypedBinary`	Типизированное двоичное поле	–	–

Таблица 19.2 *(окончание)*

Значение	Описание	Значение	Описание
Композитные поля			
ftADT	Абстрактный тип данных	ftArray	Массив
ftReference	Ссылочное поле	ftDataSet	Поле набора данных
Прочие типы данных			
ftUnknown	Неизвестный тип данных	ftVariant	Неопределенный тип данных
ftBoolean	Логическое поле	ftInterface	Ссылка на интерфейс IUnknown
ftCursor	Курсор, применяемый в хранимых процедурах СУБД Oracle	ftIDispatch	Ссылка на интерфейс IDispatch
ftBCD	Двоично-кодированный тип, применяется для хранения вещественных чисел с высокой точностью	ftFMTBcd	Расширенный вариант двоично-кодированного (BCD) типа данных. Может применяться для хранения действительных чисел с высокой точностью

Обращение к полю

Своим появлением на свет экземпляр поля целиком и полностью обязан компоненту-набору данных. Для того чтобы выяснить, какому набору данных принадлежит наше поле, следует обратиться к свойству:

```
property DataSet: TDataSet;
```

Каждое поле владеет информацией об имени связанного с ним столбца таблицы:

```
property FieldName : string;
```

Напомню, что в наборе данных `TDataSet` поля хранятся в списке `Fields`. В этом списке каждое поле описывается уникальным индексом:

```
property Index: Integer;
```

Изменяя индекс, мы получаем возможность корректировать последовательность полей в наборе. А вот порядковый номер поля в наборе данных редактировать нельзя. Это свойство доступно только для чтения:

```
property FieldNo: Integer;//только для чтения
```

Отмечу, что в классе `TDataSet` предусмотрено чисто техническое свойство. Оно по задумке создателей Delphi предназначено для хранения реального имени поля в таблицы:

```
property Origin: String;
```

Что мы имеем в виду, говоря о "реальном" имени? Дело в том, что имя столбца может быть изменено внутри запроса. Синтаксис запроса позволяет переименовывать имена в результирующем наборе данных. Например, запрос

```
SELECT VendorName AS Vendor FROM Vendors
```

в результирующем отношении переименовывает столбец `VendorName` в `Vendor`. А теперь поясню, почему я говорил о "техническом" свойстве. Особенность `Origin` в том, что свойство может использоваться только на стадии разработки проекта, и только в том случае, если поле принадлежит компоненту, построенному на базе класса `TADOQuery`. Тогда `Origin` доступно в редакторе полей.

Доступ к данным поля

При обращении к ячейкам таблицы начинающему программисту необходимо, прежде всего, усвоить, что реляционная модель данных допускает хранение неопределенных значений `NULL`. Не путайте `NULL` ни с нулем (нулевым значением), ни с символом пробела, ни с чем-нибудь другим. `NULL` — это вообще НИЧЕГО — косвенный признак того, что программист, создавший базу данных, допускает, что данный столбец не обязателен для заполнения. Для того чтобы проверить, есть ли в поле какое-то содержимое, всегда обращайтесь к свойству

```
property IsNull: Boolean;
```

Если поле пусто, то свойство вернет `true`, если же там что-то окажется, возвращаемое значение будет `false`. Если поле не пусто, то вы имеете полное право выяснить, что там находится. Для чтения и записи данных в поле предназначено свойство

```
property Value: Variant;
```

Возвращаемый тип данных описывается как универсальный, вместе с тем имеется возможность конвертировать возвращаемые данные к допустимому типу, например, строковому. В листинге 19.1 продемонстрирован способ чтения данных из поля.

Листинг 19.1. Проверка поля на наличие данных перед началом чтения

```
If ADOTable1.Fields[0].IsNull=False Then
                Result:= Table1.Fields[0].Value;
```

Для полной очистки содержимого поля следует применять процедуру

```
procedure Clear;
```

После вызова метода значение поля установится в неопределенность `NULL`.

Для копирования содержимого одного поля в другое предназначена процедура

```
procedure Assign(Source: TPersistent);
```

Обязательное условие — совместимость типов полей.

Еще один способ передачи данных в поле обеспечивает процедура

```
procedure AssignValue(const Value: TVarRec);
```

Поле можно перевести в режим "только для чтения", применив свойство

```
property ReadOnly: Boolean;
```

Свойство `ReadOnly` позволяет программным образом запрещать или разрешать редактировать данные. Но сначала нужно проверить, доступно ли для правки поле на физическом уровне, ведь оно, например, может оказаться вычисляемым, т. е. не редактируемым в принципе. Для этого требуется проверить состояние свойства:

```
property CanModify: Boolean;//только для чтения
```

Свойство информирует нас о том, разрешено (`true`) или запрещено (`false`) модифицировать данные поля.

При вводе новой или редактировании существующей записи важную роль играет свойство

```
property Required: Boolean;
```

Значение `true` свидетельствует о том, что это обязательное поле. Другими словами, оно должно быть заполнено, иначе (если поле окажется пустым) мы получим сообщение об ошибке.

Если поле `Field` принадлежит компоненту, который научен не просто сразу передавать все исправления в файл таблицы, а предварительно кэшировать результаты в памяти (компонент обладает свойством `CachedUpdates`) и сохранять все изменения позднее, хорошую службу могут сослужить свойства поля:

```
property OldValue: Variant;//значение поля перед началом редактирования
property NewValue: Variant;//текущее значение поля (после изменений)
```

Допустим, что пользователь редактирует содержимое поля и меняет какое-то старое значение "A" на новое "B". Пока изменения не были сохранены в БД и находятся в локальном буфере, в `NewValue` будет храниться новое значение "B", в свойстве `OldValue` — старое "A". Другими словами, у программиста пока существует возможность отката к предыдущему значению. Кроме того, подобное поведение возможно, если доступ к набору данных реализован с помощью класса `TClientDataSet`. Если база данных открыта в многопользовательском режиме, и имеется вероятность, что к одной и той же записи одновременно имеют доступ несколько операторов, то целесообразно проверять текущее значение поля в свойстве

```
property CurValue: Variant;
```

Дело в том, что в многопользовательском режиме значение `OldValue` может отличаться от `CurValue`. Такая ситуация возникает, когда первый пользователь открыл запись для просмотра, а вслед за этим другой пользователь внес в нее изменения и сохранил. В этом случае первый пользователь "отстал от жизни", и в свойстве `OldValue` поля окажется устаревшее значение, т. к. в кэше компонента содержатся данные, полученные до изменений.

Внимание!

При разработке приложения, нацеленного на эксплуатацию в многопользовательском режиме работы, будьте готовы к тому, что два разных пользователя одновременно попытаются внести исправления в одну и ту же запись. Также высока вероятность редактирования записи, находящейся в просмотре у другого пользователя. Ошибки такого рода называют *конфликтом доступа*. При появлении подобных коллизий набор данных (на уровне класса `TCustomClientDataSet`) генерирует ошибку `OnReconcileError`.

На запись значения поля при удаленном доступе влияет свойство

```
property ProviderFlags: TProviderFlags;
```

Оно определяет, как задействовать это поле при отправке данных в БД для обновления, здесь

```
type TProviderFlag = (pfInUpdate, pfInWhere, pfInKey, pfHidden);
TProviderFlags = set of TProviderFlag;
```

Возвращаемое свойством значение может включать комбинацию флагов. Флаг `pfInUpdate` означает, что поле может обновляться; `pfInWhere` — поле используется при отборе данных (предложение `WHERE` в SQL запросе); `pfInKey` — поле применяется для поиска записей по ключу при сбое обновления; `pfHidden` — скрытое поле.

Некоторые поля способны самостоятельно позаботиться о заполнении самих себя данными. Точнее говоря, значения в поле может вносить непосредственно БД. О таком поведении БД нам поведает свойство

```
property AutoGenerateValue: TAutoRefreshFlag;
type TAutoRefreshFlag = (arNone, arAutoInc, arDefault);
```

Для поля автоинкрементного типа свойство возвратит значение `arAutoInc`, если таблица сконфигурирована так, что в поле может подставляться значение по умолчанию, то вы получите `arDefault`, и, наконец, значение `arNone` сигнализирует, что перед нами самое обычное поле и оно, скорее всего, ожидает непосредственного ввода данных пользователем. Свойство `AutoGenerateValue` является только информационным, и нет никакого смысла изменять его значение вручную, поскольку это не повлияет на логику СУБД.

Низкоуровневый доступ к данным

Программистам, разрабатывающим свои собственные потомки `TField`, необходимо знать о существовании низкоуровневого способа доступа к содержимому поля. Для чтения данных применяется функция `GetData()`, а для записи — процедура `SetData()`:

```
function GetData(Buffer: Pointer; NativeFormat : Boolean=True): Boolean;
procedure SetData(Buffer: Pointer, NativeFormat: Boolean=True);
```

В обоих случаях данные читаются в область памяти или записываются из области памяти `Buffer`, заранее распределенной программистом, например, при помощи метода `GetMem()`. Объем буфера должен точно соответствовать размеру обрабатываемых данных. Так как в буфере данные хранятся в неформатированном виде, а просто как последовательность байтов, вопрос представления данных полностью ложится на наши "хрупкие" плечи. Впрочем, некоторую помощь в этом случае окажет флаг `NativeFormat`. Когда он установлен в `true` (а это значение по умолчанию), данные автоматически транслируются в тип данных, связанных с этим полем, иначе, повторюсь, согласование типов — наша обязанность. Если при чтении данных методом `GetData()` выясняется, что поле пусто (`NULL`), то функция возвратит `false`, иначе `true`.

ВНИМАНИЕ!

Методы `GetData()` и `SetData()` нельзя применять при работе с BLOB-полями. Для реализации операций записи/чтения в этом случае следует применять методы `SaveToStream()` и `LoadFromStream()` дочернего класса `TBLOBField`.

Преобразование типа данных

Один и тот же объект-поле способен представлять содержащееся в нем значение в виде некоторых стандартных типов данных. Для этого в классе `TField` инкапсулировано несколько свойств (табл. 19.3).

Таблица 19.3. *Приведение типов данных в поле* `TField`

Свойство	Описание
property AsBCD: TBcd;	Содержит значение поля в формате двоично-кодированного десятичного числа. Значения такого типа иногда применяются вместо денежных (currency) типов данных
property AsBoolean: Boolean;	Содержит значение в логическом формате
property AsCurrency: Currency;	Денежный формат
property AsDateTime: TDateTime;	Формат дата/время
property AsSQLTimeStamp: TSQLTimeStamp;	Штамп времени SQL
property AsFloat: Double;	Вещественное число
property AsExtended: Extended;	Вещественное число с повышенной точностью
property AsInteger: Integer;	Целое число
property AsSQLTimeStamp: TSQLTimeStamp;	Представление даты/времени в формате TSQLTimeStamp
property AsString: **String**;	Строка символов
property AsAnsiString: AnsiString;	Строка символов в кодировке ANSI
property AsWideString: UnicodeString;	Строка символов в кодировке Unicode
property AsVariant: Variant;	Универсальный формат

Ни один класс-потомок `TField` не может похвастаться полным перечнем свойств `"As…"`. Такое ограничение объясняется элементарным здравым смыслом, ведь согласитесь, что очень сложно представить в виде даты/времени (`AsDateTime`) содержимое поля неформатированного текста `TMemoField` или преобразовать в денежный формат (`AsCurrency`) картинку из поля `TGraphicField`.

Вместе с тем, существуют и некоторые допущения. Например, содержимое строкового поля можно представить в логическом виде, но при маленьком условии — свойство `AsBoolean` будет возвращать значение `true` только, если первый символ строки, хранящейся в этом поле, начинается с символов "Y", "y", "T", "t". Самые прозорливые уже догадались, что с этих символов начинаются слова "Yes" и "True". Во всех остальных случаях свойство `AsBoolean` возвратит `false`.

Второе допущение касается обратного преобразования — представления значения логического поля `TBooleanField` в виде строки символов. Если в поле находится истинное значение, то свойство `AsString` возвращает строку "True", иначе — "False".

Третье допущение касается всех типов полей, предназначенных для работы с вещественными числами. Если вы обращаетесь к свойству `AsInteger`, то будет возвращено значение, округленное к ближайшему целому числу.

И, наконец, последнее допущение разрешает представлять значение, содержащееся в полях даты-времени, в виде действительного числа при помощи свойства `AsFloat`. Здесь никаких проблем возникнуть не должно. Дата займет целую часть числа, дробная часть отводится времени. Нулевому значению соответствует дата/время: 30.12.1899 г. 00 ч 00 м 00 с, т. е. все, как у типа данных `TDateTime`.

Теперь, что касается применения этих свойств. Следующая строка кода демонстрирует способ передачи данных из поля в компонент-метку:

```
Lable1.Caption:=ADOQuery1.FieldByName('SUPPLIER').AsString;
```

Или обратный пример, показывающий, как можно забрать значение из компонента-календаря:

```
ADOTable1.Fields[1].AsDateTime:= DateTimePicker1.Date;
```

Размер поля

При обращении к значению поля иногда требуется знать физический размер, занимаемый этим значением. Для таких целей предназначено свойство

```
property DataSize: Integer;
```

Свойство работает у всех потомков `TField` (за исключением экземпляров BLOB-полей) и возвращает размер в байтах. Достаточно часто это свойство применяют для того, чтобы зарезервировать размер буфера для хранения содержимого поля.

Еще одно свойство

```
property Size: Integer;
```

для поля строкового типа указывает максимальное число символов, допустимых для ввода в это поле. А для остальных типов полей — размерность в единицах, зависящих от типа данных поля (табл. 19.4).

Таблица 19.4. *Взаимосвязь размера поля с типом обслуживаемых данных*

Тип данных	Описание
ftBCD	Число знаков после запятой
ftString	Максимальное число символов в строке
ftVarBytes	Максимальное число байтов в поле, не включая первые два байта, в которых хранится количество байтов, реально занятых данными
ftBytes	Максимальное число байтов в наборе
ftBlob	Число байтов бинарного большого объекта (Binary Large Object, BLOB), хранящегося в буфере записи
ftDBaseOle	Число байтов поля OLE BLOB таблицы dBASE, хранящегося в буфере записи
ftFMTBCD	Число знаков после запятой
ftFmtMemo	Число байтов форматированного MEMO-поля, хранящегося в буфере записи
ftGraphic	Число байтов графического поля, хранящегося в буфере записи
ftMemo	Число байтов MEMO-поля, хранящегося в буфере записи
ftParadoxOle	Число байтов поля OLE BLOB таблицы Paradox, хранящегося в буфере записи
ftTypedBinary	Число байтов типизированного BLOB-поля, хранящегося в буфере записи
ftADT	Общее число полей, входящих в состав абстрактного поля
ftArray	Число элементов в массиве

Значение по умолчанию

Полезное во всех отношениях свойство `DefaultExpression` определяет значение по умолчанию, вносимое в поле при вводе новой записи:

```
property DefaultExpression: String;
```

Как правило, в свойстве указываются наиболее вероятные данные для поля, что освобождает пользователя от многократного ввода одного и того же значения:

```
Field1.DefaultExpression:='12:00:00';
```

Для наибольшей универсальности свойство сделано строковым. Такое решение позволяет задавать значения по умолчанию для большинства типов полей, однако при этом стоит проявлять осторожность. Допустим, что вы работаете с полем вещественного типа `ftFloat` и намереваетесь определить для него значение по умолчанию "три целых и четырнадцать сотых" (или что-то в этом роде). Вы смело задаете ограничение "3.14" и… при вводе новой записи приложение начинает

"возмущаться". После непродолжительных разбирательств, вы выясняете, что в настройках операционной системы за разделитель целой и дробной части числа принята не точка, а запятая. Вы исправляете ошибку — назначаете свойству `DefaultExpression` значение "3,14", и все, кажется, работает хорошо… И будет работать до тех пор, пока какой-нибудь любопытный пользователь не захочет перенастроить региональные параметры Windows и поменяет разделитель на какой-то другой символ. После такой процедуры приложение вновь утратит работоспособность, а его пользователь расскажет своим друзьям о бестолковом программном обеспечении и его создателе. Поэтому предусмотрительный разработчик, проанализировав вероятность возникновения подобной ситуации, вспомнит, что в модуле `SysUtils` объявлен список глобальных переменных, обеспечивающих взаимодействие с операционной системой. И в частности, за символ-разделитель несет ответственность переменная

```
var DecimalSeparator: Char;
```

После этого в разделе инициализации проекта остается назначить значение по умолчанию:

```
Field1.DefaultExpression:= '3' + DecimalSeparator+ '14';
```

Список вероятных ошибок можно и продолжить. Например, при вводе даты не забывайте про разницу между российским и американским форматом представления даты. Одним словом, будьте внимательны.

Ограничения на ввод данных

Объекты поля способны осуществлять строгий контроль над корректностью вводимых данных. Набор свойств и методов поля позволяет импортировать ограничения, заложенные на уровне SQL-сервера или определенные в локальных таблицах, и определять ограничения на уровне приложения, описывая их в коде программы. Благодаря этому строится гибкая и многоуровневая система защиты от ввода недопустимых данных.

Ограничения на диапазон допустимых значений можно определить в свойстве

```
property CustomConstraint: String;
```

Это не что иное, как обычная строка, в которой описывается выражение на языке SQL. Допустим, что физическое имя поля `field1`, тогда выражение, определяющее допустимые пределы вводимого значения, должно соответствовать листингу 19.2.

При попытке пользователя ввести недопустимое значение, набор данных генерирует исключительную ситуацию. Для того чтобы сообщение об ошибке было понятно для пользователя, целесообразно описать вероятную причину ошибки в свойстве:

```
property ConstraintErrorMessage: String;
```

Порядок использования свойства иллюстрирует листинг 19.2.

Листинг 19.2. Ограничение диапазона допустимых значений для поля

```
Field1.CustomConstraint:='field1>=10 and field1<100';
Field1.ConstraintErrorMessage:= "Допустимый диапазон значений 10—99";
```

Если ограничения описаны на уровне СУБД, то вы сможете с ними ознакомиться, посмотрев содержимое свойства:

```
property ImportedConstraint: String;
```

Попытка внести изменения в импортируемые ограничения обречена на провал, т. к. эти правила определены на более низком уровне.

Еще одно справочное свойство предназначено для проверки, есть ли какие-нибудь ограничения на вводимые значения:

```
property HasConstraints: Boolean; //только для чтения
```

Маска ввода

Маска ввода предназначена для определения формата вводимых пользователем значений и для повышения наглядности представляемых данных:

```
property EditMask: TEditMask;
```

Еще один способ доступа к маске ввода обеспечивает свойство:

```
property EditMaskPtr: TEditMask; //только для чтения
```

Помощь в определении шаблона ввода окажет встроенный редактор (рис. 19.2).

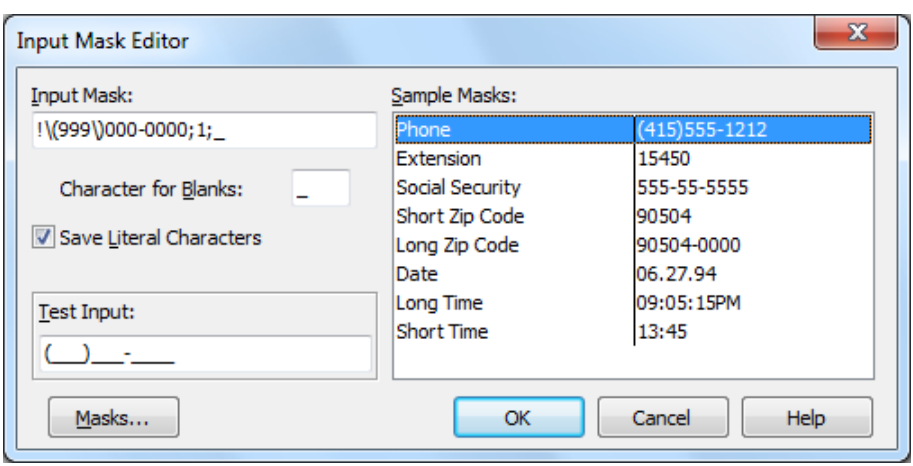


Рис. 19.2. Редактор маски ввода

Структурно маска состоит из трех полей, разделенных точкой с запятой. Обязательное для заполнения только первое из них — в нем записывается непосредственно маска ввода. Допустимые символы маски приведены в табл. 19.5. Особый интерес представляет символ наклонной черты \. Его назначение — оповещение

Delphi, что следующий за ним символ является литералом (символом оформления, отображаемым в строке ввода). В примере наклонная черта предшествует круглым скобкам, в которые заключается телефонный код города, т. е. круглые скобки и есть литералы. Второе поле маски способно принимать два значения — "0" или "1". В первом случае в передаваемый в поле текст символы-литералы не включаются (что крайне полезно для БД, т. к. снижает размерность поля данных), при втором варианте в обработанный текст войдут все символы.

Таблица 19.5. *Спецификаторы маски*

Символ	Описание	Символ	Описание
!	Подавление пробелов в тексте	>	Перевод в верхний регистр
<	Перевод в нижний регистр	<>	Отмена перевода регистров
\	Следующий символ — литерал	_	Пустое поле
L	Обязательна буква	l	Может быть буква
A	Обязательна буква или цифра	a	Может быть буква или цифра
C	Обязателен любой символ	c	Может быть любой символ
0	Обязательна цифра	9	Может быть цифра
#	Может быть цифра, знак "+" или "–"	:	Разделитель часов, минут и секунд
/	Разделитель дней, месяцев и годов	;	Разделяет поля в маске

Индексные поля

На основе поля или группы полей в таблице может быть построен индекс. Если поле входит в состав установленного в настоящий момент индекса набора данных, то свойство

```
property IsIndexField: Boolean;
```

возвращает значение `true`.

Отображение данных

Вполне естественно, что ни класс `TField`, ни его многочисленные потомки не имеют своего собственного инструментария, позволяющего отображать данные на экране компьютера. Указанная задача решается с помощью других визуальных элементов управления. Вместе с тем, класс `TField` способен существенно повлиять на особенности визуализации данных, о которых нам следует знать. В частности, отвечающие за отображение информации БД компоненты линейки **Data Controls**

палитры компонентов вполне способны узнать у класса TField, каким образом он хотел бы представлять свои данные на дисплее.

Разговор об отображении данных начнем с того, что некоторые поля вообще не стоит выводить на экран. Пользователю нет необходимости знать содержимое служебных столбцов таблиц, например с первичными и внешними ключами. А для демонстрации содержимого полей с неформатированными данными (например, BLOB) вообще требуется создать специальные условия. В связи с этим нашим первым помощником станет свойство

```
property Visible: Boolean;
```

позволяющее вообще отказаться от отображения столбца с данными. Допустим, что среди полей таблицы есть BLOB-поля, содержащие информацию, которую невозможно отобразить на экране. С отключением видимости статического поля никаких проблем нет, мы находим требуемый экземпляр поля в инспекторе объектов и устанавливаем его свойство Visible в состояние false. Но если поля создаются динамически в момент подключения набора данных к физической таблице с заранее неизвестной структурой, то потребуется немного изобретательности. В подобных случаях следует воспользоваться событием AfterOpen(), возникающим сразу после открытия набора данных (листинг 19.3).

Листинг 19.3. Настройка видимости полей набора данных

```
procedure TForm1.ADOTable1AfterOpen(DataSet: TDataSet);
var i:integer;
begin
for i:=0 to DataSet.FieldCount - 1 do
   DataSet.Fields[i].Visible:=DataSet.Fields[i].DataType<>ftBLOB;
end;
```

В рамках события производится перебор всех полей набора данных DataSet, если поле содержит данные BLOB, то мы скрываем экземпляр поля.

Данные, предназначенные для вывода на экран, хранятся в текстовом формате в свойстве

```
property DisplayText: String; //только для чтения
```

Все специализирующиеся на работе с БД элементы управления (DBText, DBEdit, DBGrid и т. д.) при отображении данных обратятся именно к этому свойству поля.

Если поле находится в режиме редактирования, то все изменения (до момента их сохранения) аккумулируются в свойстве

```
property Text: String;
```

В период редактирования содержимое свойств DisplayText и Text различается.

Замечание

При любой попытке прочитать хранящиеся в свойствах DisplayText и Text данные, возникает событие OnGetText().

При назначении имен заголовков колонок сетка TDBGrid обязательно проверит содержимое свойства

```
property FieldName: string;
```

объекта TField. В этом свойстве по умолчанию хранится физическое имя поля, сетка TDBGrid "подхватит" это имя и с его помощью определит заголовок колонки.

Если физическое имя поля не подходит для вывода в качестве заголовка сетки, то обращаемся к свойству

```
property DisplayLabel : String;
```

По умолчанию здесь находится физическое имя поля, однако программист имеет право его исправить. И тогда имена колонок в сетке TDBGrid сразу получат осмысленные названия.

Еще одно свойство, связанное с отображением данных, называется:

```
property DisplayName: String; //только для чтения
```

Содержимое этого свойства используется Delphi при выводе сообщения об ошибке, возникшей при редактировании данных в поле. И хотя свойство DisplayName доступно только для чтения, мы сумеем изменить содержащийся в нем текст. Дело в том, что DisplayName повторяет все изменения в свойстве DisplayLabel. Спросите, зачем так все лихо закручено, за ответом отправлю вас к первоисточнику — модулю DB, в котором и описан класс TField.

Ширина колонки в сетке TDBGrid зависит от свойства

```
property DisplayWidth: Integer;
```

значение которого определяет минимальное число символов, помещаемое в колонку.

Выравнивание внутри поля осуществляется в соответствии с правилами, описанными в свойстве

```
property Alignment: TAlignment;
```

Поле даже способно инициировать получение фокуса ввода компонентом, отображающим это поле. Для этого достаточно вызвать метод:

```
function FocusControl;
```

Обработка событий

Объект класса TField способен инициировать четыре события. Перед тем как отредактированное содержимое поля передается в буфер записи, вызывается обработчик события:

```
property OnValidate : TFieldNotifyEvent;
```

Это событие применяют для окончательного утверждения нового значения поля. Представьте себе, что в какой-то таблице SQLTable1 есть два поля: BDate — дата рождения и HighSchoolDate — дата окончания школы. Логика программы не должна допускать, чтобы дата рождения вдруг оказалась более поздней, чем дата получения среднего образования (листинг 19.4).

Листинг 19.4. Пример обработки события `OnValidate()`

```
procedure TForm1.SQLTable1BornDateValidate(Sender: TField);
begin
If SQLTable1BDate.Value>=SQLTable1HighSchoolDate.Value then
   raise
   EDatabaseError.Create('Ошибка при вводе даты!');
end;
```

В листинге 19.4 предложен один из вариантов реализации бизнес-правил на уровне кода клиентского приложения. Обработчик события осуществляет контроль над тем, чтобы выпускник перед завершением школы, по крайней мере, успел родиться. В противном случае принудительно генерируется исключительная ситуация `EDatabaseError` и оператор компьютера информируется об ошибке ввода данных.

Для принудительного вызова события `OnValidate()` предназначена процедура

```
procedure Validate(Buffer: Pointer);
```

В случае успешного завершения события `OnValidate` (сразу после размещения значения поля в буфере записи) у поля `TField` возникает очередное событие:

```
property OnChange: TFieldNotifyEvent;
```

Это служит признаком того, что данные успешно приняты.

При чтении содержимого свойств `DisplayText` или `Text` инициируется код обработки события:

```
type TFieldGetTextEvent = procedure(Sender: TField; var Text: String;
                                    DisplayText: Boolean) of object;
property OnGetText: TFieldGetTextEvent;
```

Параметр `Text` содержит форматированный текст. При желании можно "на лету" вносить в него корректировки. При установке в `true` параметра `DisplayText` форматирование применяется только при показе данных, иначе форматирование распространяется и на режим редактирования данных в поле.

С каждым изменением свойства `Text` поля вызывается событие

```
type TFieldSetTextEvent =procedure(Sender: TField;
                                   const Text: String) of object;
property OnSetText: TFieldSetTextEvent;
```

Поля подстановки

Поля подстановки (lookup fields) программисты активно применяют для построения связи между главной и справочной таблицей. Изучите схему данных, приведенную на рис. 19.3. В нашем распоряжении две таблицы. В справочную таблицу `DEPARTMENT` занесены названия отделов и служб предприятия. Во второй таблице `SALARY` вы найдете текстовое поле `FIO`, предназначенное для хранения фами-

лии и инициалов и одноименное с именем таблицы поле SALARY, специализирующееся на запоминании должностного денежного оклада работника предприятия.

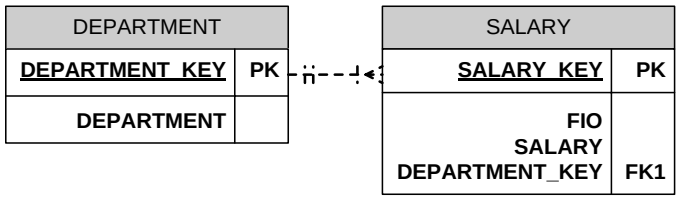


Рис. 19.3. Модель данных для примера работы с полем подстановки

Между собой таблицы связаны по ключевым полям DEPARTMENT_KEY. При вводе новой записи в таблицу SALARY в поле внешнего ключа DEPARTMENT_KEY передается значение, соответствующее первичному ключу в справочной таблице DEPARTMENT. Однако такое представление информации абсолютно неприемлемо для пользователя, вместо непонятных цифр в таблице ему хотелось бы видеть реальное название отдела. К счастью, это не сложно сделать, механизм, осуществляющий поиск данных в справочной таблице, с последующим показом найденных данных в исходной таблице реализуется при посредничестве поля подстановки.

Создадим базу данных в СУБД Microsoft Access. Для этого проведем следующие операции:

- Создайте новый проект базы данных в среде Access и сохраните ее в отдельном каталоге, например под именем demo.mdb.
- Создайте таблицу DEPARTMENT, включающую поле первичного ключа DEPARTMENT_KEY автоинкрементного типа и текстовое поле длиной 30 символов DEPARTMENT для хранения названия отдела предприятия.
- Создайте таблицу SALARY (рис. 19.4), состоящую из четырех полей: поля первичного ключа SALARY_KEY автоинкрементного типа, текстового поля длиной 30 символов FIO для хранения фамилии сотрудника, поля денежного типа SALARY с данными о ежемесячном жаловании и внешнего ключа DEPARTMENT_KEY целочисленного типа.
- У поля SALARY установите значение по умолчанию, равное нулю.

Введите в таблицу DEPARTMENT несколько записей и закройте программу Access, в ближайшее время она нам не понадобится.

Теперь приступим к проектированию приложения Delphi. Начните новый проект. Переименуйте главную форму проекта Name:=frmMain и расположите на ее поверхности следующие компоненты (рис. 19.5):

- Две таблицы (компоненты класса TADOTable, вы их найдете на странице **dbGO**). Первый компонент предназначен для взаимодействия с таблицей DEPARTMENT, поэтому переименуем его в tblDEPARTMENT. Второй компонент станет обслуживать таблицу SALARY, назовем его tblSALARY.
- Источник данных DataSource1 (страница **Data Access**). Компонент предназначен для взаимодействия с таблицей SALARY, переименуйте его в dsSALARY.

С помощью свойства `DataSet` подключите источник данных к компоненту `tblSALARY`.

- Сетку для отображения данных `DBGrid1` и компонент для управления данными `DBNavigator1` (оба компонента вы найдете на странице **Data Controls**). С помощью свойства `DataSource` присоедините оба компонента к `dsSALARY`.

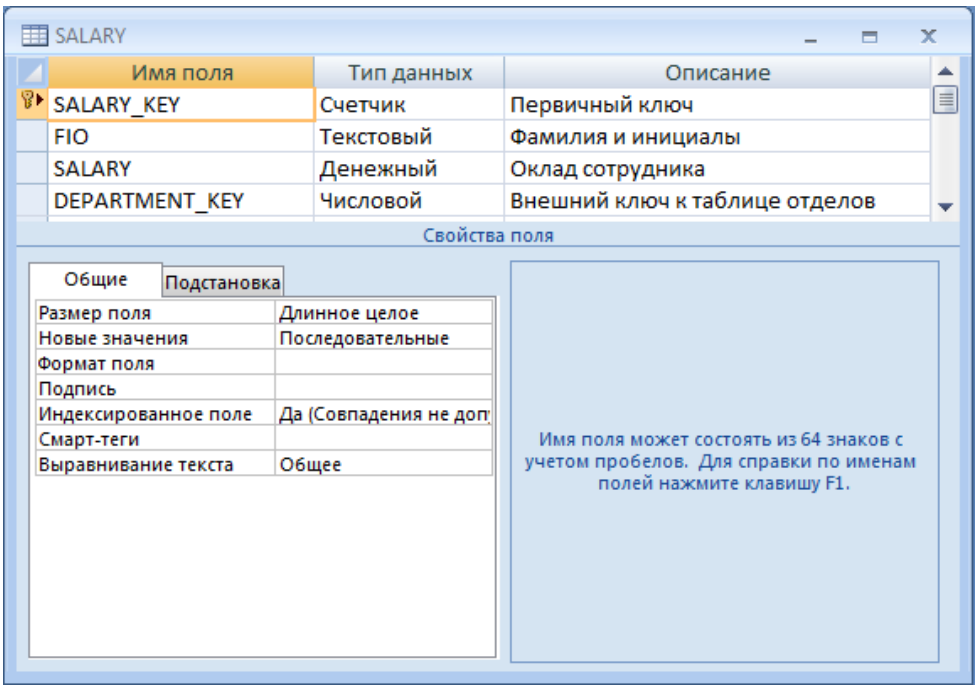


Рис. 19.4. Таблица `SALARY` в режиме конструктора

Замечание

Если проект БД содержит большое число таблиц и запросов, то для размещения невизуальных компонентов целесообразно использовать специальный контейнер — модуль данных `TDataModule`.

Сохраните проект под именем fielddemo.dpr в том же каталоге, где находится созданная ранее база данных demo.mdb. Выберите компонент `tblDEPARTMENT` и, воспользовавшись свойством `ConnectionString`, подключите его к нашей базе данных. При этом помните, что для подключения к базам данных Access ранних версий в качестве провайдера (поставщика данных) следует выбирать Microsoft.Jet.OLEDB.4.0, начиная с Access 2007 — Microsoft.Jet.OLEDB.12.0. Внимательно изучите содержимое текстовой строки в свойстве `ConnectionString`, здесь окажется примерно такой текст:

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Examples\demo.mdb;
Persist Security Info=False
```

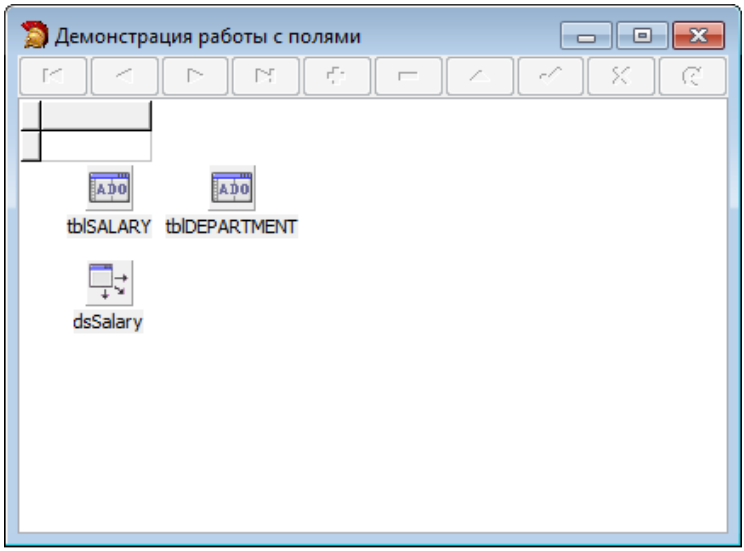


Рис. 19.5. Вид главного окна приложения

Скопируйте эту же строку в свойство `ConnectionString` второго компонента — таблицы `tblSALARY`, ведь ему нужен доступ к той же БД.

Замечание

В dbGO имеется специальный компонент, отвечающий за организацию централизованного подключения наборов данных (компонентов класса `TADOTable`, `TADOQuery`, `TADOStoredProc` и т. д.) к СУБД, он называется `TADOConnection`. С ним мы познакомимся в *главе 22*.

Закончив работать с соединительной строкой, подключите компоненты к соответствующим таблицам:

- компонент `tblDEPARTMENT` к таблице отделов и служб

  `TableName:= 'department';`
- компонент `tblSALARY` к таблице сотрудников

  `TableName:='salary'.`

Проверьте корректность подключения, активизировав свойства `Active` обеих таблиц.

Рассмотренный способ подключения к БД не лишен недостатков. Основной из них — жесткая привязка проекта к файловому пути, сохраненному в соединительной строке. Любая попытка переноса файла приложения или файла базы данных из первоначального каталога приведет к нарушению работоспособности проекта. Поэтому значительно лучшим решением станет применение адаптивного механизма подключения компонентов к БД. Решение несложное, для его воплощения в жизнь достаточно написать несколько строк кода в процедуре обработки события создания главной формы проекта `OnCreate()`. Исследуя строку соединения, вы заметили, что путь к файлу с базой данных хранится в параметре `DataSource`. Воспользу-

емся этим в листинге 19.5. Наша идея проста, она заключается в динамической компоновке строки подключения.

Листинг 19.5 Настройка строки подключения во время создания главной формы

```
procedure TfrmMain.FormCreate(Sender: TObject);
var S:string;
begin
S:=ExtractFilePath(Application.ExeName)+ '\demo.mdb';
if FileAge(s)>0 then \\ проверка на существование файла
  begin              \\ подготовим новую соединительную строку
   tblSALARY.ConnectionString:=
        'Provider=Microsoft.Jet.OLEDB.4.0;Data Source='+
        s+';Persist Security Info=False';
   tblSALARY.TableName:='SALARY';
   tblDEPARTMENT.ConnectionString:=tblSALARY.ConnectionString;
   tblDEPARTMENT.TableName:='DEPARTMENT';
   tblDEPARTMENT.Open;
   tblSALARY.Open;
  end else
  begin {если файл БД не обнаружен, прекращаем работу приложения}
  Application.MessageBox(pChar('Не найден файл БД!'),'Ошибка',MB_OK);
  Application.Terminate;
  end;
end;
```

Во время создания формы с помощью функции `ExtractFilePath()` и свойства приложения `ExeName` извлекаем путь к исполняемому файлу приложения и сохраняем его в переменной `S`. Поскольку файл с БД расположен в этом же каталоге, нам осталось лишь добавить к переменной `S` имя файла demo.mdb. Затем, убедившись в наличии файла с данными, мы склеиваем соединительную строку ADO, присваивая параметру `Data Source` значение из переменной `S`. В итоге мы добиваемся важного преимущества — наш проект вместе с файлом БД можно переносить в любой каталог, файл проекта сам найдет путь к данным, лишь бы БД находилась в одной папке с исполняемым файлом приложения.

Вновь вернемся к теории работы с полями подстановки. Наша задача — научить компонент-таблицу `tblSalary` вместо числового значения поля внешнего ключа `DEPARTMENT_KEY` отображать на экране текстовое название отделов и служб предприятия.

Механизм подстановки реализован на уровне поля — класса `TField`. Для определения поля такого типа программисту потребуется настроить четыре свойства: `KeyFields`, `LookupDataSet`, `LookupKeyFields` и `LookupResultField`.

В первую очередь, требуется указать имя внешнего ключа главной таблицы, на основании которого будет осуществляться поиск данных в справочной таблице:

```
property KeyFields: String;
```

Для нашего примера схемы данных (рис. 19.3) создаем поле подстановки в главной таблице SALARY, а в свойство следует передать имя поля DEPARTMENT_KEY.

Справочный набор данных определяется свойством

```
property LookupDataSet: TDataSet;
```

В роли такого набора могут выступать все потомки класса TDataSet, способные подключиться к таблице или таблицам. Опять же (в соответствии с примером) справочной таблицей служит таблица отделов — DEPARTMENT.

Далее нужно указать имя первичного ключа справочного набора данных:

```
property LookupKeyFields: String;
```

Поле, играющее роль первичного ключа в таблице отделов и служб, называется DEPARTMENT_KEY. По значению этого ключа производится синхронный поиск данных.

И, наконец, свойство

```
property LookupResultField: String;
```

определяет имя поля справочного набора данных "подставляемого" в главную таблицу. В нашем случае это поле DEPARTMENT.

Свидетельством того, что поле превратилось в поле подстановки, станет значение true, возвращаемое свойством

```
property Lookup: Boolean;
```

Если справочная таблица невелика, то имеет смысл забрать ее в буфер компонента. Тогда значительно повысится скорость поиска значения подстановки, особенно если мы работаем с базой данных в сети. Для включения режима кэширования следует установить в true свойство

```
property LookupCache: Boolean;
```

Если режим кэширования таблицы синхронного просмотра включен, то необходимо помнить, что, единожды заполнив кэш, приложение перестанет отслеживать все изменения в справочной таблице, которые, например, могут производиться другими пользователями в сети. В таких случаях для принудительного обновления данных в кэш компьютера следует вызывать процедуру

```
procedure RefreshLookupList;
```

Вновь возвращаемся к нашему проекту Delphi. Мы оказались у порога самого ответственного момента — создания подстановочного поля.

Дважды щелкните левой кнопкой мышки по компоненту tblSALARY для вызова редактора полей этой таблицы. Во всплывающем меню редактора полей выберите пункт **Add all fields** (Добавить все поля). После этого в редакторе появятся все поля таблицы SALARY. Осуществленную нами операцию называют определением статических полей. В результате для каждого из столбцов таблицы SALARY среда Delphi поставила в соответствие отдельный объект поля — потомок класса TField. Класс поля определяется типом хранящихся в нем данных, например, для текстового поля создается объект класса TWideStringField, для денежного — TFMTBCDField; для поля счетчика — TAutoIncField и т. д. Каждый объект-поле

получает имя путем сложения имен компонента-таблицы и физического имени поля. В нашем случае поле, хранящее данные о фамилии сотрудника, будет названо tblSALARY+FIO=tblSALARYFIO.

ВНИМАНИЕ!

Редактор "New Field" не предназначен для физического добавления поля в структуру таблицы. Он лишь способен создавать экземпляры поля (потомки класса TField) на программном уровне. В качестве основных клиентов редактора выступают вычисляемые (*calculated*) поля и поля подстановки (*lookup*).

Вновь вызовите контекстное меню редактора полей и найдите пункт меню **New field…** (Новое поле). Это действие вызовет окно настройки (рис. 19.6), в котором следует описать характеристики поля подстановки.

- В строке **Name** (имя поля) укажем имя создаваемого поля — luDEPARTMENT.
- В списке **Type** (тип данных) определим тип данных этого поля — String.
- В группе переключателей **Field type** (тип поля) установите флажок **Lookup** (Просмотр).

Заполняем четыре комбинированных списка в группе **Lookup definition** (Определение просмотра). Данные вводим слева направо, сверху вниз. Приступим. Выпадающий список **Key fields** (Ключевые поля) определяет поле таблицы сотрудников, являющееся внешним ключом, т. е. предназначенным для взаимодействия с таблицей отделов и служб. Внешний ключ в таблице сотрудников — поле DEPARTMENT_KEY, поэтому останавливаем свой выбор на нем. В выпадающем списке **Dataset** указываем имя справочного набора данных. У нас это таблица tblDEPARTMENT. Во втором ряду, в списке **Lookup Keys** (Ключи просмотра) требуется ввести имя первичного ключа таблицы-справочника отделов и служб — поле DEPARTMENT_KEY. И, наконец, в последнем списке **Result Field** (Результирующее поле) необходимо выбрать имя поля, которое мы хотим видеть в результирующем наборе данных — название отдела (DEPARTMENT). Закончив описание нового поля, нажимаем кнопку **OK**.

С этого момента наш проект приобретает работоспособность, но, к сожалению, нельзя сказать, что он лишен недостатков. Один из них — избыточность информации, отображаемой в сетке DBGrid1. Этот компонент выводит на экран столбцы, которые совсем не обязательно видеть пользователю, это служебные поля первичного (SALARY_KEY) и вторичного (DEPARTMENT_KEY) ключей таблицы tblSALARY. Рекомендую найти свойства Visible названных полей и присвоить им значения false, что сделает поля невидимыми. Кроме того, имеет смысл настроить свойство DisplayLabel всех оставшихся видимыми полей так, чтобы колонки сетки DBGrid1 стали отражать осмысленные названия полей (рис. 19.7).

ЗАМЕЧАНИЕ

Помимо компонента TDBGrid в среде Delphi реализовано еще два элемента управления, специализирующиеся на работе с полями подстановки. Это компоненты TDBLookupListBox и TDBLookupCombpBox, их вы найдете на странице Data Access палитры компонентов.

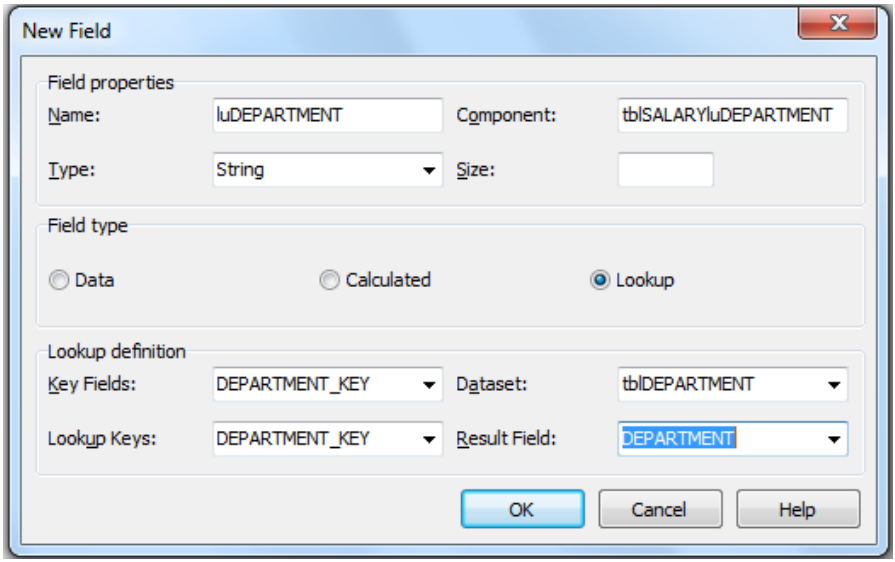


Рис. 19.6. Создание подстановочного поля

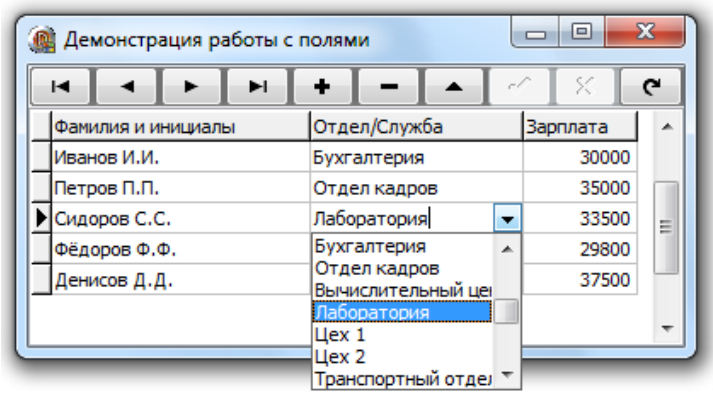


Рис. 19.7. Поле подстановки в действии

Вычисляемые поля

Опытные программисты знают, что без особой необходимости в таблицах не следует хранить информацию, дублирующую уже существующие данные. Также нужно избегать накопления информации, которую можно получить, выполнив простейшие операции над существующими данными. Поясним это утверждение на примере. Допустим, что нам следует доработать предыдущую БД так, чтобы она информировала нас о величине подоходного налога, взимаемого у каждого из сотрудников. В такой ситуации излишне заводить отдельное физическое поле в таб-

лице SALARY для внесения в него суммы налога, ведь зная жалование сотрудника, величину налога можно получать при помощи элементарной математики.

Вновь возвращаемся к примеру. Еще раз обратимся к всплывающему меню редактора полей tblSALARY и выберем пункт **New Field**. На экран будет вызван редактор, позволяющий создавать новый экземпляр поля.

В нашем случае конструируем вычисляемое поле. Для этого в строке **Name** указываем имя нового поля — CALC_TAX, в комбинированном списке **Type** определяем тип поля — Currency. Убедитесь, что в группе переключателей **Field type** отмечен флажок **Calculated** (Вычислить), и смело жмите на кнопку **OK** (рис. 19.8). После того как окно редактора закроется, загляните в инспектор объектов. В нем появился новый объект — поле tblSALARYCALC_TAX, а его свойство FieldKind установилось в состояние fkCalculated.

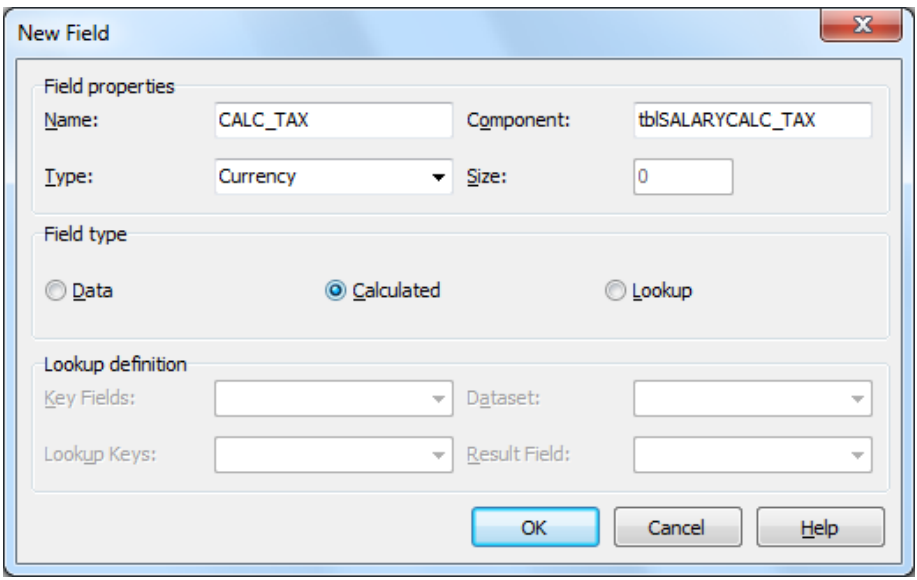


Рис. 19.8. Создание нового вычисляемого поля

Наступил черед поломать голову над листингом программы. У компонента tblSALARY находим обработчик события OnCalcFields() и внутри него напишем всего одну строку (листинг 19.6).

Листинг 19.6. Расчет величины подоходного налога

```
procedure TfrmMain.tblSALARYCalcFields(DataSet: TDataSet);
begin
tblSALARYCALC_TAX.Value:=tblSALARYSALARY.AsCurrency*0.13;
end;
```

Проверьте состояние свойства `AutoCalcFields` таблицы, оно должно быть установлено в `true`. Опять включаем таблицу (`Active:=True`) и запускаем проект на выполнение. Теперь отображающая данные сетка `DBGrid1` дополнилась новой колонкой, содержащей величину подоходного налога (рис. 19.9).

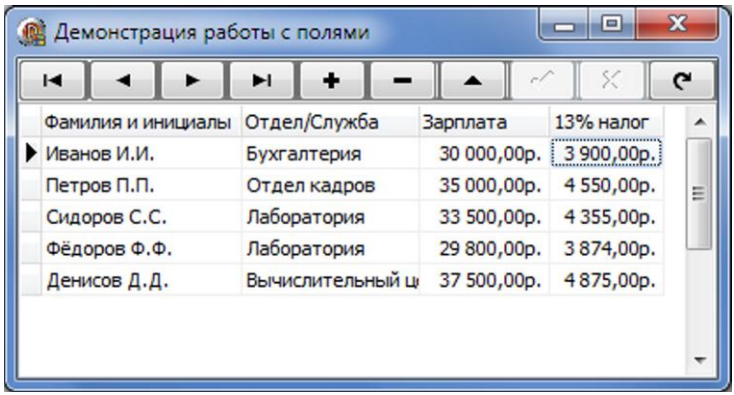


Рис. 19.9. Работа вычисляемого поля

Числовые поля, класс *TNumericField*

В соответствии со своим названием, абстрактный класс `TNumericField` инкапсулирует свойства и методы, общие для всех полей, предназначенных для обслуживания числовых данных, и служит родоначальником для ряда важных классов: `TIntegerField`, `TLargeintField`, `TFloatField`, `TBCDField` и `TFMTBCDField`.

Способ представления содержимого поля на экране определяется свойством

```
property DisplayFormat: String;
```

Используя спецификаторы из табл. 19.6, программист определяет формат вывода значений в компонентах отображения данных.

Таблица 19.6. *Спецификаторы, используемые в свойстве* `DisplayFormat`

Спецификатор	Описание
0	Позиция для обязательных цифровых символов
#	Позиция для необязательных цифровых символов
.	Разделитель целой и дробной части
,	Разделитель тысяч
E+, E-	Научный способ вывода значения

Таблица 19.6 *(окончание)*

Спецификатор	Описание
'xx'/"xx"	Символы-комментарии вводятся внутри одинарных или двойных кавычек
;	Сепаратор, разделяющий положительные, отрицательные и нулевые значения

Рассмотрим несколько элементарных примеров форматирования (листинг 19.7).

Листинг 19.7. Примеры форматирования с помощью свойства `DisplayFormat`

```
Field1.DisplayFormat:= '.00';
{Округляет представляемое значение до второго знака после запятой. Если в поле
хранится целое число, например 1, то все равно при показе его на экране будет
выведено два знака после запятой — 1.00}

Field2.DisplayFormat:= '.####';
{Округляет значение до четвертого знака после запятой. Однако, если число
целое или не располагает таким числом знаков после запятой, оно не будет
дополняться нулями}

Field3.DisplayFormat:= '###,###,###';
{Целая часть числа будет разбита на тысячи. Например, исходное значение
999999999 будет представлено в виде 999 999 999 (с пробелом через каждые три
знака)}

Field4.DisplayFormat:= '00E+'; //для больших чисел
Field5.DisplayFormat:= '#.E-'; //для малых чисел
{Научное представление, как правило, применяется только при выводе очень
больших или сверхмалых значений. Например:
значение 1000000  (формат 00E+) отобразиться на экране, как 10E+6,
значение 0,000001 (формат #.E-)                           как  1E-6}

Field6.DisplayFormat:= '.## "кг"';
{Допускается ввод комментариев, текст комментария должен заключаться в двойные
или одинарные кавычки. Таким образом, мы можем пояснить пользователю, что в
этом поле мы работаем с килограммами, квадратными метрами или с чем-то другим}

Field7.DisplayFormat:= '"+" .##; "-" .## ';
{Пример работы с символом, предназначенным для настройки способа вывода поло-
жительных, отрицательных и нулевых значений. Предложенная строка форматирова-
ния разбита на две секции. Первая секция настраивает вывод положительных и ну-
левых значений. Вторая секция — отрицательных значений. Вообще допускается не
более трех секций, строка будет содержать два символа-сепаратора ";". В таком
случае первая секция отвечает за показ положительных, вторая — отрицательных и
третья — нулевых значений}
```

Еще одно свойство класса TField

```
property EditFormat: String;
```

определяет формат поля при его редактировании пользователем. Спецификаторы строки формата идентичны свойству DisplayFormat (см. табл. 19.6) и отдельно рассматриваться не будут.

Поля целых чисел *TLargeintField*, *TIntegerField*, *TSmallintField*, *TWordField*, *TAutoIncField*

Классы, TLargeintField берут свое начало от TNumericField, классы TSmallintField и TAutoIncField, в свою очередь, базируются на классе TIntegerField.

Объекты полей этих классов отличаются друг от друга в основном диапазоном допустимых значений. Самый большой спектр значений реализован в классе TLargeIntField, поле этого типа допускает ввод 64-битовых значений. Несколько ниже предел значений у классов TIntegerField и TAutoIncField, он составляет от 2 147 483 648 до 2 147 483 647. Диапазон значений класса TWordField от 0 до 65537. И, наконец, поле типа TSmallintField предназначено для работы в диапазоне значений от –32768 до 32767.

Еще одна особенность поля автоинкрементного типа TAutoIncField заключается в том, что оно не доступно для редактирования, и в столбце автоинкрементного типа не могут находиться повторяющиеся значения.

Диапазон допустимых значений описывается с помощью свойств:

```
property MinValue: Longint;
property MaxValue: Longint;
```

ЗАМЕЧАНИЕ

Хотя свойства MinValue и MaxValue и объявлены в классе TAutoIncField, но в этом типе поля они не несут никакой смысловой нагрузки.

Все четыре класса TLargeintField, TIntegerField, TSmallintField и TAutoIncField содержат данные в свойстве Value и могут представить эти данные в виде целого числа (AsInteger), вещественном (AsFloat), строковом (AsString) и универсальном (AsVariant) форматах.

Поля вещественных чисел *TFloatField* и *TCurrencyField*

Спектр допустимых значений поля, построенного на классе TFloatField, составляет от $5{,}0\times10^{-324}$ до $1{,}7\times10^{308}$ и допускает хранение чисел с плавающей точкой с точностью до 15 знаков после запятой. Так как денежное поле TCurrencyField базируется на классе TFloatField, то это утверждение справедливо и для него.

Оба класса способны ограничивать предел допустимых значений уже знакомыми нам свойствами MinValue и MaxValue.

Число знаков после запятой определяется свойством

```
property Precision: Integer;
```

При попытке указать значение меньше двух свойство автоматически установит точность два знака.

Поля вещественных чисел позволяют представлять свои значения в виде числа с плавающей точкой (AsFloat), округлять до целого (AsInteger), в виде строки (AsString) и в универсальном виде (AsVariant).

Бинарно-кодированные десятичные поля *TBCDField* и *TFMTBCDField*

Бинарно-кодированные десятичные (Binary Coded Decimal, BCD) поля предназначены для хранения вещественных чисел без потерь точности. В Delphi объявлены два класса BCD-полей — TBCDField и TFMTBCDField. Первый из них специализируется на обработке денежных данных, он способен хранить число размерностью двадцать знаков перед запятой и четыре после. Поле типа TFMTBCDField обладает более серьезными возможностями. Основное отличие бинарно-кодированных полей от уже изученных нами полей вещественных чисел TFloatField и TCurrencyField состоит в организации хранения данных. Теперь это не просто 8-байтовая последовательность, а более сложная структура, описываемая записью TBcd. Оговоримся сразу — в формате BCD может быть представлено число, состоящее не более чем из 64 знаков (не включая знак разделителя целой и дробной части). Чем бóльшую точность представления числа (больше знаков после запятой) выбирает программист, тем меньше остается позиций для хранения целой части. Если мы планируем описывать число с точностью 16 знаков после запятой, то на целую часть останется 64 – 16 = 48 позиций. Пример приведен в листинге 19.8.

Листинг 19.8. Объявление записи TBcd

```
type TBcd  = packed record
     Precision: Byte;
     SignSpecialPlaces: Byte;
     Fraction: packed array [0..31] of Byte;
end;
```

Здесь поле Precision определяет число значащих цифр до запятой, результат может находиться в пределе от 1 до 64 знаков. SignSpecialPlaces — 8-битовая последовательность, в которой первый бит определяет знак выражения. Если в нем ноль — то число положительное, иначе — отрицательное. Если второй бит установлен в единицу, то поле пусто и не хранит никакого значения. Оставшиеся шесть битов определяют число знаков после запятой. Здесь может находиться значение от

нуля до числа Precision. Значение поля хранится в массиве Fraction, особенность записи числа в этот массив в том, что в каждом его байте (а из всего 32) может содержаться значение от 0 до 99. И если вы захотите представить в формате BCD вещественное число 7658.786621, то в нулевом байте массива окажется 76, в первом 58, во втором 78 и т. д.

Delphi накладывает ограничение на максимальное число знаков после запятой — не более 32. Это ограничение инкапсулировано в свойствах

```
property Precision: Integer; //не более 32 знаков
```

полей TBCDField и TFMTBCDField.

Для представления значения поля в денежном формате установите в true свойство

```
property Currency: Boolean;
```

Поля, построенные на основе TBCDField и TFMTBCDField, обладают уже привычными нам возможностями: ограничение на максимальное MaxValue и минимальное MinValue значения, возможность преобразования содержимого поля к форматам AsBCD, AsCurrency, AsFloat, AsInteger, AsString и AsVariant.

ЗАМЕЧАНИЕ

Вопросами работы с данными, представленными в виде BCD, ведает отдельный модуль FMTBcd. Именно в нем реализованы методы преобразования значения из одного формата в другой, основные математические операции с числами BCD, операции сравнения и многое другое.

Текстовые поля, *TStringField*

Для обслуживания текстовых данных в визуальной библиотеке компонентов реализован класс TStringField. Как мы уже привыкли, все объекты-поля являются наследником базового класса TField. TStringField в этом плане не исключение, следовательно, о нем мы практически все знаем. Осталось уяснить тонкости.

Первый нюанс состоит в том, что размер строки, которую способен обрабатывать объект TStringField, ограничен 8192 символами, включая последний нулевой символ. Если СУБД работает не на ANSI-совместимом языковом драйвере, то нужно устанавливать в true свойство

```
property Transliterate: Boolean;
```

Еще одна особенность заключается в том, что система управления базами данных способна работать с двумя типами текстовых полей CHAR (WIDECHAR) и VARCHAR (NCHAR VARYING). Поле типа CHAR, например, размерностью в 20 символов при вводе каждой новой записи будет заполнять все 20 символов. И если вы ввели в него меньше знаков, то оставшееся свободное место СУБД заполнит пробелами. Поэтому поле типа CHAR часто называют *полем фиксированной длины*. Поле типа VARCHAR (*поле переменной длины*) к памяти относится не столь расточи-

тельно, и если длина строки меньше установленного размера, то поле переменной длины не станет захватывать не нужные ему байты. Чтобы выяснить, с каким из полей мы работаем, обращаемся к свойству

```
property FixedChar: Boolean;
```

Значение `true` свидетельствует, что это поле фиксированной длины `CHAR`, иначе — поле `VARCHAR`.

Текстовые поля могут представлять данные в классических форматах: `AsBCD`, `AsBoolean`, `AsDateTime`, `AsFloat`, `AsInteger`, `AsSQLTimeStamp`, `AsString` и `AsVariant`.

Класс `TStringField` не абстрактный, т. е. из него можно создать и объект. Кроме того, класс является родительским для `TWideStringField` и `TGuidField`.

Поле глобального идентификатора, *TGuidField*

Класс `TGuidField` построен на основе текстового поля `TStringField`. Назначение этого объекта — поддержка *глобальных уникальных идентификаторов* (Globally Unique Identifiers, GUIDs). Названными идентификаторами снабжаются объекты автоматизации Windows при их регистрации в системе. Если хотите взглянуть на хотя бы один из них, то откройте ветвь HKEY_CLASSES_ROOT реестра Windows. Как только обнаружите строку подобную этой:

```
{A5BA7591-DF36-11D3-B697-009027BFA736}
```

так знайте, это он!

Способ описания идентификатора строг и определяется структурой `TGUID` (листинг 19.9).

Листинг 19.9. Объявление записи `TGUID`

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Для работы с форматом `TGUID` объект обеспечен свойством

```
property AsGuid: TGUID;
```

Для упрощения процесса преобразования строки в идентификатор и для обратной операции в модуле `SysUtils` объявлены две функции: `GUIDToString()` и `StringToGUID()`. В остальном класс аналогичен своему предку `TStringField`.

Логическое поле, *TBooleanField*

Объект предназначен для представления в наборе данных полей логического типа. Непустое поле в состоянии возвращать всего два значения: `true` и `false`. Кроме того, эти значения могут быть представлены в виде: `AsBoolean`, `AsString`, `AsVariant`. Дополнительного внимания достойно уже знакомое нам свойство

```
property DisplayValues: string;
```

Свойство определяет, какие сообщения будут выводиться в компонентах отображения данных при истинном и ложном значении поля. Сообщения разделяются точкой с запятой, причем на первой позиции требуется указать сообщение для значения `true`, а на второй — `false`, например:

```
BooleanField.DisplayValues:= 'Да;Нет';
```

Бинарные поля *TBinaryField*, *TBytesField* и *TVarBytesField*

Класс `TBinaryField` служит основой для построения полей, работающих с двоичными данными. К ним относятся битовые массивы постоянной `TBytesField` и переменной `TVarBytesField` длины. Так как данные хранятся в двоичном виде, то их структура остается неизвестной, поэтому невозможно говорить о каких-либо способах форматирования этих данных. Содержащиеся в полях значения могут быть представлены в виде `AsString` и `AsVariant`.

Дата и время, поля *TDateTimeField*, *TDateField* и *TTimeField*

В перечень объектов-полей, специализирующихся на работе с датой и временем, входят три класса: `TDateTimeField`, `TDateField` и `TTimeField`. Два последних объекта являются наследниками более общего класса `TDateTimeField`.

Способ форматирования выходных данных описывается в свойстве

```
property DisplayFormat: String;
```

Значение поля, доступное в свойстве `Value`, может быть представлено в следующих типах данных: `AsDateTime`, `AsFloat`, `AsSQLTimeStamp`, `AsString` и `AsVariant`.

Классы `TDateField` и `TTimeField` специализируются на работе только со значением даты и времени соответственно. Они обладают набором свойств, практически идентичным своему родительскому классу `TDateTimeField`. Единственное отличие от `TDateTimeField` заключается в размере данных (`DateSize`). У обоих объектов-полей он вдвое меньше и составляет четыре байта.

Дата и время, поле *TSQLTimeStampField*

Поле класса `TSQLTimeStampField` предназначено для альтернативного способа хранения информации о дате и времени. Программисты Delphi привыкли, что обычно дата и время описывается типом данных `TDateTime`, который представляет собой вещественное число размерностью 8 байтов. Целая часть числа определяет дату, а дробная — время. Однако такой способ представления не нашел поддержки в ряде СУБД, взамен ему предложена структура `TSQLTimeStamp` (листинг 19.10).

Листинг 19.10. Объявление записи `TSQLTimeStamp`

```
type TSQLTimeStamp = packed record
    Year: SmallInt;       //год
    Month: Word;          //месяц
    Day: Word;            //день
    Hour: Word;           //часы
    Minute: Word;         //минуты
    Second: Word;         //секунды
    Fractions: LongWord; //дробная часть
  end;
```

В этой записи для каждой характеристики предусмотрено отдельное поле, для года — Year, месяца — Month и т. д. Можно спорить о достоинствах и недостатках такого представления даты и времени, мы этим заниматься не будем, а просто констатируем факт, что такой формат существует и он поддерживается в Delphi.

Возвращаясь к полю `TSQLTimeStampField`, заметим, что оно обладает стандартным набором характеристик. Неотформатированное значение содержится в свойстве `Value`, это значение может быть представлено в виде `AsDateTime`, `AsFloat`, `AsSQLTimeStamp`, `AsString` и `AsVariant`. Способ представления данных можно определить в строке форматирования `DisplayFormat`.

Поля больших бинарных объектов, *TBlobField*, *TGraphicField*, *TMemoField* и *TWildMemoField*

Класс `TBlobField` реализован для обеспечения работы с *двоичными большими объектами* (Binary Large Object, BLOB) переменной длины. Поля этого типа и его производные представляют собой универсальное хранилище большого объема разнотипной информации. Поля BLOB широко представлены практически во всех СУБД.

Если рассуждать о специализации BLOB-полей, то стоит обратить внимание на свойство

```
property BlobType: TBlobType;
```

где

```
type TBlobType = ftBlob..ftOraClob;
```

Тип данных `TBlobType` — всего-навсего подмножество `TFieldType`. Значения, которые может принимать этот тип данных, вы можете просмотреть, вернувшись к табл. 19.2. Хотя свойство `BlobType` доступно как для чтения, так и для записи, но если возникла необходимость поменять тип поля, целесообразнее вызвать метод

```
procedure SetFieldType(Value: TFieldType); override;
```

Если БД по умолчанию не использует языковой драйвер ANSI и в полях таблиц могут находиться символы из расширенного набора ASCII, то для автоматического преобразования символов в кодировку ANSI предназначено свойство:

```
property Transliterate: Boolean;
```

Если свойство установлено в `true`, то будет вызван метод `Translate()`, преобразующий символы UTF-8 или ASCII в кодировку базы данных.

Не так давно на рынке локальных СУБД активно эксплуатировалась СУБД Paradox. Поэтому некоторые свойства компонентов предназначены специально для работы с этим форматом БД. Класс `TBlobField` не стал исключением. Свойство-индикатор

```
property GraphicHeader: Boolean;
```

примет значение `true`, если объект является графическим полем формата Paradox.

Если изученные ранее классы полей не вносили новых методов, за исключением конструктора `Create`, то BLOB-поле в этом плане несколько богаче. Почти все методы BLOB-поля нацелены на решение вопросов чтения или записи данных.

Для загрузки данных в поле предназначены два метода:

```
procedure LoadFromFile(const FileName: string);
procedure LoadFromStream(Stream: TStream);
```

В первом случае требуется передать имя графического файла `FileName`. Реализация второго метода несколько сложнее, в нем работа осуществляется с областью памяти — потоком `Stream`.

Для выгрузки данных из поля реализованы два симметричных метода с аналогичными параметрами:

```
procedure SaveToFile(const FileName: string);
procedure SaveToStream(Stream: TStream);
```

Для копирования данных из другого BLOB-поля определена процедура

```
procedure Assign(Source: TPersistent);
```

Здесь в качестве параметра передается источник данных — `Source`. В роли источника данных могут выступать: другие BLOB-поля, набор строк `TStrings`, графические объекты, например `TBitmap`.

Факт изменения данных внутри поля можно проверить, просмотрев свойство

```
property Modified: Boolean;
```

Значение `true` — признак того, что поле изменялось.

Для очистки содержимого BLOB-поля вызывайте процедуру

```
procedure Clear;
```

Поля типа `TGraphicField` и `TMemoField` практически повторяют своего предка — `TBlobField`. Классы не обладают ни своими свойствами, ни собственными методами. Единственное исключение — конструкторы классов, а о "глубине" отличий от предка судите сами по листингам 19.11 (конструктор графического поля) и 19.12 (конструктор многострочного неформатированного текстового поля).

Листинг 19.11. Конструктор графического поля

```
constructor TGraphicField.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  SetDataType(ftGraphic);
end;
```

Листинг 19.12. Конструктор многострочного неформатированного текстового поля

```
constructor TMemoField.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  SetDataType(ftMemo);
  Transliterate := True;
end;
```

Рассмотрим небольшой пример, демонстрирующий работу с BLOB-полем. Задача заключается в следующем: нам требуется научить приложение сохранять в таблице растровую картинку и имя файла, соответствующего этому изображению. Начнем с проектирования структуры таблицы. Для решения поставленной задачи нам достаточно определить три поля: первичный ключ `PICTURE_KEY`, поле `PICTUREBLOB` для хранения графики и текстовое поле для хранения имени файла `PICTURENAME`. Воспользуемся услугами СУБД Microsoft Access 2007 и создадим в ней файл БД с именем blobdemo.accdb. База данных будет содержать единственную таблицу `PICTURETABLE` (рис. 19.10) с описанными ранее характеристиками.

PICTURETABLE

Имя поля	Тип данных	
PICTURE_KEY	Счетчик	Ключевое поле
PICTURENAME	Текстовый	Название файла с рисунком
PICTUREBLOB	Поле объекта OLE	Объект BLOB

Рис. 19.10. Структура таблицы для работы с полем BLOB

Приступаем к работе в Delphi. Создаем новый проект и размещаем на главной форме следующий перечень компонентов:

- таблицу — `ADOTable1:TADOTable`;
- диалог доступа к файлам в стиле Microsoft Vista — `FileOpenDialog1:TFileOpenDialog`;
- объект для отображения рисунков — `Image1:TImage`;
- метку для отображения названия файла — `StaticText1:TStaticText`;
- семь кнопок класса `TSpeedButton`.

Первые четыре кнопки (рис. 19.11) предназначены для перемещения по записям внутри таблицы. Назовем их следующим образом: `btnFirst` — для перемещения на самую первую запись, `btnPrior` — предыдущая запись, `btnNext` — следующая запись, `btnLast` — последняя запись. Кроме имен кнопок потребуется изменить значения их свойств `tag`. Для `btnFirst` — 0, `btnPrior` — 1, `btnNaxt` — 2, `btnLast` — 3. И осталось три кнопки, отвечающие непосредственно за редактирование данных в таблице: `btnAppend` — добавляет новую запись в конец набора, `btnEdit` — редактирует текущую запись и `btnDelete` — отвечает за удаление.

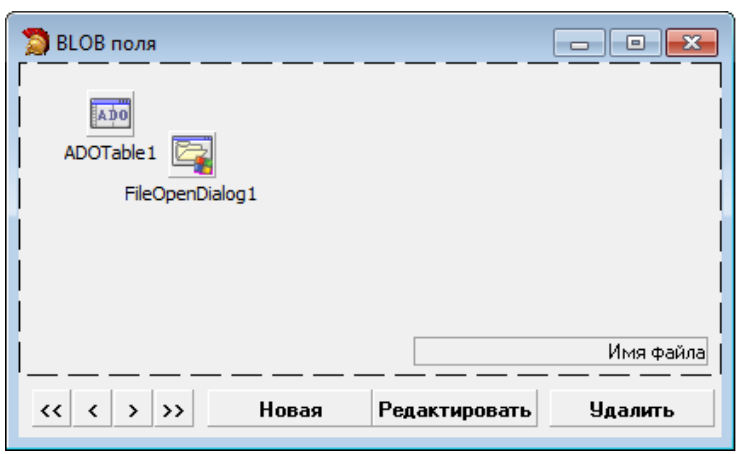


Рис. 19.11. Внешний вид формы для работы с BLOB-полем

Сохраните проект в том же каталоге, где расположен файл blobdemo.accdb. Главную форму проекта по традиции назовем `frmMain.dfm`, соответствующий ей модуль — `main.pas`, а весь проект — `blobdemo.dpr`.

Настройте строку соединения для компонента `ADOTable1`. Для взаимодействия с Access 2007 следует задействовать провайдера Microsoft.ACE.OLEDB.12.0. В результате в свойстве `ConnectionString` компонента окажется примерно такая строка:

```
Provider=Microsoft.ACE.OLEDB.12.0;Data Source=
C:\Examples\blobdemo.accdb;Persist Security Info=False
```

Возвращаемся к нашему проекту. В секции частных объявлений опишем заголовки двух процедур `SetData()` и `GetData()`. Процедура `SetData()` предназначена для организации загрузки файла рисунка в BLOB-поле. Для этого вызывается диа-

лог открытия изображения. Если пользователь выберет файл и при этом таблица находится в режиме вставки (dsInsert) или редактирования (dsEdit), оба поля записи будут заполнены соответствующими данными (листинг 19.13).

Листинг 19.13. Процедура сохранения изображения в BLOB-поле таблицы

```
procedure TfrmMain.SetData;
begin
if (FileOpenDialog1.Execute) and
        ((ADOTable1.State=dsInsert) or (ADOTable1.State=dsEdit)) then
    begin
   (ADOTable1.FieldByName('PICTUREBLOB') as
    TBlobField).LoadFromFile(FileOpenDialog1.FileName);
    ADOTable1.FieldByName('PICTURENAME').AsString:=
                            ExtractFileName(FileOpenDialog1.FileName);
   end;
end;
```

Процедура GetData() предназначена для извлечения данных из текстового и BLOB-поля (листинг 19.14). Название файла загружается в Caption компонента StaticText1. Если поле Image содержит данные, то мы создаем поток M:TMemoryStream, загружаем в него данные из поля и отображаем их в компоненте Image1. Обращаю внимание на то, что перед передачей данных из потока его следует установить на нулевую позицию. В заключение освобождаем ресурсы, занимаемые потоком, вызвав метод Free(), код предложен в листинге 19.14.

Листинг 19.14. Процедура чтения данных из BLOB-поля таблицы

```
procedure TfrmMain.GetData;
var M : TMemoryStream;
begin
with ADOTable1 do
     begin
      StaticText1.Caption:=FieldByName('PICTURENAME').AsString;
      if (FieldByName('PICTUREBLOB') as TBlobField).IsNull=False then
       begin
         M:=TMemoryStream.Create;
         (FieldByName('PICTUREBLOB') as TBlobField).SaveToStream(M);
         M.Position:=0;
         Image1.Picture.Bitmap.LoadFromStream(M);
         M.Free;
       end else Image1.Picture.Bitmap.FreeImage;
    end;
end;
```

После того как были созданы ключевые процедуры чтения и записи данных, переходим к описанию основных событий. Первое из них — событие `OnCreate()` главной формы проекта. В нем мы осуществим подключение компонента `ADOTable1` к базе данных, определим параметры диалога открытия изображений (листинг 19.15).

Листинг 19.15. Событие создания главной формы проекта

```
procedure TfrmMain.FormCreate(Sender: TObject);
var s:string;
    FT:TFileTypeItem;
    const db='\blobdemo.accdb';
begin
{подключаемся к базе данных}
s:=ExtractFilePath(Application.ExeName)+db;
if FileAge(s)>0 then
  begin
   ADOTable1.ConnectionString:='Provider=Microsoft.ACE.OLEDB.12.0;
                  Data Source='+s+';Persist Security Info=False';
   ADOTable1.TableName:='PICTURETABLE';
   ADOTable1.Open;
   GetData;
   {настраиваем файловый диалог в стиле Vista}
   FT:=FileOpenDialog1.FileTypes.Add;
   FT.FileMask:='*.bmp';
   FT.DisplayName:='Растровый файл';
 end else
  begin
   Application.MessageBox(pChar('Не найден файл'+#13+S),'Ошибка',MB_OK);
   Application.Terminate;
  end;
end;
```

Выберите кнопку `btnFirst`, отвечающую за перевод курсора на самую первую запись в таблице и опишите ее обработчик события `OnClick()`. Особенность события в том, что оно будет применяться для всех четырех кнопок перемещения. Если помните, кнопки перемещения по записям таблицы различаются свойством `tag`, `btnFirst` — 0, `btnPrior` — 1, `btnNext` — 2, `btnLast` — 3. Анализируя свойство `tag` в селекторе `Case`, мы вызываем соответствующий метод таблицы (листинг 19.16).

Листинг 19.16. Перемещение по записям набора данных

```
procedure TfrmMain.btnFirstClick(Sender: TObject);
begin
with ADOTable1 do
```

```
Begin
{перемещаем курсор в соответствии со значением свойства tag кнопок перемещения
по записям}
     case (sender as TComponent).Tag of
        0: First; //на первую запись
        1: Prior; //на предыдущую запись
        2: Next;  //на следующую запись
        3: Last;  //на последнюю запись
     end;
     GetData;//получаем данные из таблицы
{с помощью методов BOF и EOF проверяем местоположение текущей записи}
     btnFirst.Enabled:=NOT(BOF);{кнопка отключается,
                                 если курсор на первой записи}
     btnPrior.Enabled:=NOT(BOF);
     btnNext.Enabled:=NOT(EOF);{кнопка отключается,
                                если курсор на последней записи}
     btnLast.Enabled:=NOT(EOF);
end;
end;
```

Теперь осталось описать щелчки по кнопкам добавления, редактирования и удаления записи (листинг 19.17).

Листинг 19.17. События добавления, редактирования и удаления записи

```
procedure TfrmMain.btnAppendClick(Sender: TObject);//добавление
begin
 ADOTable1.Append; {добавление новой записи в конец набора}
 SetData; {сохраняем данные}
 GetData; {передаем данные в компонент Image1 и StaticText1}
end;

procedure TfrmMain.btnEditClick(Sender: TObject);//редактирование
begin
 ADOTable1.Edit;{перевод таблицы в режим редактирования}
 SetData;       {сохраняем данные}
 GetData;       {получаем данные}
end;

procedure TfrmMain.btnDeleteClick(Sender: TObject);//удаление
begin
if MessageBox(frmMain.Handle,
              pWideChar('Удалить запись?'),
              pWideChar('Удаление'),
              MB_YESNO + MB_ICONQUESTION)=idYes then
```

```
    begin
      ADOTable1.Delete;                  {удаление записи}
      Image1.Picture.Bitmap.FreeImage; {освобождаем ресурс картинки}
      StaticText1.Caption:='';           {очищаем заголовок метки}
      GetData;                           {получаем данные}
    end;
end;
```

Проект завершен. Экранный снимок работающего приложения приведен на рис. 19.12.

Рис. 19.12. Внешний вид приложения, работающего с BLOB-полем

Композитные поля, *TObjectField*

Абстрактный класс `TObjectField` служит основой для построения композитных полей (см. рис. 19.1). Композитное поле создается из нескольких простых полей (`TADTField` и `TArrayField`) или полей набора данных, ссылающихся на поля из других наборов (`TDataSetField` и `TReferenceField`). Применительно к `TDataSetField` и `TReferenceField` можно сказать, что композитное поле предназначено для представления отношений "один ко многим".

Класс `TADTField` называют абстрактным типом данных (Abstract Data Type). Объект такого типа работает с нестандартными полями, тип которых определен пользователем на сервере.

Класс `TArrayField` способен представлять массив дочерних полей одного типа.

Класс `TDataSetField` образует поле, связывающее одно или несколько полей из другого набора данных с текущей записью своего набора данных.

Класс `TReferenceField` ссылается на другое поле абстрактного типа данных.

Весь набор полей содержится в списке:

```
property Fields: TFields;//только для чтения
```

Общее число полей в списке окажется в свойстве

```
property FieldCount: Integer;//только для чтения
```

Для доступа к значению отдельного поля требуется передать его индекс в списке `Fields`:

```
property FieldValues[Index: Integer]: Variant;
```

Листинг 19.18 иллюстрирует код программы при получении данных из ADT-поля.

Листинг 19.18. Получение данных из составного поля

```
var I : Integer;
    s : string;
    d : TDateTime;
begin
...
with ADOTable1.FieldByName('Field1') do
begin
  I:= FieldValues[0];
  s:= FieldValues[1];
  d:= FieldValues[2];
end;
end;
```

Доступ к подполям ADT возможен и на уровне привычного свойства `Value`, но в таком случае значения предаются в виде массива:

```
Table1.FieldByName('Field1').Value := VarArrayOf([10, Edit1.Text, d]);
```

Композитное поле также умеет представлять данные в форматах `AsString` и `AsVariant`.

Следующий пример (листинг 19.19) демонстрирует способ обнуления всех дочерних полей массива `TArrayField`.

Листинг 19.19. Очистка массива

```
var IntegerArrayField : TArrayField;
    I : Word;
begin
For I:=0 to IntegerArrayField.Size-1 do
            IntegerArrayField.Fields[I].Value:=0;
end;
```

Вообще поля типа TArrayField — большие "затейники" по организации доступа к своим данным. В листинге 19.20 приведены три строки кода, делающие одну и ту же работу (запись содержимого нулевого элемента массива полей в переменную X) разными способами.

Листинг 19.20. Варианты обращения к нулевому элементу массива

```
X:=TArrayField(Table1.FieldByName('ArrayField'))[0];
X:=TArrayField(Table1.FieldByName('ArrayField')).Fields[0].AsInteger;
X:=TArrayField(Table1.FieldByName('ArrayField')).FieldValues[0];
```

Теперь несколько замечаний по выводу на экран компьютера содержимого поля ADT и поля-массива. Стандартные элементы управления, такие как TDBEdit и TDBGrid, вполне способны отобразить содержимое полей такого рода. Есть некоторые особенности в порядке отображения этих полей в сетке TDBGrid, зависящие от состояния свойства ObjectView набора данных, к которому подключена сетка. Разработчики Delphi рекомендуют устанавливать данное свойство в true. В этом случае дочерние поля разворачиваются/сворачиваются при щелчке курсором по заголовку колонки.

Если набор данных получен от сервера Oracle, то в свойстве

```
property ObjectType: string;
```

вы прочитаете классификатор объекта.

Еще на уровне класса TField предусмотрен ряд свойств и методов для составных полей. Если поле является частью более сложного объекта, то метод

```
function HasParent: Boolean; override;
```

вернет значение true.

Ссылка на родительское поле окажется в свойстве

```
property ParentField: TObjectField;
```

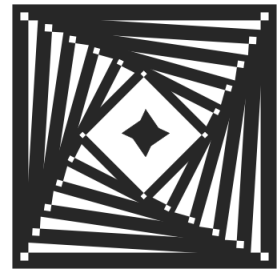
Комбинацию имени дочернего и родительского полей вы обнаружите в свойстве

```
property FullName : string;
```

Резюме

Без всякого преувеличения классы TDataSet и TField можно назвать фундаментом для всех компонентов доступа к данным. Основная особенность перечисленных классов в их независимости от механизма доступа к данным. Благодаря этому они и стали отправной точкой для обширного спектра элементов управления, предназначенных для обслуживания технологий dbExpress, ADO и Interbase.

Ознакомившись с общими чертами всех компонентов доступа к данным, в следующих главах книги мы изучим специфичные особенности ключевых элементов управления, применяемых в приложениях БД.



Глава 20

Вспомогательные классы набора данных

В этой главе мы продолжим изучать классы, оказывающее прямое влияние на функционирование компонентов-наборов данных. В число наших новых знакомых войдут:

- Коллекция параметров `TParams` и отдельный параметр `TParam`. Экземпляры этих классов предназначены для передачи значений в инструкции на языке SQL и в параметры хранимых процедур.
- Коллекция описаний полей таблицы `TFieldDefs` и элемент этой коллекции — определение поля `TFieldDef`. Благодаря перечисленным классам мы приобретаем возможность получать подробную информацию о структуре таблицы и, в ряде случаев, создавать таблицы без явного применения SQL.
- Коллекция описаний индексов таблицы `TindexDef` и элемент коллекции — определение отдельного индекса таблицы `TIndexDef`.
- Все указанные классы инкапсулируются в состав компонентов-наборов данных.

Коллекция *TParams* и динамический SQL

Основным средством общения между клиентским приложением и реляционной СУБД выступает структурированный язык запросов SQL. С помощью инструкций структурированного языка запроса программист приобретает возможность передавать в БД и получать из нее данные, необходимые для работы приложения. Разработчики Delphi предусмотрели несколько способов отправки инструкций SQL. В простейшем случае в адрес БД отправляется статичная команда SQL, например, добавляющая в таблицу `PUBLISHERS` новое издательство:

```
INSERT INTO PUBLISHERS (PUBLISHER) VALUES ("БХВ")
```

Предложенная инструкция заносится в свойство `SQL` компонента-запроса во время визуального проектирования и запускается однократно. Несложно догадаться, что это чересчур прямолинейное решение, оно обычно неприемлемо, в особен-

ности, если логика работы приложения предполагает активное редактирование строк таблиц БД.

Выходом из сложившегося положения может стать безусловный отказ от статических конструкций и переориентация компонентов-запросов на применение приемов работы с динамическим (легко модифицируемым во время выполнения приложения) SQL. В таком случае содержание SQL-запроса формируется во время выполнения программы и может перенастраиваться сколько угодно раз. Так, для обеспечения добавления записей в таблицу издательств более опытный программист может воспользоваться листингом 20.1.

Листинг 20.1. Пример создания динамической инструкции SQL

```
With SQLQuery1 do
begin
  SQL.Clear;       //очистка предыдущей инструкции SQL
  SQL.Add('INSERT INTO PUBLISHERS'); //формируем новую инструкцию
  SQL.Add('(PUBLISHER) VALUES ('+Edit1.Text+')');
  Prepared:=True; //подготовка запроса
  ExecSQL;         //отправка новой инструкции на выполнение
end;
```

Безусловно, это более гибкое решение. Теперь значения полей для записи, добавляемой к таблице, забираются из соответствующих элементов управления (в нашем примере из строки ввода `Edit1`), далее вызывается метод `ExecSQL()`, отправляющий SQL-инструкцию в БД. Но и такой подход не лишен недостатков, основной из них — необходимость многократного выполнения одних и тех же операций очистки предыдущей инструкции SQL и заполнения очередной команды с новым названием издательства.

Для устранения указанного недостатка создатели Delphi научили работающие с SQL компоненты-наборы данных (запросы `T…Query` и хранимые процедуры `T…StoredProc`) передавать в инструкцию SQL внешние параметры без необходимости модификации текста всего запроса. В проектах, опирающихся на применение параметров, текст SQL-запроса формируется единожды. Это может произойти во время проектирования или, как в очередном примере (листинг 20.2), во время выполнения проекта.

Листинг 20.2. Динамическая инструкция SQL с параметрами

```
With SQLQuery1 do
begin
  SQL.Clear; //очистка свойства
  SQL.Add('INSERT INTO PUBLISHERS'); //формируем инструкцию
  SQL.Add('(PUBLISHER) VALUES :PUBLISHER');//объявляем параметр
  Prepared:=True; //подготовка запроса
end;
```

Обратите внимание на основное отличие нового варианта от листинга 20.1. В строке SQL `(PUBLISHER) VALUES :PUBLISHER` вместо ссылки на элемент управления, в котором находится очередное название издательства, мы передаем имя параметра, начинающееся с символа двоеточия. Теперь для отправки в БД команды на добавление новой строки достаточно следующих строк кода (листинг 20.3).

Листинг 20.3. Заполнение параметров

```
SQLQuery1.Params.ParamByName('PUBLISHER'):=Edit1.Text;
SQLQuery1.ExecSQL;
```

Таким образом, применение параметров в SQL-запросе существенно упрощает процесс разработки приложения и, что очень важно, ускоряет отправку и выполнение запроса на сервере.

После того как мы познакомились с практической стороной задачи, углубим наши теоретические познания о параметрах. Благодаря листингу 20.3 нам уже известно, что при возникновении необходимости передать в запрос SQL или в хранимую процедуру внешние данные сразу следует воспользоваться помощью специализированного свойства

```
property Params: TParams;
```

Это коллекция, специализирующаяся на обслуживании параметров запросов и процедур. Свойство доступно не только во время выполнения приложения, но и в период проектирования.

Отдельный параметр описывается классом `TParam`. Для того чтобы обратиться к параметру, нам понадобится свойство

```
property Items[Index: Integer]: TParam;
```

в которое следует передать индекс параметра.

Для автоматического заполнения перечня параметров можно вызвать метод

```
function ParseSQL(SQL: string; DoCreate: Boolean): string;
```

Он проанализирует текст запроса SQL, и если аргумент `DoCreate` принимает значение `true`, создаст список параметров.

> ***ВНИМАНИЕ!***
>
> При подключении компонента к описанной на сервере БД хранимой процедуре создание параметров и настройка их свойств осуществляется автоматически в момент щелчка по свойству `Parameters` компонента.

Для обращения к параметру из текста программы можно воспользоваться методом

```
function ParamByName(const Value: string): TParam;
```

Функция возвратит параметр по его имени.

Для доступа к значению параметра следует вызвать свойство коллекции

```
property ParamValues [const ParamName: string]: Variant;
```

Если вам известен порядковый номер параметра, то для присвоения этому параметру значения достаточно повторить следующую строку кода:

```
SQLStoredProc1.Params[0].Value:='EXB';
```

Число параметров в коллекции можно уточнить с помощью свойства

```
property Count: Integer;//только для чтения
```

Для очистки всей коллекции предназначена процедура

```
procedure Clear;
```

Для удаления определенного параметра используем метод

```
procedure Delete(Index: Integer);
```

и указываем порядковый номер ненужного параметра.

Для добавления в коллекцию нового параметра следует обратиться к функции

```
function AddParameter: TParameter;
```

Метод создает новый экземпляр класса `TParameter` и помещает его в коллекцию.

Параметр *TParam*

Параметр обладает именем:

```
property Name: WideString;
```

Параметр характеризуется типом данных:

```
property DataType: TDataType;
```

Для параметра строкового типа следует указать размерность (число символов):

```
property Size: Integer;
```

Тип параметра также нуждается в настройке:

```
property ParamType: TParamType;
```

```
TParamType = (ptUnknown,     //неизвестен
              ptInput,       //входной параметр
              ptOutput,      //выходной параметр
              ptInputOutput, //универсальный параметр
              ptResult);     //параметр для возврата результата
```

Если речь идет о параметрах числового типа, то возможно, придется установить значения для свойств:

```
property Precision: Byte;    //всего знаков
property NumericScale: Byte; //знаков после запятой
```

Свойство `Precision` определяет максимальное число значащих цифр, описывающих все значение (до и после запятой), а свойство `NumericScale` уточнит точность данных (максимальное число значащих цифр после запятой).

Строки кода из листинга 20.4 добавляют новый параметр в список параметров хранимой процедуры `SQLStoredProc1`.

Листинг 20.4. Создание параметра в коде программы

```
with SQLStoredProc1.AddParameter do
begin
  Name:='@WRITER'; //имя параметра
```

```
  DataType := ftString; //тип данных параметра
  Size:=20;             //размер
  ParamType := ptInput; //входной параметр
end;
```

Для чтения значения, хранящегося в параметре, или передачи в него нового значения применяют свойство

```
property Value: Variant;
```

Например, так:

```
SQLStoredProc1.Params.ParamByName('@WRITER').Value:= 'Шишков В.Я.';
```

Так как свойство `Value` возвращает универсальный тип данных `Variant`, то зачастую целесообразно явное приведение типов. Для этого у параметра имеются свойства `AsString`, `AsInteger` и т. д. Большинство названий свойств приведения типов `TParam` аналогичны именам соответствующих свойств поля `TField` (см. табл. 19.3), поэтому мы не станем повторяться.

```
S:= SQLStoredProc1.Params.ParamByName('@WRITER').AsString;
```

Если параметр возвращает значение, то не исключено, что в нем окажется неопределенность `NULL`. При попытке присвоить неопределенное значение `NULL` любой переменной немедленно сгенерируется исключительная ситуация, поэтому, прежде чем считать данные, нужно проверять, что, собственно, находится в параметре (листинг 20.5).

Листинг 20.5. Чтение данных из параметра с контролем неопределенного значения

```
var S:string;
begin
if SQLStoredProc1.Params.ParamValues['PUBLISHER']<>NULL then
   S:=SQLStoredProc1.Params.ParamValues['PUBLISHER']
   else
   S:='';
end;
```

При желании можно спросить параметр напрямую, есть ли в нем значение или он пуст. Для этого предназначено свойство

```
property IsNull: Boolean;
```

Очистку содержимого параметра осуществляет процедура

```
procedure Clear;
```

Если параметр специализируется на обслуживании больших бинарных данных (BLOB), то для передачи в параметр данных целесообразно воспользоваться методами:

```
procedure LoadFromFile(const FileName: string; DataType: TDataType);
procedure LoadFromStream(Stream: TStream; DataType: TDataType);
```

Первый из них забирает данные из файла, а второй — из потока.

Внимание!

Пока не упоминавшиеся на этих страницах компоненты `TADODataSet`, `TADOQuery` и `TADOStoredPorc` также умеют работать с параметрами. Но так как для доступа к данным они используют не разработки Embarcadero Technologies, а созданную в стенах корпорации Microsoft технологию ADO (ActiveX Data Objects), то организация работы с параметрами у них имеет незначительные отличия. В частности, коллекция параметров компонента ADO описывается классом `TParameters`, а отдельный параметр — классом `TParametr`.

Параметры хорошо подходят для поиска данных в таблицах средствами SQL. Допустим, что в нашем распоряжении есть компонент-запрос `ADOQuery1`, в котором хранится инструкция:

```
SELECT TITLE FROM BOOKS WHERE TITLE LIKE :TITLE
```

Инструкция позволяет осуществлять поиск книги по названию с помощью шаблона `LIKE`. Как вы заметили, инструкция содержит единственный входной параметр — `:TITLE`.

Разместите на форме обычную строку ввода `Edit1:TEdit` и найдите обработчик события `OnKeyUp()`, это событие генерируется после того, как пользователь ввел очередной символ названия книги и уже отпускает кнопку на клавиатуре. В составе параметров события `OnKeyUp()` есть аргумент `Key`, хранящий числовой код нажатой клавиши (листинг 20.6).

Листинг 20.6. Использование параметра в шаблоне LIKE

```
procedure TfrmEditDeliveries.Edit1KeyUp(Sender: TObject;
                                  var Key: Word; Shift: TShiftState);
var pos : word;
begin
if KEY>47 then {нас интересуют только клавиши с кодом больше 47}
   begin
   Edit1.ClearSelection;     //снимаем выделение
   Pos:=Length(Edit1.Text); //запоминаем число введенных знаков

 with ADOQuery1 do
 begin
  If Active then Close;
  Parameters.ParamValues['Title']:=Trim(Edit1.Text)+'%';
  {<%> указывает, что далее могут следовать любые символы}
  Open;

  if FieldByName('Title').IsNull=False then
     begin //если нашли совпадение, помогаем пользователю
       Edit1.Text:=FieldByName('Title').AsString;
       {выделяем добавленную часть строки}
```

```
        Edit1.SelStart:=Pos;
        Edit1.SelLength:=Length(Edit1.Text)-pos;
      else MessageBeep(0); //названий, соответствующих шаблону, нет
    end;
  end;
end;
```

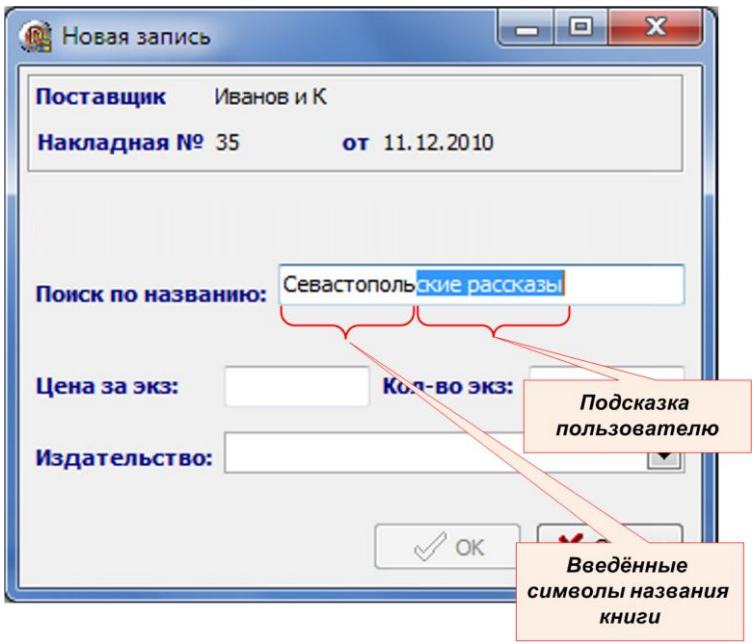


Рис. 20.1. Реализация подсказки пользователю при вводе названия

Основное достоинство приведенного примера в том, что он не просто находит название книги, соответствующее шаблону `LIKE`, но и помогает пользователю закончить ввод, подставляя наиболее подходящее название в поле поиска (рис. 20.1).

Описание структуры таблицы и ее индексов

Каждая система управления базами данных предоставляет разработчику средства проектирования структуры таблиц. В минимальной нотации для этих целей задействованы команды языка SQL `CREATE TABLE` и `ALTER TABLE`. Опытный администратор БД без особого труда напишет необходимые строки SQL-инструкции в обычном консольном окне. Более доброжелательные СУБД обладают развитым пользовательским интерфейсом, в них есть удобные и интуитивно понятные сред-

ства визуального проектирования таблиц. Современная среда Delphi XE, а точнее ряд ее компонентов, также обладает средствами, позволяющими создавать реляционные таблицы. Но на этот раз основным инструментом проектирования выступает язык Delphi.

Процесс создания новой таблицы можно условно разбить на три этапа.

Во-первых, в свойстве

```
property FieldDefs: TFieldDefs;
```

описываются физические параметры всех полей будущей таблицы.

Во-вторых, при необходимости в свойстве

```
property IndexDefs: TIndexDefs;
```

программист определяет характеристики вторичных индексов новой таблицы.

И, в-третьих, для создания таблицы на основе структур, описанных в свойствах `FieldDefs` и `IndexDefs` компонента `TClientDataSet`, вызывается метод

```
procedure CreateDataSet;
```

В результате в памяти компонента создается набор данных с заданными нами характеристиками.

Познакомимся со свойствами `FieldDefs` и `IndexDefs` подробнее. Технически названные свойства представляют собой коллекции, построенные на основе общего предка — класса `TDefCollection` (рис. 20.2). Смысл существования `TDefCollection` сводится к умению хранить и обслуживать специальные объекты — потомки класса `TNamedItem` (рис. 20.3). В свою очередь, экземпляр `TNamedItem` предоставляет нам способ описания каких-то, если хотите, инженерных характеристик будущего объекта, конкретно в нашем случае идет речь о "чертеже" поля таблицы.

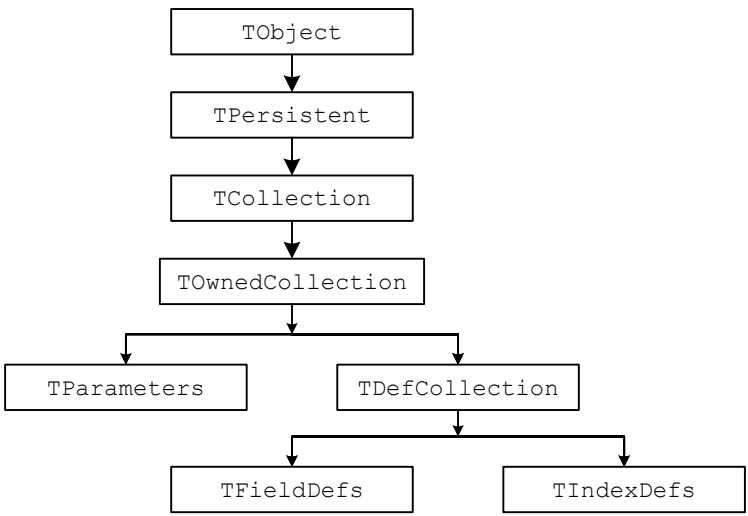


Рис. 20.2. Иерархия коллекций, инкапсулированных в наборах данных

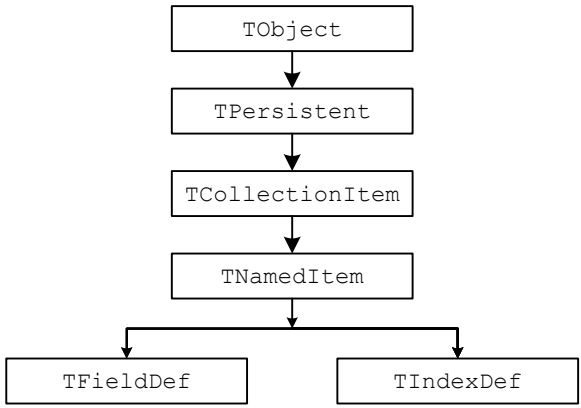


Рис. 20.3. Описание поля `TFieldDef` и индекса `TIndexDef`

Коллекция полей структуры таблицы *TFieldDefs*

Как мы уже говорили, свойство набора данных `FieldDefs` инкапсулирует класс `TFieldDefs` — коллекцию "чертежей" будущих полей. Имеющиеся в составе коллекции свойства и методы позволяют:

- получить доступ к характеристикам любого поля коллекции;
- добавлять описание поля к списку полей будущей таблицы;
- передавать весь список полей в другой набор данных.

Доступ к любому из определенных в коллекции полей обеспечивает свойство

```
property Items[Index: Integer]: TFieldDef;
```

Как всегда, отсчет начинается с поля с нулевым индексом и заканчивается полем с порядковым номером `Count-1`. Иногда для обнаружения в коллекции поля по его имени удобнее обратиться к методу

```
function Find(const Name: String): TFieldDef;
```

Функция самостоятельно переберет все элементы коллекции и при удачном стечении обстоятельств вернет экземпляр описания поля `TFieldDef`.

Для обновления коллекции допускается вызов метода

```
procedure Update;
```

Обращение к процедуре может пригодиться после изменения физической структуры, подключенной к набору данных таблицы.

Удаление элемента коллекции осуществит процедура

```
procedure Delete(Index: Integer);
```

Полную очистку коллекции обеспечит метод

```
procedure Clear;
```

В случае, когда идет речь о составных полях (таких, как `TADTField`), ссылка на родительское поле окажется в свойстве

```
property ParentDef: TFieldDef; //только для чтения
```

Если же поля не составные, то в свойстве мы обнаружим неопределенный указатель `nil`.

Пополняет коллекцию чертежом нового поля метод

```
function AddFieldDef: TFieldDef;
```

Описание характеристик отдельного экземпляра поля предоставляется классом `TFieldDef`.

Если логика программы требует сохранения описаний структуры таблицы и ее индексов в коде модуля данных или формы, на которой расположена таблица `TClientDataset` (`TIBTable`, `TTable`), то установите в `true` свойство

```
property StoreDefs: Boolean; //по умолчанию flase
```

Такая услуга может пригодиться при создании таблицы с помощью метода `CreateTable()`, имеющегося в распоряжении компонентов `TTbale` и `TIBTable`, или метода `CreateDataSet()`, предусмотренного у компонентов `TSQLTable` и `TClientDataSet`.

После соединения с файлом таблицы свойство `FieldDefs` компонента-набора данных автоматически заполняется описанием физических полей только что открытой таблицы. Эту особенность коллекции полей `FieldDefs` часто используют для создания новой таблицы, повторяющей структуру исходной (листинг 20.7).

Листинг 20.7. Создание копии таблицы

```
with IBTable1 do
begin
  Database:=IBDataBase1;
  TableName:='WRITERS';
  Open;
end;
with IBTable2 do
begin
  Database:=IBDataBase1;
  FieldDefs.Assign(IBTable1.FieldDefs);
  IndexDefs.Assign(IBTable1.IndexDefs);
  TableName:='COPY_WRITERS';
  if Exists=False then CreateTable;
end;
```

Компонент `IBTable1` соединяется с файлом таблицы, структуру которой мы планируем продублировать. После того как `IBTable1` откроет исходный набор данных (и заполнит свою коллекцию описаний полей `FieldDefs`), передаем ссылку на эту коллекцию в компонент `IBTable2` и методом `CreateTable()` создаем "генетический клон" исходной таблицы.

ВНИМАНИЕ!

Создавать таблицу на основе описания `TFieldDefs` умеют компоненты: `TTable`, `TIBTable` (для этой цели задействуется метод `CreateTable`) и `TClientDataSet` (метод `CreateDataSet`).

Определение поля, класс *TFieldDef*

Для описания технических характеристик поля в Delphi предусмотрен специальный класс — `TFieldDef`. В отличие от нашего старого знакомого, объекта `TField` (инкапсулирующего столбец таблицы и предоставляющего доступ к хранящимся в нем данным), экземпляр класса `TFieldDef` (от англ. *Field Definition*) представляет собой чертеж поля, на основании которого можно создать столбец таблицы. Включаемая в чертеж информация приведена в табл. 20.1.

***Таблица 20.1.** Основные свойства класса `TFieldDef`*

Свойство	Описание
`property Name : String;`	Физическое имя поля таблицы
`property DisplayName : String;`	Используемый при выводе на экран заголовок поля
`property FieldClass: TFieldClass;` `type TFieldClass = class of TField;`	Свойство (только для чтения), информирующее о классе поля
`property DataType: TFieldType;`	Тип хранящихся данных
`property Precision: Integer;`	Точность хранения данных, определяет число значащих цифр
`property Attributes: TFieldAttributes;` `type TFieldAttribute = (faHiddenCol, faReadonly, faRequired, faLink, faUnNamed, faFixed);`	Физические атрибуты поля. В самом простом случае могут не определяться
`property Required: Boolean;`	Признак обязательности, значение `true` говорит о том, что поле не допускает неопределенных значений `NULL`
`property InternalCalcField: Boolean;` `{только для чтения}`	Значение `true` свидетельствует о том, что поле вычисляемое
`property FieldNo: Integer;`	Физический номер поля в таблице
`function HasChildDefs: Boolean;` `{только для чтения}`	Метод возвращает `true` в случае, когда поле владеет дочерними элементами
`property ChildDefs: TFieldDefs;`	Определяет объект `ChildDefs`, содержащий массив дочерних полей

Таблица 20.1 *(окончание)*

Свойство	Описание
property ParentDef: TFieldDef;	Если рассматриваемое описание поля представляет собой элемент составного поля, то ParentDef хранит ссылку на родительское определение поля

Коллекция структуры индексов таблицы, класс *TIndexDefs*

Коллекция TIndexDefs предназначена для хранения метаданных о вторичных индексах таблицы — TIndexDef. В компонентах TIBTable и TClientDataSet доступ к описанию индексов обеспечивает свойство IndexDefs. Информация об имеющихся у таблицы индексах автоматически заполняется в момент подключения компонента к набору данных или ее описывает программист перед созданием экземпляра новой таблицы. Вся техническая информация об отдельных индексах находится в массиве:

```
property Items[Index: Integer]: TIndexDef;
```

Каждый элемент списка представлен классом TIndexDef — описанием индекса. При необходимости обновить данные об индексах, хранящихся в свойстве IndexDefs, программист вызывает метод

```
procedure Update; reintroduce;
```

Для передачи метаданных об индексах между компонентами класса TClientDataSet обращаются к методу

```
procedure Assign(ASource: TPersistent);
```

Процедура проверяет корректность параметра ASource и копирует описание индексов.

Добавление к коллекции описания нового индекса осуществляется вызовом метода

```
function AddIndexDef: TIndexDef;
```

или

```
procedure Add(const Name, Fields: String; Options: TIndexOptions);
```

Выбрав метод AddIndexDef(), мы создаем новое описание, а затем определяем его свойства так, как показано в листинге 20.8.

Листинг 20.8. Описание индекса

```
var IndexDef : TIndexDef;
…
with IBTable1.IndexDefs do
```

```
begin
    IndexDef:=AddIndexDef;
    IndexDef.Name:='indDemo';
    IndexDef.Fields:='Field1;Field2';
    IndexDef.Options:=[ixUnique];
end;
```

Метод `Add()` делает то же самое, но все параметры создаваемого им описания индекса передаются сразу, в момент вызова метода. Имя индекса передается в параметр `Name`, список полей — в параметр `Fields` и опции — в `Options`:

```
IBTable1.IndexDefs.Add('indDemo','Field1;Field2',[ixUnique]);
```

Группа методов специализируется на поиске индекса в списке. Например, чтобы обнаружить описание вторичного индекса с именем `Name`, обращаемся к функции

```
function Find(const Name: String): TIndexDef;
```

Еще один метод возвратит описание индекса, начинающегося с имени поля (или полей, если речь идет о составном индексе), переданного в параметре `Fields`:

```
function FindIndexForFields(const Fields : String): TIndexDef;
```

При обращении к функции `FindIndexForFields()` нужно помнить, что ее работоспособность сохраняется только при поиске индексов, упорядочивающих данные по возрастанию. Если же речь идет о вторичном индексе, сортирующем записи по убыванию значений, то целесообразно воспользоваться методом

```
function GetIndexForFields(const Fields: String;
                           CaseInsensitive: Boolean): TIndexDef;
```

Здесь (как и в случае с функцией `FindIndexForFields`) в первый параметр метода передаются разделенные точкой с запятой имена полей, входящих в состав индекса. Кроме того, установленный в `true` параметр `CaseInsensitive` указывает, что при поиске описания индекса будут учитываться только нечувствительные к регистру символов индексы.

Замечание

При работе с методами `FindIndexForFields()` и `GetIndexForFields()` необходимо помнить о том, что при отсутствии точного соответствия полей, входящих в индекс, и имен полей (переданных в параметр `Fields`) методы возвратят индекс, наиболее похожий на описание.

Определение индекса, класс *TIndexDef*

Отдельный вторичный индекс описывают в рамках экземпляра класса `TIndexDef`. Эксплуатируя заложенные в классе свойства и методы, программист приобретает возможность:

- назначать имя для будущего индекса;
- указывать входящие в индекс поля;

- определять атрибуты и характеристики индекса;
- строить индексные выражения для таблиц формата dBase.

Основные характеристики класса приведены в табл. 20.2.

Таблица 20.2. *Основные свойства* `TIndexDef`

Свойство	Описание
`property Fields: String;`	Определяет входящий в индекс перечень полей. Имена полей разделяются точкой с запятой. Последовательность полей в строке описания назначает порядок следования полей в индексе таблицы
`property CaseInsFields: String;`	При необходимости определяет, какие из полей индекса нечувствительны к регистру символов. Имена полей разделяются точкой с запятой
`property DescFields: String;`	Перечень полей обратной сортировки
`property Expression: String;`	Содержит описание выражения для индекса таблицы dBase
`property FieldExpression: String;`	Доступ к списку имен поля либо индексного выражения
`property GroupingLevel: Integer;`	Указывает обеспечиваемую индексом глубину группировки

`property Options: TIndexOptions;`
`type`
`TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive, ixExpression, ixNonMaintained);`
`TIndexOptions = set of TIndexOption;`
	Опции индекса. Значение `ixPrimary` указывает на то, что описываемый индекс будет выступать в роли первичного ключа таблицы. Контроль за уникальностью значений обеспечит `ixUnique`. Признак `ixNonMaintained` определяет, что индекс не обновляется даже после изменения данных в таблице. Отказ от учета регистра символов при сортировке данных в таблице обеспечит ключ `ixCaseInsensitive`. Для таблиц формата dBase состояние `ixExpression` свидетельствует о том, что индекс построен на основе выражения. Значение `ixDescending` укажет на обратный порядок сортировки данных

`property Source: String;`	В тех случаях, когда таблица `dBase` обслуживается индексом MDX, свойство `Source` определяет имя индексного файла

Пример создания таблицы

После такого объема теории, посвященной процессу создания новой таблицы, стоит попробовать свои силы на практике. В листинге 20.9 приведен код примера,

создающего в корне диска C: таблицу "Writer". В состав таблицы входит три поля и два вторичных индекса.

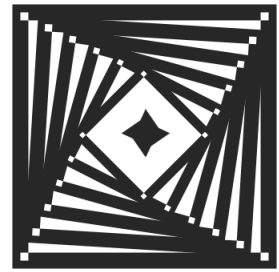
Листинг 20.9. Создание таблицы

```
with ClientDataSet1 do
begin
  Active := False;
  {ОПИСЫВАЕМ ПОЛЯ ТАБЛИЦЫ}
    with FieldDefs do
    begin
      Clear;                    //очищаем структуру
      with AddFieldDef do      //добавляем к коллекции полей новое поле
       begin
        Name := 'Wrt_Key';      //имя поля
        DataType := ftAutoInc; //признак автоинкрементного типа поля
      end;
      with AddFieldDef do
      begin
        Name := 'Surname';      //имя поля
        DataType := ftString;  //текстовое поле
        Size := 30;             //размер 30 символов
        Required := True;       //признак обязательности заполнения
      end;
      with AddFieldDef do
      begin
        Name := 'Born';       //имя поля
        DataType := ftDate;  //поле для хранения даты
      end;
    end;
{ОПИСЫВАЕМ ИНДЕКСЫ ТАБЛИЦЫ}
    with IndexDefs do
     begin
      Clear;                //очищаем структуру
      with AddIndexDef do   //добавляем описание 1-го индекса
      begin
        Name := 'indPrimaryKey'; //имя индекса
        Fields := 'Wrt_Key';     //поля в индексе
        Options := [ixPrimary];  //признак первичного ключа
      end;
```

```
    with AddIndexDef do      //добавляем описание 2-го индекса
    begin
      Name := 'indSurname'; //имя индекса
      Fields := 'Surname';  //поля в индексе
      Options := [ixCaseInsensitive]; //чувствителен к регистру
    end;
  end;
  CreateDataset;               //создаем таблицу в памяти компонента
  SaveToFile('c:\writer.cds',dfBinary); //сохраняем таблицу на диск
end;
```

Резюме

В этой главе мы познакомились с дополнительными возможностями наборов данных, приобретенными благодаря инкапсуляции в компоненты ряда вспомогательных классов. Так, компоненты-запросы и хранимые процедуры для отправки (получения) параметров в БД пользуются услугами классов `TParams` и `TParam`. Компоненты-таблицы описывают структуру физических таблиц с помощью классов `TFieldDefs` и `TFieldDef`, описание индексов обеспечивают классы `TIndexDefs` и `TIndexDef`.



Глава 21

Клиентский набор данных *TClientDataSet*

На страничке **Data Access** палитры компонентов Delphi разместился на первый взгляд ничем не примечательный компонент `TClientDataSet`. Видимо, в силу своей скромности он не часто встречается в проектах БД, а зря! Наш новый знакомый являет собой весьма остроумный способ работы с клиентскими наборами данных, что делает честь авторам идеи. Изюминка в том, что компонент представляет собой полнофункциональный программный модуль, способный обрабатывать данные даже в отрыве от СУБД, с этой целью он размещает полученные из БД записи в своем локальном буфере.

Хотя основное назначение клиентского набора данных сводится к построению клиентских приложений в многоуровневых клиент-серверных БД, но благодаря уникальной автономности на базе `TClientDataSet` можно создавать вполне эффективные, и самое главное самодостаточные настольные проекты БД.

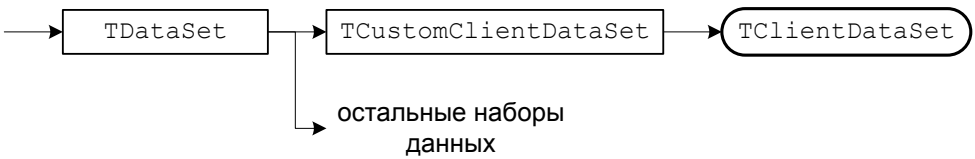


Рис. 21.1. Место класса `TClientDataSet` в иерархии VCL

Как и принято у всех компонентов, обслуживающих таблицы БД, наш новый знакомый построен на фундаменте абстрактного класса `TDataSet` (рис. 21.1). Поэтому клиентский набор данных `TClientDataSet` впитал в себя все родовые черты, присущие универсальному набору данных. Напомню, что это самый общий функционал, связанный с редактированием данных; перемещением по данным; доступом к отдельному полю; поиском и фильтрацией данных, оценкой текущего состояния набора (табл. 21.1).

Таблица 21.1. *Ключевые свойства и методы, унаследованные от* `TDataSet`

Методы и свойства	Описание
`Active`, `Open()`, `Close()`	Открытие и закрытие набора данных
`First()`, `Prior()`, `Next()`, `Last()`, `MoveBy()`	Перемещение курсора по набору данных
`AppendRecord()`, `InsertRecord()`, `Insert()`, `Append()`, `Delete()`, `Edit()`, `Cancel()`	Редактирование данных в локальном кэше компонента
`Fields()`, `FieldByName()`	Доступ к отдельному полю таблицы
`Filter`, `Filtered`, `FilterOptions`	Фильтрация данных
`Locate()`, `Lookup()`	Поиск записи
`State`	Состояние набора данных

В отличие от абстрактного универсального набора данных `TDataSet`, его совсем не дальний родственник, класс `TClientDataSet` — полноценный компонент, специализирующийся на решении двух задач:

- Обслуживание небольших однопользовательских БД, хранящихся в обычных файлах (flat-file database). Причем компонент не нуждается в услугах каких-либо внешних драйверов и провайдеров. Весь необходимый функционал по обслуживанию таблиц-файлов инкапсулирован в классе, так что, если хотите, этот компонент сам по себе СУБД. Благодаря тому, что клиентский набор `TClientDataSet` способен кэшировать данные, построенные на его основе приложения, сохраняют работоспособность даже в отрыве от сервера БД. Это так называемая *модель портфеля* (briefcase model), когда принадлежащие серверу данные временно сохраняются в локальном буфере и пользователь продолжает работать с записями без соединения с серверной частью приложения.
- Обслуживание клиентских приложений многоуровневых БД. Это разновидность распределенных приложений, в которых доступ клиента к БД осуществляется через посредника — сервер приложений (см. *главу 31*).

Проекты БД, основанные на файлах

Если вы внимательно читали первые главы книги, то не исключаю, что название подраздела вас несколько насторожило. "Как же так?" — спросите вы. — Все беды первых баз данных начались именно с того, что они разрабатывались на фундаменте систем, основанных на файлах! И вдруг мы вновь наступаем на грабли!"

Спасибо за замечание. Отвечу, что у медали всегда есть обратная сторона. Несмотря на существенные недостатки систем, основанных на файлах, они вполне сносно справляются с обработкой и хранением сравнительно небольших объемов данных в настольных БД, рассчитанных на одного пользователя. Проект на базе `TClientDataSet` не потребует от нас развертывания на компьютере дорогостоящего программного обеспечения с СУБД различных производителей.

Чтобы с первых секунд понять заложенную в компонент идею, сразу ринемся в бой — создадим новый проект и разместим на главной форме компонент `ClientDataSet1`. В данный момент нас интересует его свойство `FieldDefs`, предназначенное для определения состава полей и хранимых в них данных. Воспользуемся встроенным редактором полей (он вызывается щелчком по свойству `FieldDefs`) и опишем структуру будущей таблицы. Наша таблица предназначается для обслуживания контактных данных ваших знакомых, и в ее составе будет всего шесть полей (рис. 21.2).

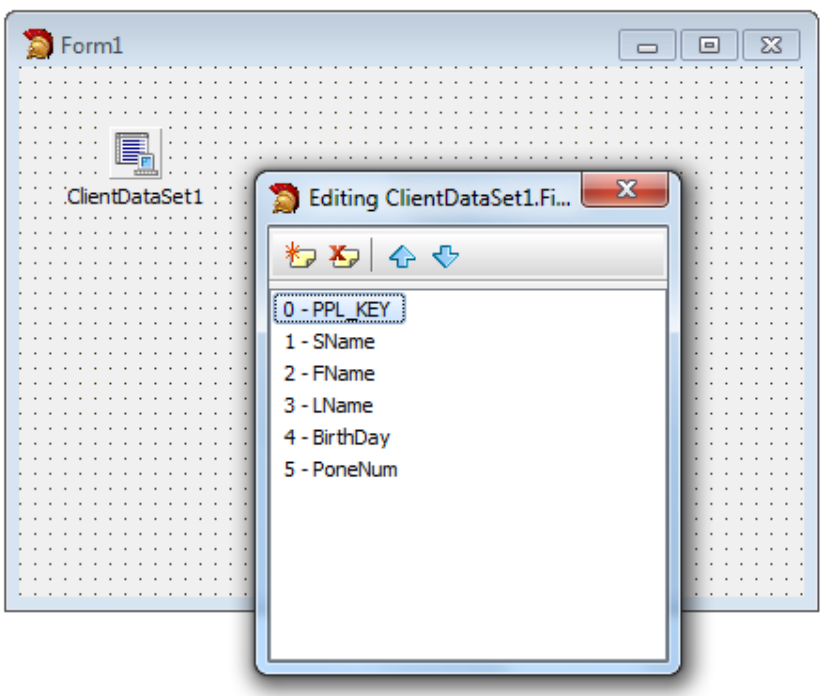


Рис. 21.2. Редактор настройки полей компонента `ClientDataSet1`

Для добавления нового поля проще всего обратиться за помощью к контекстному меню редактора полей, для этого щелкаем правой кнопкой мыши по редактору и выбираем пункт меню **Add** (Добавить). В ответ на каждый щелчок в редакторе будет появляться новая строка с "дежурными" названиями полей `ClientDataSet1Field1`, `ClientDataSet1Field2` и т. д. Вполне естественно, что та-

кие имена полей нас не устроят, поэтому обращаемся к инспектору объектов и вносим исправления в названия и тип данных:

- поле первичного ключа — `Name:='PPL_KEY'; DataType:=ftAutoInc;`
- поле фамилии — `Name:='SName'; DataType:=ftString; Size:=20;`
- поле имени — `Name:='FName'; DataType:=ftString; Size:=20;`
- поле отчества — `Name:='LName'; DataType:=ftString; Size:=20;`
- поле даты рождения — `Name:='BirthDay'; DataType:=ftDate;`
- поле номера телефона — `Name:='PoneNum'; DataType:=ftString; Size:=11;`

Сохраните проект в отдельной папке. Теперь наступает самое интересное. Вызовите всплывающее меню компонента и найдите в нем пункт **Create Dataset** (Создать набор данных). Щелчок по пункту меню активизирует деятельность встроенного метода:

```
procedure CreateDataSet;
```

Обращение к процедуре приведет к созданию в памяти компьютера нового пустого набора данных. Как вы уже поняли, обязательным условием, предваряющим вызов метода `CreateDataSet()`, должно быть заполнение структуры будущей таблицы в свойстве

```
property FieldDefs: TFieldDefs;
```

Описание структуры можно осуществить не только в период визуального проектирования, но и программным способом. Например, в момент создания экземпляра формы, которой принадлежит компонент `ClientDataSet1` (листинг 21.1).

Листинг 21.1. Создание таблицы в памяти компонента `TClientDataSet`

```
if ClientDataSet1.FieldDefs.Count=0 then
begin
  with ClientDataSet1.FieldDefs do
  begin
    with AddFieldDef do
    begin
      Name := 'PPL_KEY';
      DataType := ftAutoInc;
    end;
    with AddFieldDef do
    begin
      Name := 'SName';
      DataType := ftString;
      Size := 20;
    end;
    {Описание остальных полей набора данных}
    ClientDataSet1.CreateDataSet;
  end;
end;
```

Однако в нашем примере описание структуры будущей таблицы уже подготовлено и находится в памяти компонента. Следующим шагом должно стать сохранение этой структуры на жестком диске. Во время проектирования достаточно вновь вызвать контекстное меню компонента и выбрать в нем один из перечисленных далее пунктов меню:

- **Save to MyBase Xml Table…** (Сохранить таблицу в формате XML);
- **Save to MyBase Xml UTF8 Table…** (Сохранить таблицу в формате XML в кодировке UTF8);
- **Save to binary MyBase file…** (Сохранить таблицу в бинарный файл).

Выберем первый пункт и сохраним описание будущей таблицы в файл contacts.xml.

Загрузка и сохранение данных

Вновь ненадолго вернемся к теории. Операцию сохранения несложно вызвать из кода программы. Отметим, что компонент запомнит не только описание структуры таблицы, но и все содержащиеся в ней данные. В зависимости от пожеланий программиста сохранение производится в файл или в поток в памяти:

```
procedure SaveToFile(const FileName: string = '';
                     Format: TDataPacketFormat = dfBinary);

procedure SaveToStream(Stream: TStream;
                       Format: TDataPacketFormat = dfBinary); );
```

Второй параметр процедуры определяет формат сохраняемых данных (табл. 21.2).

Таблица 21.2. *Формат данных* `TDataPacketFormat`

Значение	Описание	Рекомендуемое расширение имени файла
`dfBinary`	Бинарный формат	*.cds
`dfXML`	Формат XML	*.xml
`dfXMLUTF8`	Формат XML, символы представляются в кодировке UTF8	*.xml

ЗАМЕЧАНИЕ

Бинарный формат CDS (видимо сокращение от ClientDataSet) разработан специально для обслуживания набора данных `TClientDataSet` и поэтому наиболее близок к формату внутреннего хранения данных в этом компоненте.

Несложно догадаться, что кроме методов сохранения в распоряжении клиентского набора данных имеется возможность загрузки структуры таблицы и данных:

```
procedure LoadFromFile(const FileName: string = '');

procedure LoadFromStream(Stream: TStream);
```

Примерно такого же результата можно достичь с помощью свойства
`property FileName: string;`

Но в последнем случае для активизации компонента после указания имени файла потребуется обращение к унаследованному от `TDataSet` методу `Open()` или свойству `Active`. Заметьте, что при загрузке данных нет необходимости явным образом указывать формат данных — с этой задачей компонент разберется самостоятельно.

Предлагаю немного попрактиковаться с операциями загрузки и сохранения данных. Добавим к начатому немного ранее проекту новый перечень компонентов:

- Источник данных `DataSource1`. Этот компонент вы найдете на той же странице палитры компонентов, где и расположен "герой" `TClientDataSet` — страница **Data Access**.
- Сетка `DBGrid1` (страница **Data Controls**).
- Диалоги загрузки `OpenDialog1` и сохранения `SaveDialog1` файлов. Названные компоненты ожидают нас на страничке **Dialogs**.
- Менеджер операций `ActionManager1` и меню операций `ActionMainMenuBar1` (страница **Additional**).

Воспользовавшись свойством `DataSet`, подключите источник данных `DataSource1` к клиентскому набору данных. Затем выберите сетку `DBGrid1` и с помощью свойства `DataSource` присоедините ее к источнику данных `DataSource1`.

Теперь настал звездный час компонентов-диалогов доступа к файлам. Нам следует настроить их свойства `Filter`, чтобы они открывали только файлы с подходящим нам расширением. Это можно сделать как с помощью инспектора объектов, так и в коде программы. Предлагаю выбрать второй путь и воспользоваться событием `OnCreate()`, создающим главную форму проекта (листинг 21.2).

Листинг 21.2. Инициализация диалогов открытия и сохранения файла

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 OpenDialog1.Filter:='Файл XML|*.xml|Двоичный файл|*.cds';
 OpenDialog1.DefaultExt:='xml';
 SaveDialog1.Filter:=OpenDialog1.Filter;
 SaveDialog1.DefaultExt:='xml';
end;
```

В событии, генерируемом в момент создания формы, описываются параметры фильтрации обоих диалогов с конкретизацией того, что расширением файлов по умолчанию следует считать xml.

Для загрузки данных и начала работы проекта выберем другое событие главной формы — `OnShow()`, код предложен в листинге 21.3.

Листинг 21.3. Подключение файла с данными к приложению

```
procedure TForm1.FormShow(Sender: TObject);
begin
if FileExists('contacts.xml') then
   begin
   ClientDataSet1.FileName:='contacts.xml';
   ClientDataSet1.Open
   end else
      if OpenDialog1.Execute then
          try
          ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
          except
          Application.MessageBox('Ошибка чтения файла!','Ошибка');
          end;
end;
```

В момент показа формы на экране подключаем к набору данных файл contracts.xml, если же по какой-то причине он не был найден, то вызываем диалог открытия файла `OpenDialog1` и предлагаем пользователю найти требуемый документ самостоятельно.

Дважды щелкните мышью по менеджеру операций `ActionManager1`. В результате на экране отобразится редактор команд. Пока список пуст, его заполнение начнем с определения двух операций:

- Первая команда (ей по умолчанию присвоится имя `Action1`) возьмет на себя ответственность за вызов диалога открытия файла, поэтому переименуйте команду в `acOpenFile`. Заголовку `Caption` присвойте название `'&Открыть'`. Свойство `Category='&Файл'`.
- Вторая команда позволит сохранять данные в другой файл. Назовем этот элемент управления `acSaveAs`, заголовок `Caption='&Сохранить как…'`. Свойство `Category='&Файл'`.

В списке **Categories** (Категории) менеджера операций выберите строку `&Файл` и, удерживая левую кнопку мыши в нажатом состоянии, перетащите выделенную категорию на расположенный в верхней части формы элемент управления `ActionMainMenuBar1`. В результате меню проекта приобретет свой первый элемент "Файл" (рис. 21.3).

В инспекторе объектов найдите действие `acOpenFile`, перейдите на страницу событий и дважды щелкните кнопкой мыши в строке `OnExecute()`. В появившейся в редакторе кода заготовке процедуры напишем код запуска диалога открытия файла (листинг 21.4).

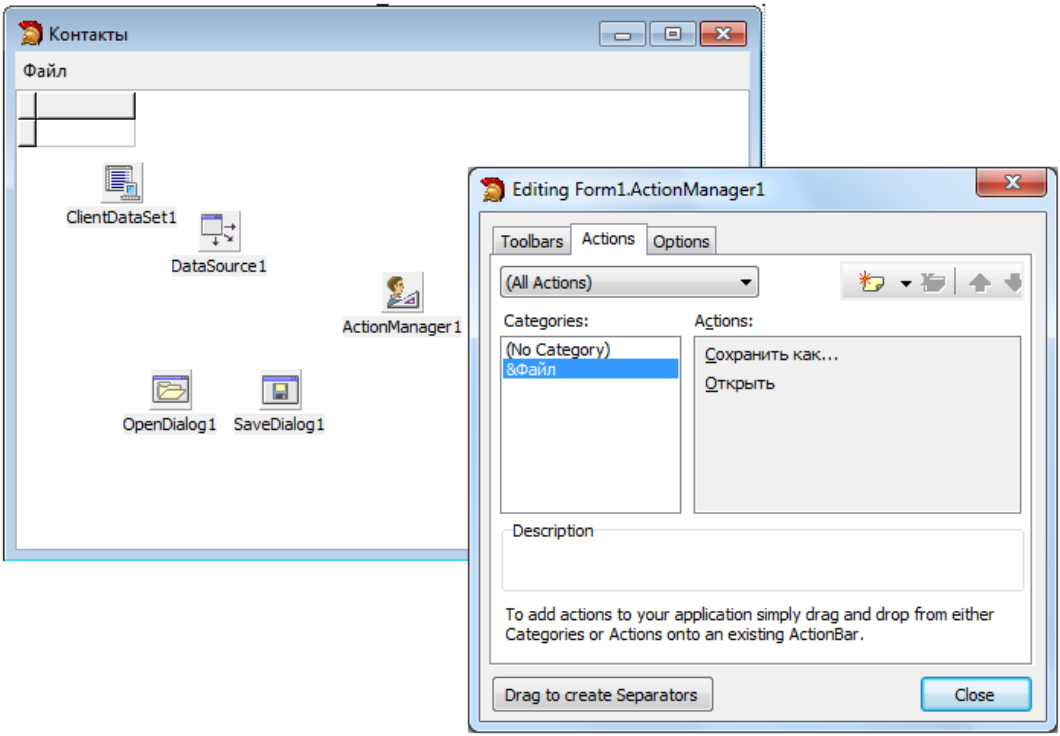


Рис. 21.3. Внешний вид приложения с редактором менеджера ActionManager1

Листинг 21.4. Обращение к диалогу открытия файла

```
procedure TForm1.acOpenFileExecute(Sender: TObject);
begin
 if OpenDialog1.Execute then
    try
    ClientDataSet1.Active:=false;
    ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
    except
    Application.MessageBox('Проверьте формат файла!','Ошибка');
    end;
end;
```

Обращение к команде открытия файла закрывает прежний набор данных и пробуждает диалог открытия файлов, в котором пользователь выберет подходящий ему документ. Если компонент `ClientDataSet1` не может обработать содержимое открываемого файла, то выводится предупреждающее сообщение.

В листинге 21.5 приведен код вызова события `OnExecute()` команды сохранения файла `acSaveAs`.

Листинг 21.5. Обращение к диалогу сохранения файла

```
procedure TForm1.acSaveAsExecute(Sender: TObject);
var s:string;
begin
  if SaveDialog1.Execute then
     begin
       s:=SaveDialog1.FileName;
       if UPPERCASE(ExtractFileExt(s))='XML' then
          ClientDataSet1.SaveToFile(s,dfXML)
       else //иначе бинарный файл
          ClientDataSet1.SaveToFile(s,dfBinary);
     end;
end;
```

Для того чтобы определить, в каком формате следует сохранить файл, мы проверяем выбранное пользователем расширение имени файла. Если выбрано расширение xml, то в метод `SaveToFile()` передается значение `dfXML`, иначе файл будет сохранен в двоичном формате.

Требования к структуре XML

Читатель, внимательно изучивший *главы 15* и *16*, наверняка уже сделал правильный вывод — компоненты класса `TClientDataSet` способны обрабатывать данные, представленные в формате расширяемого языка разметки (eXtensible Markup Language, XML). Это действительно так, но с существенными оговорками. Сохраните структуру таблицы, созданной в нашем примере, в файл contacts.xml в ту же папку, в которую мы с вами сохранили проект Delphi. Откройте документ contacts.xml в любом редакторе (лучше воспользоваться услугами утилиты XML Mapping Tool, она вызывается из главного меню Delphi: **Tools | XML Mapper**). В результате вы увидите документ, построенный в соответствии с требованиями стандарта XML (рис. 21.4).

К сожалению компонент `TClientDataSet` далеко не всеяден и не в состоянии "переварить" любой XML-документ, пусть даже и вполне корректный с точки зрения действующего стандарта. Нашему герою по зубам только документы XML, состоящие из заранее оговоренного набора элементов:

- `<DATAPACKET>` — корневой элемент документа, в котором находится информация о версии структуры;
- `<METADATA>` — предваряет описание структуры нашей таблицы;

- `<FIELDS>` — перечень полей таблицы, содержащий атрибуты, описывающие отдельные поля `<FIELD>`;
- `<PARAMS>` — дополнительные параметры.

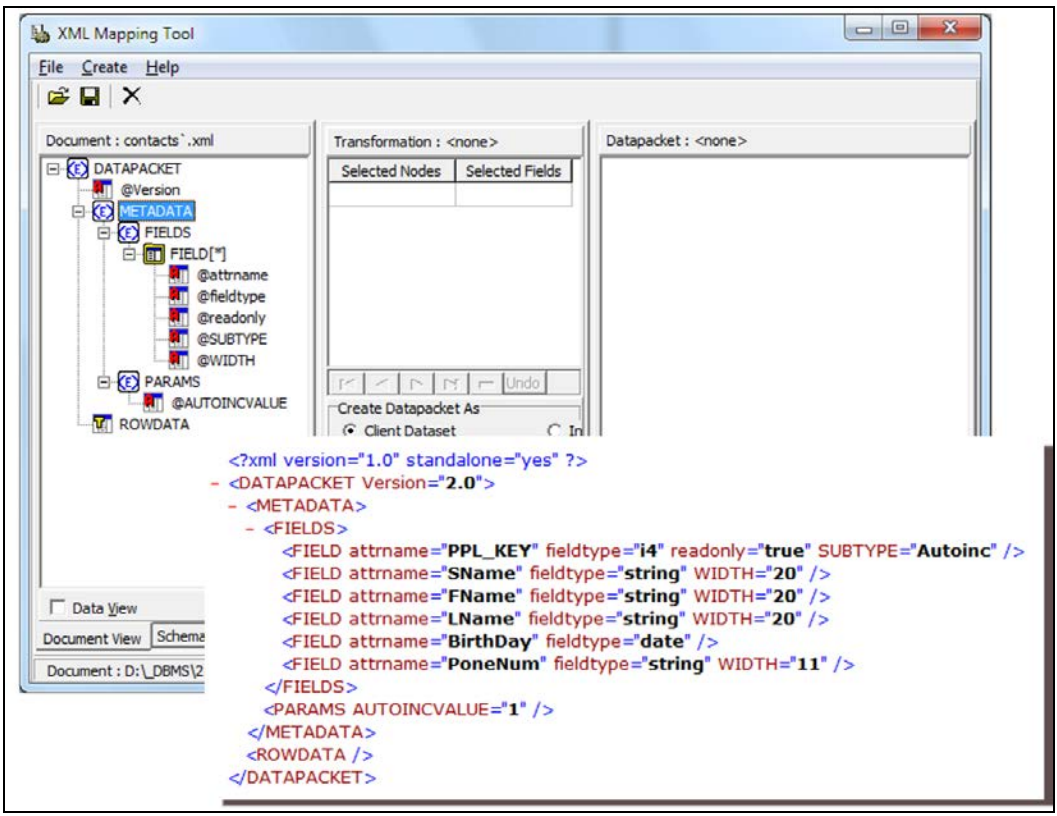


Рис. 21.4. Документ contracts.xml в редакторе XML Mapping Tool

Замечание

При работе с кириллицей целесообразно в первую строку XML-документа добавить параметр кодировки `<?xml version="1.0" encoding="WINDOWS-1251" standalone="yes"?>` или работать в формате UTF-8.

Хранение данных в компоненте

Независимо от того, каким образом данные попали в руки компонента `TClientDataSet` (то ли это был локальный файл таблицы, то ли данные были получены от сервера приложений многоуровневого проекта БД), все записи загружаются в локальный кэш компонента.

Доступ к массиву данных осуществляется с помощью любого из приведенных далее свойств:

```
property Data: OleVariant; //данные в двоичном формате
property XMLData: OleVariant;//данные в формате XML
```

В настольных проектах БД свойства `Data` и `XMLData` обслуживают один и тот же буфер данных. Разница между ними в том, что первое из свойств рассматривает данные как обычную битовую последовательность, а второе — как XML-документ.

Нужно иметь в виду, что хранимые в свойстве `Data` данные могут представлять лишь часть записей из подключенной к компоненту таблицы БД. Это вполне нормально для многоуровневых проектов БД. В таких проектах передача данных между компонентом `TClientDataSet` и серверной частью проекта организуется в форме пакетов. Если пакет не способен вместить все строки обслуживаемой таблицы (см. свойство `PacketRecords`), то в свойстве `Data` окажется лишь последняя поместившаяся в пакет порция записей.

О том, какой объем памяти занимают данные, знает свойство

```
property DataSize: Integer; //только для чтения
```

Единица измерения — байт.

Так как обрабатываемые данные хранятся в памяти компонента, то ряд действий с ними осуществляется с молниеносной быстротой. Например, высокоскоростной метод

```
procedure EmptyDataSet;
```

полностью очистит набор данных за доли секунд.

> ***Внимание!***
>
> Одной из уникальных черт `TClientDataSet` является то, что все измененные в процессе редактирования строки (новые, исправленные и удаленные записи) не направляются в БД немедленно, а накапливаются в локальном буфере компонента и находятся там до поступления команды на сохранение (или отката к первоначальному состоянию). Доступ к отредактированным данным обеспечивает свойство `Delta`.

Управление данными

Клиентский набор данных `TClientDataSet` целиком и полностью унаследовал весь спектр свойств и методов у своего универсального предка `TDataSet`. Благодаря этому читатель, внимательно ознакомившийся с *главой 18*, без особого труда сможет решить задачи, связанные с редактированием данных, перемещением по набору данных, поиском и фильтрацией и т. п. Мы не станем повторять уже пройденную теорию, а вновь перейдем к практике.

Обратитесь к менеджеру операций `ActionManager1` проекта и дополните его элементами, отвечающими за вставку `Name=acInsert`, редактирование `Name=acEdit`, сохранение `Name=acPost` и удаление данных `Name=acDelete`. Последовательно вызовите обработчики событий `OnExecute()` каждой из команд и заполните их программным кодом, предложенным в листинге 21.6.

Листинг 21.6. Вставка, редактирование, сохранение и удаление записей

```
procedure TForm1.acInsertExecute(Sender: TObject);
begin
  ClientDataSet1.Insert; //вставка новой записи
end;
procedure TForm1.acEditExecute(Sender: TObject);
begin
  ClientDataSet1.Edit; //перевод компонента в режим редактирования
end;
procedure TForm1.acPasteExecute(Sender: TObject);
begin
  ClientDataSet1.Post; //сохранение изменений
end;
procedure TForm1.acDeleteExecute(Sender: TObject);
begin
 if Application.MessageBox('Удалить запись?','Удаление',MB_YESNO)=mrYes then
    ClientDataSet1.Delete; //удаление текущей записи
end;
```

Вряд ли приведенный код нуждается в дополнительных комментариях. В обработчиках события вызываются знакомые нам еще по `TDataSet` методы `Insert()`, `Edit()`, `Post()` и `Delete()`.

Состояние клиентского набора отражается его свойством `State` (см. табл. 18.1). Это весьма полезное свойство сослужит нам хорошую службу при описании событий `OnUpdate()` только что созданных команд. Благодаря `OnUpdate()` мы сможем установить команду в актуальное (отвечающее текущей обстановке) состояние.

ВНИМАНИЕ!

У элементов `TAction` событие `OnUpdate()` вызывается в отдельном фоновом потоке именно в тот момент, когда связанный с командой элемент готовится к выводу на экран. Благодаря этому команда способна контролировать текущую обстановку с минимальным расходом системных ресурсов. Такое решение называется подходом обновления по требованию (update-on-demand approach).

Команда на вставку новой записи может быть доступна только после того, как набор данных перейдет в состояние просмотра данных (листинг 21.7).

Листинг 21.7. Установка командных элементов в актуальное состояние

```
procedure TForm1.acInsertUpdate(Sender: TObject); //КОМАНДА ВСТАВКИ
begin
{Команда сохранения активируется только в случае, когда клиентский
набор находится в состоянии редактирования или вставки}
```

```
acInsert.Enabled:=(ClientDataSet1.State=dsBrowse);
end;


procedure TForm1.acPasteUpdate(Sender: TObject); //КОМАНДА СОХРАНЕНИЯ
begin
{Активность команды определяется состоянием набора данных}
 acPaste.Enabled:=(ClientDataSet1.State=dsEdit) or
                  (ClientDataSet1.State=dsInsert);
end;


procedure TForm1.acDeleteUpdate(Sender: TObject); //КОМАНДЫ УДАЛЕНИЯ И
РЕДАКТИРОВАНИЯ
begin
{Обработчик события OnUpdate() команды удаления включит команду только при ус-
ловии, что набор данных находится в состоянии просмотра и содержит хотя бы од-
ну запись}
(Sender as TAction).Enabled:=(ClientDataSet1.State=dsBrowse)
                             and
                             (ClientDataSet1.RecordCount>0);
end;
```

Обработчик события `OnUpdate()` команды удаления можно сделать общим с командой редактирования. Поэтому в листинге мы явным образом не указываем, что код принадлежит команде удаления (`acDelete.Enabled=…`), а применяем более хитрое решение. Мы говорим, что событие управляет активностью той команды, которая к нему обратилась (`Sender as TAction`). Сделайте это событие общим для команд удаления и редактирования. Выберите команду редактирования `acEdit` в инспекторе объектов и перейдите на вкладку событий. Выделите строку `OnUpdate` и в выпадающем списке найдите событие `acDeleteUpdate` (рис. 21.5).

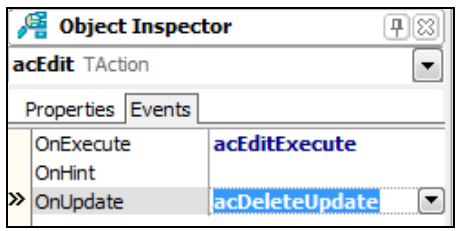


Рис. 21.5. Разделение одного события несколькими командами

Наш проект уже вполне жизнеспособен. Запустите программу и введите несколько строк, содержащих данные о ваших друзьях и знакомых.

Работа с индексами

Если вы заполнили наш пример хоты бы десятком записей, то уже наверняка заметили один существенный недостаток проекта — записи не упорядочены по алфавиту. Это весьма печальный факт, он приведет к тому, что с увеличением числа строк пользователю будет все сложнее найти необходимую запись. Чтобы устранить указанный недостаток, потребуется поработать с индексами.

Для описания структуры будущего индекса клиентского набора данных следует обратиться к коллекции доступных индексов компонента `ClientDataSet1` — свойству `IndexDefs`. Щелчок по кнопке с многоточием заставит появиться на экране редактор с перечнем имеющихся в распоряжении компонента индексов. Пока список индексов пуст. Для нашего примера БД крайне необходим индекс, упорядочивающий контактные данные по фамилии, имени и отчеству. Для этого нажимаем кнопку **Add New** (Добавить новый) в редакторе. Наше действие приводит к появлению нового индекса с названием `ClientDataSet1Index1`. Если индекс только один, то возможно, название по умолчанию нас устроит, если же планируется создать несколько индексов, то целесообразно присвоить им содержательные имена. Щелчок по строке с именем индекса активизирует деятельность инспектора объектов — он отобразит свойства выбранного индекса (рис. 21.6). Первым делом, переименуйте индекс в `indexSName` и перейдите к его свойству `Fields`.

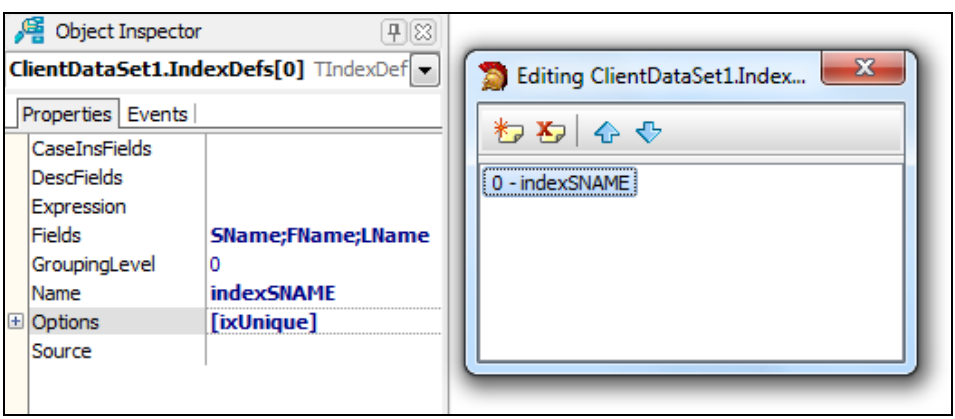


Рис. 21.6. Редактор индекса компонента `ClientDataSet1` и описание индекса в инспекторе объектов

Среди всего перечня свойств наиболее значимым является свойство `Fields`, в него следует передать имена полей таблицы, входящих в индекс. Если полей несколько, то их имена следует разделять точкой с запятой. Заполните свойство, перечислив названия полей, хранящих фамилию, имя и отчество. Если требуется, чтобы индекс не допускал повторений, то для обеспечения уникальности данных воспользуйтесь свойством `Options`, присвоив ему значение `ixUnique`. Описание индекса завершено.

Для того чтобы индекс начал приносить пользу, его следует подключить к набору данных `TClientDataSet`. Возможны два пути. Первое решение задействует свойство компонента

```
property IndexName: Widestring;
```

В него достаточно направить имя индекса, в нашем примере — `indexSName`.

Второй путь основан на обращении к свойству

```
property IndexFieldNames: Widestring;
```

В последнем случае требуется явно указать имена полей, входящих в индекс.

ВНИМАНИЕ!

Свойства `IndexFieldNames` и `IndexName` взаимоисключающие — они не могут применяться одновременно, т. к. установка значения в одно из них обнуляет другое свойство.

Все операции, которые мы только что провели в редакторе индексов, совсем несложно реализовать и программно. В листинге 21.8 вашему вниманию предложен доработанный обработчик события выполнения команды `acOpen`.

Листинг 21.8. Доработка события открытия файла

```
procedure TForm1.acOpenFileExecute(Sender: TObject);
begin
 if OpenDialog1.Execute then
    try
    ClientDataSet1.Active:=false;
    ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
       {если отсутствует индекс, то его следует создать}
       if ClientDataSet1.IndexDefs.Count=0 then
        Begin
           with ClientDataSet1.IndexDefs.AddIndexDef do
           begin
           Name := 'indexSName';
           Fields:='SName;FName;LName';
           Options:=[ixUnique];
           end;
        ClientDataSet1.IndexName:='indexSName';
        End;
    except
    Application.MessageBox('Проверьте формат файла!','Ошибка');
    end;
end;
```

После загрузки файла и открытия набора данных в коде проверяется факт существования индекса. Если такового нет, то индекс создается и подключается к компоненту `ClientDataSet1`.

Доработайте проект и запустите его на выполнение. Признаком успешной работы индекса станет автоматическое упорядочение строк таблицы по алфавиту.

Индекс можно создать и без явного обращения к коллекции индексов `IndexDefs` компонента. Вместо этого следует воспользоваться методом

```
procedure AddIndex(const Name: string; const Fields: string;
                   Options: TIndexOptions; const DescFields: string = '';
                   const CaseInsFields: string = '';
                   const GroupingLevel: Integer = 0);
```

Часть параметров нам уже знакома. Здесь `Name` — имя индекса; `Fields` — перечень полей, входящих в состав индекса; `Options` — опции индекса (табл. 21.3); `DescFields` — перечень полей индекса, учитываемых при сортировке в обратном направлении; `CaseInsFields` — перечень полей, у которых не будет учитываться регистр символов; `GroupingLevel` — уровень группировки, позволяющий получать результат, схожий с результатом выполнения запроса SQL с директивой `GROUP BY`.

Таблица 21.3. *Опции индекса* `TIndexOptions`

Значение	Описание
`ixPrimary`	Первичный ключ таблицы
`ixUnique`	Уникальный индекс, не допускающий повторения данных во входящих в его состав полях
`ixDescending`	Индекс обратной сортировки (по убыванию значений)
`ixExpression`	Индекс основан на ключевом выражении (предназначен только для таблиц dBase)
`ixCaseInsensitive`	Индекс не чувствителен к регистру символов
`ixNonMaintained`	Индекс не обновляется автоматически после редактирования данных

Если параметр `GroupingLevel` установлен в состояние 0, то режим группировки отключается. Значение 1 указывает, что в процессе группировки используется только первое поле в индексе, 2 — первое и второе поля и т. д. Номер уровня группировки не должен превышать общего числа полей, входящих в индекс. Уровень группировки окажется важен, если набор данных будет пользоваться услугами агрегирующих функций.

В заключение заметим, что метод `AddIndex()` следует вызывать только для активного набора данных.

Внимание!

Индекс, созданный средствами метода `AddIndex()`, существует только во время выполнения программы и с завершением ее работы исчезает.

В состав индекса может входить несколько полей таблицы. О числе полей, входящих в составной индекс, проинформирует свойство

```
property IndexFieldCount: Integer;
```

Перечень всех полей текущего индекса доступен благодаря списку

```
property IndexFields: [Index: Integer]: TField;
```

Ставшие ненужными индексы следует удалять. Для этого предназначен метод

```
procedure DeleteIndex(const Name: string);
```

Единственное, что потребуется для удаления, — знать имя индекса.

Определение диапазона

Диапазон (англ. *Range*) производит выборку строк из таблицы, пропуская лишь записи со значениями полей, попадающих вовнутрь назначенных программистом границ. По сравнению с потенциальными возможностями свойств `Filter` и `Filtered`, позволяющих описывать сложные условия фильтрации данных, способности диапазона не столь примечательны. Но этот недостаток меркнет в ситуации, когда на первый план выходит скорость отбора данных. Анализируя проиндексированные поля, диапазон формирует результирующий набор строк практически мгновенно.

Процесс определения и применения диапазона включает ряд обязательных шагов. Во-первых, вызывается метод

```
procedure SetRangeStart;
```

Процедура переводит набор данных в состояние `dsSetKey` и обнуляет предыдущее значение начала диапазона. Далее программист определяет начало диапазона. Второй шаг повторяет ту же операцию, но уже для конца диапазона. Для этого вызывается метод

```
procedure SetRangeEnd;
```

Третий шаг — вызов метода, применяющего диапазон:

```
procedure ApplyRange;
```

Процедура отберет записи, содержимое которых удовлетворяет границам диапазона, и установит курсор на первую строку сокращенного набора данных

Внимание!

Диапазон можно применять только к полям, входящим в текущий индекс таблицы.

В свете полученных знаний предлагаю доработать стартовавший в начале главы проект так, чтобы пользователь получил возможность отбирать записи только из определенного диапазона дат. Для того чтобы воплотить задуманное в жизнь, требуется провести ряд подготовительных мероприятий. Первое из них — определение еще одного индекса, в состав которого будет входить единственное поле `BirthDay`, хранящее дату рождения. Вы уже умеете создавать индекс с помощью коллекции полей `IndexDefs`, кроме того, знаете о формировании индекса программным способом. Какой из путей вы выберете, не важно, главное, чтобы индекс получил название `indexBirthDay`.

Вторым этапом нашей работы станет проектирование новой формы, позволяющей выбрать диапазон дат.

Для этого проделаем следующие шаги:

- Создайте новую форму и переименуйте ее в `frmDateRange`. Удалите форму из списка автоматически создаваемых. Для этого выберите пункт меню **Project | Options** (Проект | Опции), в появившемся окне опций проекта выберите вкладку **Forms**. Перенесите только что созданную форму из списка **Auto-create forms** (Автоматически создаваемые формы) в список **Available forms** (Доступные формы). Установите свойство `BorderStyle` в `bsDialog`, это действие укажет Delphi, что форма предназначена для выполнения функций диалога.
- Разместите на форме два компонента `TDateTimePicker` (вы их найдете на странице **Win32**). Переименуйте первый элемент управления в `dtStartDate`, это напомнит нам, что он предназначен для ввода даты начала диапазона. Второй элемент назовем `dtEndDate`, в нем пользователь укажет дату окончания диапазона.
- В нижней части формы разместите две кнопки `TBitBtn` (страница **Additional**). Свойству `Kind` первой кнопки присвойте значение `bkOK`, второй — `bkCancel`.

В результате вы получите диалоговое окно с интерфейсом, представленным на рис. 21.7.

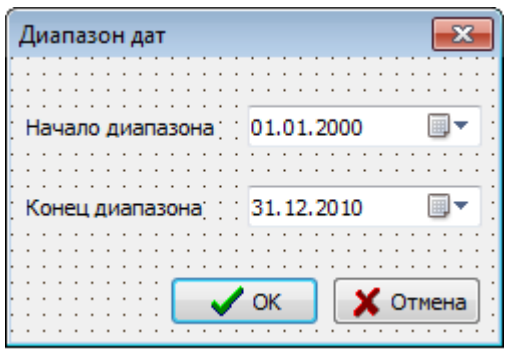


Рис. 21.7. Диалог выбора диапазона дат

Прежде чем мы приступим к программированию, добавьте в менеджер `ActionManager1` еще пару команд. Команда `acRangeOn` предназначена для вызова диалога выбора дат, команда `acRangeOf` понесет ответственность за отключение диапазона. Двойной щелчок по команде `acRangeOn` отправит нас к событию `OnExecute()`. Код события приведен в листинге 21.9.

Листинг 21.9. Применение диапазона

```
procedure TForm1.acRangeOnExecute(Sender: TObject);
var frmDateRange: TfrmDateRange;
begin
 frmDateRange:=TfrmDateRange.Create(Application.MainForm);
```

```
 if frmDateRange.ShowModal=mrOK then
      With ClientDataSet1 do
       begin
          IndexName:='indexBirthDay'; //выбор индекса
          SetRangeStart;              //начало диапазона
          FieldByName('BirthDay').Value := frmDateRange.dtStartDate.Date;
          SetRangeEnd;                //окончание диапазона
          FieldByName('BirthDay').Value := frmDateRange.dtEndDate.Date;
          ApplyRange;                 //применяем диапазон
       end;
 frmDateRange.Release;
end;
```

В коде обработчика события создается и выводится в модальном режиме на экран экземпляр формы `frmDateRange`. Пользователь выбирает диапазон дат и нажимает кнопку **OK**, после чего значения дат передаются в соответствующие свойства компонента `ClientDataSet1`. Заметьте, что перед применением диапазона мы выбираем индекс, в составе которого имеется поле даты рождения `BirthDay`.

Для отмены диапазона воспользуемся процедурой

```
procedure CancelRange;
```

После обращения к методу набор данных вернется в исходное состояние, отобразив весь набор строк. В нашем примере полномочия на отказ от услуг диапазона предоставлены команде `acRangeOf` (листинг 21.10).

Листинг 21.10. Отмена диапазона

```
procedure TForm1.acRangeOfExecute(Sender: TObject);
begin
  ClientDataSet1.CancelRange; //отмена диапазона
  ClientDataSet1.IndexName:='indexSName';//возврат к исходному индексу
end;
```

Для изменения границ диапазона предназначены методы

```
procedure EditRangeStart;//редактируем начало диапазона
procedure EditRangeEnd;  //редактируем окончание диапазона
```

Вместо последовательного вызова процедур `SetRangeStart()`, `SetRangeEnd()` и `ApplyRange()` зачастую удобнее обратиться к одному-единственному методу:

```
procedure SetRange(const StartValues, EndValues : array of const);
```

Границы диапазона определяются параметрами `StartValues` и `EndValues` (листинг 21.11).

Листинг 21.11. Изменение границ диапазона

```
ClientDataSet1.IndexName:='indexBirthDay';
ClientDataSet1.SetRange([frmDateRange.dtStartDate.Date],
                        [frmDateRange.dtEndDate.Date]);
```

Обратите внимание на то, что перед вызовом метода `SetRange()` мы, с помощью свойства `IndexName`, подключаем вторичный индекс, благодаря которому и осуществится отбор данных из заданного диапазона.

Некоторые изменения в порядок включения границ в диапазон вносит свойство

```
property KeyExclusive: Boolean; //по умолчанию false
```

Установив свойство в `true`, программист указывает программе, что граничные значения не должны входить в диапазон.

Ограничения

Клиентский набор данных позволяет читать, а при необходимости и назначать пользовательские ограничения на значения полей записи набора данных. Доступ к ограничениям предоставляет свойство-коллекция

```
property Constraints: TCheckConstraints;
```

Коллекция `TCheckConstraints` содержит свои элементы (экземпляры класса `TCheckConstraint`.) в свойстве `Items`. Основной метод коллекции `Add()` предназначен для добавления нового ограничения.

Экземпляр `TCheckConstraint` характеризуется тремя базовыми свойствами. Если ограничение было определено на уровне БД, то его характеристику мы обнаружим в свойстве

```
property ImportedConstraint: string;
```

Описание ограничений хранится в формате выражения SQL. Для определения дополнительных локальных ограничений следует воспользоваться свойством

```
property CustomConstraint: string;
```

Пользовательские ограничения накладываются на уровне клиентского набора данных и описываются по правилам SQL-выражения, например:

```
field1>10 and field1<99
```

Для уведомления пользователя о несоответствии вводимых данных требованиям к значению поля применяют свойство

```
property ErrorMessage: string;
```

В свойство заносится текстовое сообщение, автоматически выводимое на экран в случае нарушения ограничения на значение поля.

Применение агрегирующих функций

Любой опытный разработчик БД знает, какую неоценимую помощь способны оказать имеющиеся в составе SQL агрегирующие функции `MIN`, `MAX`, `SUM`, `AVG` и

COUNT (см. табл. 12.2). Перечисленные функции позволяют легко найти минимальное или максимальное значения в наборе данных, рассчитать среднее значение, получить сумму и узнать общее число записей в наборе.

Так как компонент TClientDataSet способен выполнять задачи по обработке данных автономно (без опоры на всемогущий SQL-сервер), разработчики Delphi решили научить компонент применять агрегирующие функции, не используя при этом структурированный язык запросов.

В перечень свойств клиентского набора данных включено свойство-коллекция:

```
property Aggregates: TAggregates;
```

предназначенное для хранения агрегатов — экземпляров класса TAggregate. Агрегаты могут создаваться и добавляться в коллекцию как при проектировании, так и во время выполнения приложения. В последнем случае для этого следует воспользоваться функцией

```
function Add: TAggregate;
```

Каждый из агрегатов позволяет осуществлять самостоятельные расчеты, базирующиеся на простейших математических выражениях.

Централизованное включение/отключение агрегатов производится с помощью свойства

```
property AggregatesActive: Boolean; //по умолчанию false
```

Для старта расчетов следует установить свойство в состояние true.

Агрегат, класс *TAggregate*

Рассмотрим основные черты отдельного агрегата. Пояснительная надпись, отражающая назначение агрегата, описывается в свойстве

```
property AggregateName: string;
```

Выражение, содержащее агрегирующую функцию, заносится в свойство

```
property Expression: string;
```

Здесь могут использоваться знакомые по SQL ключевые слова MIN, MAX, SUM, AVG, COUNT и имена доступных полей. Например, листинг 21.12 демонстрирует порядок применения функции COUNT.

Листинг 21.12. Работа с агрегирующей функцией Count

```
With ClientDataSet1.Aggregates.Add do
 begin
    AggregateName:='Всего строк';
    Expression:='Count(field1)';
    Active:=True;
 end;
```

Благодаря выражению Count() мы подсчитаем число строк в группе.

Для определения уровня группировки используют свойство

```
property GroupingLevel: Integer; //по умолчанию 0
```

По умолчанию свойство хранит нулевое значение (обработке подвергнутся все строки набора данных). Любое другое превышающее ноль значение указывает на то, что состав анализируемой группы определяют записи, входящие в состав индекса с именем

```
property IndexName: string;
```

Если свойство `GroupingLevel` принимает значение 1, то записи в наборе группируются по первому полю индекса, 2 — по первому и второму и т. д.

Чтобы агрегат смог начать вычисления, его следует активировать с помощью свойства

```
property Active: Boolean;
```

Готовность агрегата выдать результаты расчетов контролирует свойство

```
property InUse: Boolean;
```

Если оно возвращает `true`, то это свидетельствует об окончании расчетов и разрешает обращаться к функции

```
function Value: Variant;
```

возвращающей запрашиваемые результаты. Любой вызов функции инициирует автоматический пересчет результирующего значения. Например, следующий фрагмент кода (листинг 21.13) отобразит результаты выполнения агрегирующей функции на панели статусной строки `StatusBar1`.

Листинг 21.13. Проверка готовности агрегата

```
var x:integer;
…
if ClientDataSet1.Aggregates[0].InUse=true then
   begin
   x:=ClientDataSet1.Aggregates[0].Value;
   StatusBar1.Panels[0].Text:=
                ClientDataSet1.Aggregates[0].AggregateName +IntToStr(X);
   end;
```

Агрегатные поля *TAggregateField*

Клиентский набор данных (компонент `TClientDataSet`) позволяет создавать особый класс агрегатных полей `TAggregateField`. Агрегатные поля предназначены для совместной работы с агрегирующими функциями, объединяющими данные из некоторой группы записей таблицы.

Для создания экземпляра агрегатного поля следует вызвать редактор полей компонента `TClientDataSet` и добавить к нему новое поле в диалоге **New Field** (рис. 21.8).

Работа с агрегатным полем во многом напоминает организацию расчетов в агрегате `TAggregate`. Выражение, основанное на применении функций `MIN`, `MAX`, `SUM`, `AVG` и `COUNT`, записывается в строку:

```
property Expression: string;
```

Вычисления производятся только при активизации свойства

```
property Active: Boolean;
```

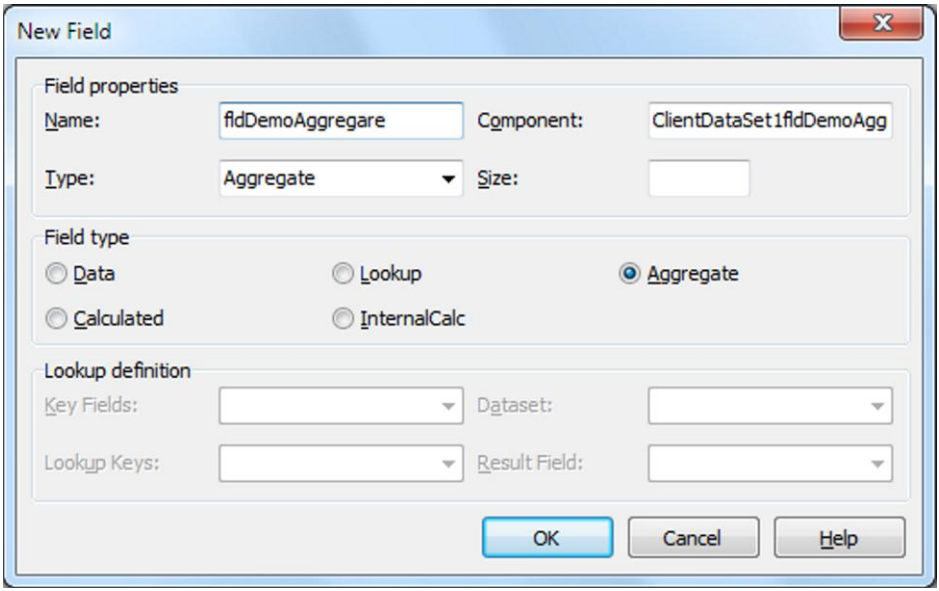


Рис. 21.8. Добавление агрегатного поля

ВНИМАНИЕ!

Для представления агрегатного поля на экране не применяют компоненты-сетки `TDBGrid`. Для этой цели обычно задействуют компоненты, специализирующиеся на работе с отдельным полем, например `TDBText`.

Место *TClientDataSet* в многоуровневых проектах БД

Обсудим еще один аспект применения клиентского набора данных. На этот раз речь пойдет о свойствах и методах компонента `TClientDataSet`, имеющих прямое отношение к построению многоуровневых проектов БД.

Главное отличие многоуровневых проектов БД от классических систем "клиент-сервер" заключается во внедрении в проект промежуточного звена — сервера приложений. Основная задача сервера приложений — поддержка бизнес-логики БД, что снимает нагрузку как с СУБД, так и с клиентских приложений. В результате клиентское приложение превращается в так называемого тонкого клиента, который, по сути, отвечает лишь за визуализацию данных.

Процесс создания многоуровневого проекта мы изучим в *главе 31*, но уже сейчас вы можете получить представление о месте компонента `TClientDataSet` в проекте DataSnap в Delphi 2010 (см. рис. 31.1). В подобных приложениях, для организации взаимодействия с удаленными наборами данных клиентский `TClientDataSet` сотрудничает с цепочкой компонентов-посредников. В их числе компоненты `T…Connection`, отвечающие за соединение клиентского модуля с сервером приложений, метод сервера `TSqlServerMethod` и провайдер `TDataSetProvider`, предоставляющий клиентскому набору услуги сервера.

В обновленной технологии (она появилась в Delphi 2010) в приложениях DataSnap для подключения клиентского набора следует воспользоваться свойством

```
property ProviderName: string;
```

Задача провайдера — обеспечить мост между клиентом и сервером приложений, а точнее, между компонентом `TClientDataSet` и соответствующим ему набором данных на сервере приложений.

Работа в клиентских приложениях DataSnap ранних версий

В более ранних версиях Delphi место компонента `TClientDataSet` в многоуровневом проекте БД было несколько другим (рис. 21.9). Основу взаимодействия между клиентом и сервером приложений составлял интерфейс `IAppServer`.

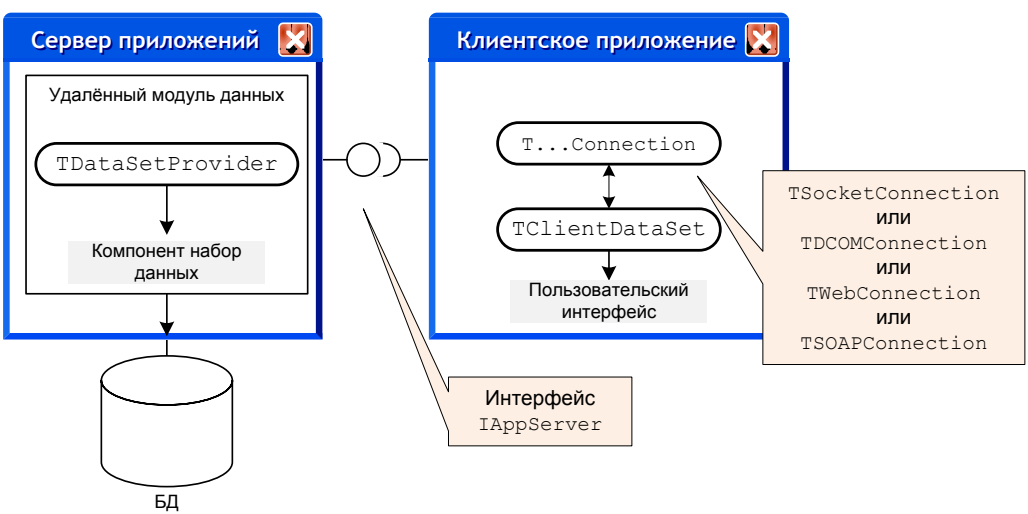


Рис. 21.9. Место компонента `TClientDataSet` в многоуровневых БД ранних версий

Проектирование клиентского приложения DataSnap до выхода Delphi 2010 начиналось с настройки соединения клиентского набора данных с удаленным сервером, это зона ответственности свойства

```
property RemoteServer: TCustomRemoteServer;
```

Благодаря компоненту-соединителю клиент получает сетевой доступ к расположенному на другом компьютере серверу приложений. Свойство может ссылаться на один из следующих компонентов: `TDCOMConnection`, `TSocketConnection`, `TWebConnection` или `TSOAPConnection` (см. рис. 21.9). Компоненты `TDCOMConnection` и `TSocketConnection` обеспечивают работу приложений в сетях предприятия. Разница между классами в том, что `TDCOMConnection` применяется в приложениях, ориентированных на технологию Microsoft (Distributed Component Object Model), а `TSocketConnection` позволяет строить многоуровневые приложения на основе универсальной технологии сокетов TCP/IP. Третий по счету компонент-соединитель `TWebConnection` ориентирован на разработку приложений, поддерживающих обмен данных на основе протокола HTTP. Наконец, компонент `TSOAPConnection` используется при разработке многоуровневых БД, взаимодействующих на основе Web-протоколов.

Замечание

Компоненты, обеспечивающие соединение клиентского приложения с сервером, собраны на страничке **Datasnap Client** палитры компонентов Delphi. Для доступа к серверу приложений компоненты DataSnap задействуют интерфейс `IAppServer`.

Сохранение данных на сервере

Несмотря на множество общих черт с универсальным набором данных, компонент `TClientDataSet` обладает рядом уникальных возможностей, одна из них — ведение протокола всех действий пользователя, связанных с добавлением, редактированием и удалением данных. Все изменения в записях, сделанные оператором компьютера, не сразу попадают в физическую таблицу, а сначала заносятся в дополнительный кэш и находятся там до особого распоряжения. Можно сказать, что компонент ведет протокол всех действий пользователя.

Доступ к кэшу обеспечивает свойство

```
property Delta: OleVariant;
```

Разработчики Delphi допускают, что у программиста может возникнуть острое желание заставить компонент направлять данные не в протокол `Delta`, а непосредственно в свойство `Data`. В этом случае следует перевести в `false` свойство

```
property LogChanges: Boolean; //по умолчанию true
```

Однако в многоуровневых приложениях так поступать не стоит. Поэтому названное свойство следует оставлять в состоянии по умолчанию, тогда все пользовательские правки отправятся в буфер `Delta`.

Замечание

Протокол `Delta` содержит не все строки данных пользовательского набора, а лишь информацию о записях, подвергшихся изменению.

Об общем числе редакторских правок в хранимых в свойстве `Delta` нас проинформирует свойство

```
property ChangeCount: Integer; //только для чтения
```

Счетчик получает приращение с каждой новой модификацией данных (вставкой, удалением или редактированием).

В многоуровневых проектах для отправки всех пользовательских правок на сервер следует вызывать метод

```
function ApplyUpdates(MaxErrors: Integer): Integer; virtual;
```

Функция учитывает вероятность возникновения ошибок при передаче данных. Порядок реакции на ошибки определяется параметром `MaxErrors`, он назначает предельно допустимое число ошибок. Нулевое значение требует прекратить отправку данных на сервер при возникновении первой же ошибки. Значение –1 указывает на то, что процесс сохранения данных не прерывается при любом количестве сбоев. По окончании выполнения метод возвратит число успешно сохраненных записей. Записи, сохранение которых завершилось некорректно, возвращаются в кэш `Delta`.

Обращение к методу `ApplyUpdates()` инициирует череду событий. Сначала генерируется событие

```
property BeforeApplyUpdates: TRemoteEvent;//перед началом обновления
```

в рамках которого клиентское приложение отправляет свои данные. Затем сервер анализирует полученную информацию и вносит изменения в БД. При возникновении ошибок сервер собирает о них данные и возвращает назад в клиентское приложение. В заключение генерируется событие

```
property AfterApplyUpdates: TRemoteEvent; //обновление завершено
```

в котором, клиентское приложение обрабатывает ответ сервера.

Оба события обновления типизированы особым классом обработчика событий, позволяющим организовать взаимодействие между клиентским приложением и провайдером:

```
TRemoteEvent = procedure(Sender: TObject; var OwnerData: OleVariant) of
object;
```

Информация между участниками обмена передается благодаря параметру `OwnerData`.

Отмена изменений

Если по каким-то причинам необходимость сохранения на сервере БД редакторских правок отпадает, следует обратиться к методу

```
procedure CancelUpdates;
```

Такая операция полностью обнулит протокол изменений `Delta` и сбросит счетчик правок `ChangeCount` к нулевому значению. Менее радикальное решение возвратит в исходное состояние только последнюю операцию с данными. Для этого следует вызвать функцию

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Параметр `FollowChange` определяет, перевести курсор на восстановленную строку (значение `true`) или оставить на текущей записи (`false`).

Метод

```
procedure RevertRecord;
```

также против радикальных мер, он отменит изменения, которым подверглась только текущая запись.

Применение точек сохранения

Кроме отката всех изменений единым пакетом, класс `TClientDataSet` поддерживает механизм точек сохранения, позволяющий пользователю осуществлять пошаговый откат изменений, или даже реализовать на стороне клиента простые транзакции. Точка сохранения характеризуется порядковым номером:

```
property SavePoint: Integer;
```

Для запоминания точки сохраните значение свойства в переменной, после этого вы сможете вернуться к точке сохранения так, как показано в листинге 21.14.

Листинг 21.14. Точки сохранения

```
var MyPoint : integer;
begin
MyPoint:=ClientDataSet1.SavePoint; //запоминаем точку сохранения
{операции с данными}
if <условие=true> then
    ClientDataSet1.SavePoint:=MyPoint //возврат к точке сохранения
else ClientDataSet1.ApplayUpdates(-1);
end;
```

Обработка ошибок

После вызова метода `ApplayUpdates()` клиентский набор данных отправляет в адрес провайдера `TDataSetProvider` пакет, содержащий данные из буфера `Delta`. Провайдер обрабатывает полученные данные (например, проверяет их на соответствие бизнес-правилам), а затем ретранслирует пакет в адрес сервера БД.

Если в процессе сохранения строки данных возникает ошибка, то провайдер извещает о ней клиента, а тот, в свою очередь, генерирует событие

```
property OnReconcileError: TReconcileErrorEvent;
```

Здесь

```
TReconcileErrorEvent = procedure (DataSet: TCustomClientDataSet;
                                  E: EReconcileError; UpdateKind: TUpdateKind;
                                  var Action: TReconcileAction) of object;
```

`DataSet` — "виновный" в ошибке набор данных, если набор данных обслуживает связанные (например, отношением "один ко многим") друг с другом наборы, то через параметр может прийти ссылка на подчиненный набор данных; `E` — исклю-

чительная ситуация, предоставляющая информацию об ошибке; `UpdateKind` — детализация причины ошибки (`ukModify` — ошибка правки; `ukInsert` — вставки, `ukDelete` — удаления); `Action` — указывает, как следует отреагировать на возникшую ошибку (табл. 21.4).

Таблица 21.4. *Реакция на ошибку при сохранении записи* `TReconcileAction`

Действие	Описание
`raSkip`	Отменить операцию сохранения для вызвавшей ошибку записи, запись-неудачница возвращается в буфер `Delta` клиентского набора данных
`raAbort`	Прервать процесс сохранения записи
`raMerge`	Объединить обновляемую запись с данными на сервере, другими словами, добавить в таблицу запись, вызвавшую ошибку, как новую
`raCorrect`	Повторить попытку сохранения
`raCancel`	Отменить все изменения для данной записи и восстановить значения полей, считав запись из буфера `Data` клиентского набора данных
`raRefresh`	Отменить все изменения для данной записи

В составе шаблонов Delphi имеется специализированный диалог, существенно упрощающий труд программиста, "колдующего" над обработкой события `OnReconcileError()`. Для подключения модуля к проекту необходимо воспользоваться меню **File | New | Other**, затем в появившемся окне следует найти пиктограмму **Reconcile Error Dialog** (Диалог согласования ошибки) и нажать кнопку **OK**.

Для того чтобы подключенный модуль начал взаимодействовать с обработчиком события `OnReconcileError()`, следует включить название модуля диалога в строку `uses` и написать единственную строку кода (листинг 21.15).

Листинг 21.15. Вызов диалога согласования ошибки

```
procedure TForm1.ClientDataSet1ReconcileError(
                            DataSet: TCustomClientDataSet;
                            E: EReconcileError; UpdateKind: TUpdateKind;
                            var Action: TReconcileAction);
begin
Action:=HandleReconcileError(DataSet, UpdateKind, E );
end;
```

На рис. 21.10 представлен экранный снимок диалога, в котором смоделирована ошибка, возникающая при попытке одновременного редактирования одной и той же записи несколькими пользователями. Оператору компьютера предоставляется возможность выбрать способ устранения ошибки совместного доступа,

указания пользователя передаются в параметр `Action` обработчика события `OnReconcileError()`.

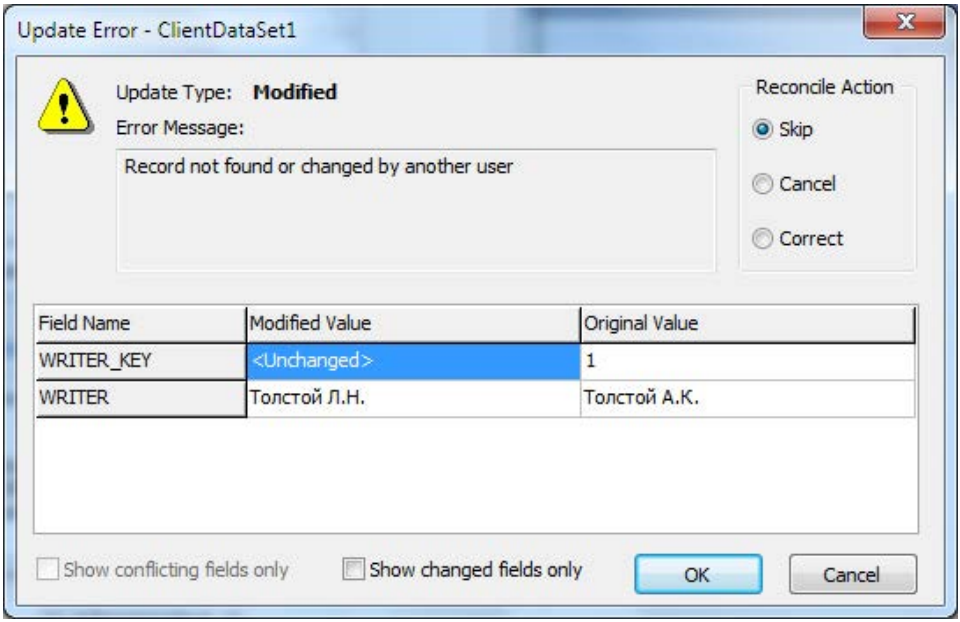


Рис. 21.10. Диалог обработки ошибки

Ошибки могут возникать не только при отправке данных к серверу БД, но и при модификации записей в локальном кэше компонента. В этом случае на сцену выходят события, унаследованные от родительского класса `TDataSet`:

```
property OnPostError: TDataSetErrorEvent;//ошибка сохранения данных
property OnEditError: TDataSetErrorEvent;//ошибка редактирования данных
property OnDeleteError: TDataSetErrorEvent;//ошибка удаления данных
```

Оптимизация пакета с данными

Обмен данными между расположенным на стороне сервера компонентом-провайдером и клиентским набором данных `TClientDataSet` проводится в форме пакетов. Размерность пакетов допускается регулировать, для этого предназначено свойство

```
property PacketRecords: Integer;//по умолчанию -1
```

По умолчанию свойство хранит значение –1. Это означает, что в пакет включаются все записи результирующего набора. Подобное решение оправдано только в случае, когда мы работаем с малыми БД. Нулевое значение свидетельствует о том, что клиентский набор нуждается только в метаданных. Если клиенту необходимо просмотреть определенное число строк, то целесообразнее ограничить размер

передаваемых порций данных, например `PacketRecords:=10`. Заметим еще одну положительную черту компонента `TClientDataSet` — способность отслеживать состояние присоединенных к нему элементов пользовательского интерфейса. Допустим, что размеры подключенной к клиентскому набору данных сетки `TDBGrid` позволяют одновременно отобразить 20 строк. В этом случае сразу после открытия клиентского набора данных он затребует от провайдера сразу 20 строк, и только потом удовлетворится пакетами по 10 записей в каждом.

По мере необходимости клиентский набор самостоятельно запросит очередной пакет с данными, это произойдет при условии, что свойство

```
property FetchOnDemand: Boolean; //по умолчанию true
```

установлено в состояние `true`. Если же режим автоматической подкачки данных отключен (и свойство `PacketRecords` явным образом определяет размерность пакета), то для получения от сервера очередной порции данных клиентский набор должен воспользоваться методом

```
function GetNextPacket: Integer; virtual;
```

Если провайдер отказывается передавать клиентскому приложению данные полей BLOB (в опциях компонента установлено значение `poFetchBlobsOnDemand`), то клиент может потребовать предоставить ему содержимое полей явным образом. Для этого следует обратиться к методу

```
procedure FetchBlobs;
```

ЗАМЕЧАНИЕ

Если клиентский набор работает совместно с провайдером данных `TDataSetProvider`, то порядок загрузки BLOB-данных зависит от состояния опции `poFetchBlobsOnDemand` и `poFetchDetailsOnDemand`.

В процессе запроса от провайдера пакета с данными возникает пара событий:

```
property BeforeGetRecords: TRemoteEvent; //до запроса пакета
property AfterGetRecords: TRemoteEvent;  //после получения пакета
```

Обновление данных

Для обновления данных, хранящихся в локальном кэше клиентского набора, следует вызвать метод

```
procedure RefreshRecord;
```

Процедура инициирует отправку запроса на получение от сервера копии данных. Процесс обновления сопровождается парой событий:

```
property BeforeRowRequest: TRemoteEvent; //перед началом обновления
property AfterRowRequest: TRemoteEvent;  //по окончании обновления
```

Для себя отметим, что обращение к методу `RefreshRecord ()` очистит свойство `Delta`, обнулив все несохраненные пользовательские правки на стороне клиента.

Выполнение команд SQL

Для передачи в адрес сервера баз данных инструкции на языке SQL понадобится свойство

```
property CommandText: string;
```

Обычно команда определяет, какие именно данные мы рассчитываем получить от поставщика услуг (провайдера), кроме того, это может быть выражение или запрос на SQL или имя таблицы, хранимой процедуры.

ЗАМЕЧАНИЕ

Находящийся на сервере приложений провайдер данных `TDataSetProvider` не станет реагировать на поступающие от клиента команды SQL до тех пор, пока в его свойстве `Options` не будет включена опция `poAllowCommandText`.

Если команда SQL содержит параметры — потребуется помощь свойства

```
property Params: TParams;
```

Для того чтобы компонент `TClientDataSet` смог уточнить перечень параметров подключенного к нему набора данных сервера, следует обратиться к методу

```
procedure FetchParams;
```

Это приведет к последовательному вызову двух обработчиков событий:

```
property BeforeGetParams: TRemoteEvent;
property AfterGetParams: TRemoteEvent;
```

Отправка команды на выполнение осуществляет процедура

```
procedure Execute; virtual;
```

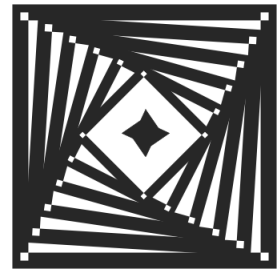
Выполнение команды SQL сопровождается двумя событиями:

```
property BeforeExecute: TRemoteEvent; //перед отправкой запроса
property AfterExecute: TRemoteEvent;  //после выполнения запроса
```

Событие `BeforeExecute()` вызывается в момент отправки запроса компоненту-провайдеру. Провайдер инициирует выполнение команды SQL на стороне сервера, собирает результаты выполнения запроса и возвращает информацию клиенту в событии `AfterExecute()`.

Резюме

Клиентский набор данных `TClientDataSet` — один из уникальных компонентов Delphi, предназначенных для разработки приложений БД. Благодаря неординарной идее создателей компонента, снабдивших набор данных локальным буфером, герой этой главы одинаково хорошо себя чувствует как в простейших проектах БД, так и в многоуровневых клиент-серверных базах данных.



Глава 22

Введение в технологию ADO

Технология *объектов данных ActiveX* (Microsoft ActiveX Data Objects, ADO) появилась на свет в стенах корпорации Microsoft и считается логическим развитием системы доступа к данным OLE DB. У ADO много достоинств, основное из них в том, что технология обеспечивает единый механизм доступа к разнотипным хранилищам данных. Приложение ADO обращается не к конкретной СУБД, а к объекту COM. *Многокомпонентная модель объектов* (Component Object Model, COM) описывает способ взаимодействия программ любого типа, программа-сервер предоставляет в распоряжение окружающих собственные службы, программа-клиент пользуется услугами доступных служб.

Для доступа к данным в рамках ADO (рис. 22.1) реализована коллекция интерфейсов COM-объектов, предназначенных для работы как с реляционными, так и с нереляционными наборами данных, включая майнфреймы, иерархические базы данных, текстовые, графические, географические данные, XML, e-mail и многое другое. Доступ к данным предоставляют драйверы, называемые *провайдерами*.

Приложение, ориентирующееся на технологию ADO, умеет работать как с локальными данными, так и со сложными платформами "клиент-сервер". Кроме того, применение ADO вполне возможно и в Web-приложениях. В рамках программной модели ADO достаточно просто создавать объекты, обеспечивающие не только элементарный функционал по редактированию данных в таблице, но и способных:

- поддерживать хранимые процедуры с объявлением как входных, так и выходных параметров;
- работать с разнообразными типами курсоров, создаваемых в СУБД;
- обслуживать составные поля;
- создавать многопоточные приложения для работы с данными в Web.

Еще один плюс технологии в том, что операционная система Windows по умолчанию включает программную поддержку ADO-приложений в виде пакета компонентов доступа к данным (Microsoft Data Access Components, MDAC). Таким образом, написав в среде Delphi приложение ADO, нам не потребуется инсталлировать дополнительные файлы.

ВНИМАНИЕ!

Компоненты ADO, входящие в состав Delphi 2010, предназначены для работы с ADO версии не ниже 2.1. На момент написания этой книги уже активно эксплуатировалась ADO версии 6.1. Справедливости ради нужно отметить, что в природе не существовало версий ADO 3.0, 4.0 и 5.0. ADO 6.0 пришел на смену ADO 2.8. Такой скачок в нумерации произошел вместе с выходом в свет Windows Vista, видимо, с появлением новой ОС в Microsoft перешли на новые правила идентификации версий. С появлением Windows 7 нумерация нормализовалась — в этой ОС предустановлен пакет ADO 6.1. Отметим, что с выходом .NET 3.5 в Microsoft сделан сильный крен в пользу развития пакета ADO .NET, который существенно отличается от исходной объектной модели ADO.

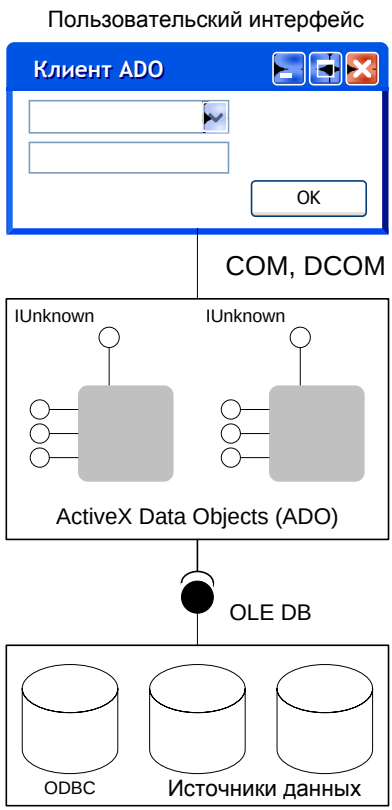


Рис. 22.1. Архитектура универсального доступа к данным ADO

Взаимодействие ADO и компонентов VCL

Объектная модель Microsoft ADO включает девять объектов (рис. 22.2):

- Объект соединения `Connection` отвечает за осуществление и поддержку подключения к хранилищу данных.
- Все исключительные ситуации `Error`, возникающие во время сессии, доступны благодаря коллекции `Errors`.

- Командный объект `Command` предназначен для отправки в адрес БД команд на языке SQL.
- Параметры `Parameter`, используемые в рамках команды SQL, хранятся в коллекции `Parameters`.
- Множество записей `Recordset` инкапсулирует в себе набор строк, полученный из таблицы БД.
- Доступ к отдельной записи обеспечивает объект `Record`.
- Доступ к отдельному полю записи `Field` осуществляется при посредничестве коллекции полей `Fields`.
- Объекты соединения, команда, параметры, записи и поля записей обладают свойствами `Property`. Все свойства хранятся в коллекции свойств `Properties`.
- Стоящий особняком объект `Stream` специализируется на доставке потоковых данных, таких как XML.

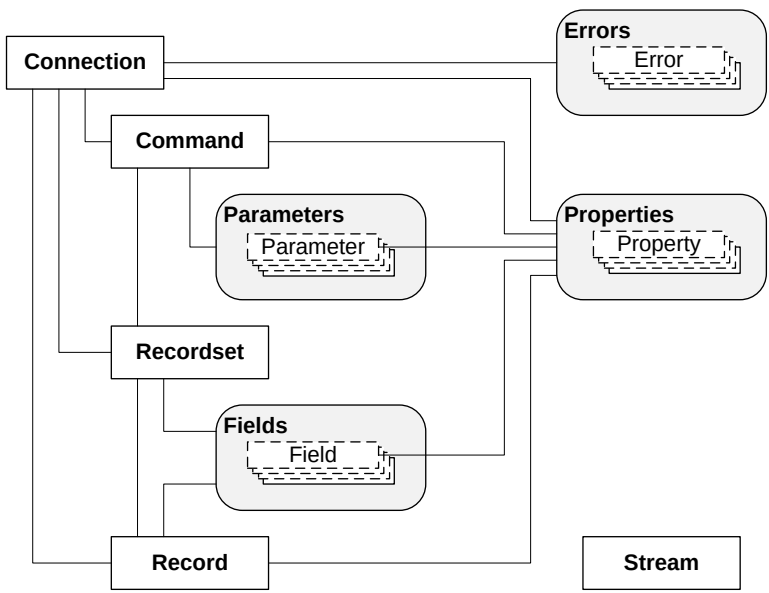


Рис. 22.2. Объектная модель ADO 6.1

Так как программисты Microsoft все объекты ADO построили на основе COM-технологии, то доступ к этим объектам предоставляется посредством соответствующих интерфейсов. Разработчики Delphi существенно упростили работу обычного программиста, инкапсулировав интерфейсы ADO в компоненты визуальной библиотеки. В результате в компонентах VCL интерфейсы COM объектов ADO представлены в виде классических для Delphi (а значит и удобных) свойств и методов.

Все элементы управления VCL, нацеленные на работу с объектами данных ADO, собраны на странице **dbGo** палитры компонентов Delphi XE. Основной про-

граммный код всех интересующих нас сегодня классов сосредоточен в модуле ADODB, а описания их интерфейсов — в ADOInt. Семерку компонентов ADO можно условно разделить на три категории (рис. 22.3):

1. Компоненты обеспечения централизованного доступа к данным — TADOConnection и TRDSConnection. Эти компоненты обычно используются при разработке сложных приложений, базирующихся на архитектуре клиент-сервер. Экземпляр класса TADOConnection отвечает за установку соединения и взаимодействие проекта с БД, все остальные компоненты dbGo пользуются его посредническими услугами.
2. Компоненты обслуживания данных в стиле Microsoft ADO — командный компонент TADOCommand и набор данных TADODataSet.
3. Компоненты, работающие с данными в классическом для программистов Delphi духе. Это компонент-таблица TADOTable, запрос TADOQuery и хранимая процедура TADOStoredProc.

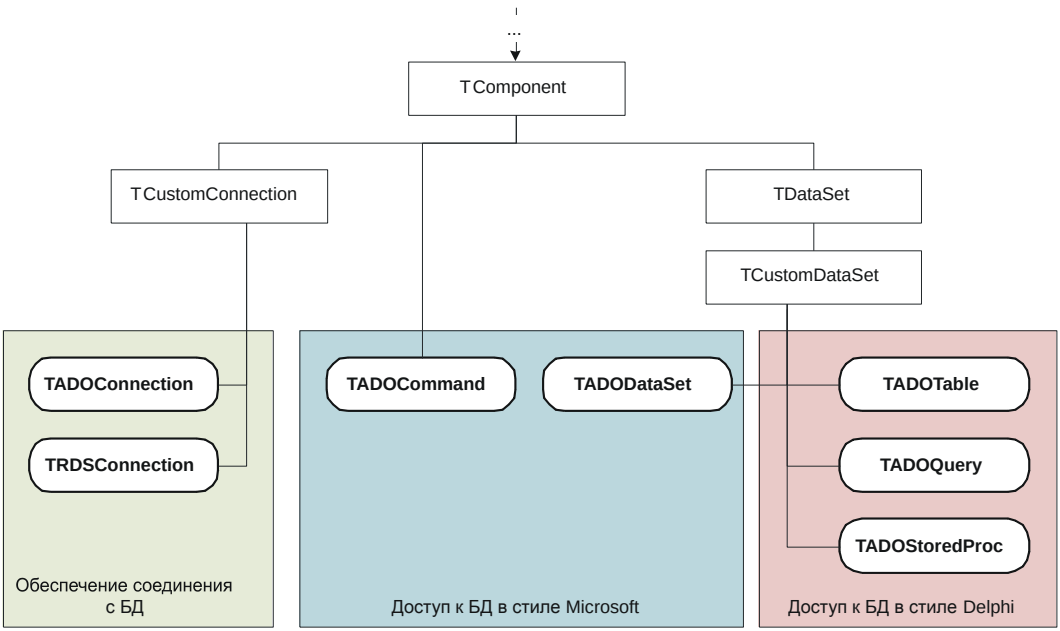


Рис. 22.3. Иерархия компонентов ADO в библиотеке VCL

Замечание

Компонент TRDSConnection с проектами БД связан косвенно. Элемент управления поддерживает функционал *службы удаленных данных* (Remote Data Services, RDS). Эта технология обеспечивает удаленный доступ к источникам данных ADO и, в первую очередь, предназначена для построения многоуровневых систем.

Строка соединения ADO

В главах, посвященных универсальному набору данных `TDataSet` и абстрактному полю `TField`, мы с вами уже применяли технологию ADO для демонстрации тех или иных особенностей работы с БД. Теперь настал час разобраться со всеми тонкостями организации взаимодействия между проектом Delphi и БД с помощью технологии ADO. Начнем с изучения строки соединения.

Строка соединения ADO представляет собой обычную текстовую строку, содержащую описание параметров соединения с интересующей нас БД. Соединительная строка — это последовательность аргументов и значений, разделяемых точкой с запятой. В качестве символа присвоения применяется знак равенства.

Например, так выглядит строка доступа к базе данных DEMODB, развернутой на Microsoft SQL Server с именем MSSQL_HOST:

```
Provider=SQLOLEDB.1;Persist Security Info=False;User ID=SA;Initial
Catalog=DEMODB;Data Source=MSSQL_HOST
```

Перечень аргументов, наиболее часто встречающихся в соединительной строке, приведен в табл. 22.1.

Таблица 22.1. *Аргументы соединительной строки*

Параметр	Описание
`Provider`	Имя поставщика интерфейса OLE DB. Если этот аргумент не определен, то по умолчанию задействуется провайдер Microsoft OLE DB; для ODBC — "MSDASQL"
`Persist Security Info`	Принимает значение `true` или `false`. В состоянии `true` уведомляет, что в строке соединения имеется конфиденциальная информация о порядке соединения (например, имя пользователя и пароль доступа), состояние `false` говорит о том, что такая информации отсутствует
`File name`	Название файла, содержащего информацию о соединении
`User ID`	Регистрационное имя пользователя по умолчанию
`Password`	Пароль пользователя
`Data Source`	Имя источника данных, например SQL-сервера
`Initial Catalog`	Каталог инициализации — имя базы данных
`Remote Provider`	Имя провайдера, применяемого на клиентской стороне соединения. Параметр необходим только при организации работы со службами удаленных данных RDS
`Remote Server`	Имя удаленного сервера, параметр задействуется только при работе с RDS
`URL`	Абсолютный URL-адрес

ЗАМЕЧАНИЕ

Строку соединения допускается хранить в специальном файле связи с данными (UDL-файле). Для создания такого файла достаточно сформировать пустой текстовый файл и заменить расширение txt на udl. Запуск такого файла автоматически вызовет редактор настройки соединительной строки.

Соединение с хранилищем данных, компонент *TADOConnection*

Созданный в корпорации Microsoft объект `Connection` (см. рис. 22.2) предназначен для организации соединения между приложением и БД. Объект позволяет поддерживать как простейшие соединения с однопользовательскими базами, так и работать с серьезными клиент-серверными СУБД. Интерфейсная часть объекта называется `_Connection`, интерфейс объявлен в модуле `ADOInt`. Интерфейс `Connection` поддерживается всеми компонентами линейки dbGo (за исключением `TRDSConnection`).

Разработчики Delphi спрятали интерфейс `_Connection` в недра `TADOConnection`, в результате компонент превратился в универсального специалиста по осуществлению соединения с источником данных ADO.

ЗАМЕЧАНИЕ

Как правило, в приложении применяется не более одного компонента `TADOConnection`, к которому, в свою очередь, подключаются все остальные элементы доступа к данным ADO.

Хотя основной задачей компонента считается установка соединения с базой данных ADO, но, ко всему прочему, `TADOConnection` умеет:

- регистрировать пользователя в СУБД;
- управлять транзакциями;
- отправлять в адрес СУБД различные команды (инструкции SQL);
- взаимодействовать с подчиненными компонентами (наборами данных ADO);
- получать от БД информацию о ее объектах и особенностях соединения.

ЗАМЕЧАНИЕ

Компоненты `TADODataSet`, `TADOTable`, `TADOQuery`, `TADOStoredProc`, `TADOCommand` способны поддерживать соединение с базой данных как при помощи `TADOConnection`, так и самостоятельно.

Установка соединения

Для описания параметров подключения к БД текст соединительной строки направляется в свойство

```
property ConnectionString: WideString;
```

Если в строке отсутствует информация о поставщике услуг OLE DB, то его необходимо определить в свойстве

```
property Provider: WideString;
```

По умолчанию роль провайдера исполняет MSDASQL.

Процесс соединения с указанной в соединительной строке базой данных инициируется посредством обращения к методу `Open()`. В простейшей ситуации процедура даже не требует параметров:

```
procedure Open; overload;
```

В перегружаемой версии процедуры `Open()` предусмотрена возможность передачи в адрес сервера имени и пароля пользователя. Но только для случая, когда эта информация не включена в соединительную строку:

```
procedure Open(const UserID: WideString; const Password: WideString);
overload;
```

Если вы полагаете, что при подключении к базе данных пользователь обязан пройти регистрацию и своими собственными руками ввести имя и пароль, то целесообразно установить в `true` свойство

```
property LoginPrompt: Boolean;
```

В результате перед соединением с базой вызывается стандартное диалоговое окно регистрации.

Для проверки факта подключения компонента `TADOConnection` к БД контролируют состояние свойства

```
property Connected: Boolean;
```

Значение `true` свидетельствует о наличии соединения с базой данных, обратное состояние `false` — констатирует факт, что соединение отсутствует. Свойство доступно как для чтения, так и для записи, поэтому оно вполне подходит для управления подключением или отключением приложения от БД.

Для исключения несанкционированного отключения от БД, даже в тех случаях, когда приложение не обращается ни к одному из наборов данных, переведите в `true` свойство

```
property KeepConnection: Boolean;
```

В противном случае клиентское приложение порвет контакт с сервером сразу после закрытия последнего активного набора данных, а это не всегда удобно, т. к. для повторного соединения потребуется вновь проходить утомительные этапы регистрации пользователя и подключения приложения к БД.

Различают два типа соединения: синхронное и асинхронное. За определение, каким образом наше приложение намерено работать с БД, отвечает свойство

```
property ConnectOptions: TConnectOption;
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

По умолчанию оно установлено в состояние `coConnectUnspecified`, в этом случае по возможности обеспечивается синхронное соединение. Другими словами, наш проект станет ожидать от СУБД реакцию на каждое свое действие. Принудительная организация асинхронного соединения (значение `coAsyncConnect`) реко-

мендуется только, когда производительность СУБД оставляет желать лучшего, и мы предполагаем, что у приложения могут возникнуть проблемы с получением данных.

Отнюдь не всегда факт успешного соединения свидетельствует о том, что пользователь приобрел абсолютные права на работу с данными. В реальности перечень возможностей пользователя может быть ограничен. Для того чтобы уточнить ограничения, следует обратиться к информационному свойству

```
property Mode: TConnectMode;
```

Восемь возможных состояний свойства приведены в табл. 22.2.

Таблица 22.2. *Режимы соединения* `TConnectMode`

Константа	Описание
`cmUnknown`	Тип доступа явно не указан, предоставляется установленный по умолчанию тип доступа
`cmRead`	Данные доступны только для чтения
`cmWrite`	Данные доступны только для записи
`cmReadWrite`	Права на чтение и запись данных
`cmShareDenyRead`	Монопольный доступ "только для чтения", остальные пользователи не смогут читать данные
`cmShareDenyWrite`	Монопольные права на запись, запрет на совместную запись
`cmShareExclusive`	Данные забраны в эксклюзивное пользование и недоступны для других
`cmShareDenyNone`	Монопольный доступ к источнику данных, другие пользователи не смогут открыть источник

Время ожидания компонента ответа от сервера баз данных после вызова метода `Open()` называется временем тайм-аута. Оно определяется свойством

```
property ConnectionTimeout: Integer;
```

и по умолчанию составляет 15 с. Если за установленный промежуток времени компонент `TADOConnection` не получит ответа от СУБД, то принимается решение о том, что попытка соединения завершилась неудачей. Поэтому (по истечении времени тайм-аута) компонент `TADOConnection` самостоятельно обратится к методу

```
procedure Cancel;
```

прекращающему попытки вызова сервера.

С процессом установки соединения связана строгая последовательность событий (табл. 22.3), позволяющая программисту практически на всех этапах помогать или мешать (кому что нравится) приложению получать доступ к БД.

Таблица 22.3. *Последовательность событий при соединении с набором данных*

№ п/п	Событие	Описание
1	**property** BeforeConnect: TNotifyEvent;	Вызывается перед началом соединения
2	**property** OnWillConnect: TWillConnectEvent;	Генерируется после отправки запроса на соединение
3	**property** OnLogin: TLoginEvent;	Регистрация пользователя
4	**property** OnConnect-Complete: TConnectErrorEvent;	Процесс соединения завершается
5	**property** OnInfoMessage: TInfoMessageEvent;	Вызывается после успешного соединения, в тот момент, когда провайдер возвращает дополнительную информацию о соединении
6	**property** AfterConnect: TNotifyEvent;	Возникает по окончании всего процесса соединения

В перечне событий из табл. 22.3 наибольший интерес представляет `OnWillConnect()`. Это событие вызывается в тот момент времени, когда провайдер уже ответил согласием на соединение, но физически соединение еще не установлено.

```
TWillConnectEvent = procedure(Connection: TADOConnection;
                    var ConnectionString, UserID, Password: WideString;
                    var ConnectOptions: TConnectOption;
                    var EventStatus: TEventStatus) of object;
```

Изменяя значения параметров события `OnWillConnect()`, мы получаем возможность внести последние редакторские правки в текст соединительной строки ADO. Здесь `Connection` — ссылка на компонент; `ConnectionString`, `UserID`, `Password` — соответственно строка соединения, имя пользователя и пароль; `ConnectOptions` — опции соединения (синхронное `coConnectUnspecified` и асинхронное `coAsyncConnect`); `EventStatus` — позволяет программисту определить статус операции, и если по какой-то причине в момент возникновения события `OnWillConnect()` вы вдруг захотите отказаться от соединения, присвойте этому параметру значение `esCancel`:

```
type   TEventStatus = (esOK, esErrorsOccured, esCantDeny,
                       esCancel, esUnwantedEvent);
```

Наиболее часто употребляемое значение — `esOK`, оно свидетельствует о том, что все хорошо, и мы готовы к подключению. Нужно отметить, что параметр `EventStatus` еще неоднократно нам встретится в других событиях, поэтому познакомьтесь с предложенным в табл. 22.4 описанием статуса события.

Таблица 22.4. *Описание статуса события* `TEventStatus`

Значение	Описание
`esOK`	Продолжаем выполнение операции
`esErrorsOccured`	Во время выполнения операции произошла ошибка или ряд ошибок
`esCantDeny`	Запрет на отмену текущей операции
`esCancel`	Отмена операции
`esUnwantedEvent`	Прекратить уведомления о последующих событиях до тех пор, пока продолжает выполняться текущая операция

После успешной регистрации пользователя генерируется событие `OnConnectComplete()`:

```
TConnectErrorEvent = procedure(Connection: TADOConnection;
                               Error: Error;
                               var EventStatus: TEventStatus) of object;
```

Здесь `Connection` — ссылка на компонент, осуществивший соединение. `Error` — зарегистрированный в ADO интерфейс, описывающий ошибку. По умолчанию во время обработки события параметр `EventStatus`, содержащий информацию о статусе события, принимает значение `esOK`, что свидетельствует о нормальном выполнении операции. Манипулируя другими значениями, программист приобретет некоторое влияние на процесс обработки события. Уже знакомый нам параметр `EventStatus` может принимать значения, приведенные в табл. 22.4.

Событие `OnInfoMessage()` позволяет программисту отреагировать на дополнительную информацию, возвращаемую провайдером клиентскому приложению после успешного соединения. Все параметры обработчика события вам уже знакомы:

```
TInfoMessageEvent = procedure(Connection: TADOConnection;
                              const Error: Error;
                              var EventStatus: TEventStatus) of object;
```

ВНИМАНИЕ!

Если в приложении планируется обрабатывать события, включающие параметр `Error` (например, `OnConnectComplete` и `OnInfoMessage`), то проверьте, чтобы к проекту был подключен модуль `ADOInt`, содержащий описания интерфейсов ADO.

Пример соединения без регистрации пользователя

Благодаря тому, что параметры соединения ADO содержатся в обычной текстовой строке, программист получает отличную возможность управлять процессом подключения к любому хранилищу данных, просто редактируя параметры соединительной строки. Предлагаю вашему вниманию пример, позволяющий пользователю выбирать расположение исходного файла с данными в момент старта приложения настольной БД.

Допустим, что в нашем распоряжении имеется однопользовательская база данных книжного магазина в формате Microsoft Access. В составе форм проекта Delphi имеется модуль данных `frmDM:TDataModule`, на котором размещен компонент `ADOConnection1 :TADOConnection`. Перечень других компонентов и форм проекта сейчас для нас несущественен, наша задача — предоставить пользователю возможность самостоятельно указать место расположения файла с БД.

Вполне очевидно, что для выбора файла БД в момент старта приложения следует выводить окно, уточняющее путь к файлу (в нашем примере — файл books.accdb в формате Microsoft Access 2007). Для этого дополняем наш проект новой формой (рис. 22.4) и размещаем на ее поверхности следующий перечень компонентов:

- Строка ввода `TEdit` для редактирования пути к файлу. Переименовываем компонент в `edFilePath`.
- Диалог открытия файла `OpenDialog1`. Для того чтобы компонент реагировал только на файлы Access, следует отредактировать отвечающее за фильтрацию файлов свойство `Filter:= Книжный магазин Access 2003|books.mdb|Книжный магазин Access 2007|books.accdb|Все файлы Microsoft Access|*.mdb; *.accdb`.
- Справа от строки ввода расположите кнопку `TButton` и переименуйте ее в `btnChangeFilePath`. Щелчок по кнопке пробудит от сна диалог открытия файла `OpenDialog1`.
- Два последних компонента — кнопки класса `TBitBtn`. Первую кнопку называем `btnOK` и ее свойство `Kind` установим в `bkOK`. Свойству `Kind` второй кнопки присвоим значение `bkCancel`.

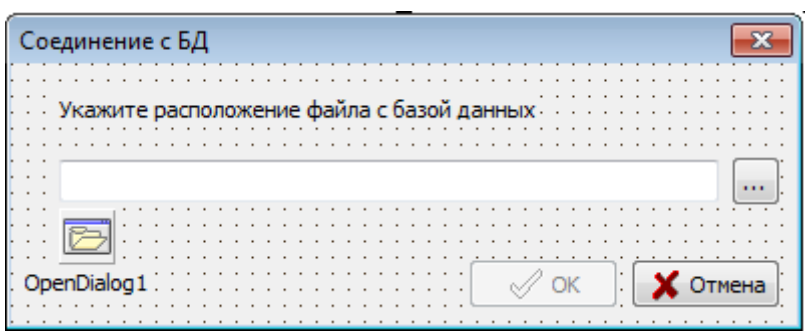


Рис. 22.4. Форма ввода пути к БД

Сохраняем только что созданную форму, предварительно переименовав ее в `frmConnectToDB`. Теперь напишем два простых обработчика события. Первое из событий вызывается щелчком по кнопке `btnChangeFilePath` (листинг 22.1).

Листинг 22.1. Изменение пути к файлу БД

```
procedure TfrmConnectToDB.btnChangeFilePathClick(Sender: TObject);
begin
```

```
if OpenDialog1.Execute then
   edFilePath.Text:=OpenDialog1.FileName;
end;
```

В рамках события `OnClick()` мы выводим на экран диалог открытия файла (в котором пользователь выберет имя файла с БД) и передаем имя и путь к файлу в строку `edFilePath`.

Второе событие отвечает за то, чтобы кнопка `btnOK` активизировалась только в случае, если текст в компоненте `edFilePath` действительно содержит верный путь к файлу. Эта предосторожность исключит ввод в строку ввода некорректных данных (листинг 22.2).

Листинг 22.2. Проверка наличия файла

```
procedure TfrmConnectToDB.edFilePathChange(Sender: TObject);
begin
btnOK.Enabled:=FileExists(edFilePath.Text);
end;
```

Теперь наступает самый интересный этап нашей работы. Как вы понимаете, вывод формы `frmConnectToDB`, спрашивающей у пользователя путь к файлу, должен предварять запуск всех остальных форм приложения, но при этом ее не стоит делать главной формой проекта. В таком случае целесообразно внести коррективы в головной модуль проекта (в моем примере он называется `books.dpr`).

Для вызова исходного текста головного модуля на экран выберите пункт меню **Project | View Source** (Проект | Вид источника). По умолчанию код модуля выглядит так, как в листинге 22.3.

Листинг 22.3. Исходный код головного модуля приложения

```
program books;

uses
  Forms,
  main in 'main.pas' {frmMain},
  dm in 'dm.pas' {frmDM: TDataModule},
  ConnetcToDB in 'ConnetcToDB.pas' {frmConnectToDB},
  {ДРУГИЕ МОДУЛИ ПРОЕКТА};

{$R *.res}

begin
  Application.Initialize; //инициализация приложения
  Application.MainFormOnTaskbar := True;
```

```
  Application.CreateForm(TfrmMain, frmMain); //создание главной формы
  Application.CreateForm(TfrmDM, frmDM); //создание модуля данных
 {СОЗДАНИЕ ДРУГИХ ФОРМ ПРОЕКТА}
  Application.Run; //запуск приложения
end.
```

В списке `uses` перечисляется перечень используемых в проекте программных модулей и в составном операторе `begin..end` осуществляется инициализация и старт приложения и автоматически создаваемых форм проекта.

Нам предстоит внести некоторые доработки в главный модуль проекта. Во-первых, добавляем в список используемых модулей `uses` ссылки на модули `SysUtils`, `Registry` и `Controls`. Во-вторых, создаем функцию `ConnectToDB()`, она возьмет на себя обязанности по вызову только что разработанной нами формы настройки соединения с базой данных. В-третьих, внесем незначительные изменения в процесс создания форм проекта (листинг 22.4).

Листинг 22.4. Изменения в коде головного модуля приложения

```
program books;

uses
  Forms, SysUtils, Registry, Controls,
  main in 'main.pas' {frmMain},
  dm in 'dm.pas' {frmDM: TDataModule},
  ConnetcToDB in 'ConnetcToDB.pas' {frmConnectToDB},
  {ДРУГИЕ МОДУЛИ ВХОДЯЩИЕ В ПРОЕКТ};

{$R *.res}
{ФУНКЦИЯ ПОДКЛЮЧЕНИЯ К БД}
Function CoonectToDB:boolean;
 var RI:TRegIniFile; //переменная для работы с реестром
begin
  RI:=TRegIniFile.Create('Software'); //обращаемся к реестру
{!!! Переменная FilePath объявляется в секции VAR модуля данных frmDM!!!}
 FilePath:=RI.ReadString('bookshop','path','');
 frmConnectToDB:=TfrmConnectToDB.Create(application);
 if FileExists(FilePath)=true then
               frmConnectToDB.edFilePath.Text:=FilePath;
    if frmConnectToDB.ShowModal=mrOK then //если нажата кнопка "OK"
    begin
     FilePath:=frmConnectToDB.edFilePath.Text;//читаем путь
     Result:=TRUE; //путь к файлу существует и находится в FilePath
    end
```

```
    else Result:=False;
    frmConnectToDB.Release; //уничтожаем форму выбора файла
 RI.Free; //освобождаем реестр
end;

begin
  Application.Initialize; //инициализация приложения
  Application.MainFormOnTaskbar := True;
  if CoonectToDB=true then //если соединение корректно, то
     begin                 //создаем все формы и стартуем приложение
      Application.CreateForm(TfrmMain, frmMain);
      Application.CreateForm(TfrmDM, frmDM);
      {СОЗДАНИЕ ДРУГИХ ФОРМ ПРОЕКТА}
      Application.Run;
     end
     else Application.Terminate; //если соединение не корректно
end.
```

Обсудим функцию `ConnectToDB()` подробнее. Задача функции заключается в запросе у пользователя пути к файлу с БД и передаче полученной строки компоненту соединения `ADOConnection1`. Чтобы избавить пользователя от каждодневного рутинного поиска файла с данными, сразу после старта функция обращается к системному реестру Windows. Для этого мы создаем экземпляр класса `TRegIniFile` и готовим его к чтению текстовых данных из подраздела реестра `HKEY_CURRENT_USER\Software\bookshop`, в предположении, что там в текстовом параметре `path` содержится информация о пути к файлу с БД. Полученная с помощью метода `ReadString()` текстовая строка передается в глобальную переменную `FilePath`, объявленную в разделе `var` модуля данных `frmDM`. После перечисленных действий функция создает описанный двумя страницами ранее экземпляр формы `frmConnectToDB` и выводит его на экран в модальном режиме. Пользователь уточняет путь к файлу и нажимает кнопку **OK** на форме. Если все подготовительные операции завершились корректно — наша функция возвращает `true`.

Вызов функции `ConnectToDB()` осуществляется внутри составного оператора `begin..end` основного кода программы сразу после инициализации приложения. Главная форма проекта `frmMain`, модуль данных `frmDM` подлежат созданию, только если функция возвратит (свидетельствующее о наличии файла с данными) значение `true`. В противном случае работа приложения прекращается.

Для завершения проекта нам остается сделать два последних штриха: создать строку соединения ADO и научить проект сохранять в реестре путь к файлу с данными. Эти задачи мы решим в модуле данных `frmDM`.

Работу с `frmDM` начинаем с объявления глобальной переменной `FilePath`. Именно она использовалась для хранения пути к файлу с данными. Затем воспользуемся двумя обработчиками событий модуля данных. Событие `OnCreate()` возни-

кает в момент создания модуля данных, здесь мы заполним основные параметры соединительной строки ADO. Событие `OnDestroy()` вызывается при завершении работы модуля данных, оно нам пригодится для сохранения в реестре пути к базе данных (листинг 22.5).

Листинг 22.5. События создания и уничтожения модуля данных приложения

```
var
  frmDM: TfrmDM;
  FilePath:string; //путь к базе данных
implementation

{$R *.dfm}
procedure TfrmDM.DataModuleCreate(Sender: TObject);
begin
{формируем соединительную строку}
ADOConnection1.ConnectionString:='Provider=Microsoft.ACE.OLEDB.12.0;
                  Data Source='+FilePath+';Persist Security Info=False';
ADOConnection1.Connected:=True;
end;

procedure TfrmDM.DataModuleDestroy(Sender: TObject);
var RI:TRegIniFile;
begin
{сохраняем путь к файлу БД в реестре}
  RI:=TRegIniFile.Create('Software');
  RI.WriteString('bookshop','path',FilePath);
  RI.Free;
end;
```

Проект завершен. Предложенное решение может оказаться полезным при работе не только с настольными СУБД, но и при разработке клиент-серверных проектов БД.

Регистрация пользователя

Одним из составных элементов безопасности любой СУБД выступает процесс регистрации пользователей. Перед предоставлением клиентскому приложению услуг сервер запрашивает имя и пароль пользователя и сверяет полученные данные с хранящимся в своем системном каталоге списком учетных записей. В случае совпадения контрольных данных СУБД предоставляет клиентскому приложению доступ к базе данных, но в строгом соответствии с определенными для пользователя правами. Таким образом, рядовой пользователь вряд ли получит разрешение на ре-

дактирование данных, а уж тем более не сможет добраться до конфиденциальной информации.

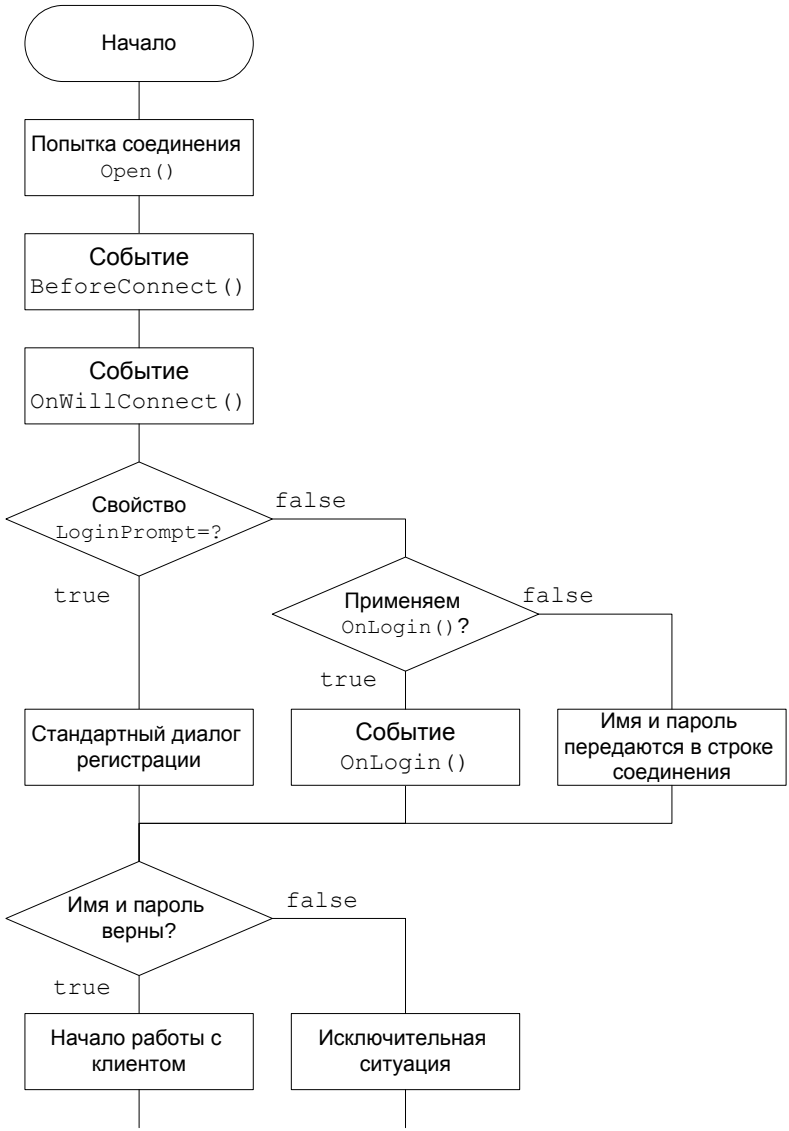


Рис. 22.5. Фрагмент блок-схемы регистрации пользователя

В классе `TADOConnection` предусмотрен механизм, снимающий рутинную задачу написания килобайтов кода, управляющих регистрацией пользователя. В простейшем случае программист может положиться на умение компонента самостоятельно вызывать диалоговое окно регистрации и отправлять учетные данные в адрес СУБД.

Алгоритм поведения класса (рис. 22.5) определяется состоянием свойства

```
property LoginPrompt: Boolean; //по умолчанию true
```

По умолчанию элемент `TADOConnection` настроен на автоматический вызов стандартного окна регистрации (`LoginPrompt=true`), в полях которого пользователь укажет свое имя и пароль доступа.

Если стандартная процедура регистрации нас чем-то не устраивает, то установите `LoginPrompt` в состояние `false`. Теперь стандартный диалог регистрации перестает подлежать вызову, но это не означает, что SQL-сервер больше не захочет аутентифицировать пользователя. Просто теперь управление процессом регистрации на клиентской стороне возлагается на программиста. Самое тривиальное развитие событий сводится к передаче имени и пароля через параметры соединительной строки. Напомню (см. табл. 22.1), что имя пользователя передается в параметр `User ID`, а пароль — в `Password`:

```
ADOConnection1.ConnectionString:=Provider=SQLOLEDB.1;Password=MYPASS;
Persist Security Info=True;User ID=SA;Initial Catalog=master;
Data Source=MSSQLSERVER_HOST
```

Но хранение учетных данных в исполняемом коде — далеко не самое грамотное решение, поскольку даже для начинающего хакера не составляет особого труда их оттуда "выудить". Кроме того, исполняемый модуль с "намертво" зашитыми в него паролями теряет свою работоспособность после изменения учетных записей на стороне SQL-сервера. Поэтому описанный пример — это вариант временного кода, который допустимо применять только на стадии разработки проекта, чтобы не регистрироваться на сервере каждый раз при старте приложения.

На рис. 22.5 представлен фрагмент блок-схемы регистрации пользователя на сервере БД. Обычно вопрос регистрации решается в рамках события `OnLogin()`. Мы несколько изменим традицию. Приведенный в листинге 22.6 пример выводит диалог ввода имени и пароля несколько раньше — в момент возникновения события `OnWillConnect()`. Перед вами листинг модуля данных `dm.pas`, демонстрирующий порядок регистрации пользователя в Microsoft SQL Server. В коде есть два обработчика событий. Первое по очередности событие вызывается в момент создания модуля данных `OnCreate()`, второе событие `OnWillConnect()` генерируется перед началом соединения.

Листинг 22.6. Модуль данных с кодом регистрации пользователя

```
unit dm;

interface

uses SysUtils, Classes, DB, ADODB;

type
  TfrmDM = class(TDataModule)
    ADOConnection1: TADOConnection;
```

```
    procedure DataModuleCreate(Sender: TObject);
    procedure ADOConnection1WillConnect(Connection: TADOConnection;
              var ConnectionString, UserID, Password: WideString;
              var ConnectOptions: TConnectOption;
              var EventStatus: TEventStatus);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

var frmDM: TfrmDM;

implementation
uses DBLogDlg,ComObj, Variants;
{$R *.dfm}
{создание модуля данных}
procedure TfrmDM.DataModuleCreate(Sender: TObject);
var  i:integer;
begin
ADOConnection1.ConnectionString:='Provider=SQLOLEDB.1;
                Persist Security Info=False;User ID=SA;
                Initial Catalog=master;Data Source=MSSQLSERVER_HOST';
{отключаем автоматический вызов диалога регистрации}
ADOConnection1.LoginPrompt:=false;
try
ADOConnection1.Open;
except on EOLEException do {Обработка вероятной исключительной ситуации}
     for i:=0 to ADOConnection1.Errors.Count-1 do
       begin
       //выводим сообщения о всех возникших ошибках
       MessageBox(Application.Handle,
       pWideChar(ADOConnection1.Errors.Item[i].Description),
       pWideChar('Ошибка!'),MB_OK+MB_ICONWARNING);
       end;
end;
end;

{событие, предваряющее соединение с БД}
procedure TfrmDM.ADOConnection1WillConnect(Connection: TADOConnection;
  var ConnectionString, UserID, Password: WideString;
  var ConnectOptions: TConnectOption; var EventStatus: TEventStatus);
var aUserID, aPassword: String;
```

```
begin
if LoginDialog('HostName',aUserID,aPassword)=true then
  begin
    UserID:=aUserID;
    Password:=aPassword;
    EventStatus:=esOK;
  end
 else EventStatus:=esCancel; //отказ от регистрации
end;

end.
```

В событии `OnCreate()` мы описываем основные параметры строки соединения, но не включаем в строку пароль пользователя. Затем, устанавливая свойство соединения `ADOConnection1.LoginPrompt` в состояние `false`, отключаем режим регистрации по умолчанию. И пытаемся установить соединение с СУБД, обратившись к методу `Open()`. Последнее действие приводит к генерации события `OnWillConnect()` у компонента `ADOConnection1`. В рамках этого события вызываем функцию `LoginDialog()`, выводящую на экран диалог регистрации, и направляем в процедуру обработки события значения имени и пароля пользователя. Если пользователь отказывается от регистрации (функция возвращает значение `false`), мы отменяем операцию регистрации.

Для максимального удобства программиста, специализирующегося на создании приложений баз данных, в модуле `DBLogDlg` объявлен ряд методов, формирующих наиболее часто применяемые диалоги доступа пользователя к БД (табл. 22.5).

***Таблица 22.5.** Функции вызова диалога регистрации пользователя*

Функция	Описание
`function RemoteLoginDialog( var AUserName, APassword: string): Boolean;`	Метод регистрации пользователя `AUserName` с паролем `APassword` на сервере. Если пользователь отказался от регистрации и нажал кнопку **Cancel**, то метод возвращает значение `false`
`function LoginDialog( const ADatabaseName: string; var AUserName, APassword: string): Boolean;`	Метод вызывает стандартное диалоговое окно регистрации пользователя в базе данных `ADatabaseName`. Пользователь вводит свое имя и пароль, эти значения передаются в переменные `AUserName`, `APassword`
`function LoginDialogEx( const ADatabaseName: string; var AUserName, APassword: string; NameReadOnly: Boolean): Boolean;`	Расширенная версия метода `LoginDialog()`. Установив параметр `NameReadOnly` в `true`, программист может запретить пользователю изменять имя

Зачастую, вместо невыразительного стандартного диалога регистрации, лучше создавать свой собственный вариант окна ввода имени и пароля. Для этого достаточно добавить к рассмотренному ранее проекту еще одну форму и разместить на ней ряд элементов управления (рис. 22.6):

- две строки ввода (компоненты класса `TEdit` или `TLabeledEdit`). Первую строку переименуем в `edUser`, она отвечает за ввод имени пользователя. Вторую строку назовем `edPassword`, т. к. компонент предназначен для ввода пароля, присвойте его свойству `PasswordChar` символ звездочки "*";
- две кнопки `TBitBtn`. Свойству `Kind` первой присвойте значение `bkOK`, второй — `bkCancel`.

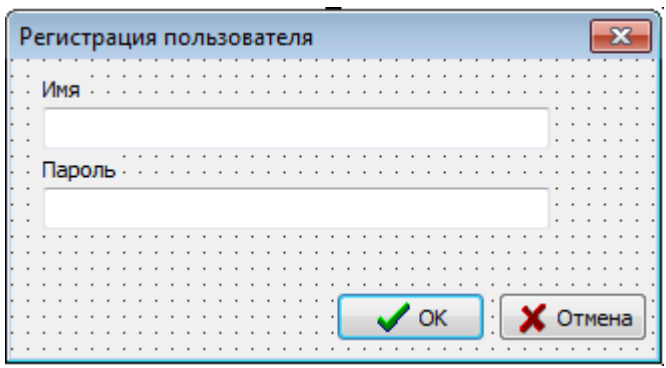


Рис. 22.6. Пользовательская форма регистрации

Переименуйте форму в `frmUserLogin` и обязательно удалите ее из списка автоматически создаваемых форм проекта. С этой целью выберите пункт меню **Project | Options**, в появившемся окне найдите узел **Forms** и перенесите форму `frmUserLogin` из списка **Auto-create forms** в список доступных форм **Available forms**. Сохраните изменения в проекте и внесите правку в процедуру обработки события `OnWillConnect()` компонента `ADOConnection1` (листинг 22.7).

Листинг 22.7. Подключение пользовательской формы регистрации

```
procedure TfrmDM.ADOConnection1WillConnect(Connection: TADOConnection;
                var ConnectionString, UserID, Password: WideString;
                var ConnectOptions: TConnectOption;
                var EventStatus: TEventStatus);
var frmUserLogin: TfrmUserLogin;
begin
frmUserLogin:=TfrmUserLogin.Create(application);
if frmUserLogin.ShowModal=idOK then
   begin
    UserID:=frmUserLogin.edName.Text;
```

```
    Password:=frmUserLogin.edPassword.Text;
    EventStatus:=esOK;
   end
   else EventStatus:=esCancel;
frmUserLogin.Release;
end;
```

Теперь вместо стандартного англоязычного диалогового окна пользователя встретит приветливая русскоязычная форма.

Разрыв соединения

Для разрыва подключения к БД воспользуйтесь методом `Close()` или установите в `false` свойство `Connection`. При отключении от набора данных последовательно генерируются три события (табл. 22.6).

***Таблица 22.6.** Последовательность событий при разрыве соединения с набором данных*

	Событие	Описание
1.	`property BeforeDisconnect: TNotifyEvent;`	Вызывается перед разрывом соединения

2.	`property OnDisconnect: TDisconnectEvent;`
	
	`TDisconnectEvent = procedure(Connection: TADOConnection; var EventStatus: TEventStatus) of object;`
		Осуществляется разрыв. Подробности статуса события передаются через параметр `EventStatus` (см. табл. 22.4)

3.	`property AfterDisconnect: TNotifyEvent;`	Генерируется после разрыва

События, связанные с разрывом соединения, обычно задействуют, чтобы обеспечить сохранение всех изменений в БД, сделанных пользователем.

Информирование о соединении

Для выяснения номера установленной на компьютере версии технологии ADO, обращаемся к свойству

```
property Version: WideString; //только для чтения
```

При этом вовсе не обязательно, чтобы элемент управления `TADOConnection` состоял в соединении с какой-либо базой данных. Благодаря свойству `Version` программист приобретает возможность проверять наличие пакета ADO на компьютере

пользователя. В ОС класса Windows XP\Server 2003 была предустановлена ADO 2.8, в Windows Vista\Server 2008 — 6.0, в Windows 7 — 6.1.

Текущее состояние соединения описывается свойством

```
property State: TObjectStates; //только для чтения
```

Перечень возможных состояний, описываемых типом данных `TObjectStates`, будет рассмотрен несколько позднее при обсуждении интерфейса `Recordset`.

Весьма информативен метод `OpenSchema()`, если название процедуры перевести дословно, то это "открытие схемы". Напомню, что в терминологии реляционной модели данных под схемой понимают взаимосвязанную совокупность объектов, составляющих базу данных. В перечень объектов схемы входят таблицы, домены, индексы, хранимые процедуры и т. п. Заголовок метода выглядит следующим образом:

```
procedure OpenSchema(const Schema: TSchemaInfo;
                     const Restrictions: OleVariant;
                     const SchemaID: OleVariant; DataSet: TADODataSet);
```

Первый параметр метода `Schema: TSchemaInfo` определяет, какую именно информацию мы рассчитываем получить. Например, передав в параметр константу `siIndexes`, мы приобретем информацию об индексах БД, `siTables` — таблицах, `siViews` — представлениях. Второй параметр `Restrictions` представляет собой массив ограничений на запрашиваемую информацию. Третий по счету параметр `SchemaID` применяется только в ситуации, когда глобальный уникальный идентификатор GUID провайдера данных не определен в стандартной спецификации OLE DB. Как правило, такая необходимость возникает, когда аргумент `Schema` установлен в `siProviderSpecific`. И, наконец, последний параметр `DataSet` — ссылка на элемент управления класса `TADODataSet`, в его множество записей `Recordset` будут направлены результаты выполнения процедуры. Полученные данные программист может обрабатывать по своему усмотрению, например, просто отобразить их в сетке `TDBGrid`.

Апробируем приобретенные знания на практике. Разместите на форме нового проекта (рис. 22.7) следующие компоненты:

- соединение ADO — `ADOConnection1`;
- набор данных ADO — `ADODataSet1`;
- источник данных — `DataSource1`;
- сетку — `DBGrid1`.

Перейдите к инспектору объектов. С помощью свойства `Connection` элемента управления `ADODataSet1` свяжите компонент с `ADOConnection1`. Затем выберите источник данных `DataSource1` и, воспользовавшись свойством `DataSet`, соедините его с `ADODataSet1`. В заключение подключите сетку `DBGrid1` к источнику данных `DataSource1`. На этом манипуляции с инспектором объектов завершились.

Переходим к программированию. На свое усмотрение создайте строку соединения с интересующей вас базой данных. В этом примере мы соединяемся с системной базой данных Master из состава СУБД Microsoft SQL Server, размещенной на сервере MSSQLSERVER_HOST. Для этого опишем событие создание формы так, как предложено в листинге 22.8.

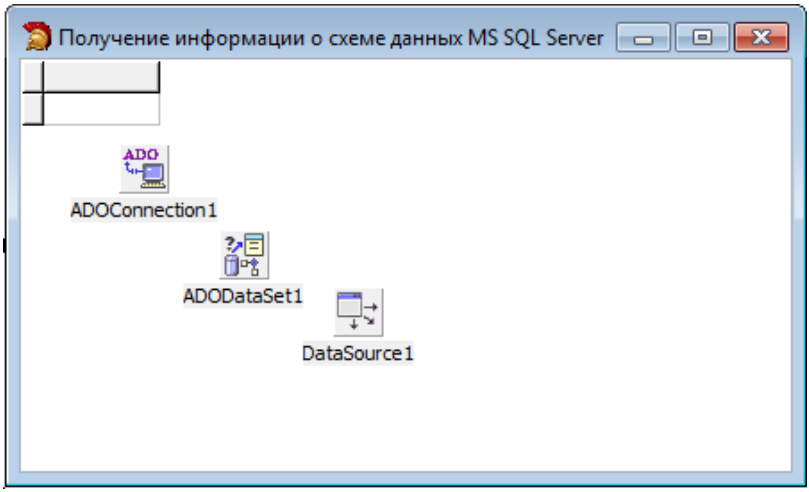


Рис. 22.7. Сбор данных о системной БД Microsoft SQL Server

Листинг 22.8. Получение системной информации о БД

```
procedure TForm1.FormCreate(Sender: TObject);
begin
ADOConnection1.ConnectionString:='Provider=SQLOLEDB.1;
                  Persist Security Info=False;User ID=sa;
                  Initial Catalog=Master; Data Source=MSSQLSERVER_HOST';
ADOConnection1.Open;
ADOConnection1.OpenSchema(siTables, EmptyParam, EmptyParam, ADODataSet1);
end;
```

Если все сделано безошибочно, то в результате запуска проекта мы получим данные обо всех таблицах системной БД Master.

На основе процедуры `OpenSchema()` построен ряд справочных методов, способных снабдить нас данными о таблицах, полях таблиц и хранимых процедурах базы данных.

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: String; List: TStrings);
procedure GetProcedureNames(List: TStrings);
```

Результаты выполнения процедур передаются в список строк `List`.

Отправка команд

В терминологии ADO под командой понимается инструкция на языке SQL. Например, в качестве команды можно рассматривать операции удаления, вставки и редактирования строк набора данных.

Для отправки провайдеру SQL-инструкции компонент `TADOConnection` имеет право пойти двумя путями:

- воспользоваться услугами любого подключенного к нему компонента линейки ADO Delphi, инкапсулирующего интерфейс `_Command`;
- передать команду, вызвав метод `Execute()`.

Замечание

Для хранения и передачи провайдеру наиболее часто вызываемых команд в VCL разработан специальный элемент управления — `TADOCommand`.

Для обеспечения эффективного взаимодействия компонента `TADOConnection` и подключенных к нему командных компонентов в классе `TADOConnection` объявлен ряд свойств. Число подключенных компонентов, специализирующихся на отправке команд, хранится в свойстве

```
property CommandCount: Integer; // только для чтения
```

Список связанных с `TADOConnection` активных командных компонентов находится в массиве

```
property Commands[Index: Integer]: TADOCommand;
```

Если следует выполнить все команды, то достаточно перебрать элементы массива:

```
for i := 0 to (ADOConnection1.CommandCount – 1) do
                    ADOConnection1.Commands[i].Execute;
```

Если нам требуется передать в базу данных какую-то единичную инструкцию SQL, то вместо услуг командных компонентов, целесообразнее сформировать команду динамически и отправить ее, воспользовавшись одной из вариаций перегружаемого метода `Execute()`.

```
function Execute(const CommandText: WideString; //инструкция SQL
             const CommandType: TCommandType = cmdText; {тип команды}
          const ExecuteOptions: TExecuteOptions = [] {опции выполнения}
             ): _RecordSet; overload;//возвращаем интерфейс Recordset

procedure Execute(const CommandText: WideString;//инструкция SQL
                  var RecordsAffected: Integer; //обработано строк
                  ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]
               ); overload;
```

В обоих случаях единственный обязательный параметр — константа `CommandText`, содержащая инструкцию SQL (листинг 22.9).

Листинг 22.9. Пример работы с текстом команды

```
var SQLText : String;
begin
```

```
  SQLText := 'CREATE TABLE Peoples
             (FName Char(30), LName Char(30), BDay Date)';
  SQLConnection1.Execute(SQLText, nil, nil);
end;
```

С выполнением команды связаны два события.

Перед тем как команда будет выполнена, генерируется событие:

```
property OnWillExecute: TWillExecuteEvent;
TWillExecuteEvent = procedure(Connection: TADOConnection;
           var CommandText: WideString;//текст команды
           var CursorType: TCursorType;//тип курсора
           var LockType: TADOLockType;//особенности блокировки транзакции
           var CommandType: TCommandType;//тип команды
           var ExecuteOptions: TExecuteOptions;//опции выполнения
           var EventStatus: TEventStatus; //индикатор выполнения операции
           const Command: _Command;       //интерфейс командного объекта
           const Recordset: _Recordset    //интерфейс множества записей
           ) of object;
```

После успешного запуска команды вызывается событие:

```
property OnExecuteComplete: TExecuteCompleteEvent;
TExecuteCompleteEvent = procedure(Connection: TADOConnection;
          RecordsAffected: Integer;
          const Error: Error; //интерфейс ошибки
          var EventStatus: TEventStatus;//индикатор выполнения операции
          const Command: _Command;  //интерфейс командного объекта
          const Recordset: _Recordset//интерфейс множества записей
          ) of object;
```

По параметру `RecordsAffected` программист может судить о том, сколько строк набора данных было охвачено командой; наличие ошибок при выполнении команды отразится на параметре `Error`. При необходимости переменная `EventStatus` позволит обработать ошибку. Если для отправки инструкции использовался командный компонент ADO, то ссылку на него мы обнаружим в свойстве `Command`.

Внимание!

Ряд параметров событий класса `TADOConnection` предназначен для взаимодействия с реальными объектами ADO, а не с инкапсулирующими их компонентами библиотеки VCL. К таким параметрам относятся: `Error` — специализированный интерфейс ошибки; `Command` — интерфейс командного объекта; `RecordSet` — интерфейс множества записей.

Управление подчиненными компонентами dbGo

Подключившись к хранилищу данных, соединительный компонент предоставляет всем остальным компонентам dbGo свои услуги по доступу к БД. Компоненты класса `TADOTable`, `TADOQuery` и т. д. для подключения к компоненту `TADOConnection` используют свое свойство `Connection`. С этого момента соединительный компонент берет в свои руки управление процессом обмена данными между вашим приложением и СУБД.

Удостоившись звания "главного" компонента ADO, элемент управления `TADOConnection` приобретает некоторые права по отношению к своим подчиненным собратьям. Во-первых, соединительный компонент знает количество подключенных к нему наборов данных (компонентов-потомков `TCustomADODataSet`) и способен обратиться к любому из них благодаря свойствам:

```
property DataSetCount: Integer;
property DataSets [Index: Integer]: TCustomADODataSet;
```

Во-вторых, соединительный компонент обладает аналогичной информацией о применяемых в проекте командных объектах `TADOCommand`:

```
property CommandCount: Integer;
property Commands [Index: Integer]: TADOCommand;
```

Транзакции

Для старта новой транзакции предназначена функция

```
function BeginTrans: Integer;
```

Метод возвращает целочисленное значение, соответствующее уровню вложения транзакции. Если запуск новой транзакции осуществился без заминок, то это приведет к установке в `true` свойства

```
property InTransaction: Boolean;
```

Значение `true` свидетельствует о том, что `TADOConnection` инициировал транзакцию и эта транзакция находится в стадии выполнения. С окончанием транзакции свойство вернется в состояние `false`. Сразу после успешного запуска новой транзакции происходит событие

```
property OnBeginTransComplete: TBeginTransCompleteEvent;
TBeginTransCompleteEvent = procedure(Connection: TADOConnection;
                           TransactionLevel: Integer; const Error: Error;
                           var EventStatus: TEventStatus) of object;
```

Параметр `EventStatus` информирует нас, насколько успешно прошел процесс запуска транзакции (см. табл. 22.4). Если операция выполнилась корректно, то в параметре окажется значение `esOK`.

Для сохранения результатов выполнения транзакции программист вызывает процедуру

```
procedure CommitTrans;
```

Сразу после сохранения транзакции возникает событие

```
property OnCommitTransComplete: TConnectErrorEvent;
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
                               var EventStatus: TEventStatus) of object;
```

Если в процесс выполнения транзакции вкралась ошибка, то для ее отмены воспользуйтесь методом

```
procedure RollbackTrans;
```

Откат транзакции сопровождается событием

```
property OnRollbackTransComplete: TConnectErrorEvent;
```

Если разрабатываемая база данных подлежит эксплуатации в многопользовательском режиме, то особое внимание нужно уделить вопросам обеспечения параллельной обработки данных. Накал борьбы клиентских приложений за записи таблиц можно снизить, установив соответствующий уровень изоляции транзакций:

```
property IsolationLevel: TIsolationLevel;
```

Уровень разграничения транзакций определяет возможность одновременного совместного доступа пользователей к одним и тем же данным. В табл. 22.7 приведены допустимые значения уровней изоляции транзакций в ADO. Блокировки упорядочены по степени возрастания пессимистических настроений. Другими словами, каждая последующая строка все более и более затрудняет параллельную работу нескольких пользователей с одними и теми же данными, но зато снижает вероятность появления конфликтов.

Таблица 22.7. *Возможные уровни изоляции транзакции* `TIsolationLevel`

Уровень изоляции	Описание
`ilUnspecified`	Сервер может остаться в любом режиме изоляции транзакций, т. к. приложение не предъявляет к СУБД каких-либо особенных требований
`ilChaos`	Изменения, сделанные транзакцией с более высоким уровнем изоляции, не могут быть переписаны текущей транзакцией
`ilReadUncommitted` или `ilBrowse`	*Грязное чтение*, допустимо чтение данных незавершенных транзакций
`ilCursorStability` или `ilReadCommitted`	*Исключение грязного чтения*, данные доступны только после завершения транзакции
`ilRepeatableRead`	*Повторяющееся чтение*, изменения, произведенные другими транзакциями к предварительно прочитанным данным, не видны, пока не будет осуществлено повторное чтение данных
`ilSerializable` или `ilIsolated`	*Полная изоляция*, максимальная степень изоляции транзакций

Процесс инициализации очередной транзакции можно автоматизировать. Для этого предназначено свойство

```
property Attributes: TXactAttributes; //по умолчанию автоматизация отключена
```

В свойство может устанавливаться любая комбинация флагов из множества TXactAttributes:

```
TXactAttributes = set of TXactAttribute;
type TXactAttribute = (xaCommitRetaining, xaAbortRetaining);
```

Наличие флагов TXactAttributes влияет на процесс инициализации очередной транзакции после сохранения или отката текущей транзакции. Флаг xaCommitRetaining определяет, что сохранение изменений, сделанных корректно завершенной транзакцией, автоматически вызывает следующую транзакцию. Наличие флага xaAbortRetaining служит признаком того, что после отката сбойной транзакции вновь предпринимается повторная попытка ее выполнения.

ВНИМАНИЕ!

При назначении атрибутов соединения необходимо ознакомится с документацией провайдера ADO, иначе можно установить недопустимые флаги.

Кэширование

Полученный в результате выполнения запроса набор строк может обрабатываться на стороне сервера или на стороне клиента. Предпочтительный для нас режим обработки результирующего набора определяется свойством

```
property CursorLocation: TCursorLocation; //по умолчанию clUseClient
type TCursorLocation = (clUseServer, clUseClient);
```

При включении режима clUseClient запрошенные из хранилища данных записи кэшируются в клиентском приложении и обслуживаются имеющимися в его распоряжении методами. Как правило, возможности клиентского курсора значительно гибче, чем у сервера, и поддерживают расширенные возможности фильтрации и сортировки данных. Но все выгоды клиентского курсора сводятся к нулю, если мы работаем с большими блоками данных на слабой клиентской машине. В этом случае отправка необработанной информации клиенту нерациональна, т. к. пересылка и обработка данных займет много времени и ресурсов. Поэтому при обработке больших массивов записей целесообразнее выбирать серверный курсор (clUseServer).

ЗАМЕЧАНИЕ

Изменения состояния свойства CursorLocation сказывается на всех подключенных к соединению компонентов наборов данных ADO.

Сервисные методы модуля ADODB

При создании приложения Delphi, работающего с СУБД на основе технологии ADO, к файлам проекта подключается модуль ADODB. В нем сосредоточен программный код, обеспечивающий взаимодействие нашего проекта и COM-объектов Windows. В том числе в модуле ADODB объявлен ряд сервисных процедур и функций, о которых полезно знать программисту.

В первую очередь познакомлю вас с процедурой

```
procedure CreateUDLFile(const FileName, ProviderName, DataSourceName:
WideString);
```

позволяющей программисту создавать файл соединения программным способом. Здесь `FileName` — имя будущего UDL-файла; `ProviderName` — имя провайдера; `DataSourceName` — имя источника данных. Пример создания файла соединения приведен в листинге 22.10.

Листинг 22.10. Создание UDL-файла

```
Uses ComObj, ADODB;
var FileName : String;
begin
FileName:=DataLinkDir+'\tst.udl'; //формируем имя файла
try
CreateUDLFile(FileName, 'MSDASQL.1', 'Phones');
except
  on EOLESysError do
        begin
          //обработка ошибки
        end;
end;
end;
```

Обратите внимание, что при формировании имени UDL-файла с помощью метода

```
function DataLinkDir: string;
```

мы выясняем штатный каталог Windows для хранения всех UDL-файлов. По умолчанию это

```
C:\Program Files\Common Files\System\Ole DB\Data Links\
```

Замечание

Строку соединения допускается хранить в специальном файле связи с данными (UDL-файле). Для формирования содержащего описание соединения UDL-файла "вручную", создайте пустой текстовый файл. Измените его расширение с txt на udl и запустите файл на выполнение. Запуск UDL-файла автоматически вызовет редактор настройки соединительной строки.

Для того чтобы приложение смогло собрать весь перечень UDL-файлов из каталога, определенного в параметре `Directory`, нам пригодится метод

```
function GetDataLinkFiles(FileNames: TStrings; Directory: string = ''):
Integer;
```

Если параметр `Directory` оставить в покое и не передавать в него никаких значений, то поиск файлов осуществится в каталоге по умолчанию. Собранные данные передаются в список `FileNames`, а функция возвращает число найденных файлов.

Для построения полного списка имеющихся в системе провайдеров (драйверов OLE DB) пригодится перегружаемый метод

```
procedure GetProviderNames(Names: TStrings); overload;
procedure GetProviderNames(Names: TWideStrings); overload;
```

Результаты анализа помещаются в список имен `Names`.

Для вызова диалога выбора UDL-файла, воспользуйтесь методом

```
function PromptDataLinkFile(ParentHandle: THandle; InitialFile: WideString):
WideString;
```

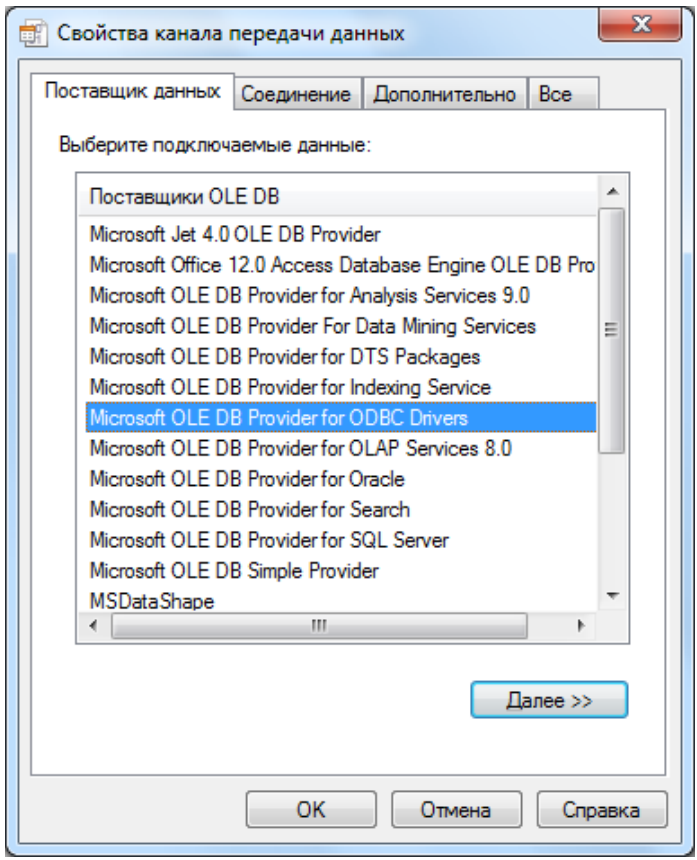


Рис. 22.8. Стандартный диалог "Свойства канала передачи данных"

Здесь `ParentHandle` — указатель на форму, владеющую диалогом, `InitialFile` — файл, инициализирующий каталог диалога и фильтр по расширению файла. Функция возвращает имя файла, выбранного пользователем.

Если вместо UDL-файла вы планируете сформировать соединительную строку, то нашим лучшим помощником станет метод

```
function PromptDataSource(ParentHandle: THandle; InitialString: WideString):
WideString;
```

который вызовет диалоговое окно определения параметров строки соединения (рис. 22.8). Параметры аналогичны функции `PromptDataLinkFile()`, но в качестве результата метод возвратит текст с параметрами соединения.

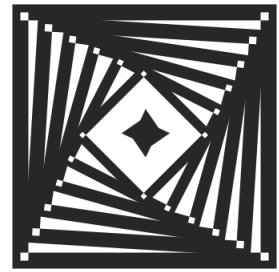
Резюме

В этой главе мы получили общее представление о возможностях разработанной в стенах корпорации Microsoft технологии *объектов данных ActiveX* (ActiveX Data Objects, ADO). Из всего перечня имеющихся в распоряжении Delphi способов доступа к хранилищам данных механизм ADO наиболее предпочтителен для работы с БД Access и SQL Server корпорации Microsoft.

Ответственность за подключение к БД с помощью технологии ADO возлагается на компонент класса `TADOConnection`. Названный компонент держит в своих руках все основные бразды управления проектом БД и реализует следующий функционал:

- Установка и разрыв соединения.
- Регистрация пользователя в СУБД.
- Настройка тайм-аута.
- Управление транзакциями и установка уровня их изоляции.
- Настройка поведения курсора.

В следующей главе мы продолжим изучение технологии ADO и сосредоточим свое внимание на компонентах-наборах данных, позволяющих читать, добавлять, редактировать и удалять записи из таблиц БД.



Глава 23

Наборы данных ADO

Продолжим работу с технологией ADO, но на этот раз сосредоточим свое внимание на компонентах VCL, предназначенных для непосредственной работы с таблицами БД. В первую очередь, это командный объект `TADOCommand` и набор данных `TADODataSet`. Компонент `TADOCommand` инкапсулирует командный объект ADO (см. рис. 22.2), он предназначен для отправки в БД простейших инструкций SQL, например, связанных с созданием каких-либо объектов БД (таблиц, представлений, триггеров). Набор данных `TADODataSet` специализируется на представлении записей из таблиц БД. Названные компоненты целиком и полностью соответствуют идеологии, заложенной Microsoft в объектную модель ADO, и вдвоем способны решить подавляющее большинство задач, связанных с обработкой данных в БД.

Командный объект — *TADOCommand*

Класс `TADOCommand` инкапсулирует командный объект ADO, а точнее интерфейс команды `_Command`. Элемент управления предназначен для отправки в адрес базы данных отдельных команд. Синтаксис команд должен соответствовать диалекту SQL целевой СУБД. Кроме того, есть одно небольшое ограничение, оно заключается в том, что, отправив в адрес провайдера команду, элемент `TADOCommand` весьма неохотно интересуется результатами ее выполнения. Исходя из этой особенности в качестве команд целесообразно передавать SQL-инструкции, не возвращающие приложению никаких данных. Вместе с тем, в порядке исключения командный объект способен вернуть набор записей `_Recordset`, но при условии, что обработкой полученных результатов займется подключенный к нему экземпляр класса `TADODataSet`.

Замечание

С задачами командного компонента превосходно справится и экземпляр класса `TADOQuery`. Однако применение не столь ресурсоемкого элемента `TADOCommand` может оказаться более эффективным при отправке SQL-команд, не возвращающих результата (`CREATE`, `DROP`, `INSERT`, `UPDATE`, `DELETE` и т. д.).

Для соединения с провайдером командный объект может выбрать один из двух путей. Если приложение подключается к данным централизованно (при посредничестве элемента управления `TADOConnection`), то следует воспользоваться этой возможностью и связать командный компонент с элементом управления `TADOConnection` при помощи свойства `Connection`. В таком случае процесс подключения перекладывается на `TADOConnection` и перестает интересовать командный элемент управления. Второй путь — самостоятельное подключение с помощью соединительной строки `ConnectionString`.

Текст отправляемой команды описывается в свойстве `CommandText`:

```
property CommandText: WideString;
```

Как уже отмечалось ранее, это не что иное, как инструкция на языке SQL. Если команда содержит какие-то параметры, например аргументы хранимой процедуры, то укажите их в свойстве

```
property Parameters: TParameters;
```

Для того чтобы список параметров регенерировался автоматически при изменении команды, установите в `true` свойство

```
property ParamCheck: Boolean;
```

Для удобства работы команды систематизировали по типам с помощью свойства

```
property CommandType: TCommandType;//по умолчанию cmdText
```

Определяя тип команды (табл. 23.1), программист явным образом указывает объекту ADO, какого рода инструкцию он планирует описать в его свойстве `CommandText`. Например, определив тип команды `cmdStoredProc`, мы говорим компоненту, что в его свойстве `CommandText` находится имя хранимой процедуры, которую мы планируем выполнить.

Таблица 23.1. *Перечень типов команд* `TCommandType`

Тип команды	Описание	Примечание
`cmdUnknown`	Неизвестный тип команды	—
`cmdText`	В свойстве `CommandText` находится явное текстовое описание инструкции SQL	
`cmdTable`	Рассматривает содержимое свойства `CommandText` как имя таблицы, данные из которой следует получить	
`cmdStoredProc`	Команда выполняет хранимую процедуру	
`cmdFile`	Рассматривает содержимое свойства `CommandText` как имя файла с данными	Для выполнения команды требуется помощь интерфейса `_RecordSet`
`cmdTableDirect`	Прямой доступ к таблице, в свойстве `CommandText` должно находиться имя таблицы	

Для упрощения работы с командным объектом в текстовом режиме (`CommandType=cmdText`) компонент предоставляет разработчику простой редактор (рис. 23.1), который позволит начинающему программисту создать SQL-запрос на основе инструкции `SELECT` и передать его в свойство `CommandText`.

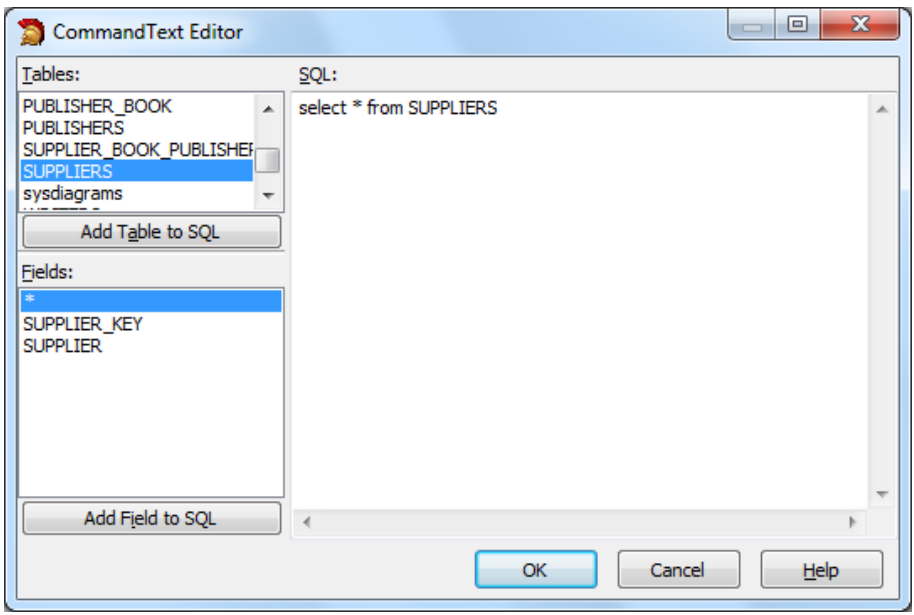


Рис. 23.1. Редактор запроса SQL-компонента `TADOCommand`

Самый главный метод командного объекта ADO доступен в трех вариациях:

```
function Execute: Recordset; overload;
function Execute(const Parameters: OleVariant):_Recordset; overload;
function Execute(var RecordsAffected: Integer;
                 const Parameters: OleVariant):_RecordSet; overload;
```

Он отправляет хранящиеся в `CommandText` инструкции SQL на выполнение.

Допустим, что мы намерены создать хранимую процедуру для нашей БД книжного магазина, развернутой на сервере MS SQL Server. Назначение процедуры — обеспечить добавление к таблице `SUPPLIERS` нового поставщика. С такой задачей командный компонент справится в два счета (листинг 23.1).

Листинг 23.1. Пример создания хранимой процедуры средствами команды ADO

```
with ADOCommand1 do
begin
  CommandType:=cmdText;
  CommandText:='CREATE PROCEDURE pr_SUPPLIERS_INSERT
               @SUPPLIER VARCHAR(100)
               AS
```

```
                BEGIN
                 INSERT INTO SUPPLIERS
                 (SUPPLIER) VALUES (@SUPPLIER)
                END';
  Execute;
end;
```

Теперь, когда хранимая процедура с именем `pr_SUPPLIERS_INSERT` уже существует, попробуем воспользоваться ее услугами и добавим к таблице поставщиков очередную запись (листинг 23.2).

Листинг 23.2. Пример обращения командного объекта ADO к хранимой процедуре

```
with ADOCommand1 do
begin
  CommandType:=cmdStoredProc;{тип команды — хранимая процедура}
  CommandText:='pr_SUPPLIERS_INSERT'; {имя хранимой процедуры}
  Parameters.Clear; {очищаем коллекцию параметров}
      with Parameters.AddParameter do {создаем новый параметр}
      begin
       Name:='@SUPPLIER'; {имя параметра}
       DataType:=ftString; {тип данных — текстовый}
       Direction:=pdInput; {параметр предназначен для передачи значений}
      end;
  Parameters.ParamValues['@SUPPLIER']:='Новый поставщик';{заполняем параметр}
  Execute; {выполняем команду}
end;
```

Листинг 23.2 демонстрирует не только порядок обращения команды ADO к хранимой процедуре, но и процесс создания параметра в коде приложения.

Замечание

Для создания параметров совсем не обязательно писать код. Компоненты ADO способны самостоятельно собрать служебную информацию о параметрах хранимых процедур. Единственное условие — свойство `ParamChek` должно быть установлено в состояние `true`.

Еще один пример (листинг 23.3), но на этот раз в адрес СУБД направляется текстовая SQL-инструкция на основе команды `SELECT`.

Листинг 23.3. Получение записей командным объектом и набором данных ADO

```
with ADOCommand1 do
 begin
  CommandType:=cmdText; {тип команды — инструкция SQL}
```

```
  CommandText:='SELECT * FROM SUPPLIERS';
 {для обработки множества строк нужна помощь компонента ADODataSet1}
  ADODataSet1.Recordset:=Execute;
 end;
```

Сразу заострим внимание на том факте, что множество записей, возвращаемое в результате выполнения запроса, командный компонент не в состоянии обслужить самостоятельно. Для этой цели ему требуется помощник, способный работать с интерфейсом множества записей `_RecordSet`, в нашем примере это набор данных — компонент `ADODataSet1: TADODataSet`.

> ***ВНИМАНИЕ!***
>
> Основная специализация командного объекта `TADOCommand` заключается в отправке команд в адрес СУБД без обработки возвращаемых данных. Если же программе необходимо получать от БД данные, то следует научить командный объект взаимодействовать с компонентами, способными обслуживать интерфейс `_RecordSet`.

Время тайм-аута, по истечению которого компонент станет полагать, что выполнение команды закончилось неудачей, определяется целым числом в свойстве

```
property CommandTimeout: Integer;
```

По умолчанию тайм-аут равен 30 с. Если за отведенное время отправленная к СУБД команда не будет выполнена, то она автоматически отменяется.

На порядок выполнения инструкции влияют опции (табл. 23.2), установленные в свойстве

```
property ExecuteOptions: TExecuteOptions;
```

По умолчанию свойство содержит пустое множество.

Таблица 23.2. *Опции* `TExecuteOptions`

Опция	Описание
`eoAsyncExecute`	Команда выполняется в асинхронном режиме, компонент не станет ожидать ответа сервера
`eoAsyncFetch`	Получение данных из БД в асинхронном режиме
`eoAsyncFetchNonBlocking`	Получение данных из БД в асинхронном режиме, во время которого основной поток никогда не блокируется
`eoExecuteNoRecords`	Независимо от содержания инструкции SQL, передаваемой в адрес БД, командный объект не будет возвращать результирующие записи

Для того чтобы СУБД заранее подготовилась к выполнению команды, перед первым обращением к методу `Execute()` следует установить в состояние `true` свойство

```
property Prepared: WordBool;
```

Впрочем, явный вызов операции подготовки команды не обязателен. При необходимости сервер баз данных самостоятельно откомпилирует текст команды.

Замечание

Асинхронной операцией с базой данных называют такую операцию, которая, отправив команду СУБД, сразу, не дожидаясь выполнения этой команды, возвращает управление приложению. При проведении *синхронной операции* управление приложению не передается, пока СУБД не выполнит команду. По умолчанию компоненты ADO настроены на работу в синхронном режиме.

Коллекция ошибок *Errors* и объект ошибки *Error*

Для обработки исключительных ситуаций, неминуемо возникающих в приложениях БД, в объектной модели ADO введены коллекция ошибок `Errors` и отдельная ошибка `Error` (см. рис. 22.2). Как и следует из названия, объект `Errors` коллекционирует ошибки, помещая информацию об исключительной ситуации (ИС) в свое хранилище. Тяга к "собирательству" ошибок не означает, что объект `Errors` помнит обо всех ошибках, произошедших с момента старта приложения. Алгоритм его работы существенно проще — в момент возникновения очередной ИС коллекция полностью освобождается от предыдущего груза ошибок и заполняется данными о новых неприятностях.

Доступ к коллекции ошибок осуществляется с помощью свойства

```
property Errors: Errors;
```

имеющегося в распоряжении компонента `TADOConnection`, либо через соответствующий параметр `Errors` многочисленных обработчиков событий компонентов линейки **dbGo**, таких как `TRecordSetEvent`, `TExecuteCompleteEvent`, `TInfoMessageEvent` и т. п.

Управление коллекцией `Errors` ничем не отличается от работы с другими коллекциями VCL. Чтобы выяснить число ошибок в коллекции, обращаемся к свойству

```
Property Count:Integer;
```

Доступ к отдельному элементу коллекции производится по его индексу с помощью свойства

```
Property Item[Index:Integer]:EADOError;
```

При необходимости можно обновить все коллекцию, в этом нам окажет помощь метод

```
Procedure Refresh;
```

Для очистки коллекции вызываем метод

```
Procedure Clear;
```

Впрочем, эта операция делается автоматически с возникновением новой ИС.

Теперь перейдем к объекту, описывающему ошибку ADO. В Delphi отдельный объект ошибки инкапсулируется в класс `Error = ADOInt.Error` и является прямым наследником общего для всех исключительных ситуаций БД Delphi класса `EDatabaseError`.

Смысловое описание ошибки находится в свойстве

```
Property Description : WideSring;
```

Код инструкции SQL, благодаря "стараниям" которой в нашем проекте возникла исключительная ситуация, находится в свойстве

```
Property SQLState:WideString;
```

Числовые значения кода зависят от СУБД, в которой мы разрабатываем БД. Источник ошибки хранится в свойстве

```
Property Source:WideString;
```

Источник обычно указывает на имя провайдера, который оказывает нам помощь в соединении с БД.

Набор данных ADO, компонент *TADODataSet*

Все компоненты наборы данных ADO (`TADODataSet`, `TADOTable`, `TADOQuery` и `TADOStoredProc`) построены на фундаменте класса `TCustomADODataSet`. Обычно класс, подобный `TCustomADODataSet`, мы называли абстрактным, но здесь такое определение не вполне точно. Более правильно классифицировать его, как виртуальный базовый класс ADO. Развивая идеи, заложенные в `TDataSet`, и являясь пропагандистом универсальной, независимой от платформы технологии доступа к набору данных, класс `TCustomADODataSet` инкапсулирует практически все интерфейсы объектов ADO, рожденных в стенах Microsoft. Таким образом, `TCustomADODataSet` предоставляет доступ к "инородным" интерфейсам в привычном для программиста Delphi виде — в качестве свойств и методов.

Ключевой элемент линейки ADO и прямой наследник класса `TCustomADODataSet` — компонент `TADODataSet` — предназначен для обслуживания набора данных. Попытка однозначной классификации компонента по функциональной направленности обречена на неудачу, его нельзя отнести ни к привычным для программиста Delphi компонентам-таблицам, ни к запросам, ни к хранимым процедурам. На самом деле `TADODataSet` — это универсальный элемент доступа к БД, впитавший в себя многие положительные качества своих коллег по цеху обслуживания данных.

Например, о наличии черт запроса говорит то, что для подключения таблицы (или нескольких таблиц) к набору данных необходимо описать текст запроса SQL в свойстве `CommandText`, а также, что текст запроса допускает использование параметров.

Как и у классической таблицы, в компоненте TADODataSet полученный набор данных представляется в виде строк и столбцов, а в свойствах IndexName и IndexFileName мы можем управлять подключением индексов.

Схожесть с хранимыми процедурами в том, что компонент позволяет выполнять описанные на сервере процедуры.

И самое главное в том, что TADODataSet инкапсулирует важнейшее качество объекта ADO — интерфейс множества записей _RecordSet.

Интерфейс множества записей *_Recordset*

В рамках технологии ADO программисты корпорации Microsoft реализовали специализированный объект _Recordset, называемый множеством записей. Этот объект обеспечит доступ к набору записей таблицы или записей, полученных в результате выполнения какой-то команды. Физически _Recordset представляет собой совокупность записей, создаваемых в момент обращения к данным или выполнения SQL-инструкции. Что касается Delphi, то в составе изучаемого нами класса TCustomADODataSet множество записей инкапсулировано в свойстве

```
property Recordset: _Recordset;
```

Свойство представляет собой указатель на интерфейс _Recordset и позволяет программисту манипулировать его свойствами и методами напрямую.

Текущее состояние объекта Recordset описывается свойством

```
property RecordsetState: TObjectStates;
type TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
```

Состояние stClosed удостоверяет факт отключения объекта от хранилища данных. Пока потомок класса TCustomADODataSet находится в процессе соединения, свойство возвратит значение stConnecting. Значение stOpen свидетельствует о том, что набор данных открыт и множество записей доступно. Значение stExecuting указывает, что элемент управления выполняет команду. Значение stFetching информирует нас о том, что в данный момент времени производится выборка строк из таблиц.

Чтобы набор данных мог выяснить, каким перечнем возможностей обладает обслуживаемый им объект Recordset, в его состав включен метод

```
function Supports(CursorOptions: TCursorOptions): Boolean;
```

Возможные значения, которые может принимать единственный параметр метода, приведены в табл. 23.3. Если указанная опция поддерживается, то метод возвратит значение true.

Таблица 23.3. *Возможные значения класса* `TCursorOptions`

Значение	Описание возможностей множества записей Recordset
coHoldRecords	Большинство записей можно восстановить без подтверждения отложенных обновлений

Таблица 23.3 *(окончание)*

Значение	Описание возможностей множества записей Recordset
coMovePrevious	Курсор можно переместить назад без использования закладки
coAddNew	Поддерживается вставка новых записей
coDelete	Поддерживается удаление записей
coUpdate	Поддерживается модификация записей
coBookmark	Закладку (свойство Bookmark) объекта _Recordset можно использовать для перемещения курсора или доступа к отмеченной записи
coApproxPosition	Множество записей поддерживает свойство RecNo, что позволяет идентифицировать номер записи
coUpdateBatch	Поддерживается пакетное обновление данных (допустим вызов методов UpdateBatch и CancelBatch)
coResync	Данные курсора могут обновляться асинхронным способом
coNotify	Объект _Recordset поддерживает уведомления и может генерировать события
coFind	Для поиска данных допустимо применение метода Locate()
coSeek	Для поиска записи может использоваться метод Seek()
coIndex	Задание текущего индекса возможно через свойство IndexName

Доступ к хранилищу данных

Для подключения к набору данных TADODataSet способен воспользоваться услугами компонента TADOConnection. Для этого достаточно обратиться к подробно рассмотренному в предыдущей главе компоненту-соединителю, сославшись на него в свойстве

```
property Connection: TADOConnection;
```

Если же в приложении компоненту TADODataSet места не нашлось (что может произойти только в простейших проектах), то компоненту TADODataSet допускается самостоятельно сформировать строку соединения (свойство ConnectionString) или подключиться к файлу UDL.

Покончив с описанием параметров соединения, необходимо конкретизировать, какой именно набор данных нас интересует. И здесь нам пригодится свойство CommandText:

```
property CommandText: WideString;
```

В нем мы описываем инструкцию выборки строк из таблицы (либо объединения таблиц) так, как в листинге 23.4.

Листинг 23.4. Получение записей с помощью инструкции SELECT

```
With ADODataSet1 do
begin
    CommandType:=cmdText;
    CommandText:='SELECT SUPPLIERS.SUPPLIER, CONTRACTS.CONTRACT,
                        CONTRACTS.CONTRDATE FROM SUPPLIERS
                  LEFT JOIN CONTRACTS ON
                        CONTRACTS.SUPPLIER_KEY=SUPPLIERS.SUPPLIER_KEY
                  ORDER BY CONTRDATE, CONTRACT';
   Open;
end;
```

В конце концов, указав интересующий нас набор данных, вызываем метод `Open()` либо устанавливаем в `true` свойство `Active`.

После завершения работы метода `Open()`, в тот момент, когда набор данных будет готов к использованию, генерируется событие `OnRecordsetCreate()`, информирующее программиста о том, что множество записей получено и готово к дальнейшей работе.

При необходимости обновить множество записей, полученное в результате выполнения команды, обращаемся к процедуре `Refresh()` или `Requery()`. Вызов метода инициирует повторную отправку компонентом SQL-запроса в адрес провайдера, т. е. это аналог последовательного вызова методов `Close()` и `Open()`.

Выбор библиотеки курсора

По аналогии с `TADOConnection` класс позволяет выбирать библиотеку курсора с помощью свойства `CursorLocation`. По умолчанию набор данных ориентирован на работу с курсором на клиентской стороне (`clUseClient`). В этом случае данные с сервера сначала передаются на клиентскую машину и затем обрабатываются ее средствами. Курсор такого рода обладает максимально гибкими возможностями, в особенности по сортировке и фильтрации данных. При использовании курсором клиентских библиотек роль сервера ограничивается только отбором данных согласно инструкции SQL. Серверный тип курсора (`clUseServer`) целесообразно выбирать лишь в случаях пересылки на пользовательскую машину больших объемов информации при очевидной слабости клиентского и сетевого оборудования.

ЗАМЕЧАНИЕ

Для централизованного назначения типа курсора для всех компонентов-наборов данных приложения разумно воспользоваться однотипным свойством `CursorLocation` компонента `TADOConnection`.

Редактирование данных

Названия почти всех методов, обеспечивающих процесс редактирования данных, унаследованы от универсального набора данных `TDataSet`, и при практическом применении почти ничем от них не отличаются (листинг 23.5).

Листинг 23.5. Вставка строки средствами таблицы `TADOTable`

```
With ADOTable1 do
begin
   if Active=True then close;
   TableName:='Writers';
   Open;
   Insert;
   FieldByName('Writer').Value:='Цветаева Марина';
   Post;
end;
```

Заметьте, что операции вставки, редактирования и удаления записей без явного применения команд SQL поддерживается только компонентом-таблицей `TADOTable`. В остальных компонентах придется немного поработать — указать источник данных на языке SQL. В листинге 23.6 предложен пример решения аналогичной задачи вставки новой записи в таблицу авторов с помощью набора данных `ADODataSet1`.

Листинг 23.6. Вставка строки средствами набора данных `TADODataSet`

```
With ADODataSet1 do
begin
   if Active=True then close;
   CommandText:='SELECT * FROM WRITERS';
   Open;
   Insert;
   FieldByName('Writer').Value:='Пастернак Борис';
   Post;
end;
```

Листинг 23.7 содержит еще один пример вставки новой записи методом `Insert()`, но на этот раз с помощью компонента-запроса `ADOQuery1`.

Листинг 23.7. Вставка строки средствами запроса `TADOQuery`

```
With ADOQuery1 do
begin
```

```
    if Active=True then close;
    SQL.Clear;
    SQL.ADD('SELECT * FROM WRITERS');
    Open;
    Insert;
    FieldByName('Writer').Value:='Куприн А.И.';
    Post;
  end;
```

Обратите внимание на различия в коде вставки новой записи в таблицу с помощью компонента-таблицы `ADOTable1`, компонента набора данных `ADODataSet1` и запроса `ADOQuery1`. В первом случае мы, воспользовавшись свойством `TableName`, просто указываем имя таблицы, во втором и в третьем нам пришлось явно задать инструкцию `SELECT`.

ВНИМАНИЕ!

Методы редактирования данных `Insert()`, `Append()`, `Update()`, `Delete()` и т. п. сохраняют работоспособность только в ситуации, когда компонент подключен к единственной таблице. Если же наследник `TCustomADODataSet` класса обслуживает объединение таблиц (например, полученное в результате применения инструкции `JOIN`), то методы редактирования могут стать недееспособными.

Кроме методов редактирования, унаследованных от универсального класса `TDataSet`, у потомков `TCustomADODataSet` имеется ряд процедур и функций, управляющих редактированием записей во время пакетного обновления, о них мы поговорим несколько позже.

Перемещение по множеству строк

Общее число записей и номер текущей записи доступных в объекте `_RecordSet` выясняется из справочных свойств:

```
property RecordCount: Integer;//всего строк
property RecNo: Integer;      //номер текущей строки
```

Перемещение по открытому набору данных осуществляется с помощью методов, знакомых нам еще по классическому классу `TDataSet`:

```
procedure First; //переход к первой записи во множестве
procedure Prior; //предыдущая запись
procedure Next; //следующая запись
procedure Last; //последняя запись в множестве
function MoveBy(Distance: Integer): Integer;{переход на несколько записей}
```

Метод `MoveBy()` перемещает курсор на `Distance` записей вперед или назад (при отрицательном значении аргумента). Функция возвращает число строк, на которые действительно удалось переместиться. О нахождении на первой записи во множе-

стве свидетельствует значение `true`, возвращаемое свойством `BOF`. Если выбрана последняя запись набора — в `true` установится свойство `EOF`.

В следующем фрагменте кода (листинг 23.8) предложен классический пример перебора всех записей набора данных.

Листинг 23.8. Перебор всех записей в наборе данных

```
ADODataSet1.DisableControls;
try
 ADODataSet1.First; //переход к первой записи
  while ADODataSet1.EOF=False do //пока не достигнут конец набора данных
  begin
  {Операции с записями набора}
  ADODataSet1.Next; //переход к очередной записи
  end;
finally
ADODataSet1.EnableControls;
end;
```

Особое место в примере занимает обращение к методам `DisableControls()` и `EnableControls()`. Процедуры отключают и включают связанные с набором данных элементы управления (в первую очередь отвечающие за визуализацию данных). Предложенный в листинге 23.8 код позволит значительно ускорить перебор строк во множестве `_Recordset`, т. к. исключает необходимость постоянной перерисовки содержимого записей в элементах пользовательского интерфейса.

Помимо унаследованных от класса `TDataSet` обработчиков событий, связанных с перемещением по набору данных, в `TCustomADODataSet` представлены некоторые новинки — события `OnWillMove()`, `OnMoveComplete()` и `OnEndOfRecordset()`, описание которых вы найдете в табл. 23.6.

Размер записи в байтах укажет свойство

```
property RecordSize: Word; //только для чтения
```

Состояние текущей записи определяется при помощи свойства

```
property RecordStatus: TRecordStatusSet; //только для чтения
```

Свойство возвращает константу (табл. 23.4). Большинство констант указывает причину отказа при сохранении записи.

Таблица 23.4. *Статус записи, возможные значения* `TRecordStatusSet`

Константа	Описание
`rsOK`	Строка была успешно модифицирована
`rsNew`	Это новая строка
`rsModified`	Строка изменилась

Таблица 23.4 *(окончание)*

Константа	Описание
rsDeleted	Строка была удалена
rsUnmodified	Строка не подвергалась изменениям
rsDBDeleted	Строка уже была удалена из источника данных
Причина отказа сохранить изменения в строке	
rsInvalid	Запись не сохранена из-за неверной закладки
rsMultipleChanges	Недопустимое воздействие на другие записи
rsPendingChanges	Процесс обновления не завершен
rsCanceled	Отмена операции
rsCantRelease	Строка заблокирована
rsConcurrencyViolation	Запись захвачена конкурирующей транзакцией
rsIntegrityViolation	Нарушение целостности данных
rsMaxChangesExceeded	Ожидание большого числа изменений
rsObjectOpen	Конфликт с открытым объектом
rsOutOfMemory	Нехватка памяти
rsPermissionDenied	Ограничение прав пользователя
rsSchemaViolation	Сохранение приведет к нарушению структуры основной базы данных

Особенности применения поля BCD

При работе с действительными числами разработчики ADO вместо вещественных полей `TFloatField` рекомендуют использовать *бинарно-кодированные десятичные поля* (Binary Coded Decimal, BCD). Главное достоинство BCD-поля заключается в отсутствии ошибок округления, характерных для поля класса `TFloatField`.

Для перехода к формату BCD следует установить в `true` свойство

```
property EnableBCD: Boolean; // по умолчанию true
```

Активизировав свойство `EnableBCD`, мы добьемся того, что компонент, столкнувшись с любым полем вещественного типа, станет представлять его в формате `TBCDField`.

События *TCustomADODataSet*

Помимо событий, унаследованных от универсального набора данных `TDataSet`, в классе `TCustomADODataSet` реализована дюжина своих собственных обработчиков

событий. Во многих событиях объявлены схожие параметры. В первую очередь, это `DataSet: TCustomADODataSet` — ссылка на компонент, ставший источником события. Второе место по частоте появления занимает параметр — `EventStatus: TEventStatus`. Это переменная, в которую возвращается описание статуса события (см. табл. 22.4). Исследовав `EventStatus`, программист делает вывод о том, насколько успешно завершилась та или иная операция с данными.

В момент создания и инициализации множества записей `Recordset` генерируется событие:

```
property OnRecordsetCreate: TRecordsetCreate;
type TRecordsetCreate = procedure(DataSet: TCustomADODataSet;
                                  const Recordset: _Recordset) of object;
```

Второй параметр события предоставляет ссылку на интерфейс множества данных.

Во время асинхронного получения данных периодически (по мере поступления пакетов с данными) генерируется событие

```
property OnFetchProgress: TFetchProgressEvent;
TFetchProgressEvent = procedure(DataSet: TCustomADODataSet;
                                Progress, MaxProgress: Integer;
                                var EventStatus: TEventStatus) of object;
```

Параметры `Progress` и `MaxProgress` возвращают значения, позволяющие программисту информировать пользователя о проценте выполнения задачи, например:

```
Trunc(Progress/MaxProgress*100).
```

Здесь `MaxProgress` хранит информацию об общем числе строк, задействованных в этой операции, а `Progress` — номер сроки, получаемой в данный момент времени. После того как все строки данных будут "закачаны", генерируется очередное событие:

```
property OnFetchComplete: TRecordsetEvent;
TRecordsetEvent = procedure(DataSet: TCustomADODataSet;
                            const Error: Error;
                            var EventStatus: TEventStatus) of object;
```

Половина событий, предусмотренных у `TCustomADODataSet`, парные: `OnWill...()` и `On...Complete()`. Событие `OnWill...()` сообщает о начале какого-то действия, а `On...Complete()` информирует о его завершении.

Первая пара таких событий связана с изменением содержимого поля: `OnWillChangeField()` и `OnFieldChangeComplete()`. Перед изменением значения в поле (или в группе полей) множества записей `_Recordset` генерируется:

```
property OnWillChangeField: TWillChangeFieldEvent;
TWillChangeFieldEvent = procedure(DataSet: TCustomADODataSet;
                                  const FieldCount: Integer;
                                  const Fields: OLEVariant;
                                  var EventStatus: TEventStatus) of object;
```

Параметр `FieldCount` — число полей, подвергаемых изменениям; `Fields` — массив полей. Заметьте, что речь ведется не о привычных для программистов Delphi классах `TField`. Ссылка на поля возвращается в виде универсального типа данных `OLEVariant`.

После того как в поле (поля) множества записей вносятся какие-то изменения, генерируется событие

```
property OnFieldChangeComplete: TFieldChangeCompleteEvent;
TFieldChangeCompleteEvent = procedure(DataSet: TCustomADODataSet;
                        const FieldCount: Integer;
                        const Fields: OleVariant;
                        const Error: Error;
                        var EventStatus: TEventStatus) of object;
```

Все параметры события нам уже встречались.

Очередная пара событий обслуживает все изменения, происходящие с отдельной записью множества данных `_Recordset`. При любых изменениях в состоянии строки (или нескольких строк) вызывается событие

```
property OnWillChangeRecord: TWillChangeRecordEvent;
TWillChangeRecordEvent = procedure(DataSet: TCustomADODataSet;
                                const Reason: TEventReason;
                                const RecordCount: Integer;
                                var EventStatus: TEventStatus) of object;
```

Что именно послужило причиной генерации этого события, выясняется по состоянию параметра `Reason` (табл. 23.5), параметр `RecordCount` возвратит число измененных строк.

Таблица 23.5. *Причина вызова события, возможные значения* `TEventReason`

Значение	Описание
erAddNew	Добавление новой строки
erDelete	Удаление записи
erUpdate	Внесение изменений в строку
erUndoUpdate	Отмена последних изменений
erUndoAddNew	Отмена вставки новой строки
erUndoDelete	Отмена удаления строки
erRequery	Множество записей обновлено методом `Requery()`
erResynch	Множество записей обновлено методом `Resynch()`
erClose	Множество записей закрыто
erMove	Курсор переместился
erFirstChange	Первое изменение данных
erMoveFirst	Курсор переместился на первую строку
erMoveNext	Курсор переместился на следующую строку
erMovePrevious	Курсор переместился на предыдущую строку
erMoveLast	Курсор переместился на последнюю строку

После изменения одной (или нескольких) записей вызывается завершающее событие:

```
property OnRecordChangeComplete: TRecordChangeCompleteEvent;
TRecordChangeCompleteEvent = procedure(DataSet: TCustomADODataSet;
                          const Reason: TEventReason;
                          const RecordCount: Integer;
                          const Error: Error;
                          var EventStatus: TEventStatus) of object;
```

Очередная пара событий `OnWillChangeRecordset()` и `OnRecordsetChangeComplete()` реагирует на изменения во множестве записей:

```
property OnWillChangeRecordset: TRecordsetReasonEvent;
property OnRecordsetChangeComplete: TRecordsetErrorEvent;
TRecordsetReasonEvent = procedure(DataSet: TCustomADODataSet;
                             const Reason: TEventReason;
                             var EventStatus: TEventStatus) of object;
```

Параметр `Reason` описан в табл. 23.5, остальные параметры нам уже знакомы.

В табл. 23.6 описаны три события, связанные с перемещением курсора по множеству записей.

Таблица 23.6. *События, вызываемые при перемещении по множеству строк*

Событие	Описание

`property OnWillMove: TRecordsetReasonEvent;`
`TRecordsetReasonEvent = procedure(`
`DataSet: TCustomADODataSet;`
`const Reason: TEventReason;`
`var EventStatus: TEventStatus) of object;`
	Событие вызывается перед началом перемещения к новой строке. Событие может быть инициировано методами `First`, `Last`, `Next`, `Prior`, `MoveBy`, `Insert`, `Delete`, `Post`, `Requery` и `Resynch`. Именно об этом и уведомляет параметр `Reason`

`property OnMoveComplete:`
`TRecordsetErrorEvent;`
`TRecordsetErrorEvent = procedure(`
`DataSet: TCustomADODataSet;`
`const Reason: TEventReason;`
`const Error: Error;`
`var EventStatus: TEventStatus) of object;`
	Сигнализирует о факте перехода к новой строке

`property OnEndOfRecordset:`
`TEndOfRecordsetEvent;`
`TEndOfRecordsetEvent = procedure (`
`DataSet: TCustomADODataSet;`
`var MoreData: WordBool;`
`var EventStatus: TEventStatus) of object;`
	Событие генерируется в момент достижения последней строки множества `RecordSet`. Здесь параметр `MoreData` в состоянии `true` сигнализирует о том, что в наборе `RecordSet` появились новые строки

Работа с индексами

Для сбора имен доступных индексов предназначена процедура

```
procedure GetIndexNames(List: TStrings);
```

Для определения текущего активного индекса необходимо передать его название в свойство

```
property IndexName: String;
```

Число полей, входящих в состав индекса, мы обнаружим в свойстве

```
property IndexFieldCount: Integer; //только для чтения
```

Зная общее число полей в индексе, мы сможем получить доступ к отдельному полю индекса через свойство-массив

```
property IndexFields[Index: Integer]: TField;
```

Сортировка записей

Для изменения порядка следования строк в результирующем наборе пригодится свойство

```
property Sort: WideString;
```

В свойство передается строка с текстом запроса, но без упоминания инструкции упорядочивания `ORDER BY`:

```
ADODataSet1.Sort := 'WRITER DESC';
```

Альтернативный способ сортировки записей в наборе обеспечивает свойство

```
property IndexFieldNames: String;
```

Это перечень названий полей, участвующих в сортировке. По умолчанию сортировка осуществляется на стороне сервера, а клиент только получает результирующий набор данных. Если провайдер не способен поддерживать такой способ упорядочивания результирующего набора данных, то перед обращением к свойству `IndexFieldNames` необходимо перевести свойство `CursorLocation` в состояние `clUseClient`.

Поиск данных

Поиск входящей в текущий индекс записи осуществляет функция

```
function Seek(const KeyValues: Variant;
              SeekOption: TSeekOption = soFirstEQ): Boolean;
```

При нахождении строки, соответствующей критериям поиска, на нее устанавливается курсор. Возможные опции поиска приведены в табл. 23.7.

Таблица 23.7. *Возможные опции поиска* `TSeekOption`

Значение	Описание
`soFirstEQ`	Курсор устанавливается на первую найденную запись, если ни одной записи не обнаружено, указатель переходит к последней строке набора

Таблица 23.7 *(окончание)*

Значение	Описание
`soLastEQ`	Курсор позиционируется на последней, удовлетворяющей условиям поиска записи. Если ни одной записи не обнаружено, то указатель переходит к последней строке набора
`soAfterEQ`	Если запись обнаружена, то курсор позиционируется на ней или сразу после нее
`soAfter`	Курсор устанавливается на следующей записи после найденной строки
`soBeforeEQ`	Если запись обнаружена, то курсор позиционируется на ней или перед ней
`soBefore`	Курсор устанавливается на строке, предшествующей найденной

Кроме метода `Seek()` наборы данных ADO поддерживают для поиска метод `Locate()` (он подробно рассмотрен в *главе 18*). Отличительная особенность метода `Locate()` в том, что он способен добраться до искомой записи даже при отсутствии индекса. Безусловно, отсутствие индекса существенно замедляет процесс поиска необходимой записи, но с другой стороны, не всегда имеется возможность проиндексировать таблицу вдоль и поперек.

Особенности изоляции транзакций

При организации многопользовательской обработки данных обязательно следует обратить внимание на определение уровня изоляции между параллельно выполняющимися транзакциями. Мы уже говорили об этом при обсуждении элемента управления `TADOConnection`. В свою очередь, в классе `TCustomADODataSet` предусмотрен дополнительный механизм, определяющий уровни изоляции отдельной записи во множестве строк:

```
property LockType: TADOLockType; //по умолчанию ltOptimistic

type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic,
                     ltOptimistic, ltBatchOptimistic);
```

Значение `ltUnspecified` свидетельствует о том, что тип блокировки выяснить не удалось; `ltReadOnly` — данные доступны только для чтения; `ltPessimistic` — пессимистическая блокировка, после того, как вы обратились к записи, она блокируется для других пользователей; `ltOptimistic` — оптимистическая блокировка, заставляет провайдера блокировать запись только в том случае, когда транзакция приступит к ее обработке; `ltBatchOptimistic` — оптимистическая блокировка, допускающая пакетные обновления.

Листинг 23.9 демонстрирует пессимистический подход к настройке уровня изоляции транзакции.

Листинг 23.9. Установка уровня изоляции

```
With ADODataSet1 do
Begin
 LockType:= ltPessimistic;
 CursorLocation:=clUseServer;
 CursorType:=ctKeyset;
 CommandType:=cmdText;
 CommandText:='SELECT * FROM WRITERS';
Open;
End;
```

ВНИМАНИЕ!

Уровень блокировки строки `LockType` должен быть назначен еще до обращения к набору данных. По умолчанию он оптимистический — `ltOptimistic`.

Фильтрация множества записей

Несколько свойств и методов класса предназначены для обеспечения процесса фильтрации данных во множестве строк:

```
property Filter: String;                //текст фильтра
property FilterOptions: TFilterOptions;//опции фильтрации
property Filtered: Boolean;             //включение/отключение фильтра
```

На синтаксисе описания фильтра мы останавливаться не станем, т. к. он подробно изложен в *главе 18*. В процедуру передаются два аргумента: параметр `DataSet` — ссылка на экземпляр класса `TCustomADODataSet`, к которому мы планируем применить фильтр, и `FilterText` — текст фильтра.

Кэширование записей

Еще одна характерная черта наборов данных ADO — возможность применения локального буфера для хранения и обработки множества строк, полученного из хранилища данных. Механизмом кэширования управляет свойство

```
property CursorLocation:TCursorLocation;
```

По умолчанию свойство установлено в состояние `clUseClient`. Это свидетельствует о том, что режим кэширования включен, и обслуживание хранящихся в нем записей обеспечивается с помощью локального курсора. Для отказа от кэширования данных на клиентской стороне переводим свойство в режим `clUseServer`.

Замечание

Если наборы данных подключаются к хранилищу с помощью компонента-соединителя `TADOConnection`, то целесообразно управлять курсором централизованно, с помощью свойства `CursorLocation` компонента `TADOConnection`.

Таблицы БД способны содержать десятки или даже сотни тысяч записей. Как вести себя клиентским приложениям, запросившим от СУБД весь набор данных? Ведь может статься, что клиентский компьютер будет не в состоянии "проглотить" такой кусок пирога за один присест. Простейшее решение подобной проблемы — чтение данных порциями. Размерность порции данных задает свойство

```
property MaxRecords: Integer; //по умолчанию 0
```

По умолчанию это свойство установлено в 0, что свидетельствует об отсутствии каких-либо ограничений. Можно долго обсуждать вопрос о том, какое значение `MaxRecords` станет наиболее оптимальным для соблюдения баланса между скоростью доступа к данным и ресурсными ограничениями клиентского компьютера. Ведь результат зависит от многих показателей: пропускной способности сетевого оборудования, производительности сервера, числа пользователей, характеристик клиентского компьютера и качества нашего программного обеспечения.

Другим приемом оптимизации работы клиентских приложений БД может стать настройка размера кэша. Объем памяти, отводящейся для кэширования строк, определяется свойством

```
property CacheSize: Integer;//по умолчанию 1
```

В качестве единицы измерения используются не байты, а записи. По умолчанию размер буфера установлен в 1, это значит, что в памяти хранится всего одна строка данных и, чтобы обратиться к очередной записи, элемент управления будет вынужден отправить запрос к провайдеру. Если увеличить размер кэша, скажем, до сотни, то в момент обращения к набору данных элемент управления затребует от провайдера сразу 100 строк и разместит их в памяти. Теперь перемещение по строкам будет происходить значительно быстрее, но до тех пор, пока не потребуется 101-я строка. В этот момент повторится процедура запроса очередной сотни строк. У буфера большого размера еще один недостаток проявляется при работе в многопользовательском режиме. При параллельной обработке данных, считанные в кэш записи не обновляются, т. е. не отражают изменения, которые, возможно, сделали другие пользователи.

Ускорить чтение записей можно, немного поэкспериментировав с размером блока чтения:

```
property BlockReadSize: Integer; {по умолчанию 0 — чтение блоками отключено}
```

Свойство определяет количество буферизируемых записей за одну операцию чтения. При этом связанные с набором данных элементы отображения не обновляются. Для отключения режима поблочного чтения установите `BlockReadSize` в нулевое значение. Когда `BlockReadSize>0`, состояние набора данных установится в `State=dsBlockRead`.

Фильтрация записей в кэше

В ADO предусмотрен оригинальный способ фильтрации данных, учитывающий техническое состояние записи в локальном буфере набора данных:

```
property FilterGroup: TFilterGroup;
```

Свойство назначает групповой фильтр множеству записей на основе статуса каждой из входящих в набор записей (табл. 23.8).

Таблица 23.8. *Вид фильтрации* `TFilterGroup`

Значение	Описание
`fgUnassigned`	Используется только внутри ADO-компонента, признак того, что фильтрация не применялась
`fgNone`	Отключает любую фильтрацию. Аналог установки свойства `Filtered` в `false`
`fgPendingRecords`	Показывает только строки, которые были модифицированы, но изменения пока не сохранялись (не был вызван метод `UpdateBatch`) или отменены (метод `CancelBatch`)
`fgAffectedRecords`	Покажет только последние модифицированные строки
`fgFetchedRecords`	Показывает только строки в текущем кэше. Это результаты последнего запроса к базе данных
`fgPredicate`	Покажет только удаленные строки
`fgConflictingRecords`	Покажет только строки, которые подвергались модификациям, но не были сохранены из-за ошибок в процессе сохранения

Для работы с фильтром `FilterGroup` необходимо, чтобы набор данных находился в пакетном режиме модификации (`LockType=ltBatchOptimistic`) и свойство `Filtered` было переведено в `true` (листинг 23.10).

Листинг 23.10. Получение строк, не сохраненных в БД из-за ошибок

```
if (ADODataSet1.LockType = ltBatchOptimistic) then
begin
  ADODataSet1.FilterGroup := fgConflictingRecords;
  ADODataSet1.Filtered := True;
end;
```

В предложенном примере благодаря фильтру мы узнаем, какие записи набора данных нам не удалось сохранить.

Организация отложенного обновления данных в ADO

Модель ADO поддерживает два основных режима обновления данных — немедленное и отложенное (пакетное) обновление.

Обычная работа пользователя с базой данных сводится к процессу вставки, редактирования и удаления данных. При поступлении команды на изменение записи в режиме немедленного обновления СУБД сразу инициирует транзакцию, в рамках которой в БД моментально вносятся все изменения, инициированные пользовательским приложением. Все идет гладко до тех пор, пока доступ клиентского приложения к ресурсу базы данных не ограничен ни скоростью обмена сети или модема, ни занятостью сервера на обслуживание транзакций, инициированных другими клиентами, ни уборщицей, выдернувшей шваброй сетевой кабель, ни чем-то еще подобным. Но как только возможности СУБД или сети начинают расходиться с пожеланиями пользователей, то клиент сталкивается с проблемами внесения обновлений в данные. В таком случае, по истечении времени тайм-аута приложение сигнализирует о невозможности сохранения модификаций, а работа оператора превращается в сплошное мучение.

В качестве одного из решений изложенной проблемы программисты предложили отложенное обновление данных. Идея заключается в том, что все изменения в данных накапливаются в локальном буфере пользовательского компьютера и отправляются в адрес сервера одним пакетом в рамках единой транзакции. Такой подход значительно снижает нагрузку на СУБД и минимизирует сетевой трафик. Вместе с тем отложенное обновление данных может стать потенциальным источником ошибок, особенно в случае многопользовательской работы с БД. Ведь нельзя исключить вероятность одновременного захвата одних и тех же данных с целью редактирования двумя пользователями, в результате можно получить абсолютно непредсказуемые последствия. Поэтому при пакетном обновлении особое внимание необходимо уделить вопросу сериализации транзакций.

Степень защиты данных, загруженных в локальный кэш пользовательского компьютера, определяется не только уровнем изоляции транзакций, но и типом курсора, который будет "пробегать" строки набора данных:

```
property CursorType: TCursorType;

type TCursorType = (ctUnspecified, ctOpenForwardOnly, ctKeyset,
                    ctDynamic, ctStatic);
```

Тип курсора (табл. 23.9) устанавливается еще до обращения к набору данных. В ситуации, когда провайдер не в состоянии обеспечить запрашиваемый тип курсора, он имеет право назначить курсор самостоятельно.

Таблица 23.9. *Значения* `TCursorType`

Тип курсора	Описание	Курсор
`ctUnspecified`	Неопределенный (назначаемый по умолчанию) тип курсора	_

Таблица 23.9. *Значения* `TCursorType`

Тип курсора	Описание	Курсор
`ctDynamic`	Динамический курсор. Добавления, изменения и удаления видны для других пользователей	Двунаправленный
`ctStatic`	Статический курсор. Статическая копия множества записей может использоваться для поиска данных или генерации отчетов. Добавления, изменения или удаления записи другим пользователям не видны	Двунаправленный
`ctOpenForwardOnly`	Однонаправленный курсор (настройка по умолчанию). Курсор идентичен статическому, за исключением того, что перемещение по записям доступно только в одном направлении — вперед. Это весьма быстрый курсор при условии, что планируется только один проход по множеству данных	Однонаправленный
`ctKeySet`	Поведение подобно динамическому курсору, за исключением того, что вы не можете видеть записи, вносимые другими пользователями. Записи, удаленные другими пользователями, также недоступны	Двунаправленный

Внимание!

Для обеспечения пакетной передачи данных курсор должен находиться в состоянии `ctKeySet` или `ctStatic`.

В результате работы с записями, помещенными в буфер, часть из них может быть модифицирована, часть удалена, часть оставлена без изменений. Поэтому в ADO предусмотрен фильтр, позволяющий отбирать записи из множества `Recordset` не на основе фактических данных, содержащихся в полях записи, а путем контроля технического показателя состояния модифицируемой строки (см. свойство `FilterGroup` и табл. 23.8).

Для отправки содержимого кэша в таблицы базы данных для сохранения вызывается процедура

```
procedure UpdateBatch(AffectRecords: TAffectRecords = arAll);
type TAffectRecords = (arCurrent, arFiltered, arAll, arAllChapters);
```

По умолчанию единственный параметр процедуры установлен в состояние `arAll` — это приказ обработать все строки из буфера. Значение `arCurrent` отправит для сохранения только текущую строку, `arFiltered` — только строки, отобранные в результате фильтрации, `arAllChapters` — воздействует на все записи во всех подчиненных множествах `Recordset`, включая даже те строки, которые скрыты в результате применения фильтра.

Метод, специализирующийся на удалении записей:

```
procedure DeleteRecords(AffectRecords: TAffectRecords = arAll);
```

Поведение процедуры вновь определяется параметром `AffectRecords`.

Замечание

Если множество записей обслуживается в пакетном режиме, то их физическое удаление произойдет только после вызова метода `UpdateBatch()`.

Для отмены пакетной передачи данных обращаемся к процедуре

```
procedure CancelBatch(AffectRecords: TAffectRecords = arAll);
```

Указанный метод можно вызывать не только в пакетном режиме, но и при работе в режиме немедленного обновления данных, но при условии, что в параметр `AffectRecords` передано значение `arCurrent`, что восстановит текущую запись. Значение `arFiltered` укажет на то, что метод будет применен только к записям, отобранным текущим фильтром. Значение `arAll` установит режим по умолчанию, в этом случае функция возвратит в исходное состояние все записи. Последний вариант `arAllChapters` определит, что обновление затронет все данные ADO, включая даже записи, которые не видны из-за включенного фильтра.

Подытожим все сказанное. Для организации пакетного обновления данных в ADO программист должен выполнить следующий перечень действий:

- Убедиться в том, что включен локальный кэш (свойство `CursorLocation` находится в режиме `clUseClient`).
- Свойство `LockType` у потомка класса `TCustomADODataSet` перевести в режим пакетного обновления — `ltBatchOptimistic`.
- Определить степень изоляции пользовательского набора данных. Для этого тип курсора `CursorType` установить в режим `ctKeySet` или `ctStatic`.
- При необходимости настроить фильтр `FilterGroup`.
- Для отправки пакета с измененными данными на сервер вызвать метод `UpdateBatch()`, при отказе от обновления измененные на клиентской стороне данные восстанавливаются методом `CancelBatch()`.

Если мы работаем с клиентским курсором (`CursorLocation=clUseClient`), то некоторые особенности возврата отредактированных данных на сервер зависят от состояния свойства

```
property MarshalOptions: TMarshalOption; //по умолчанию moMarshalAll
type TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
```

При установке свойства в состояние `moMarshalModifiedOnly` набор данных отправит на сервер только те записи, которые подверглись изменениям на клиентской стороне. Если же это свойство останется в состоянии по умолчанию (`moMarshalAll`), то в адрес сервера передаются все данные.

Портфельный режим обработки данных

В наборе данных ADO предусмотрена пара процедур, позволяющих организовать обслуживание набора данных (например, полученного от удаленного сервера)

на компьютере, временно отключенном от сети. Это так называемый режим портфельной обработки данных. Пользователь загружает требуемые данные с сервера и сохраняет их в файл. Путешествуя на самолете из Калининграда во Владивосток, пользователь спокойно обрабатывает множество записей на ноутбуке. Во Владивостоке компьютер вновь включается в сеть и сбрасывает обновленные данные на сервер:

```
procedure SaveToFile(const FileName: String = '';
                     Format: TPersistFormat = pfADTG);
procedure LoadFromFile(const FileName: WideString);
```

У обоих методов первый параметр `FileName` определяет имя файла. Формат хранения данных определяется параметром `Format`. Выбор у нас не богат, всего два варианта:

```
type TPersistFormat = (pfADTG, pfXML);
```

По умолчанию Delphi предлагает двоичный формат хранения данных ADTG (Advanced Data Tablegram). Второй вариант — это хорошо нам знакомый расширенный язык разметки XML.

Управление данными ADO в стиле Delphi

В отличие от мастера на все руки класса `TADODataSet`, классические для проектов БД Delphi компоненты-наборы данных `TADOTable`, `TADOQuery` и `TADOStoredProc` имеют узкую специализацию. Перечисленные классы продолжают славные традиции своих предшественников — старых компонентов BDE, разработанных еще для первых версий Delphi. Таблица `TADOTable` предназначена для организации работы с отдельной таблицей БД. Указанный компонент очень удобен в эксплуатации и как нельзя хорошо подходит начинающему разработчику БД, слабо разбирающемуся в структурированном языке запросов SQL. Компоненты-таблицы рекомендуется применять при проектировании простейших настольных приложений БД. В противовес компоненту-таблице, запрос `TADOQuery` специализируется на работе с БД с только с помощью SQL. Это настоящий универсал, с его помощью можно решать практически все задачи, возникающие как при разработке простейших локальных БД, так и при работе с серьезными клиент-серверными СУБД. Хранимая процедура `TADOStoredProc` предназначена только для БД, функционирующих под управлением СУБД класса Oracle, MS SQL Server, InterBase и т. п., а в настольных БД услуги компонента нам не понадобятся, потому что простейшие однопользовательские СУБД хранимые процедуры просто не поддерживают.

Чтобы задействовать `TADOTable`, `TADOQuery` и `TADOStoredProc` в наших проектах БД, следует совершить несколько последовательных операций. Первая заключается в обеспечении соединения с хранилищем данных. Если доступ к БД осуществляется при посредничестве компонента-соединителя `TADOConnection`, то необходимо установить ссылку на него с помощью свойства `Connection`. Иначе текст соединительной строки нужно передать в свойство `ConnectionString` компонента.

ЗАМЕЧАНИЕ

Хотя наборы данных ADO способны самостоятельно подключаться к хранилищу данных, для этой цели настоятельно рекомендуется задействовать соединение `TADOConnection`. Централизация рычагов управления в руках одного компонента значительно упрощает разработку и отладку проекта.

Таблица *TADOTable*

Таблица `TADOTable` — наиболее предпочтительный компонент для начинающего разработчика проектов БД, т. к. объединяет в себе удобство и простоту, присущую компонентам VCL, и универсальность наборов данных ADO. Одно из ключевых достоинств компонента — возможность создать приложение БД без применения языка SQL.

После подключения к хранилищу данных компонент `TADOTable` следует присоединить к таблице набора. Для этого предназначено свойство

```
property TableName: WideString;
```

После выбора имени таблицы остается вызвать метод `Open()`, и вот уже компонент готов предоставить программисту широкий спектр свойств и методов, позволяющих просматривать, редактировать, перемещаться по строкам, фильтровать и искать данные и т. д. Список свойств и методов компонента почти на 100% повторяет функционал своего идейного вдохновителя — дальнего родственника абстрактного класса `TDataSet`.

Обращение к свойствам и методам таблицы `TADOTable` автоматически конвертируются в инструкции SQL, которые отправляются в адрес провайдера. Вместе с тем, некоторые провайдеры БД поддерживают прямой доступ к таблицам. В этом случае целесообразно установить в `true` свойство

```
property TableDirect: Boolean; //по умолчанию false
```

Работа компонента несколько ускоряется, т. к. в случае прямого доступа отпадает необходимость применения инструкций `SELECT`.

Перевод таблицы в режим только для чтения осуществляет свойство

```
property ReadOnly: Boolean; //по умолчанию false
```

Подключение индексов по его имени или по названиям входящих в индекс полей осуществляется с помощью свойств

```
property IndexName: WideString; //имя индекса
property IndexFieldNames: string; //поля индекса
```

Список индексов, имеющихся в распоряжении таблицы, построит процедура

```
procedure GetIndexNames(List: TStrings);
```

Все остальные свойства, методы и обработчики событий компонента унаследованы от набора данных ADO `TCustomADODataSet` (см. рис. 22.3) и абстрактного класса `TDataSet`.

Ранее мы уже неоднократно применяли таблицу `TADOTable` в наших примерах. Предлагаю написать еще один пример, на этот раз демонстрирующий порядок построения отношения "главная – подчиненная таблица".

Организация отношения "главная – подчиненная таблица"

Один из полезных приемов, которым владеют компоненты-таблицы, и в том числе `TADOTable`, связан с построением отношения "главный – подчиненный" (master-detail). Подобное отношение имеется и в нашей демонстрационной БД книжного магазина (рис. 23.2). Таблице поставщиков `SUPPLIERS` подчиняется таблица контрактов `CONTRACTS`. Отношение подчиненности реализуется через ключевое поле `SUPPLIER_KEY`, хранящее ключ поставщика.

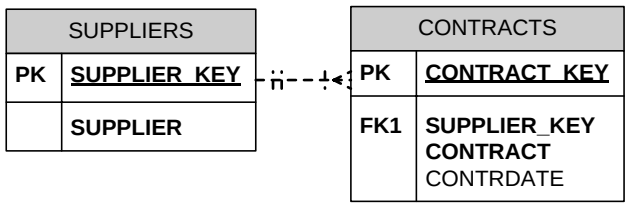


Рис. 23.2. Отношение "главный – подчиненный" на ER-модели

Функциональные возможности компонента `TADOTable` при желании позволяют формировать отношение "главный – подчиненный" даже без единой строки кода. Для претворения такого плана в жизнь достаточно помнить о существовании свойств `MasterSource` и `MasterFields`. Все перечисленные свойства имеют прямое отношение к подчиненной таблице и настраиваются в строгой последовательности.

Сначала в свойстве

```
property MasterSource: TDataSource;
```

указывают на источник данных, подключенный к главной таблице (компонент `TDataSource`, подробно рассмотренный в *главе 29*).

Затем программист определяет перечень полей, участвующих в организации взаимодействия между таблицами. Для этого в свойстве

```
property MasterFields: String;
```

указывают имена столбцов, входящих в первичный ключ главной таблицы.

Для практического претворения в жизнь нашего замысла можете взять за основу один из проектов из *главы 22*. Благо он уже обучен подключаться к базе данных Microsoft Access. Добавьте к предыдущей разработке новую форму и следующий перечень компонентов (рис. 23.3):

- Две таблицы `TADOTable`. Переименуем первую таблицу в `tblSUPPLIERS`, этот компонент станет обслуживать таблицу поставщиков. Вторую таблицу назовем `tblCONTRACTS`, мы позднее подключим этот компонент к таблице контрактов.
- Два источника данных `TDataSource`. Первый источник переименуем в `dsSUPPLIERS` и с помощью инспектора объектов подключим к таблице поставщиков (свойство `DataSet:=tblSUPPLIERS`). Второй источник назовем `dsCONTRACTS` и присвоим свойству `DataSet:=tblCONTRACTS`.
- Две сетки `TDBGrid`. Сетки позволят нам увидеть содержимое таблиц. Первый компонент назовем `dbgSUPPLIERS` и расположим в левой части формы. Под-

ключим сетку к источнику данных `dsSUPPLIERS` (свойство `DataSource:= dsSUPPLIERS`). Оставшаяся сетка получит имя `dbgCONTRACTS`, свойство `DataSource:= dsCONTRACTS`.

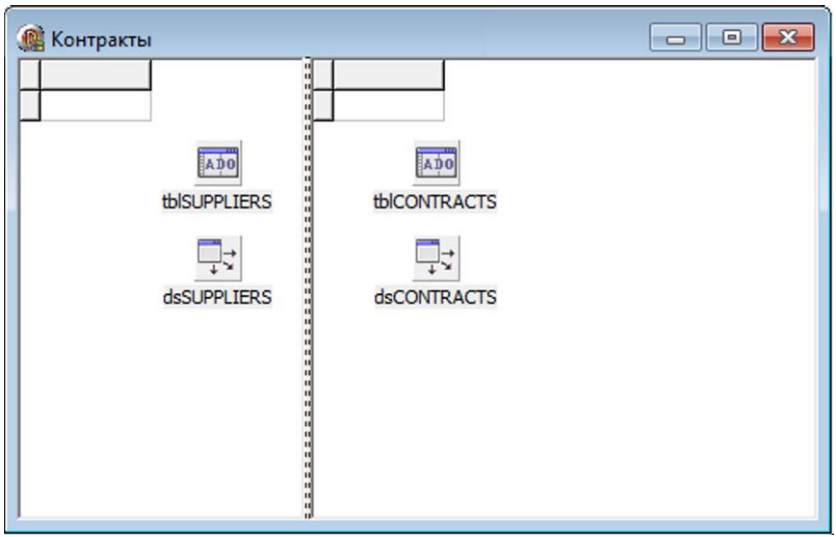


Рис. 23.3. Форма, демонстрирующая отношение "главный – подчиненный"

Переименуйте форму в `frmSuppliersContracts`, найдите событие `OnShow()`, вызываемое в момент вывода формы на экран, и внесите в него строки кода из листинга 23.11.

Листинг 23.11. Реализация отношения "главный – подчиненный"

```
procedure TfrmSuppliersContracts.FormShow(Sender: TObject);
begin
 with tblSUPPLIERS do //поставщики — главная таблица
 begin
  Connection:=frmDM.ADOConnection1;
  TableName:='SUPPLIERS';
  Open;
 end;

 with tblCONTRACTS do //контракты — подчиненная таблица
 begin
  Connection:=frmDM.ADOConnection1;
  TableName:='CONTRACTS';
  MasterSource:=dsSUPPLIERS;
```

```
  MasterFields:='SUPPLIER_KEY';
  Open;
 end;
end;
```

Программа готова. Запустите приложение на выполнение и внимательно проследите за его работой. Выбор в сетке `dbgSUPPLIERS` того или иного поставщика приводит к автоматическому отображению в сетке `dbgCONTRACTS` сведений о контрактах, заключенных с ним (рис. 23.4).

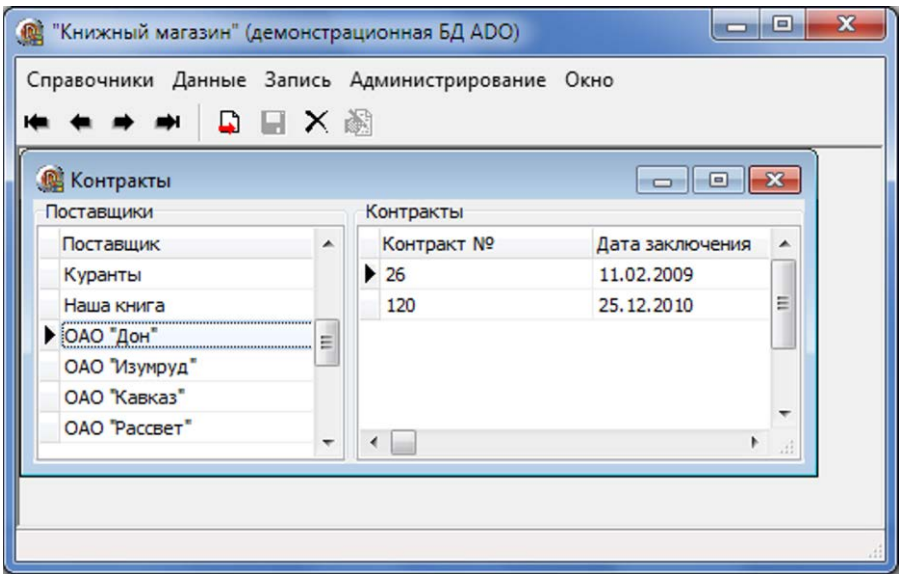


Рис. 23.4. Экранный снимок формы, реализующей отношение "главный – подчиненный"

Запрос *TADOQuery*

Таблица `TADOTable` незаменима в приложениях малой степени сложности, поскольку предоставляет разработчику весь минимально-необходимый пакет услуг по обслуживанию таблиц БД. Компонент умеет вставлять, редактировать, удалять, просматривать набор записей, осуществлять простейший поиск и ограничивать результирующий набор с помощью фильтров. Но при возникновении любой нестандартной задачи, например, связанной с выводом данных на основе объединения нескольких таблиц, компонент `TADOTable` становится бессилен. Это означает, что на сцену должен выходить запрос `TADOQuery`.

Компоненты-запросы — наиболее универсальные представители из семейства наборов-данных. Они умеют общаться с БД на ее родном языке — языке структурированных запросов SQL, значит, компонент `TADOQuery` умеет все.

Текст запроса хранится в списке

```
property SQL: TStrings;
```

Для заполнения списка строк инструкции SQL следует вспомнить о существовании пары методов: `Clear()` — очистка содержимого набора строк и `Add()` — добавить новую строку в конец набора (листинг 23.12).

Листинг 23.12. Пример получения данных с помощью `TADOQuery`

```
With ADOQuery1 do
Begin
 SQL.Clear;
 SQL.Add('Select * from writer');
 Open;
end;
```

Если запрос SQL основан на инструкции `Select`, то для его выполнения следует воспользоваться методом `Open()`. Это команда на получение запрашиваемых данных из таблиц базы данных. Во всех остальных случаях отправка инструкции SQL осуществляется методом

```
function ExecSQL: Integer;
```

Функция возвратит целое число — количество строк, оказавшихся под воздействием команды.

При формировании текста SQL-инструкции `TADOQuery` допускает применение параметров. Предположим, что в свойстве SQL компонента записана следующая команда редактирования данных таблицы заказов:

```
UPDATE ORDERS
SET
ORDERNUM=:ORDERNUM, ORDERDATE=:ORDERDATE
WHERE ORDER_KEY=:ORDER_KEY
```

Инструкция содержит три параметра (их названия начинаются с символа двоеточия `:`).

Для доступа к параметрам из кода программы следует воспользоваться коллекцией

```
property Parameters: TParameters;
```

каждый элемент которой описывается классом `TParameter`. Для обращения к значению отдельного параметра можно воспользоваться свойством `ParamValues` (листинг 23.13).

Листинг 23.13. Пример редактирования записи с помощью `TADOQuery`

```
procedure TfrmMain.UpdateOrder(ORDER_KEY, ORDERNUM: Cardinal;
                               ORDERDATE: TDate);
begin
With frmDM.ADOQuery1 do
```

```
begin
  Parameters.ParamValues['ORDERNUM']:=ORDERNUM;
  Parameters.ParamValues['ORDERDATE']:=ORDERDATE;
  Parameters.ParamValues['ORDER_KEY']:=ORDER_KEY;
  ExecSQL;
end;
end;
```

Предусмотрены и другие способы доступа к отдельному параметру коллекции: по названию параметра с помощью метода `ParamByName()` и по индексу параметра, направляемому в свойство `Items`.

Замечание

Применение коллекции параметров ADO `TParameters` и отдельного параметра ADO `TParameter` во многом аналогично классам `TParams` и `TParam`, рассмотренным в *главе 20*.

Для обеспечения автоматической регенерации коллекции параметров при любом изменении текста запроса SQL следует установить в состояние `true` свойство

```
property ParamCheck: Boolean;
```

Если SQL-команда будет выполняться многократно, то имеет смысл оптимизировать ее выполнение. Для этого после заполнения свойства SQL компонента следует установить в `true` свойство

```
property Prepared: Boolean;
```

Такое действие заставит компонент `TADOQuery` отправить серверу БД текст SQL-запроса. Получив инструкцию, СУБД подготовит ее к выполнению — откомпилирует и присвоит идентификатор вызова. Следующий раз при поступлении этого запроса от клиентского приложения сервер выполнит заранее откомпилированную версию SQL-инструкции.

О количестве записей, попавших под воздействие инструкции SQL, нам подскажет свойство

```
property RowsAffected: Integer;
```

Строки, полученные в результате выполнения инструкции SQL, можно перенаправить в другой набор данных с помощью свойства

```
property DataSource: TDataSource;
```

В остальном работа с компонентом практически ничем не отличается от его коллег `TSQLQuery`, `TIBQuery` и `TQuery`.

Хранимая процедура *TADOStoredProc*

Если компоненты-запросы можно отнести к разряду специалистов "широкого профиля" (способных выполнять практически все действия с БД), то хранимые процедуры — представители "узкоспециализированных" специалистов. Компонент

TADOStoredProc предназначен для вызова процедуры, заранее подготовленной на стороне сервера.

С точки зрения программной реализации компонента TADOStoredProc, его содержание на 99,9% повторяет код предка TCustomDataSet. Чтобы не быть голословным, предложу вашему вниманию заголовок и реализацию класса хранимой процедуры (листинг 23.14).

Листинг 23.14. Фрагмент модуля ADODB с описанием класса TADOStoredProc

```
 type TADOStoredProc = class(TCustomADODataSet)
  public
    constructor Create(AOwner: TComponent); override;
    procedure ExecProc;
  published
    property CommandTimeout;
    property DataSource;
    property EnableBCD;
    property ProcedureName: WideString read GetCommandText
                                        write SetCommandText;
    property Parameters;
    property Prepared;
  end;
//…
constructor TADOStoredProc.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Command.CommandType := cmdStoredProc;
  Command.CommandTextAlias := 'ProcedureName';
end;

procedure TADOStoredProc.ExecProc;
begin
  InitializeMasterFields(Self);
  Command.Execute;
end;
```

Как видите, все отличия хранимой процедуры TADOStoredProc от ее предка TCustomADODataset сводятся к двум дополнительным строкам конструктора и косметическим изменениям в реализации метода ExecProc().

Для вызова хранимой процедуры следует указать ее имя:

```
property ProcedureName: WideString;
```

При необходимости заполнить параметры обращаемся к коллекции

```
property Parameters: TParameters;
```

и вызываем метод

```
procedure ExecProc;
```

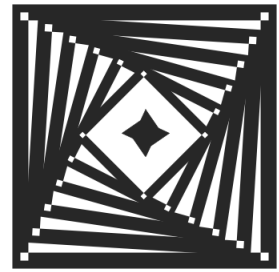
или, если процедура основана на инструкции `SELECT`, — метод `Open()`.

Резюме

Наряду со своими фирменными технологиями доступа к данным, построенными на основе компонентов dbExpress и InterBase, среда Delphi открыта для работы с альтернативными способами доступа к БД. В частности Delphi позволяет создавать высокоэффективные приложения баз данных, используя технологию Microsoft ActiveX Data Objects (ADO).

Компонент `TADOCommand` инкапсулирует командный объект ADO и служит для отправки в БД простейших инструкций SQL. Набор данных `TADODataSet` специализируется на представлении записей из таблиц БД и предлагает все необходимые услуги по управлению этими записями. Эти компоненты целиком и полностью соответствуют идеологии, заложенной Microsoft в объектную модель ADO, и способны решить подавляющее большинство задач, связанных с обработкой данных в БД.

Несмотря на самодостаточность компонентов `TADOCommand` и `TADODataSet`, создатели Delphi приняли решение включить в состав VCL компоненты ADO, исповедующие стиль программирования Delphi: `TADOTable`, `TADOQuery` и `TADOStoredProc`.



Глава 24

Доступ к БД InterBase

Если вы только начинаете изучать СУБД и мечтаете разработать свою первую клиент-серверную БД, то лучшей платформой для этого станет сервер InterBase. Тому несколько причин:

- компактность программного продукта, для инсталляции которого достаточно пары десятков мегабайт на жестком диске;
- в сравнении с такими монстрами, как Oracle, Informix и Microsoft SQL Server, система InterBase предъявляет минимальные требования к ресурсам компьютера, на котором он будет функционировать;
- на работе с InterBase специализируется более десятка компонентов Delphi;
- для взаимодействия клиентского приложения с сервером InterBase достаточно установить на компьютер всего одну библиотеку;
- поддержка не только Windows, но и Linux.

Замечание

Большинство компонентов InterBase вполне подходят для разработки клиентских приложений для бесплатных СУБД FireBird и Yaffil. Это объясняется тем, что FireBird и Yaffil были созданы на основе открытого кода InterBase 6, и поэтому хорошо совместимы со своим именитым предком.

СУБД InterBase позволяет создавать три варианта клиент-серверных приложений:

- Сервер БД и клиент развернуты на одном и том же компьютере.
- Классическая двухуровневая архитектура, в которой клиент получает доступ к серверу, установленному на другом сетевом компьютере.
- Трехуровневая архитектура, когда между сервером и клиентом появляется дополнительный уровень — сервер приложений.

Технические возможности последнего на момент написания книги сервера InterBase 9.0 приведены в табл. 24.1.

Таблица 24.1. *Основные технические характеристики InterBase 9.0*

Параметр	Значение
Максимальный размер БД	Определяется возможностями ОС и компьютера
Максимум таблиц в БД	32 640

Таблица 24.1 *(окончание)*

Параметр	Значение
Максимальный размер записи	64 Кбайт
Максимум хранимых версий одной и той же таблицы	255
Максимальное число кэшируемых страниц в БД	750 000
Максимальное число строк и колонок в таблице	2^{32}
Максимальное число индексов для одной таблицы	64
Максимальное число индексов для БД	2^{32}
Максимальный размер BLOB-данных	Определяется размером страницы в памяти. При размере страницы 8 Кбайт может достигать 32 Гбайт
Максимальное число вложенных вызовов хранимых процедур и триггеров	Для Windows — 750 Для Linux — 1000
Диапазон хранимых дат	От 1 января 100 г. н. э. до 29 февраля 32 768 г. н. э.

Доступ к базе данных, компонент *TIBDatabase*

Компонент `TIBDatabase` предназначен для организации доступа клиентского приложения к базе данных, обслуживаемой сервером InterBase. Наличие компонента `TIBDatabase` в клиентском приложении — обязательное условие, т. к. его услугами пользуются практически все остальные компоненты доступа к данным (рис. 24.1), для соединения с базой данных у них объявлено свойство `Database`. Весь механизм взаимодействия между сервером и клиентом содержится в специальном клиентском программном обеспечении, ядро которого составляет библиотека gds32.dll.

Замечание

Первая версия СУБД InterBase была создана талантливым программистом Джимом Старки (Jim Starkey) для компании Groton Database Systems. Именно поэтому в базовой библиотеке gds32.dll применяется аббревиатура gds, кроме того, gds зачастую используется в качестве расширения для файла с данными.

Для работы клиентского приложения InterBase необходимо наличие на компьютере библиотеки gds32.dll. При разработке программного обеспечения следует учитывать, что в составе клонов InterBase (СУБД FireBird и Yaffil) также имеются одноименные библиотеки gds32.dll, которые несовместимы с файлом библиотеки InterBase. Если вы хотите, чтобы ваше клиентское приложение стало максимально универсальным и смогло работать как с InterBase, так и с FireBird и Yaffil, снабдите его возможностью выбирать нужную библиотеку.

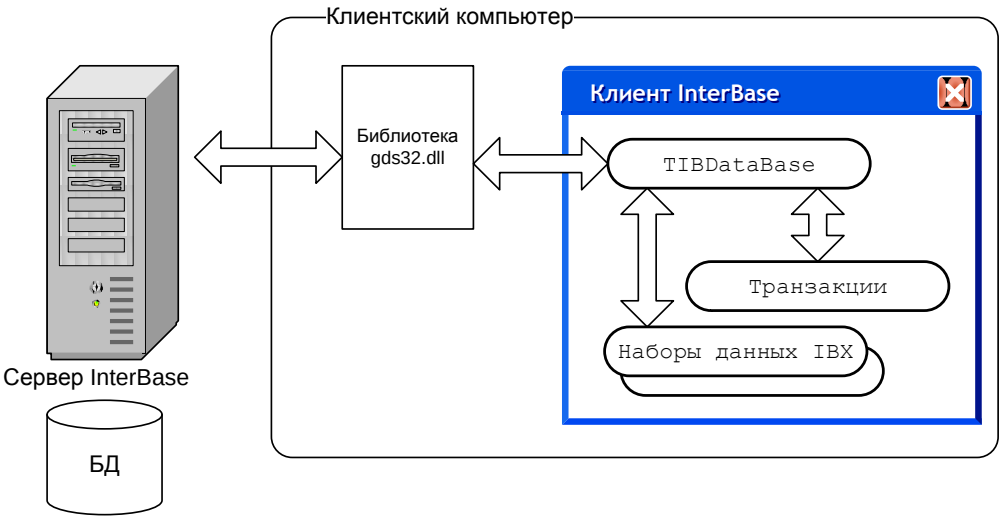


Рис. 24.1. Место компонента `TIBDataBase` в приложении БД

Кроме собственно организации доступа приложения к БД, компонент `TIBDatabase` позволяет:

- устанавливать соединение и разрывать соединение с сервером;
- осуществлять регистрацию пользователя на сервере;
- управлять транзакциями;
- создавать и удалять БД;
- получать информацию об основных объектах БД;
- реагировать на возникающие события.

Выбор диалекта ISQL

Как и все современные СУБД, сервер InterBase полностью поддерживает SQL-92, при этом существенно расширяет набор инструкций стандартного структурированного языка запросов. В рамках глав, посвященных InterBase, невозможно даже поверхностно поведать об особенностях InterBase SQL (ISQL), тем более, что к настоящему времени уже сменилось три диалекта ISQL. Отметим, что первая версия ISQL предназначалась для InterBase 5 и более ранних релизов сервера. Второй диа-

лект сыграл переходную роль и, по сути, не использовался. И, наконец, начиная с версии 7, InterBase работает на третьем по счету диалекте ISQL. По умолчанию компонент `TIBDatabase` нацелен на работу с ISQL 3, но если потребуется обслуживать более ранние версии InterBase, то понизьте номер диалекта в свойстве

```
property SQLDialect: Integer; //по умолчанию 3
```

Выбор диалекта осуществляется до подключения компонента к базе данных. Сразу оговорюсь, что все примеры этой книги ориентированы на ISQL 3. У свойства `SQLDialect` есть коллега:

```
property DBSQLDialect: Integer; //только для чтения
```

Но это свойство доступно только для чтения. При несоответствии уровней диалекта сервера и клиента `TIBDatabase` компонент генерирует событие

```
property OnDialectDowngradeWarning: TNotifyEvent;
```

Создание и уничтожение базы данных

Компонент `TIBDatabase` вполне способен справиться с задачей создания новой базы данных. Для этого предназначена процедура

```
procedure CreateDatabase;
```

Перед вызовом метода программист должен задать диалект, местоположение БД и регистрационные данные пользователя (листинг 24.1).

Листинг 24.1. Создание БД на локальном сервере InterBase

```
with IBDatabase1 do
 begin
   Params.Clear;                        //очистка параметров
   SQLDialect:=3;                       //диалект ISQL
   DatabaseName:= 'localhost:c:\booksmarket.gdb';//файл с БД
   Params.Add('user_name=sysdba');      //имя пользователя
   Params.Add('password=masterkey');    //пароль
   Params.Add('lc_ctype=WIN1251');      //кодировка
   CreateDatabase; //создаем БД с заданными параметрами
 end;
```

Во время создания новой БД компонент `TIBDatabase` должен быть свободен (не состоять в соединении ни с какой базой данных). Заметим, что после вызова метода `CreateDatabase()` компонент не только создаст БД, но и автоматически подключится к ней.

Удаляет базу данных метод

```
procedure DropDatabase;
```

Перед обращением к процедуре компонент `TIBDatabase` должен подключиться к удаляемой БД.

Соединение с базой данных

Для организации соединения с базой данных, в первую очередь, необходимо указать имя БД. Для этого предназначено свойство

```
property DatabaseName: string
```

Имя определяется названием файла базы данных, а формат его описания зависит от того, под какой операционной системой выполняется СУБД InterBase, и от сетевого протокола, с помощью которого осуществляется доступ к серверу. В простейшем случае, когда база данных размещается на том же компьютере, на котором установлено наше приложение, мы ограничиваемся обычным путем к файлу:

```
IBDatabase.DatabaseName:= 'C:\IBData\booksmarket.gdb';
```

Но если InterBase выполняется на удаленном сервере, то порядок определения имени слегка усложняется. Для соединения с СУБД мы можем выбирать любой из наиболее распространенных сетевых протоколов: TCP/IP или NetBEUI. По возможности предпочтение следует отдавать современному протоколу TCP/IP.

При подключении к сетевой БД с помощью TCP/IP путь к файлу должен начинаться с указания сетевого имени компьютера:

```
IBDatabase.DatabaseName:= 'IBComputer:C:\IBData\booksmarket.gdb';
```

Если по какой-то причине стек протоколов TCP/IP не установлен, то организуем работу с помощью протокола NetBEUI, но тогда порядок описания имени базы данных немного изменится:

```
IBDatabase.DatabaseName:= '\\IBComputer\C:\IBData\booksmarket.gdb';
```

Параметры соединения передаются при посредничестве свойства

```
property Params: TStrings;
```

Во время проектирования (чтобы избежать повторения процедуры регистрации при каждом перезапуске приложения) в это свойство можно внести данные об имени и пароле пользователя, не забыв отключить автоматический вызов диалога регистрации, установив в `False` свойство

```
property LoginPrompt : Boolean;
```

Для обращения к отдельным параметрам соединения можно обратиться к свойству

```
property DBParamByDPB[const Idx: Integer]: String;
```

В свойство следует передать имя соответствующей константы, например `user_name`. Все константы объявлены в заголовочном модуле `IBHeader`.

Для очень забывчивых разработчиков может пригодиться метод

```
procedure CheckDatabaseName;
```

Он проверяет факт заполнения свойства `DatabaseName`, и если свойство пусто — вызывает исключительную ситуацию.

Завершив все подготовительные операции, вызываем метод

```
Procedure Open;
```

либо переводим в `true` свойство

```
Property Connected : Boolean;
```

Ко всему прочему свойство `Connected` позволяет проверять факт соединения с БД (листинг 24.2).

Листинг 24.2. Пример подключения к локальной БД

```
procedure TfrmDataModule.DataModuleCreate(Sender: TObject);
begin
 with IBDatabase1 do
 begin
  DatabaseName:= 'C:\IBData\IBDemoDB.gdb';
  LoginPrompt:=False;
  Params.Clear;
  Params.Add('user_name=sysdba');
  Params.Add('password=masterkey');
  Connected:=True;
 end;
end;
```

Подчеркну, что такой код применяется только во время разработки проекта, а окончательная версия нашего программного продукта просто обязана потребовать от пользователя ввода регистрационных данных.

> ***ВНИМАНИЕ!***
>
> Для доступа к серверу InterBase необходимо указать имя и пароль пользователя. По умолчанию права системного администратора предоставит следующая учетная запись: пользователь "*SYSDBA*", пароль "*masterkey*".

Обычно программисты предпочитают управлять ходом соединения клиентского приложения с БД вручную в коде программы. Но зачастую контакт с базой данных необходим и во время разработки, например, для инициализации свойств и параметров компонентов хранимых процедур или для генерации списка статических полей у компонентов-наборов даных. После этого перед стартом программы важно не забыть вернуть свойство Connected в состояние false, в противном случае Delphi сгенерирует исключительную ситуацию. Если подключение/отключение нужно выполнить пару раз за день, то ничего страшного, но если проект перекомпилируется многократно… К счастью, в компоненте TIBDatabase острота проблемы существенно снижена благодаря свойству

```
property AllowStreamedConnected : Boolean; //по умолчанию true
```

Свойство предназначено исключительно для упрощения процесса отладки проекта. Переведя свойство `AllowStreamedConnected` в состояние `false`, мы освобождаемся от необходимости постоянно включать/отключать `Connected` во время разработки. Компонент самостоятельно, без какого-то постороннего вмешательства, отсоединится от базы данных в момент старта проекта (ожидая

явной команды на соединение в коде программы) и вновь подключится к БД после его остановки.

Еще одно полезное качество класса `TIBDatabase` — безопасное тестирование соединения. В процессе подготовки соединения перед вызовом метода `Open()` целесообразно вызвать проверочный метод:

```
procedure TestConnected: Boolean;
```

Процедура проверит корректность параметров соединения и при успехе вернет значение `true`, при ошибке мы получим `false`, но (что очень важно) исключительная ситуация сгенерирована не будет.

Для настройки параметров соединения во время разработки проекта программист может обратиться к контекстному меню компонента и воспользоваться пунктом **Database edittor** (рис. 24.2).

С процессом соединения связано два события:

```
property BeforeConnect: TNotifyEvent; //перед началом соединения
property AfterConnect: TNotifyEvent;  //после успешного соединения
```

Они могут пригодиться для инициализации других компонентов проекта.

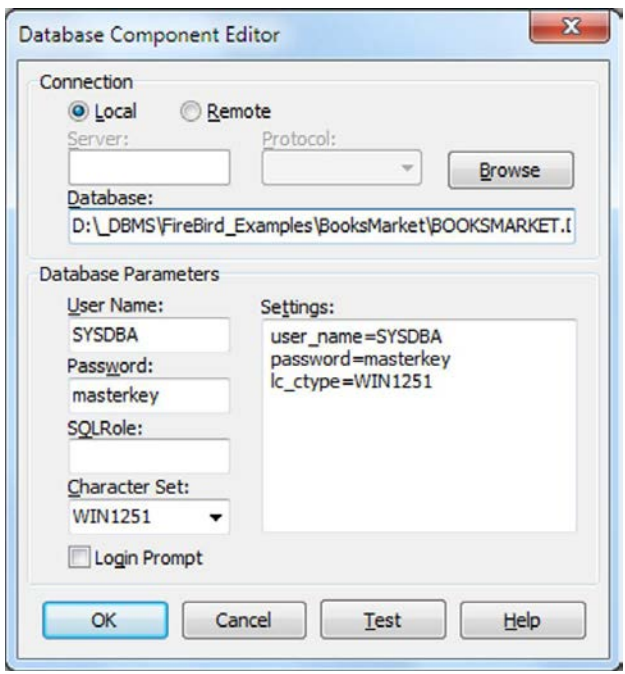


Рис. 24.2. Редактор настройки соединения

Регистрация пользователя

Для вызова штатного диалога регистрации пользователя установите в `true` свойство `LoginPrompt`.

В этом случае обращение к методу `Open()` инициирует событие:

```
property OnLogin: TIBDatabaseLoginEvent;
TIBDatabaseLoginEvent = procedure(Database: TIBDatabase;
                                  LoginParams: TStrings) of object;
```

Регистрационные данные пользователя направляются в параметр `LoginParams`.

Допустим, что у нас имеется отдельная форма `TfrmUserLogin` со строками ввода `edUser:TEdit` и `edPassword:TEdit`, предназначенными для ввода имени и пароля пользователя. Тогда процесс регистрации можно организовать так, как предложено в листинге 24.3.

Листинг 24.3. Регистрация пользователя на сервере InterBase

```
procedure TfrmDataModule.IBDatabase1Login(Database: TIBDatabase;
                                          LoginParams: TStrings);
var frmUserLogin: TfrmUserLogin;
begin
  frmUserLogin:=TfrmUserLogin.Create(nil); {создаем регистрационную форму}
  if frmUserLogin.ShowModal=mrOK then
    with LoginParams do
    begin
     Clear;
     Add('user_name='+frmUserLogin.edUser.Text); //имя пользователя
     Add('password='+frmUserLogin.edPassword.Text); //пароль
    end;
    frmUserLogin.Release;
end;
```

Разрыв соединения

Кроме анализа состояния свойства `Connect` для проверки наличия соединения с БД программист может использовать методы:

```
procedure CheckActive;   //проверка активности соединения
procedure CheckInactive; //проверка пассивности
```

Особенность процедур `Check…` в том, что при невыполнении условия проверки они генерируют исключительную ситуацию (листинг 24.4).

Листинг 24.4. Шаблон кода завершения работы с БД

```
Uses IB;
…
Procedure …
Begin
try
```

```
  IBDatabase1.CheckInactive;
  {какие-то операции, осуществление которых возможно только
   при отсутствии соединения}
 except
   on EIBClientError do //проверяем класс ИС
      begin
           MessageBox(Handle,
                      PChar('Данная операция невозможна
                             при активном соединении!'),
                      PChar('Ошибка'),
                      MB_ICONERROR+MB_OK);
      end;
end;
end;
```

Для отключения от базы данных обращаемся к методу `Close()` либо переводим в `false` свойство `Connected`. Иногда возникают случаи, когда компонент `TIBDatabase` категорически отказывается порвать все отношения с так понравившейся ему базой данных. Это может быть объяснено внутренними проблемами сервера, например, затянувшейся транзакцией. Что делать, когда нам просто позарез нужно прекратить работу клиентского приложения, а оно не собирается слушаться? Обращаемся к методу

```
procedure ForceClose;
```

Процедура форсирует процесс отключения настолько, насколько это возможно.

Для отключения всех активных наборов данных с сохранением соединения с БД вызываем метод

```
procedure CloseDataSets;
```

После обращения к методу все обслуживаемые нашим компонентом потомки `TDataSet` перейдут в пассивный режим (`Active=false`), но `TIBDatabase` по-прежнему сохранит контакт с базой.

Процесс разрыва соединения сопровождается парой событий:

```
property BeforeDisconnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
```

События возникают соответственно перед началом разрыва соединения и по факту разрыва.

При необходимости можно ограничить время простоя соединения с помощью свойства

```
property IdleTimer: Integer; //по умолчанию 0 — неограниченно
```

Время исчисляется в миллисекундах. По умолчанию свойство установлено в ноль, т. е. ограничение отсутствует. По истечении времени простоя генерируется событие

```
property OnIdleTimer: TNotifyEvent;
```

которое можно использовать для разрыва соединения с БД.

Информирование о составе БД

Для получения перечня таблиц из состава БД следует вызвать процедуру

```
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);
```

Данные передаются в список `List`. Если необходимо собрать сведения о системных таблицах, передаете `true` в параметр `SystemTables`. Зная имя таблицы, мы сможем запросить список ее полей:

```
procedure GetFieldNames(const TableName: string; List: TStrings);
```

Здесь `TableName` — название таблицы, `List` — ссылка на список, в который будут собраны данные. Допустим, что на форме проекта `frmTablesInfo: TfrmTablesInfo` размещены два компонента-списка `TListBox`, предназначенные для хранения перечня таблиц (компонент `lbTables`) и списка полей таблицы (компонент `lbFields`). Кроме того, на форме имеется компонент `CheckBox1:TCheckBox`, он отвечает за включение/отключение вывода информации о системных таблицах. Подключение к БД осуществляет компонент `IBDatabase1`, он размещен на модуле данных проекта `DataModule1`.

В листинге 24.5 приведен пример сбора данных о таблицах.

Листинг 24.5. Получение имен таблиц

```
procedure TfrmTablesInfo.FormShow(Sender: TObject);
begin
lbTables.Clear; {очищаем список}
DataModule1.IBDatabase1.GetTableNames(lbTables.Items, CheckBox1.Checked);
if lbTables.Count> 0 then
{если список не пуст, "щелкнем" по его первому элементу}
   begin
    lbTables.ItemIndex:=0;
    lbTables.OnClick(Self);
   end;
end;
```

Щелчок по элементу списка таблиц инициирует процесс сбора данных о полях выбранной таблицы (листинг 24.6).

Листинг 24.6. Получение имен полей таблицы

```
procedure TfrmTablesInfo.lbTablesClick(Sender: TObject);
var TableName : String;
begin
if lbTables.ItemIndex>-1 then
   begin
      lbFields.Clear;
```

```
      TableName:=lbTables.Items.Strings[lbTables.ItemIndex];
      DataModule1.IBDatabase1.GetFieldNames(TableName,lbFields.Items);
   end;
end;
```

В заключение в инспекторе объектов свяжите событие `OnClick()` компонента `CheckBox1` с событием `OnShow()` формы.

На рис. 24.3 представлен экранный снимок формы с информацией о БД "Книжный магазин". Заметьте, что кроме таблиц спроектированной нами демонстрационной БД (см. рис. 6.7) в список вошло много системных таблиц InterBase (их названия начинаются с аббревиатур MON$ и RDB$).

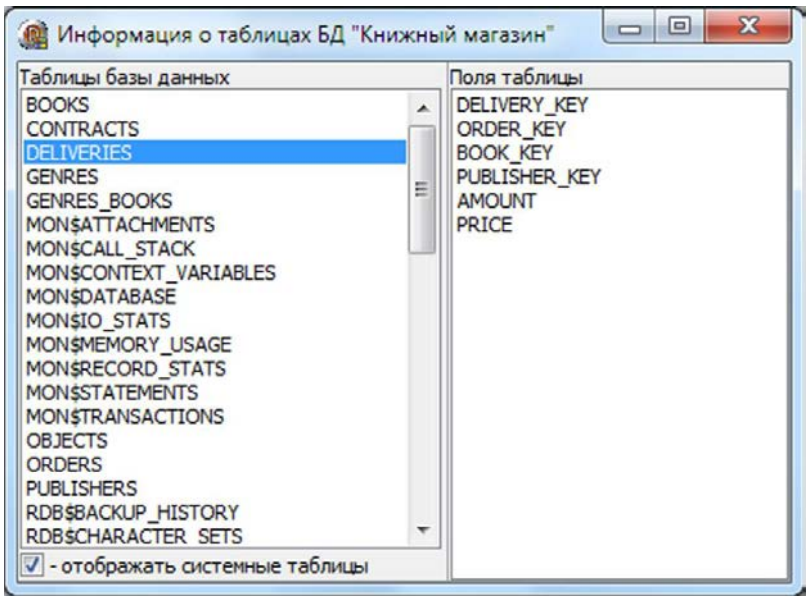


Рис. 24.3. Экранный снимок формы сбора информации о таблицах БД

Замечание

Для получения перечня таких метаданных, как список таблиц, представлений, правил, индексов, целесообразно использовать компонент `TIBExtract`. Более полную информацию о технических характеристиках БД предоставит компонент `TIBDatabaseInfo`.

Управление транзакциями

Для управления транзакциями класс `TIBDatabase` задействует помощников — компоненты `TIBTransaction`. Список всех транзакций, связанных с `TIBDatabase`, мы обнаружим в свойстве

```
property Transactions[Index: Integer]: TIBTransaction;
```

О количестве доступных транзакций все известно свойству

```
property TransactionCount: Integer; //только для чтения
```

Для выяснения индекса транзакции `TR` в массиве `Transactions` необходима функция

```
function FindTransaction (TR: TIBTransaction): Integer;
```

В поиске транзакции по умолчанию нам поможет метод

```
function FindDefaultTransaction: TIBTransaction;
```

Для того чтобы назначить транзакцию по умолчанию, обратитесь к свойству

```
property DefaultTransaction: TIBTransaction;
```

Все подключенные к `TIBDatabase` компоненты наборы данных также станут использовать эту транзакцию.

Список `Transactions` можно пополнить очередной транзакцией `TR` непосредственно из кода программы:

```
function AddTransaction(TR: TIBTransaction): Integer;
```

При успешном завершении метод возвратит новый индекс.

Обратную операцию, исключающую транзакцию из списка, осуществляет процедура

```
procedure RemoveTransaction(Idx: Integer);
```

Метод отсоединит (но не уничтожит) транзакцию с индексом `Idx` от базы данных. Процедура

```
procedure RemoveTransactions;
```

отключает все компоненты `TIBTransaction`.

Контроль за событиями

С компонентом `TIBDatabase` допустимо связывать произвольное число объектов `TIBEvents`, отслеживающих возникновение событий в базе данных InterBase:

```
procedure AddEventNotifier(Notifier: IIBEventNotifier);
```

В роли внешних контролеров могут выступать экземпляр объекта `TIBEvents` либо интерфейс `IIBEventNotifier`.

Процедура

```
procedure RemoveEventNotifier(Notifier: IIBEventNotifier);
```

удаляет связь и уведомляет о событии.

Совместная работа с SQL монитором

В состав компонентов IBX входит элемент управления `TSQLMonitor`, позволяющий проводить мониторинг базы данных и управлять сведениями о выполнении ISQL-запросов сервером InterBase.

Особенности мониторинга определяются с помощью флагов трассировки:

```
property TraceFlags: TTraceFlags; //по умолчанию [tfQExecute,tfConnect]
```

Программист включает флаги, соответствующие интересующим его операциям (табл. 24.2).

Таблица 24.2. *Возможные значения* `TTraceFlags`

Флаг	Описание
`tfQPrepare`	Мониторинг подготовительных операций
`tfQExecute`	Отслеживать выполнение инструкций SQL
`tfQFetch`	Контроль за операциями чтения
`tfError`	Мониторинг за ошибками сервера
`tfStmt`	Обрабатывать все команды SQL
`tfConnect`	Контролировать процесс соединения и разрыва соединения приложения с БД
`tfTransact`	Отслеживать команды управления транзакциями: `StartTransaction`, `Commit` и `Rollback`
`tfBlob`	Мониторинг операций с BLOB-данными
`tfService`	Обслуживать сервисы
`tfMisc`	Мониторинг за всеми операциями, не связанными с рассмотренными ранее флагами

Транзакция, компонент *TIBTransaction*

Компонент `TIBTransaction` предоставляет услуги по управлению транзакциями из клиентского приложения с использованием одного или нескольких соединений с БД. В услугах транзакций нуждаются все наборы данных InterBase (потомки класса `TIBCustomDataSet`) и компонент `TIBSQL`.

Внимание!

Клиентское приложение должно как минимум содержать один компонент-транзакцию. Но одного компонента достаточно только для простейших проектов. В реальности компонентов `TIBTransaction` должно быть ровно столько, сколько требует логика работы приложения. Конечно же, не стоит присоединять личную транзакцию к каждой таблице, хранимой процедуре и запросу InterBase, это явный перебор. Имеет смысл выделить индивидуальные компоненты `TIBTransaction` для обслуживания наиболее часто выполняемых SQL-команд и запросов.

Перечень всех обслуживаемых соединений с БД мы обнаружим в свойстве

```
property Databases[Index: Integer]: TIBDatabase;
```

Число связанных с транзакцией экземпляров `TIBDatabase` предоставит свойство

```
property DatabaseCount: Integer;
```

Ссылка на базу данных по умолчанию хранится в свойстве

```
property DefaultDatabase: TIBDatabase;
```

Для программного пополнения списка обслуживаемых соединений с БД используем метод

```
function AddDatabase(db: TIBDatabase): Integer;
```

Для удаления отдельного экземпляра `TIBDatabase` из списка `Databases` вызовем процедуру

```
procedure RemoveDatabase(Idx: Integer);
```

Метод требует указать индекс исключаемого компонента. Если этот индекс вам неизвестен, то воспользуйтесь методом

```
function FindDatabase (db: TIBDatabase): Integer;
```

Функция просмотрит весь список `Databases` с целью поиска в нем экземпляра интересующего нас экземпляра `TIBDatabase` и возвратит его порядковый номер в списке.

Для полной очистки списка понадобится процедура

```
procedure RemoveDatabases;
```

О количестве наборов данных `InterBase` и экземпляров `TIBSQL`, обслуживаемых базой данных `TIBDatabase`, мы сможем судить по свойству

```
property SQLObjectCount: Integer; //только для чтения
```

Доступ к SQL-объектам предоставит свойство

```
property SQLObjects[Index: Integer]: TIBBase; //только для чтения
```

Управление транзакцией

Об активности транзакции судят по свойству

```
property Active: Boolean;
```

Свойство позволяет стартовать транзакцию даже во время визуального проектирования, поэтому не стоит злоупотреблять доброжелательностью компонента (это может сослужить плохую службу), тем более что возврат `Active` в состояние `false` аварийно завершает транзакцию. Поэтому будем полагать, что свойство `Active` следует использовать только в режиме для чтения.

Для явного запуска новой транзакции следует обращаться к процедуре

```
procedure StartTransaction;
```

При нормальном завершении транзакции вызываем метод

```
procedure Commit;
```

фиксирующий изменения в БД.

В противном случае, для отката транзакции идем на поклон к методу

```
procedure Rollback;
```

Листинг 24.7 демонстрирует порядок управления транзакцией при вставке новой строки в таблицу авторов.

Листинг 24.7. Пример работы с компонентом `TIBTransaction`

```
Procedure Writer_InsertRecord(Writer:string);
begin
 if IBTransaction1.Active=False then
 begin
 IBTransaction1.StartTransaction; //старт транзакции
 try
  with PR_WRITERS_INSERT do // операция с хранимой процедурой
    begin
    Params.ParamByName(':IN_WRITER').Value:= Writer;
    ExecProc;
    end;
  IBTransaction1.Commit; //фиксируем транзакцию
 except {Действия при возникновении ошибки}
  IBTransaction1.Rollback; //откат транзакции
 end;
 end;
end;
```

Для проверки, выполняется ли в данный момент времени транзакция, обращаемся к свойству

```
property InTransaction: Boolean; //только для чтения
```

Значение `true` свидетельствует, что транзакция еще не завершена.

Если мы работаем с InterBase не ниже версии 6, то для отката всех изменений в данных, осуществленных в рамках текущей транзакции, разрешено воспользоваться процедурой

```
procedure RollbackRetaining;
```

При этом сохраняется контекст транзакции, другими словами, транзакция не завершается и может быть продолжена. Для фиксации изменений, сделанных транзакцией, с сохранением ее контекста используйте метод

```
procedure CommitRetaining;
```

Еще раз заострю внимание читателя на том, что метод не завершает транзакцию, а просто создает точку сохранения (к которой можно возвратиться с помощью `RollbackRetaining`). Транзакция останется открытой до тех пор, пока не будет вызван метод `Commit()` или `Rollback()`.

Компонент `TIBTransaction` достаточно "умен" для того, чтобы самостоятельно начинать и завершать выполнение транзакции. Степень доверия программиста к компоненту определяется свойствами:

```
property AllowAutoStart:Boolean; //по умолчанию true
property AutoStopAction : TAutoStopAction; //по умолчанию saNone
```

Свойство AllowAutoStart, установленное в состояние true, указывает компоненту на то, что при необходимости он должен самостоятельно запустить транзакцию. Свойство AutoStopAction определит действие по умолчанию в момент завершения транзакции. Варианты допустимых значений TAutoStopAction приведены в табл. 24.3.

***Таблица 24.3.** Возможные значения TAutoStopAction*

Значение	Описание
saNone	Компонент не может самостоятельно управлять завершением транзакций
saRollback	Компонент имеет право на самостоятельность при откате транзакции
saCommit	Компоненту разрешено фиксировать транзакцию
saRollbackRetaining	Возврат к точке сохранения. Поддерживается, начиная с InterBase 6.0 Транзакция не завершается, пока есть незакрытые наборы данных. Все изменения в связанных с транзакцией данных отменяются. Другими словами, производится откат транзакции с сохранением ее контекста
saCommitRetaining	Транзакция не завершается до тех пор, пока имеются незакрытые наборы данных. Все изменения в охваченных транзакцией записях фиксируются. Контекст транзакции сохраняется

ВНИМАНИЕ!

Если транзакция обслуживает компонент TIBSQL, то свойство AutoStopAction должно быть установлено в состояние saNone.

Тайм-аут транзакции

Для того чтобы выполнение транзакция не длилось вечно в свойстве

```
property IdleTimer: Integer;
```

назначается время тайм-аута в миллисекундах. По истечению тайм-аута судьба транзакции определяется состоянием свойства

```
property DefaultAction: TTransactionAction;
```

Это операция по умолчанию, возможные варианты которой описаны в табл. 24.4. Кроме того, по завершении времени тайм-аута генерируется событие

```
property OnIdleTimer: TNotifyEvent;
```

В рамках события следует уведомить пользователя о результатах выполнения транзакции.

Таблица 24.4. *Варианты действий по истечении тайм-аута* `TTransactionAction`

Значение	Описание
`TARollback`	Откат транзакции
`TACommit`	Фиксация изменений
`TARollbackRetaining`	Поддерживается, начиная с InterBase 6.0. Откат транзакции с удержанием ее контекста
`TACommitRetaining`	Фиксация транзакции с сохранением контекста

Диагностика состояния транзакции

Факт наличия компонентов `TIBDataBase`, связанных с транзакцией, проверит процедура

```
procedure CheckDatabasesInList;
```

Если связанные базы данных отсутствуют, то генерируется исключительная ситуация.

Чтобы узнать, выполняется ли в данный момент транзакция, обращаемся к процедуре

```
procedure CheckInTransaction;
```

Как и все начинающиеся со слова "Check" методы компонентов InterBase, процедура отличается "отвратительным характером" — она генерирует исключительную ситуацию, если транзакция не активна.

Обратную проверку проводит метод

```
procedure CheckNotInTransaction;
```

который выясняет, завершена ли в данный момент транзакция.

Для принудительного вызова действия, указанного в свойстве `AutoStopAction` (см. табл. 24.3), следует вызвать метод

```
procedure CheckAutoStop;
```

Параметры транзакции

Прикладной интерфейс программирования InterBase позволяет наделять транзакции дополнительными параметрами. Доступ из кода приложения к буферу, содержащему параметры, обеспечивает свойство

```
property Params: TStrings;
```

или

```
property TPB: PChar; //только для чтения
```

В параметрах транзакции содержится информация об уровне изоляции транзакции, порядке использования точек сохранения, разрешение на редактирование данных. Для определения параметров транзакции проще всего воспользоваться ус-

лугами редактора транзакции, который вызывается из контекстного меню компонента `TIBTransaction` (рис. 24.4).

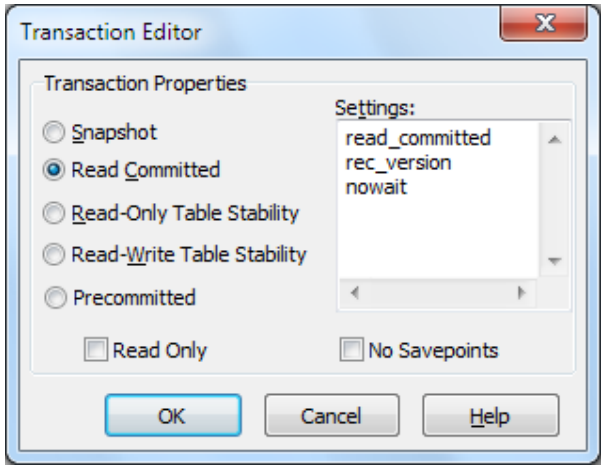


Рис. 24.4. Окно редактора параметров транзакции

О размере буфера нас проинформирует свойство

```
property TPBLength: Short;
```

Свойство

```
property Handle: TISC_TR_HANDLE;//только для чтения
```

возвращает дескриптор транзакции, который может пригодиться, если вы осуществляется низкоуровневый доступ к транзакции на уровне InterBase API.

Уровни изоляции транзакций InterBase несколько отличаются от требований стандарта SQL (см. табл. 8.4):

- **Read Committed** (неповторяемое чтение) — обеспечивает самую низкую степень изолированности. В этом режиме транзакция может видеть все изменения, сделанные другими транзакциями, при условии, что они зафиксированы. Фактически уровень Read Committed представляет собой единственное совпадение с пожеланиями SQL.
- **Pre committed** — соответствует уровню Read Committed в режиме только для чтения.
- **Snapshot** (моментальный снимок) — соответствует средней степени изоляции. Транзакция запоминает состояние БД на момент своего запуска и работает с этими данными, не обращая внимания на действия других транзакций. Основная особенность уровня в том, что он не блокирует записи — они остаются доступными другим транзакциям.
- **Read-Only Table Stability** и **Read-Write Table Stability** — гарантируют наибольшую степень изоляции. По аналогии с уровнем Snapshot в момент старта транзакции Table Stability создает моментальный снимок БД, но на этот раз транзакция блокирует записи. В режиме Read-Only операторы внутри транзак-

ции не могут модифицировать данные, в режиме Read-Write — модификации допускаются.

Замечание

Чтобы ваша транзакция видела изменения, сделанные транзакциями других пользователей, выбирайте уровень изоляции Read Committed (неповторяемое чтение).

Информация об объектах БД, компонент *TIBExtract*

Изучая компонент `TIBDatabase`, мы узнали о его способностях получать список таблиц БД и перечень входящих в них полей. Еще бóльшим спектром возможностей обладает элемент управления `TIBExtract`. Это прямой наследник класса `TComponent`, компонент описан в модуле `IBExtract`. Задача компонента — выведать у InterBase весь списочный состав объектов, имеющихся в БД, с подробным их описанием.

Для организации работы с компонентом программист должен подключить `TIBDatabase` к базе данных (свойство `Database`) и соединиться с транзакцией (`Transaction`), а далее наступает черед ключевого метода:

```
procedure ExtractObject(ObjectType : TExtractObjectTypes;
                        ObjectName : String = '';
                        ExtractTypes : TExtractTypes = []);
```

Здесь `ObjectType` — запрашиваемый у БД тип метаданных (табл. 24.5). Необязательный параметр `ObjectName` указывает имя конкретного объекта БД. Необязательный параметр `ExtractTypes` позволяет наложить дополнительный фильтр на тип извлекаемых данных (табл. 24.6).

Результат выполнения метода `ExtractObject()` помещается в свойство

```
property Items: TStrings;
```

Это список строк с текстовой информацией.

Таблица 24.5. *Перечень значений* `TExtractObjectTypes`

Значение	Описание
`eoDatabase`	Список всех определенных в БД объектов. Этот перечень можно дополнительно отфильтровать при помощи параметров `ObjectName` и `ExtractTypes`
`eoDomain`	Список всех доменов
`eoTable`	Если программист не воспользовался свойством `ObjectName` и оставил его пустым, то мы получим список всех таблиц БД. Если же в `ObjectName` были занесены данные, то процедура интерпретирует их как имя таблицы. В этом случае возвращается информация об этой таблице, причем тип информации должен быть определен в параметре `ExtractTypes`

Таблица 24.5 *(окончание)*

Значение	Описание
`eoView`	Если параметр `ObjectName` не использовался, возвращается список всех представлений, иначе мы будем проинформированы о представлении с именем `ObjectName`
`eoProcedure`	Список хранимых процедур
`eoFunction`	Список всех внешних функций. Если параметр `ObjectName` не пуст, то мы получим информацию об отдельной функции
`eoGenerator`	Список генераторов
`eoException`	Список исключений
`eoBLOBFilter`	Список фильтров BLOB
`eoRole`	Список ролей
`eoTrigger`	Перечень триггеров
`eoForeign`	Перечень внешних ключей
`eoIndexes`	Список индексов
`eoChecks`	Ограничения
`eoData`	Список всех данных с именем, определенным в параметре `ObjectName`

Таблица 24.6. *Описание* `TExtractTypes`

Значение	Описание
`etDomain`	При выборке метаданных о таблице включить информацию обо всех принадлежащих таблице доменах
`etTable`	Собрать данные только о таблице
`etRole`	При выборке метаданных о таблице включить информацию обо всех связанных с таблицей правилах
`etTrigger`	При выборке метаданных о таблице собирать данные о ее триггерах
`etForeign`	При выборке метаданных о таблице собирать данные о ее внешних ключах
`etIndex`	При выборке метаданных о таблице собирать данные о ее индексах
`etData`	При выборке метаданных о таблице получить ее данные
`etGrant`	При выборке метаданных о таблице включать информацию о правах пользователей
`etCheck`	При выборке метаданных о таблице собирать данные об ограничениях
`etAlterProc`	При сборе данных о хранимых процедурах исключить информацию о взаимных ссылках между процедурами

События InterBase, компонент *TIBEvents*

Одна из примечательных особенностей СУБД InterBase заключается в способности сервера генерировать события и уведомлять об этом подключенные к серверу клиентские приложения. Для определения события язык ISQL в распоряжение разработчика БД предоставляет инструкции `EVENT INIT` и `EVENT WAIT`. Первая регистрирует событие, вторая указывает приложению о необходимости ожидать возникновения события на сервере.

Для того чтобы клиентское приложение смогло среагировать на событие сервера, оно должно быть вооружено компонентом `TIBEvents`. Это наследник класса `TComponent`, описанный в модуле `IBEvents`.

Самым первым действием программиста должно быть подключение компонента к базе данных. Для этого предназначено свойство `Database`.

Затем в свойстве

```
property Events: TStrings;
```

соблюдая регистр символов, описываем перечень интересующих нас событий.

Заполнив список событий, переводим в `true` свойство

```
property Registered: Boolean;
```

или вызываем метод

```
procedure RegisterEvents;
```

Таким образом, приложение укажет серверу, что нас необходимо уведомлять об этих событиях (листинг 24.8).

> ***ВНИМАНИЕ!***
>
> Сервер InterBase отправит клиенту уведомление о событии только при корректном завершении транзакции (фиксации изменений командой `COMMIT`), в которой эти события произошли.

Листинг 24.8. Регистрация событий

```
with IBEvents1 do
begin
 Events.Clear();
 Events.Add('TestEvent1');
 Events.Add('TestEvent2');
 //…
 RegisterEvents();
end;
```

С этого момента, при возникновении на сервере зарегистрированного события на клиентской стороне вызывается процедура:

```
property OnEventAlert: TEventAlert;
```

```
type TEventAlert = procedure( Sender: TObject;
                              EventName: String;
                              EventCount: longint;
                              var CancelAlerts: Boolean);
```

Здесь `EventName` — название события, `EventName` информирует об общем количестве данных событий с момента последнего вызова `OnEventAlert()`. Передав в переменную `CancelAlerts` значение `true`, мы сможем указать серверу, что больше не нуждаемся в информации о его событиях. Для возобновления процесса информирования вновь установите в `true` свойство `Registered`.

Информация о БД, компонент *TIBDatabaseInfo*

При разработке административных модулей хорошую службу сослужит компонент `TIBDatabaseInfo`. Это наследник класса `TComponent`. Элемент управления описан в модуле `IBDatabaseInfo`. Подключившись к базе данных (с помощью свойства `Database`), компонент сможет проинформировать нас об основных характеристиках СУБД InterBase (табл. 24.7).

***Таблица 24.7.** Основные свойства компонента `IBDatabaseInfo`*

Свойство	Описание
Характеристики БД	
`property Allocation: Long;`	Количество страниц БД, распределенных в памяти
`property BaseLevel: Long;`	Второй байт числа содержит версию базы данных
`property DBFileName: String;`	Название файла базы данных
`property DBIinplementationClass : Long;`	Номер класса описания (значение в диапазоне от 1 до 12)
`property DBImplementationNo: Long;`	Номер описания
`property DBSiteName: String;`	Имя сайта БД, обычно соответствует имени компьютера
`property DBSQLDialect: Long;`	Диалект ISQL
`property NoReserve: Long;`	Поддержка резервирования страниц: 0 — да, 1 — нет
`property ODSMajorVersion: Long;`	Версия внутреннего формата файла (on-disk structure, ODS), старшее значение. Для InterBase версии 7.*x* – 11.*x*
`property ODSMinorVersion: Long;`	Версия ODS, младшее значение
`property PageSize: Long;`	Размер страницы

Таблица 24.7 *(окончание)*

Свойство	Описание
Характеристики БД	
property Readonly: Long;	Доступ только для чтения: 0 — да, 1 — нет
property Version: **String**;	Версия базы данных
Число операций с БД	
property ReadSeqCount: TStringList;	Число последовательных просмотров таблицы
property UpdateCount: TStringList;	Количество обновлений БД с момента последнего сохранения
property PurgeCount: TStringList;	Число удаленных записей
property InsertCount: TStringList;	Число вставок данных в БД с момента последнего сохранения
property DeleteCount: TStringList;	Возвращает количество удалений, отсчет начинается с момента последнего обновления БД
property BackoutCount: TStringList;	Возвратит число удаленных страниц
property ExpungeCount: TStringList;	Количество удаленных записей с момента последнего сохранения БД
property ReadIdxCount: TStringList;	Число чтений через индексы с момента последнего сохранения
Производительность	
property Fetches: Long;	Число чтений в локальный буфер памяти
property Marks: Long;	Число выполненных записей в кэш
property Writes: Long;	Количество постраничных записей
property Reads: Long;	Число чтений из БД
Характеристики окружения	
property CurrentMemory: Long;	Объем (в байтах) используемой СУБД памяти
property ForcedWrites: Long;	Режим операций: 0 — асинхронный, 1 — синхронный
property MaxMemory: Long;	Максимальный размер памяти (в байтах), занимаемый БД с момента последнего сохранения
property NumBuffers: Long;	Число выделенных буферов
property SweepInterval: Long;	Число зафиксированных транзакций
property UserNames: TStringList;	Список активных пользователей

Монитор SQL, *TSQLMonitor*

Компонент предназначен для мониторинга выполнения команд ISQL на стороне сервера БД. Подлежащие трассировке опции определяются в свойстве:

```
property TraceFlags: TTraceFlags;
TTraceFlags = set of TTraceFlag;
```

Элементы, содержащиеся в перечне флагов, приведены в листинге 24.9.

Листинг 24.9. Элементы перечня флагов

```
TTraceFlag = (
  tfQPrepare, //идет подготовка запроса
  tfQExecute, //выполнение запроса
  tfQFetch,   //получение данных (инструкция SELECT)
  tfError,    //возникновение ошибки
  tfStmt,     //все операции с запросами
  tfConnect,  //соединение и отключение от БД
  tfTransact, //выполнение транзакций
  tfBlob,     //операции с данными BLOB
  tfService,  //выполнение служебных операций
  tfMisc      //все остальные операции
   );
```

После настройки свойства `TraceFlags` информация об интересующих нас показателях начинает поступать компоненту через свойство

```
property OnSQL: TSQLEvent;
TSQLEvent = procedure (EventText: String;
                       EventTime : TDateTime) of object;
```

Параметр `EventText` будет содержать текстовое описание, временные параметры окажутся в параметре `EventTime`.

Файл инициализации БД, *TIBDataBaseINI*

Компонент `TIBDataBaseINI` предоставляет разработчику клиентского приложения простой механизм сохранения/загрузки основных системных настроек базы данных `TIBDataBase`. Для этого компонент пользуется одним очень старым (применявшимся еще на заре Windows) способом сохранения системной информации — файлами инициализации. Начиная с Windows 95, на смену INI-файлам пришел системный реестр, но, как оказалось, INI-файлы еще готовы "тряхнуть стариной".

В файле инициализации сохраняется "золотой минимум" данных. В первую очередь, это путь к файлу БД и данные авторизации (рис. 24.5).

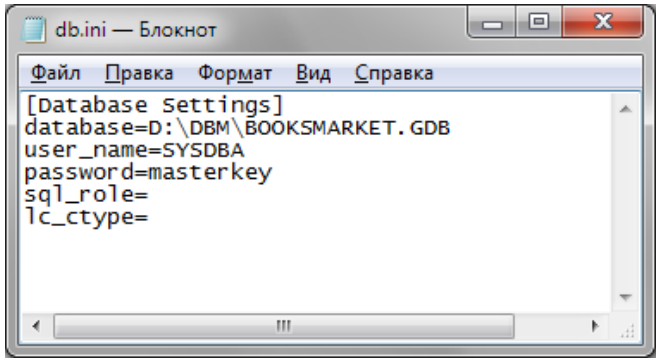


Рис. 24.5. Снимок блокнота с открытым файлом инициализации

Работать с компонентом очень просто. Для присоединения к базе данных применяется свойство

```
property Database : TIBDatabase;
```

Имя файла инициализации и путь к нему передается в свойство

```
property FileName : String;
```

Если в свойстве `FileName` путь отсутствует, то порядок его назначения зависит от состояния свойства

```
property UseAppPath : TIniFilePathOpt;
TIniFilePathOpt = (ipoPathNone,     //путь задается в FileName
                   ipoPathToServer, //системная папка Windows
                   ipoPathRelative); //в папке с исполняемым модулем
```

В любом случае функция

```
function IniFileName: string;
```

проинформирует нас о текущих настройках местоположения файла.

Во время запуска клиентского приложения загружаем данные из файла инициализации с помощью процедуры

```
procedure ReadFromINI;
```

Считав содержимое файла инициализации, передаем сохраненные в нем настройки в соответствующие свойства компонента `TIBDatabase`. с помощью метода

```
procedure WriteToDatabase(ADatabase : TIBDatabase);
```

Пример работы с методом приведен в листинге 24.10.

Листинг 24.10. Чтение файла инициализации

```
IBDatabaseINI1.ReadFromIni;
IBDatabaseINI1.WriteToDatabase(IBDatabase1);
```

По завершении работы клиентского приложения получаем информацию о текущих на данный момент настройках компонента `TIBDatabase` с помощью процедуры

```
procedure ReadFromDatabase;
```

Сохранение данных в файл производит процедура

```
procedure SaveToINI;
```

Порядок сохранения данных в файл инициализации иллюстрирует листинг 24.11.

Листинг 24.11. Запись в файл инициализации

```
IBDatabaseINI1.ReadFromDatabase;
IBDatabaseINI1.SaveToINI;
```

В табл. 24.8 приведен перечень базовых свойств компонента, благодаря которым можно читать и редактировать параметры соединения с базой данных.

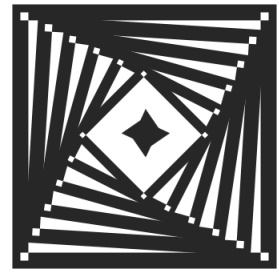
Таблица 24.8. *Свойства компонента* `TIBDataBaseINI`

Свойство	Описание
`property DatabaseName : String;`	Путь к файлу с БД
`property Username : String;`	Имя пользователя
`property Password : String;`	Пароль
`property SQLRole : String;`	Назначение сервера
`property CharacterSet : String;`	Кодовая страница
`property Section : String;`	Название секции в файле инициализации, в которую будут сохранены данные. По умолчанию "Database Settings"

Резюме

Компоненты IBX специализируются исключительно на работе с системой управления базами данных InterBase и ее клоном FireBird. В Delphi 2010 палитра компонентов, нацеленных на обслуживание InterBase, включает два десятка наименований. Эта весьма неприхотливая к ресурсам компьютера и эффективная СУБД соответствует стандарту SQL-92.

Глава 25



Наборы данных InterBase

Основная задача любого клиентского приложения баз данных — обеспечить пользователю удобный доступ к данным для их просмотра и редактирования. Далее мы познакомимся с компонентами InterBase, отвечающими за решение этой задачи. Среди них классические наборы данных — компоненты, берущие свое начало от абстрактного класса `TDataSet` (рис. 25.1). Кроме того, мы рассмотрим и ряд вспомогательных компонентов, помогающих наборам данных решать поставленные перед ними задачи.

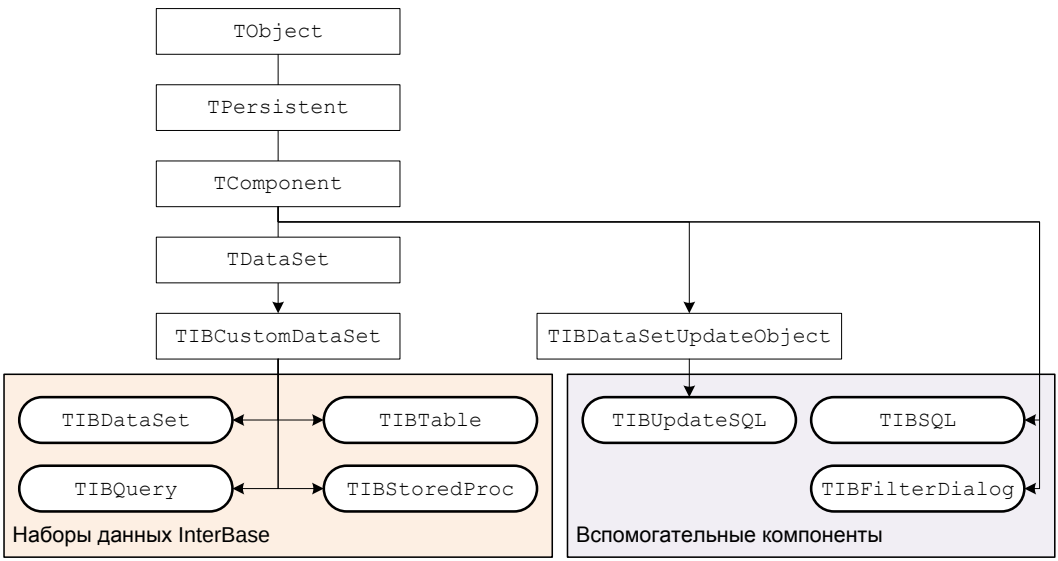


Рис. 25.1. Наборы данных и компоненты SQL для работы с InterBase

Большинство из компонентов на рис. 25.1 обладают рядом идентичных свойств. В первую очередь это свойства, предназначенные для обеспечения соединения с базой данных `DataBase` и транзакцией `Transaction`. Исключение из правил составляет модифицируемый запрос `TIBUpdateSQL` — вспомогательный компонент

(подключаемый к TIBQuery), обеспечивающий редактирование данных, полученных в результате выполнения SQL-запроса.

ЗАМЕЧАНИЕ

Все компоненты-наборы данных InterBase построены на основе класса TIBCustomDataSet. Предшественник всех наборов данных InterBase описан в модуле IBCustomDataSet. В нем сконцентрированы основные инженерные решения по обслуживанию баз данных InterBase из клиентского приложения.

Инструкция SQL, компонент *TIBSQL*

Основная отличительная черта компонента TIBSQL от своих собратьев по цеху InterBase — отсутствие в списке его родственников какого-либо намека на обязательный для всех компонентов, обслуживающих наборы данных, класс TDataSet. Непосредственным предком нашего нового знакомого выступает класс TComponent. Поэтому TIBSQL характеризуется минимальным спектром возможностей по обслуживанию данных, но это как раз тот случай, когда недостаток превращается в достоинство. Преимущество TIBSQL над другими элементами управления, специализирующимися на работе с данными, в том, что компонент ни в коем случае нельзя причислить к "пожирателям" системных ресурсов, в рассматриваемом плане он настоящий аскет.

Несмотря на отсутствие родственных связей между TIBSQL и универсальным набором данных TDataSet (от которого берут начало все известные нам компоненты-наборы данных), при встрече с нашим новым приятелем мы увидим знакомые имена свойств и методов. Дело в том, что, описывая класс TIBSQL, разработчики Delphi постарались сохранить привычные для нашего слуха названия.

Подготовка к работе

Работа с компонентом начинается с подключением запроса к базе данных. Для этого предназначено свойство

```
property Database: TIBDatabase;
```

Заручившись поддержкой БД (в лице компонента TIBDatabase), находим еще одного помощника — компонент, отвечающий за обслуживание транзакций:

```
property Transaction: TIBTransaction;
```

Собрав всех друзей компонента, переходим к подготовке инструкции ISQL. Текст запроса заносится в набор строк:

```
property SQL: TStrings;
```

В момент изменения запроса генерируется событие

```
property OnSQLChanging: TNotifyEvent;
```

Процедура

```
procedure Prepare;
```

подготавливает метод к выполнению.

После того как запрос будет подготовлен, свойство

```
property Prepared: Boolean; //только для чтения
```

перейдет в состояние `true`.

Если запрос содержит параметры, то доступ к ним осуществляется посредством свойства

```
property Params: TIBXSQLDA;
```

содержащего массив параметров `TIBXSQLDA`, или благодаря функции

```
function ParamByName(Idx: String): TIBXSQLVAR;
```

Единственный параметр метода — имя параметра.

Для передачи команды в адрес сервера находим процедуру

```
procedure ExecQuery;
```

Пример иллюстрирует листинг 25.1.

Листинг 25.1. Создание таблицы с помощью компонента `TIBSQL`

```
with IBSQL1 do
begin
  sql.Clear;
  sql.Add('CREATE TABLE "GENRES"');
  sql.Add('("GENRE_KEY" INTEGER NOT NULL PRIMARY KEY,');
  sql.Add('"GENRE" VARCHAR(30) NOT NULL UNIQUE)');
  if  Transaction.Active=false then Transaction.Active:=True;
  Prepare;
  ExecQuery;
end;
```

Для того чтобы определить, какой тип запроса был отправлен на выполнение, взгляните на свойство

```
property SQLType: TIBSQLTypes; //только для чтения
```

После выполнения предыдущего примера свойство установится в состояние `SQLDDL`. Полное описание типов запросов `TIBSQLTypes` приведено в табл. 25.1.

Таблица 25.1. *Тип выполняемого запроса* `TIBSQLTypes`

Константа	Описание
`SQLCommit`	Сохраняет результаты активной транзакции
`SQLDDL`	Запускает на выполнение инструкцию определения данных DDL (Data Definition Language)
`SQLDelete`	Удаляет записи из таблицы или из активного набора курсора
`SQLExecProcedure`	Вызывает хранимую процедуру
`SQLGetSegment`	Читает сегмент из открытого объекта BLOB

Таблица 25.1. *(окончание)*

Константа	Описание
SQLInsert	Добавляет одну или несколько новых записей в таблицу
SQLPutSegment	Записывает BLOB-сегмент
SQLRollback	Производит откат транзакции
SQLSetForUpdate	Хранимая процедура установлена для обновления
SQLSetGenerator	Устанавливает новое значение в имеющийся генератор
SQLSelect	Запрашивает данные из БД
SQLStartTransaction	Запускает новую транзакцию
SQLUnknown	Неизвестный тип SQL
SQLUpdate	Модифицирует данные в строках таблицы, в представлении или в активном наборе курсора

Обслуживание полученного набора данных

Если запрос возвращает результирующий набор данных, то для перебора всех записей в наборе пригодится приведенная далее группа свойств и методов.

Свойства

```
property Bof: Boolean; //только для чтения
property Eof: Boolean; //только для чтения
```

сигнализируют о нахождении курсора в начале (`Bof=true`) или в конце набора (`Eof=true`).

Все сведения о текущей записи возвращает метод

```
function Current: TIBXSQLDA;
```

Результат выполнения функции представляется в виде объекта `TIBXSQLDA`.

Переход к следующей записи осуществляет метод

```
function Next: TIBXSQLDA;
```

Эта функция возвращает следующую строку из полученного набора данных.

ВНИМАНИЕ!

Компонент `TIBSQL` способен просматривать записи лишь в одном направлении — от начала набора данных к его концу.

Для возврата к самой первой записи в наборе установите в `true` свойство

```
property GoToFirstRecordOnExecute: Boolean;
```

Для организации доступа к полям текущей записи необходимо воспользоваться услугами следующего "трио":

```
property Fields[const Idx: Integer]: TIBXSQLVAR;//массив полей
property FieldIndex: [FieldName: String]: Integer;//индекс поля в массиве
function FieldByName[FieldName: String]: TIBXSQLVAR;//отдельное поле
```

ВНИМАНИЕ!

Основной недостаток TIBSQL в том, что к полученному в результате выполнения инструкции SELECT набору данных нельзя обеспечить доступ с помощью источника данных — компонента TDataSource.

Наборы данных InterBase, компонент *TIBDataSet*

Открыв страницу **InterBase** палитры компонентов Delphi, мы обнаружим три элемента управления, предназначенные для обслуживания данных в классическом для Delphi стиле, основанном на идеях, заложенных в универсальном наборе данных TDataSet. Это запрос TIBQuery, хранимая процедура TIBStoredProc и таблица TIBTable. Все перечисленные компоненты построены на основе общего предка — класса TIBCustomDataSet. Но запрос, хранимая процедура и таблица — лишь дань традициям, появившимся в первых версиях Delphi. Ключевым набором данных InterBase считается элемент управления TIBDataSet — "мастер на все руки", обладающий полным спектром возможностей своих коллег.

Подготовка к работе

Жизнедеятельность набора данных начинается с момента подключения к компоненту TIBDatabase с помощью свойства

```
property Database: TIBDatabase;
```

Второй взаимодействующий элемент управления — TIBTransaction. Связь с ним определяется в свойстве

```
property Transaction: TIBTransaction;
```

Задача элемента TIBTransaction — организовать обмен между клиентским приложением и БД в границах отдельной транзакции.

Если внимательно изучить исходный код класса TIBCustomDataSet, то выяснится его главный секрет — класс инкапсулирует пять экземпляров класса TIBSQL (листинг 25.2).

Листинг 25.2. Фрагмент исходного кода класса TIBCustomDataSet

```
type
    TIBCustomDataSet = class(TDataset)
private
//...
FQDelete, FQInsert, FQRefresh, FQSelect, FQModify : TIBSQL;
```

Каждый из экземпляров TIBSQL отвечает за решение специфичных вопросов: удаление, вставка, обновление, выбор и модификация данных.

Доступ к инкапсулированным объектам обеспечивают пять свойств:

```
property QDelete: TIBSQL;
property QInsert: TIBSQL;
property QRefresh: TIBSQL;
property QSelect: TIBSQL;
property QModify: TIBSQL;
```

Соответствующие инструкции SQL передаются в следующие свойства:

```
property DeleteSQL: TStrings;  //удаление
property InsertSQL: TStrings;  //вставка
property RefreshSQL: TStrings; //обновление
property SelectSQL: TStrings;  //выбор
property ModifySQL: TStrings;  //модификация
```

Если запрос на выборку данных SelectSQL содержит параметры, то доступ к ним производится из свойства

```
property Params: TIBXSQLDA;
```

Для подготовки всех инструкций вызовите метод

```
procedure Prepare;
```

Противоположная цель достигается посредством обращения к методу

```
procedure UnPrepare;
```

В листинге 25.3 приведен пример инициализации элемента управления TIBDataSet для работы с таблицей авторов Writers.

Листинг 25.3. Пример работы с компонентом TIBDataSet

```
with IBDataSet1 do
begin
 Database:=IBDatabase1;
 Transaction:=IBTransaction1;
 ForcedRefresh:=true; //автоматически обновлять набор данных
 SelectSQL.Text:='SELECT * FROM WRITERS ORDER BY WRITER';
 RefreshSQL.Text:=SelectSQL.Text;
 InsertSQL.Text:='INSERT INTO WRITERS (WRITER)
                  VALUES (:WRITER)';
 ModifySQL.Text:='UPDATE WRITERS SET WRITER=:WRITER
                 WHERE WRITER_KEY=:OLD_WRITER_KEY ';
 DeleteSQL.Text:='DELETE FROM WRITERS WHERE WRITER_KEY=:OLD_WRITER_KEY';
 Prepare;
 Open;
end;
```

Обратите внимание на особенность определения имен параметров при описании инструкций ISQL:

- В листинге 25.3 параметр, следующий за предложением `WHERE`, называется ":OLD_"+ "Имя_поля". В `:OLD_` хранятся данные поля до модификации записи.
- Помимо префикса `:OLD_` компонент реагирует на префикс `:NEW_`. В параметрах `:NEW_` находятся новые значения, которые пользователь ввел во время редактирования, но пока еще не сохранил в БД.
- Имена всех остальных параметров повторяют имена обслуживаемых полей.

Свойство

```
property BufferChunks: Integer;//по умолчанию 1000
```

позволяет настроить размер буфера компонента, чтобы ускорить обслуживание записей.

По умолчанию буфер готов хранить 1000 записей. Отметим, что названное свойство не имеет прямого отношения к работе набора данных в режиме кэширования обновлений (это тот случай, когда все пользовательские правки записей накапливаются в памяти компонента и отправляются на сервер единым пакетом).

К подготовке наборов данных InterBase к работе убедитесь, что свойство

```
property UniDirectional: Boolean; //по умолчанию false
```

принимает значение `false`, т. е. компонент нацелен на работу с полноценными двунаправленными курсорами. Для сервера InterBase (и его клонов) это нормальный режим работы. Переключение в режим однонаправленного курсора крайне нежелательно, т. к. компоненты утратят возможность кэшировать записи.

Обработка событий

Помимо классического (унаследованного от `TDataSet`) набора событий потомки класса `TIBCustomDataSet` владеют специфичным для них перечнем событий (табл. 25.2).

***Таблица 25.2.** События класса `TIBCustomDataSet`*

Событие	Описание
`property BeforeTransactionEnd: TNotifyEvent;`	Вызывается перед окончанием транзакции
`property AfterTransactionEnd: TNotifyEvent;`	Вызывается по завершению транзакции
`property BeforeDatabaseDisconnect: TNotifyEvent;`	Генерируется перед разрывом соединения со связанной базой данных
`property AfterDatabaseDisconnect: TNotifyEvent;`	Генерируется после разрыва соединения со связанной базой данных
`property TransactionFree: TNotifyEvent;`	Освобождение связанного с набором данных экземпляра транзакции (ссылка на объект находится в свойстве `Transaction`)

Таблица 25.2 *(окончание)*

Событие	Описание
property DatabaseFree: TNotifyEvent;	Освобождение ассоциированного с набором экземпляра базы данных (свойство Database)
property OnUpdateRecord: TIBUpdateRecordEvent;	Сохранение записи
property OnUpdateError: TIBUpdateErrorEvent;	Ошибка при сохранении записи

ЗАМЕЧАНИЕ

События OnUpdateRecord() и OnUpdateError() более подробно рассмотрены в подразделе "Работа в режиме кэширования обновлений" этой главы.

Генератор значений

Одна из особенностей InterBase — наличие в СУБД специальных объектов (называемых генераторами), предназначенных для генерации числовых значений для их последующей вставки в поля таблиц, например, в поля первичных ключей.

Генераторы обслуживаются инструкциями ISQL: CREATE GENERATOR, SET GENERATOR и встроенной функцией GEN_ID(). Объекты-генераторы и соответствующие им инструкции не описаны в стандарте SQL и являются отличительным признаком InterBase.

Например, для автоматического присвоения очередного значения для поля автоинкрементного типа на ISQL пришлось бы, кроме формирования таблицы, создавать генератор и триггер (листинг 25.4).

Листинг 25.4. Создание таблицы средствами ISQL с использованием генератора

```
{создаваем таблицу}
CREATE TABLE WRITERS (
       WRITER_KEY INTEGER NOT NULL,
       WRITER Varchar(100) NOT NULL,
       CONSTRAINT PK_WRITERS PRIMARY KEY (WRITER_KEY));
   {создаем генератор}
CREATE GENERATOR GEN_WRITERS_ID;
SET GENERATOR GEN_WRITERS_ID TO 0
   {создаем триггер}
CREATE TRIGGER TR_GEN_WRITERS_ID FOR WRITERS
ACTIVE BEFORE INSERT POSITION 0
AS
```

```
BEGIN
  IF (NEW.WRITER_KEY IS NULL) THEN
     NEW.WRITER_KEY=GEN_ID(GEN_WRITERS_ID,1);
END
```

Триггер вызывается перед началом операции добавления новой записи в таблицу `WRITERS`. Его единственная задача — вставка в поле первичного ключа `WRITER_KEY` очередного значения, сгенерированного генератором `GEN_WRITERS_ID`.

При сохранении данных наследники `TIBCustomDataSet` также способны автоматически присваивать значения для одного или нескольких полей обслуживаемого набора данных. Для этого предназначено свойство

```
property GeneratorField: TIBGeneratorField;
```

Обращение к свойству во время разработки проекта приводит к вызову диалога настройки поля. Здесь программист должен выбрать (рис. 25.2):

- имя генератора (**Generator**);
- имя обслуживаемого поля таблицы (**Filed**);
- значение приращения (**Increment By**);
- событие, по которому новое значение будет передаваться в поле. Предусмотрены три варианта: в момент вставки новой записи (**On New Record**), в момент сохранения (**On Post**) и поведение определяется сервером (**On Server**).

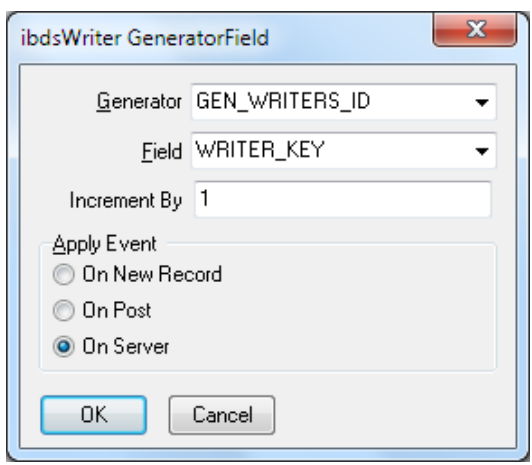


Рис. 25.2. Настройка свойства `GeneratorField`

Внимание!

Для обеспечения целостности данных и поддержки бизнес-правил БД нужно добиваться того, чтобы вся логика управления данными была сосредоточена на стороне сервера БД.

Особенности редактирования данных

Признаком того, что компонент содержит доступные для редактирования данные, будет значение `true`, возвращаемое свойством

```
property CanModify: Boolean; //только для чтения
```

Более полную информацию предоставляет свойство

```
property LiveMode: TLiveModes; //только для чтения
```

Это множество, способное хранить до четырех флагов:

```
TLiveModes = set of TLiveMode;

type TLiveMode = (lmInsert, {разрешены операции вставки}
                  lmModify, {разрешены операции модификации данных}
                  lmDelete, {разрешено удаление}
                  lmRefresh);{разрешено обновление}
```

Если свойство пусто, значит, набор данных доступен только для просмотра, подключение остальных флагов расширит возможности компонента.

Для того чтобы всякий раз после сохранения изменений в данных компонент самостоятельно обновлял набор, установите в `true` свойство

```
property ForcedRefresh: Boolean;//по умолчанию false
```

Работа в режиме кэширования обновлений

Для включения механизма кэширования установите в состояние `true` свойство

```
property CachedUpdates: Boolean; //по умолчанию false
```

С этого момента все изменения, внесенные пользователем, вместо отправки на сохранение в БД, начнут накапливаться в локальном буфере набора данных.

Если набор данных работает в режиме кэширования обновлений, то благодаря свойству

```
property UpdateRecordTypes: TIBUpdateRecordTypes;

type TIBUpdateRecordTypes = set of TCachedUpdateStatus;
```

мы получаем возможность определять, какие из находящихся в буфере записей будут видимы в наборе данных. Это своего рода фильтр, имеющий пять режимов отбора данных (табл. 25.3):

```
TCachedUpdateStatus = (cusUnmodified,//Неизменившиеся записи
                       cusModified,  //Модифицированные записи
                       cusInserted,  //Добавленные записи
                       cusDeleted,   //Удаленные записи
                       cusUninserted //Недобавленные записи);
```

По умолчанию в состав множества `UpdateRecordTypes` включены флаги `cusModified`, `cusInserted` и `cusUnmodified`. Манипулируя элементами множества, мы пополним наш проект дополнительными сервисными возможностями, например, позволим пользователю восстановить все удаленные записи из кэша (листинг 25.5).

Листинг 25.5. Восстановление записей, удаленных из кэша компонента

```
procedure RestoreRecords;
var Old_URT : TIBUpdateRecordTypes;
begin
  with IBDataSet1 do
  begin
  Old_URT:=UpdateRecordTypes; //запоминаем состояние набора данных
  {выбираем только записи, помеченные на удаление}
  UpdateRecordTypes := [cusDeleted];
    First; //переход к первой удаленной записи
    while EOF=False do //цикл просмотра всех удаленных записей
      begin
      Undelete; //восстановление записи
      Next;     //переход к очередной записи
      end;
      UpdateRecordTypes := Old_URT;//возвращаемся к исходному состоянию
  end;
end;
```

В примере мы воспользовались новым методом:

```
procedure Undelete;
```

Эта процедура восстанавливает запись, помеченную на удаление.

Если в буфере компонента содержатся модифицированные записи, ожидающие своего сохранения в БД, то свойство

```
property UpdatesPending: Boolean;
```

примет значение `true`.

В этом случае для сохранения изменений в БД обратитесь к методу

```
procedure ApplyUpdates;
```

В момент сохранения изменений в данных возникает событие

```
property OnUpdateRecord: TIBUpdateRecordEvent;
TIBUpdateRecordEvent = procedure(DataSet: TDataSet;
                                 UpdateKind: TUpdateKind;
                                 var UpdateAction: TIBUpdateAction) of object;
```

которое генерируется для каждой записи из кэша. Параметр `UpdateKind` указывает, какая именно операция (модификация `ukModify`, вставка `ukInsert` или удаление `ukDelete`) осуществляется. Переменная `UpdateAction` позволит программисту управлять развитием события (табл. 25.3).

Таблица 25.3. *Возможные значения* `TIBUpdateAction`

Значение	Описание
`uaAbort`	Прекращение обновления без генерации сообщения об ошибке
`uaApply`	Для внутреннего использования
`uaApplied`	Сохранить изменения и очистить кэш
`uaFail`	Прервать обновление и вызвать сообщение об ошибке
`uaRetry`	Повторить операцию обновления
`uaSkip`	Пропустить текущую запись с ее сохранением в кэше

Если во время сохранения данных из кэша в БД возникнет ошибка, то генерируется событие

```
property OnUpdateError: TIBUpdateErrorEvent;
TIBUpdateErrorEvent = procedure(DataSet: TDataSet;
                                E: EDatabaseError;
                                UpdateKind: TUpdateKind;
                                var UpdateAction: TIBUpdateAction)of object;
```

Практически все параметры события повторяют параметры события `OnUpdateRecord()`, за исключением параметра `E`, это объект исключительной ситуации.

Для очистки кэша без отправки пакета изменений в БД вызовите процедуру

```
procedure CancelUpdates;
```

Если же требуется вернуть в исходное состояние лишь отдельную запись в наборе, то лучше воспользоваться методом

```
procedure RevertRecord;
```

Перемещение по записям

Наборы данных InterBase владеют всеми классическими методами перемещения по записям (методы `First`, `Prior`, `Next` и `Last`). Кроме того, если набор данных поддерживает порядковые номера записей, а об этом свидетельствует значение `true` возвращаемое функцией

```
function IsSequenced: Boolean;
```

то по набору разрешено двигаться, передавая в свойство

```
property RecNo: Integer;
```

порядковый номер требуемой строки.

Фильтрация данных

Изучая универсальный набор данных `TDataSet`, мы познакомились с ключевыми свойствами и методами компонентов-наборов данных, предназначенными для

фильтрации данных. В первую очередь, это свойство `Filter`, определяющее условия фильтрации (см. табл. 18.2). Опции фильтра задаются в свойстве `FilterOptions`. Для включения/отключения фильтра обратитесь к свойству `Filtered`.

ЗАМЕЧАНИЕ

Если вы хотите реализовать в своем проекте гибкую систему настройки параметров фильтра, то обязательно обратите внимание на компонент `TIBFilterDialog`.

Запрос, компонент *TIBQuery*

Запрос `TIBQuery` построен на базе рассмотренного ранее `TIBCustomDataSet` и описан в модуле `IBQuery`. Основное назначение компонента — организация доступа к данным одной или нескольких таблиц. Как и у всех подобных элементов управления, текст запроса описывается в свойстве

```
property SQL: TStrings;
```

Текст запроса можно задать как во время проектирования, так и в коде программы. Если запрос основан на инструкции `SELECT`, то для получения записей следует воспользоваться методом

```
procedure Open;
```

Во всех остальных случаях выполнение инструкции ISQL доверяется процедуре

```
procedure ExecSQL;
```

Листинг 25.6 подготавливает компонент к получению данных из таблиц поставщиков и заказчиков.

Листинг 25.6. Пример работы с компонентом `TIBQuery`

```
With IBQuery1 do
begin
  Database:=IBDatabase1;
  SQL.Clear;
  SQL.Add('SELECT SUPPLIER, CONTRACT, CONTRDATE FROM SUPPLIERS');
  SQL.Add('LEFT JOIN CONTRACTS ON');
  SQL.Add('CONTRACTS.SUPPLIER_KEY=SUPPLIERS.SUPPLIER_KEY');
  SQL.Add('ORDER BY CONTRDATE');
  Prepared:=True;
  Open;
end;
```

Возвращаемый набор данных доступен только в режиме "для чтения". Чтобы добиться возможности его редактирования, нам понадобятся услуги `TIBUpdateSQL`, подключаемого к свойству

```
property UpdateObject: TIBDataSetUpdateObject;
```

Если запрос содержит параметры, то для их обслуживания следует воспользоваться коллекцией

```
property Params: TParams;
```

Для того чтобы компонент автоматически регенерировал список параметров запроса, следует установить в `true` свойство

```
property GenerateParamNames: Boolean;
```

Запрос InterBase обладает дескриптором, который может пригодиться при обращении к низкоуровневым функциям API сервера:

```
property StmtHandle: TISC_STMT_HANDLE;
```

Редактирование данных, доступных только для чтения

Зачастую набор данных, возвращаемый потомками `TIBCustomDataSet`, доступен только для просмотра. Если логика работы приложения требует обеспечить возможность модификации таких данных, то на помощь приходит свойство

```
property UpdateObject: TIBDataSetUpdateObject;
```

Это ссылка на компонент `TIBUpdateSQL`, способный оказать помощь при работе с недоступными для редактирования данными.

Хранимая процедура, компонент *TIBStoredProc*

Хранимая процедура `TIBStoredProc` — прямой наследник класса `TIBCustomDataSet`, компонент реализован в модуле IBStoredProc. Элемент управления предназначен для обращения к процедурам, откомпилированным и хранящимся в системном каталоге сервера БД.

Название процедуры, услугами которой мы намерены воспользоваться, указывают в свойстве

```
property StoredProcName: string;
```

Параметры процедуры содержатся в коллекции

```
property Params: TParams;
```

О готовности процедуры к выполнению сигнализирует свойство

```
property Prepared: Boolean;
```

Для отправки команды на выполнение обращаемся к процедуре

```
procedure ExecProc;
```

Внимание!

К сожалению, хранимые процедуры InterBase не любят работать с инструкциями `SELECT`, возвращающими набор из нескольких записей. Поэтому, несмотря на наличие у `TIBStoredProc` метода `Open()`, основной способ отправки команды на выполнение должен быть `ExecSQL()`.

Допустим, что в составе БД книжного магазина имеется хранимая процедура PR_WRITERS_INSERT, вставляющая новую запись в таблицу авторов. SQL-текст процедуры приведен в листинге 25.7.

Листинг 25.7. Код хранимой процедуры ISQL

```
CREATE PROCEDURE PR_WRITERS_INSERT
(IN_WRITER VARCHAR(40))
AS
BEGIN
  INSERT INTO WRITERS
    (WRITER)
  VALUES
    (:IN_WRITER);
END
```

Процедура обладает единственным входным параметром IN_WRITER.

Для подготовки компонента TIBStoredProc к работе с процедурой PR_WRITERS_INSERT следует воспользоваться фрагментом кода из листинга 25.8.

Листинг 25.8. Подключение компонента TIBStoredProc к хранимой процедуре

```
With IBStoredProc1 do
begin
  Database:=IBDatabase1;
  StoredProcName:='PR_WRITERS_INSERT';
  if Prepared=false then Prepared:=True;
end;
```

Для выполнения процедуры PR_WRITERS_INSERT достаточно обратиться к листингу 25.9.

Листинг 25.9. Вызов хранимой процедуры

```
  with IBStoredProc1 do
  begin
  {если свойство AllowAutoStart транзакции установлено в false,
   то необходимо явно стартовать транзакцию Transaction.StartTransaction}
    TRY
    Params.ParamByName('IN_WRITER').AsString:=Edit1.Text;
    ExecProc;
    Transaction.Commit;   //применение транзакции
```

```
    EXCEPT
    Transaction.Rollback; //откат транзакции
    END;
  end;
```

В них мы передаем новое значение в параметр `IN_WRITER` и выполняем процедуру. Обратите внимание: здесь предполагается, что транзакция стартует автоматически (свойство `AllowAutoStart=true`), иначе потребуется запуск транзакции с помощью метода `StartTransaction()`.

Таблица, компонент *TIBTable*

Таблица InterBase — это прямой потомок класса `TIBCustomDataSet`, компонент описан в модуле IBTable. Компонент `TIBTable` предназначен для просмотра и редактирования отдельной таблицы базы данных. Для подключения к таблице необходимо указать ее название в свойстве `TableName`. Все ключевые свойства и методы класса подробно рассмотрены в *главе 18*.

При работе с компонентом, необходимо определиться, какие типы таблиц будут доступны в свойстве `TableName`. Для этого предназначено свойство:

```
property TableTypes: TIBTableTypes;
type TIBTableType = (ttSystem, ttView);
```

Если в состав элементов множества входит значение `ttSystem`, то нам будут доступны системные таблицы и представления, при наличии элемента `ttView` — только представления, определенные пользователем.

Компонент позволяет синхронизировать позицию своего курсора с курсором другой таблицы с помощью метода

```
procedure GotoCurrent(Table: TIBTable);
```

Для успешной синхронизации оба компонента-таблицы должны быть подключены к одной и той же физической таблице БД.

Перед открытием набора данных нужно убедиться в наличии таблицы с именем `TableName`, вызвав свойство

```
property Exists: Boolean;
```

Во всем остальном наш компонент схож с любой другой таблицей `T…Table`.

Экспорт-импорт данных

Для организации экспорта записей из наборов данных InterBase в файл и для обратной операции (импорта данных из файла) компоненты `TIBSQL`, `TIBDataSet`, `TIBQuery` вооружены парой методов:

```
procedure BatchInput(InputObject: TIBBatchInput);   //импорт данных
procedure BatchOutput(OutputObject: TIBBatchOutput);//экспорт данных
```

Единственный параметр процедур — ссылка на специализированный файловый объект — потомок класса TIBBatch. На рис. 25.3 представлено дерево наследования этого класса. Сразу за абстрактным классом TIBBatch иерархия расходится в двух направлениях: ветвь TIBBatchInput отвечает за классы, обеспечивающие импорт данных, ветвь TIBBatchOutput содержит классы поддержки экспорта.

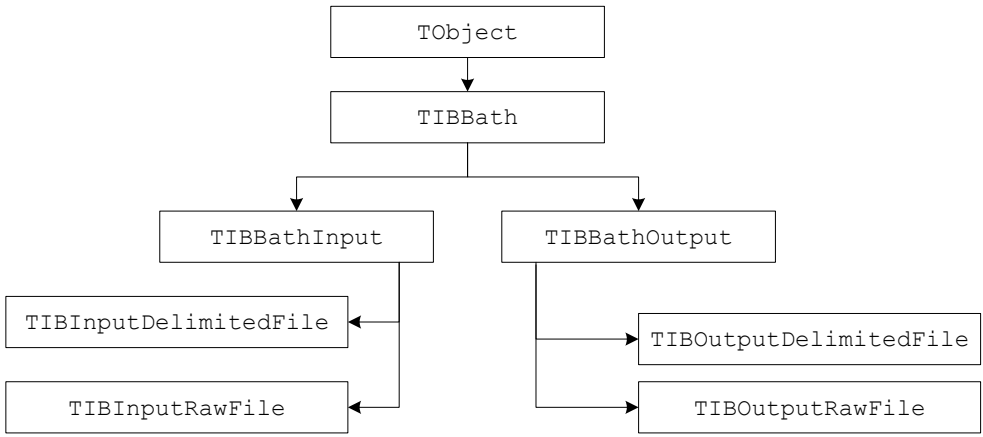


Рис. 25.3. Потомки класса TIBBath

Каждая ветвь способна работать с двумя форматами файлов:

- файл с разделителями (delimited file);
- двоичный файл (raw file).

В первом случае (классы TIBInputDelimitedFile и TIBOutputDelimitedFile) программист может определить символы-разделители строк и колонок в свойствах:

```
property RowDelimiter: String; //разделитель строк
property ColDelimiter: String; //разделитель рядов
```

Для файлов, инкапсулируемых классами TIBOutputRawFile и TIBInputRawFile, такая возможность отсутствует.

Основное свойство всех классов — имя файла:

```
property Filename: String;
```

Листинг 25.10 демонстрирует способ сброса данных из таблицы в размеченный файл.

Листинг 25.10. Экспорт данных из таблицы в файл

```
var OutFile : TIBOutputDelimitedFile;
begin
  frmDataModule.IBSQL1.SQL.Clear;
  frmDataModule.IBSQL1.SQL.Add('SELECT ORDERNUM, ORDERDATE FROM ORDERS');
  OutFile:= TIBOutputDelimitedFile.Create;
```

```
  try
    OutFile.FileName := 'c:\orders.txt';
    OutFile.RowDelimiter:=#13;//строки разделяем символом ввода
    OutFile.ColDelimiter:=#9; //колонки разделяем символом табуляции
    frmDataModule.IBSQL1.BatchOutput(OutFile);
  finally
    OutFile.Free;
  end;
end;
```

В результате мы сможем просмотреть данные из файла в обычном блокноте.

В листинге 25.11 приведен исходный код загрузки данных из двоичного (неразмеченного) файла.

Листинг 25.11. Импорт данных из файла в таблицу

```
var RawFile : TIBInputRawFile;
begin
frmDataModule.IBSQL2.SQL.Clear;
frmDataModule.IBSQL2.SQL.Add('INSERT INTO ORDERS (ORDERNUM, ORDERDATE)');
frmDataModule.IBSQL2.SQL.Add('VALUES(:ORDERNUM, :ORDERDATE)');
RawFile:= TIBInputRawFile.Create;
try
  RawFile.FileName := 'c:\orders.raw';
  frmDataModule.IBSQL2.BatchInput(RawFile);
finally
  RawFile.Free;
end;
end;
```

Модифицируемый запрос, компонент *TIBUpdateSQL*

Описанный в модуле `IBUpdateSQL` компонент `TIBUpdateSQL` приходит на помощь в тех случаях, когда возникает острая необходимость редактирования записей в наборах данных, напрямую не доступных для модификации. Например, такая ситуация характерна для компонента `TIBQuery`, возвращающего запросом `SELECT` строки таблиц в режиме только для чтения.

Можно сказать, что, подключив к `TIBQuery` нашего нового знакомого, мы "оживляем" полученные результаты. Другими словами, справедлива следующая формула:

`TIBQuery + TIBUpdateSQL = TIBDataSet`.

Замечание

Для подключения модифицируемого запроса `TIBUpdateSQL` к набору данных InterBase у потомков `TIBCustomDataSet` объявлено свойство `UpdateObject`.

"Оживление" происходит не за счет магии, а благодаря тому, что внутри компонента `TIBUpdateSQL` скрывается три экземпляра класса `TIBQuery`, каждый из которых отвечает за решение отдельной задачи: вставки, редактирования или удаления данных. Для того чтобы обратиться к требуемому инкапсулированному экземпляру `TIBQuery`, можно воспользоваться свойством

```
property Query[UpdateKind: TUpdateKind]: TIBQuery;
type TUpdateKind = (ukModify, ukInsert, ukDelete);
```

Для определения соответствующих инструкций ISQL обращаемся к свойству

```
property SQL[UpdateKind: TUpdateKind]: TStrings;
```

или к трем свойствам:

```
property InsertSQL: TStrings;
property ModifySQL: TStrings;
property DeleteSQL: TStrings;
```

Прямой доступ к инструкции ISQL, позволяющей обновить результирующий набор, обеспечивает свойство

```
property RefreshSQL: TIBSQL;
```

Для организации взаимодействия между компонентом-запросом `TIBQuery` и компонентом `TIBUpdateSQL` нужно проделать следующие шаги:

1. Выполнить запрос; `TIBQuery` соединяется с компонентом-базой данных с помощью свойства `Database`.
2. Ввести текст запроса.
3. С помощью свойства `UpdateObject` подключить `TIBQuery` к `TIBUpdateSQL`.

Изучив листинг 25.12, вы получите представление о том, как следует настроить компонент `TIBUpdateSQL` для работы с данными.

Листинг 25.12. Подготовка `TIBQuery` к совместной работе с `TIBUpdateSQL`

```
with IBQuery1 do
begin
 SQL.Clear;
 SQL.Add('SELECT WRITER_KEY, WRITER FROM WRITERS');
 UpdateObject:=IBUpdateSQL;
end;
with IBUpdateSQL do
begin
InsertSQL.Add('INSERT INTO WRITERS (WRITER) VALUES (:WRITER)');
ModifySQL.Add('UPDATE WITERS SET WRITER=:WRITER
               WHERE WRITER_KEY=:OLD_WRITER_KEY');
DeleteSQL.Add('DELETE FROM WRITERS WHERE (WRITER_KEY=:OLD_WRITER_KEY)');
end;
```

ВНИМАНИЕ!

При объявлении имен параметров программист обязан соблюдать следующие два правила: 1) в тексте команды `INSERT` имена параметров должны совпадать с физическими именами полей; 2) в командах `UPDATE` и `DELETE` (после ключевого слова `WHERE`) старые значения полей в тексте запроса указывают префиксом `:OLD_` (поскольку поиск записей для модификации или удаления проводится по старым значениям).

Перед запуском той или иной инструкции ISQL на выполнение необходимо установить параметры запроса с помощью метода

```
procedure SetParams(UpdateKind: TUpdateKind);
```

и только после этого воспользуемся методом

```
procedure ExecSQL(UpdateKind: TUpdateKind);
```

Чтобы сэкономить силы, вместо последовательного обращения к методам `SetParams()` и `ExecSQL()` вызываем процедуру

```
procedure Apply(UpdateKind: TUpdateKind);
```

Диалог фильтрации, компонент *TIBFilterDialog*

В последних версиях Delphi на страничке **InterBase** появился новый диалог, предназначенный для оказания помощи пользователю при определении параметров фильтрации данных, — компонент `TIBFilterDialog`. Класс объявлен в модуле `IBFilterDialog` и является потомком класса `TComponent`. На службе диалога состоит обычная форма `type TfrmIBFilterDialog = class(TForm)` с элементами управления, позволяющими пользователю своими руками назначить параметры фильтрации (рис. 25.4).

Компонент-диалог подключается к набору данных InterBase с помощью свойства

```
property DataSet : TIBCustomDataset;
```

и помогает пользователю настроить параметры фильтрации записей, находящихся в `DataSet`.

Традиционно для всех диалогов Delphi основной метод компонента называется

```
function Execute : Boolean;
```

Функция выводит на экран форму, в которой мы настраиваем параметры фильтрации.

Наших знаний уже вполне достаточно, чтобы задействовать всю мощь диалога. В листинге 25.13 фильтр присоединяется к компоненту `IBQuery1`, в котором подготовлен запрос, объединяющий данные из таблиц поставщиков и контрактов.

Листинг 25.13. Применение диалога `TIBFilterDialog`

```
With DataModule1.IBFilterDialog1 do
 begin
```

```
  DataSet:=DataModule1.IBQuery1;
  Execute;
end;
```

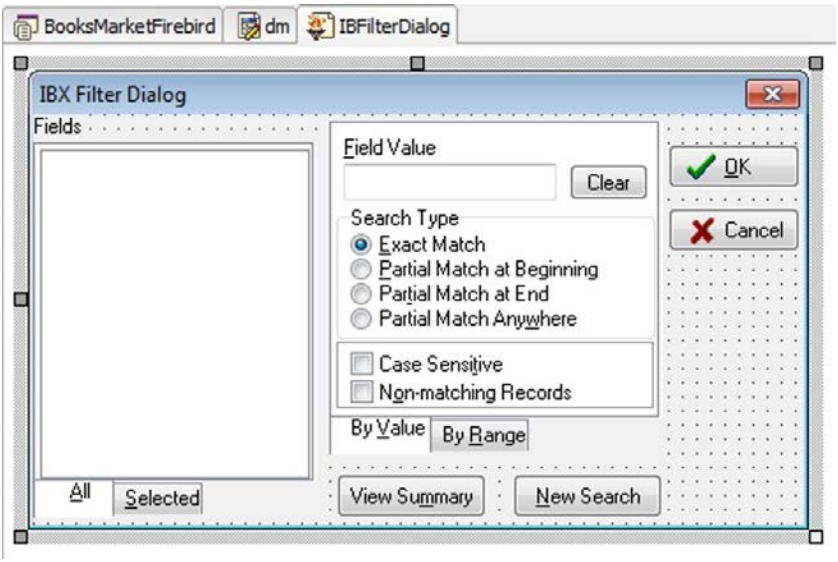


Рис. 25.4. Внешний вид формы диалога настройки фильтра

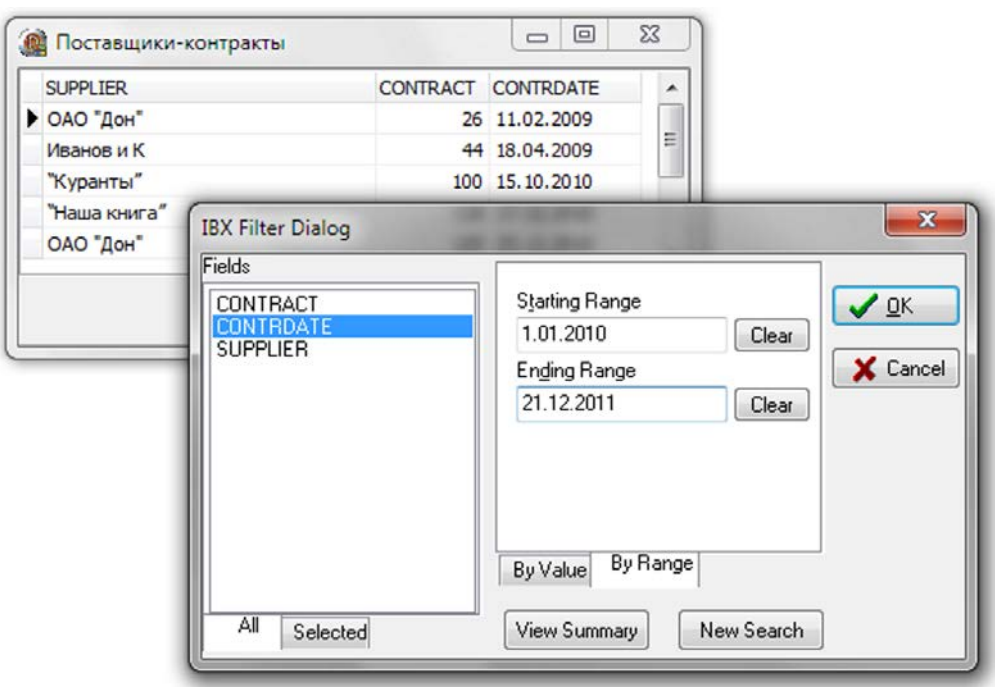


Рис. 25.5. Диалог фильтрации в работе

На экранном снимке (рис. 25.5) вы можете видеть диалог в работе. В данном случае мы ограничиваем диапазон дат заключения контрактов. Для этого в списке полей пользователь выбирает столбец CONTRDATE и назначает стартовую и конечную даты фильтра.

Перечень полей, которые разрешено использовать в фильтре, располагается в списке

```
property Fields : TStringList;
```

Свойство

```
property DefaultMatchType : TIBFilterMatchType
```

определяет выбираемый по умолчанию порядок совпадения записей отношения с образцом (листинг 25.14).

Листинг 25.14. Тип данных TIBFilterMatchType

```
TIBFilterMatchType = (fdMatchExact, //полное совпадение
                      fdMatchStart, //совпадение с начала
                      fdMatchEnd,   //совпадение в конце
                      fdMatchAny,   //совпадение в любом месте
                      fdMatchRange, //вхождение в диапазон
                      fdMatchNone); //нет значения по умолчанию
```

Опции диалога определяются свойством

```
property Options : TIBOptions;
```

Опций всего три (листинг 25.15).

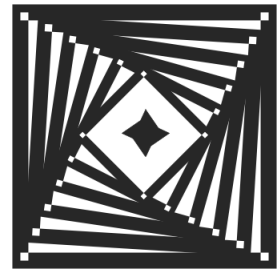
Листинг 25.15. Опции диалога

```
TIBOption = (fdCaseSensitive,     {чувствительность к регистру}
             fdShowCaseSensitive, {разрешить пользователю
                                   включать/отключать
                                   чувствительность к регистру символов}
             fdShowNonMatching  {показать не попавшие в фильтр строки});
```

Резюме

Компоненты-наборы данных InterBase продолжают традиции по управлению наборами данных, заложенные в абстрактном классе TDataSet. Для построения полноценного клиентского приложения разработчик может ограничиться применением компонентов TIBSQL и TIBDataSet. Для сторонников традиционного подхода к проектированию приложения баз данных Delphi пригодятся классические компоненты TIBTable, TIBQuery и TIBStoredProc.

Рассмотренные здесь компоненты можно применять не только в клиентских приложениях InterBase, но и в проектах БД, развернутых на основе бесплатных СУБД Firebird и Yaffil.



Глава 26

Администрирование сервера InterBase

Если вы имеете опыт конфигурирования БД с помощью стандартной консоли управления сервером **IBConsole**, входящей в инсталляционный пакет InterBase, то наверняка хорошо знаете функционал администратора:

- ведение учетных записей пользователей;
- проверка состояния БД и настройка ее параметров;
- резервное копирование и восстановление БД;
- управление лицензиями;
- ведение журнала;
- анализ статистики и производительности.

Доступ к большинству перечисленных операций предоставляет контекстное меню обслуживаемой базы данных (рис. 26.1).

Система управления базами данных InterBase обладает мощным набором системных процедур и функций API, предоставляющих программисту возможность низкоуровневого доступа к серверу. Опираясь на API, разработчик сможет осуществить более тонкую настройку всех параметров системы, начиная от повышения производительности, заканчивая управлением поведением сервера в критических ситуациях.

В составе палитры среды проектирования Delphi имеется несколько компонентов, инкапсулирующих базовый набор функций InterBase API, отвечающих за конфигурирование и администрирование сервера InterBase и развернутых на нем баз данных. Все компоненты размещены на страничке **InterBase Admin**, описаны в модуле IBServices и предназначены для обслуживания сервера InterBase не ниже версии 6.0.

Внимание!

Прикладной интерфейс программирования FireBird и Yaffil несколько отличается от функций API InterBase. Поэтому рассматриваемые в этой главе компоненты не всегда корректно работают с бесплатными клонами InterBase.

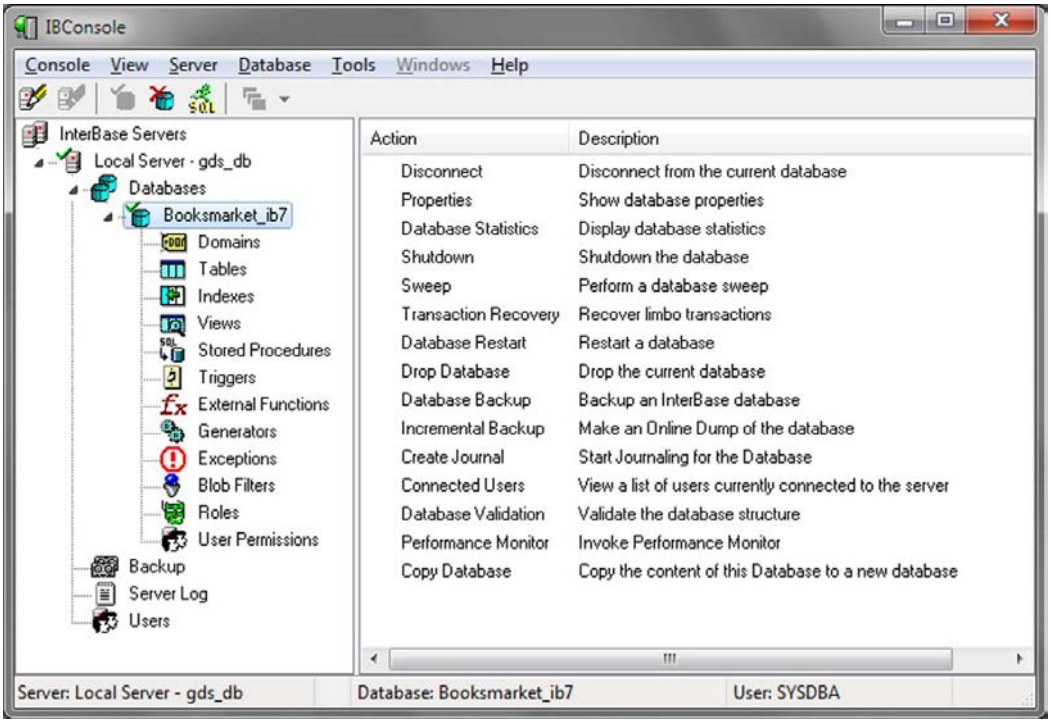


Рис. 26.1. Управление БД из консоли InterBase

В цепочке наследования (рис. 26.2) перед компонентами-сервисами имеется два промежуточных класса, определивших родовые черты всего большого семейства компонентов `TIB...Service`, — классы `TIBCustomService` и `TIBControlService`.

Первый по счету предок отвечает за организацию соединения с хостом, на котором развернут интересующий нас сервер InterBase. Имя сервера следует поместить в свойство

```
property ServerName: String;
```

Вместо символьного имени допускается и указание IP-адреса компьютера. Для доступа к локальному хосту можно воспользоваться IP-адресом 127.0.0.1 или текстовой константой `localhost`.

Следующее действие программиста — выбор сетевого протокола доступа к серверу:

```
property Protocol: TProtocol; \\ по умолчанию Local
TProtocol = (TCP, SPX, NamedPipe, Local);
```

Перечень доступных вариантов:

- транспортный протокол TCP;
- неактуальный на сегодняшний день SPX;
- именованный канал NamedPipe (это надстройка над стеком протоколов TCP/IP);
- значение `Local`, указывающее на то, что сервер и клиент размещены на одном хосте.

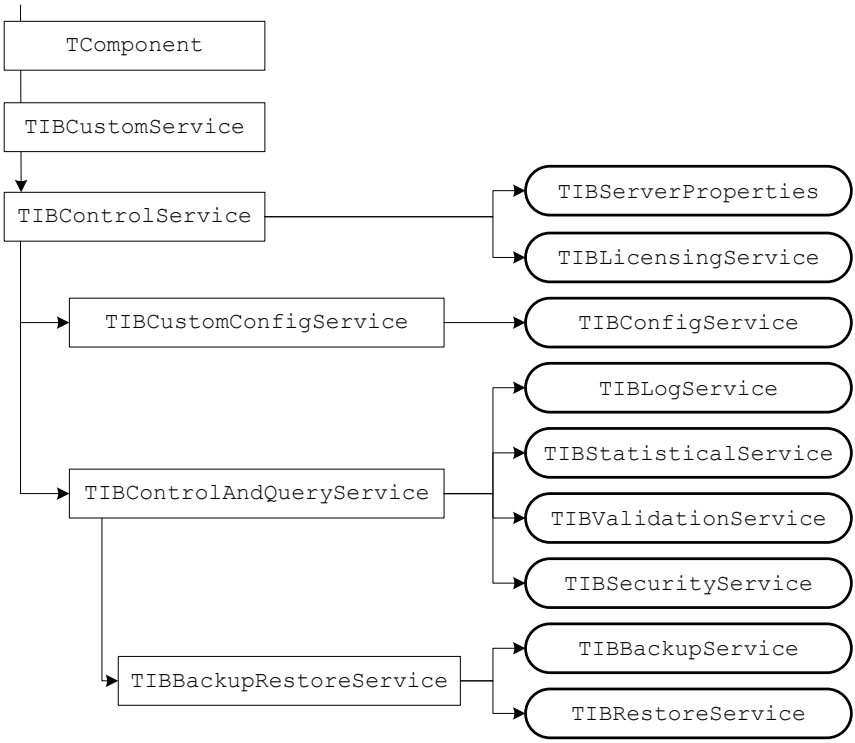


Рис. 26.2. Иерархия компонентов-сервисов InterBase

Порядок регистрации пользователя на сервере зависит от состояния свойства

```
property LoginPrompt: Boolean; \\ по умолчанию True
```

По умолчанию оно установлено в `true` — для аутентификации пользователя должен вызываться специальный диалог. Процесс аутентификации сопровождается событием

```
property OnLogin: TIBLoginEvent;
TIBLoginEvent = procedure(Database: TIBCustomService;
                          LoginParams: TStrings) of object;
```

Параметры компонента, в частности, обеспечивающие выполнение процедуры аутентификации пользователя, могут быть занесены в список

```
property Params: TStrings;
```

Подключение к серверу и отключение компонента от сервера производится с помощью свойства

```
property Active: Boolean; //по умолчанию False
```

или пары методов

```
procedure Attach;//подключение
procedure Detach;//отключение
```

В момент подключения генерируется событие

```
property OnAttach: TNotifyEvent;
```

Наличие общих черт у всех сервисных компонентов существенно упрощает управление ими. В частности, если в проекте задействовано несколько потомков класса `TIBControlService`, то всех их можно подготовить к работе в коде единой процедуры (листинг 26.1).

Листинг 26.1. Подготовка к работе административных компонентов

```
procedure TDataModule1.PrepareIBService(const Host,
                                                User, Password: string);
var i:integer;
begin
 for i := 0 to TDataModule1.ComponentCount-1 do
   if (TDataModule1.Components[i] is TIBControlService) then
   with (TDataModule1.Components[i] as TIBControlService) do
   begin
      if Active=true then Active:=False;
       LoginPrompt:=false;
       ServerName:=Host;
       Protocol:=TCP;
       Params.Clear;
       Params.Add('user_name='+User);
       Params.Add('password='+Password);
   end;
end;
```

На вход функции поступают имя хоста, имя и пароль пользователя. В цикле `for..do` мы перебираем все компоненты, принадлежащие модулю данных `DataModule1`, и если среди них нам встречаются потомки `TIBControlService`, то заполняем их ключевые свойства, отвечающие за организацию соединения.

Наиболее значимый вклад класса `TIBControlService` в своих потомков — это метод

```
procedure ServiceStart; virtual;
```

и свойство

```
property IsServiceRunning : Boolean; //только для чтения
```

Метод `ServiceStart` позволяет запустить службу InterBase, а доступное только для чтения свойство `IsServiceRunning` проверяет факт выполнения службы.

Запрашиваемые параметры передаются с помощью свойства

```
property ServiceParamBySPB[const Idx: Integer]: String;
```

Уже встречавшиеся нам в табл. 24.2 флаги трассировки, информирующие программиста о характере выполняемых действий на сервере, доступны благодаря свойству

```
property TraceFlags: TTraceFlags;
```

Свойства сервера, *TIBServerProperties*

Компонент `TIBServerProperties` позволяет получить сводную информацию о текущем состоянии сервера. После присоединения к хосту, на котором развернут сервер, компонент приобретает возможность опросить СУБД по шести направлениям.

- Для выяснения версии СУБД следует вызвать метод:
  ```
  procedure FetchVersionInfo;
  ```
  Результаты выполнения процедуры представляются в виде объекта `TVersionInfo` (листинг 26.2) и помещаются в свойство
  ```
  property VersionInfo: TVersionInfo; // только для чтения
  ```

Листинг 26.2. Объявление класса `TVersionInfo`

```
type TVersionInfo = class
     ServerVersion: String;         //версия сервера
     ServerImplementation: string; //реализация сервера
     ServiceVersion: Integer;       //версия сервиса
     function IsMinimumVersion(MinVersion : string) : Boolean;
     procedure Clear;
  end;
```

- Второй вопрос задается с помощью метода
  ```
  procedure FetchDatabaseInfo;
  ```
  Благодаря `FetchDatabaseInfo()` мы узнаем количество и названия БД, развернутых на сервере, и число текущих подключений к СУБД. На этот раз результаты опроса передаются в объект `TDatabaseInfo` (листинг 26.3).

Листинг 26.3. Объявление класса `TDatabaseInfo`

```
type TDatabaseInfo = class
  public
    NoOfAttachments: Integer;//число подключений
    NoOfDatabases: Integer;  //число БД
    DbName: array of String; //имена БД
    constructor Create;
    destructor Destroy; override;
    procedure Clear;
  end;
```

Нужно отметить, что массив `DbName`, предназначенный для хранения имен, развернутых на сервере БД, заполнится информаций, только если к БД есть хотя бы одно подключение.

Доступ к объекту `TDatabaseInfo` предоставляет свойство

```
property DatabaseInfo: TDatabaseInfo; //только для чтения
```

- Третий вопрос серверу отправляется с помощью метода

```
procedure FetchConfigParams;
```

Процедура узнает пути к основным файлам сервера и передает их в объект `TConfigParams` (листинг 26.4).

Листинг 26.4. Объявление класса `TConfigParams`

```
type TConfigParams = class
 public
   ConfigFileData: TConfigFileData;
   ConfigFileParams: array of String; //массив с параметрами
   BaseLocation: String;              //расположение сервера
   LockFileLocation: String;          //файл блокировок
   MessageFileLocation: String;       //файл сообщений
   SecurityDatabaseLocation: String;  //БД с учетными записями
   constructor Create;
   destructor Destroy; override;
   procedure Clear;
 end;
```

Здесь поле `ConfigFileData` также является объектом, но на этот раз класса `TConfigFileData` (листинг 26.5).

Листинг 26.5. Класс `TConfigFileData`

```
type TConfigFileData = class
 public
   ConfigFileValue: array of integer;
   ConfigFileKey: array of integer;
   constructor Create;
   destructor Destroy; override;
   procedure Clear;
 end;
```

Результаты выполнения метода `FetchConfigParams()` помещаются в свойство:

```
property ConfigParams: TConfigParams; //только для чтения
```

- Четвертый и пятый вопросы связаны с лицензированием СУБД InterBase. Процедура

```
procedure FetchLicenseInfo;
```

спрашивает о поддерживаемых системой лицензиях.

Метод

```
procedure FetchLicenseMaskInfo;
```

получает подробности о маске лицензии. Результаты представляются в виде объектов `TLicenseInfo` и `TLicenseMaskInfo` (листинг 26.6).

Листинг 26.6. Классы `TLicenseInfo` и `TLicenseMaskInfo`

```
type TLicenseInfo = class
  public
    Key: array of String;  //массив ключей
    Id: array of String;   //массив идентификаторов
    Desc: array of String;
    LicensedUsers: Integer; //число лицензий
    constructor Create;
    destructor Destroy; override;
    procedure Clear;
  end;

type TLicenseMaskInfo = class
  public
    LicenseMask: Integer;
    CapabilityMask: Integer;
    procedure Clear;
  end;
```

Доступ к объектам можно получить благодаря свойствам

```
property LicenseInfo: TLicenseInfo; // только для чтения
property LicenseMaskInfo: TLicenseMaskInfo; // только для чтения
```

- Заключительный, шестой по счету вопрос серверу отправляет процедура

```
procedure FetchAliasInfo;
```

Она "выпытает" информацию о псевдонимах БД. Результаты передаются в свойство

```
property AliasInfos : TIBAliasInfos; // только для чтения
```

представляющее собой массив записей, каждый элемент которого описывается структурой `TIBAliasInfo` (листинг 26.7).

Листинг 26.7. Элемент массива `TIBAliasInfo`

```
type  TIBAliasInfo = class
  public
    Alias : String;  //псевдоним
    DBPath : String; //путь
  end;
```

Число доступных псевдонимов помещается в свойство

```
property AliasCount : Integer; // только для чтения
```

При желании, можно обратиться к любому элементу массива. Для этого предназначено свойство

```
property AliasInfo[Index : Integer] : TIBAliasInfo; // только для чтения
```

Пара методов позволяет добавлять и удалять псевдонимы

```
procedure AddAlias(Alias, DBPath : String);
procedure DeleteAlias(Alias : String);
```

Если программиста по какой-то причине не устраивает последовательность вопросов, то никто не запрещает переставить их в любом подходящем порядке. Более того, все вопросы можно задать одновременно, вызвав метод

```
procedure Fetch;
```

Но перед этим убедитесь, что в свойстве

```
property Options : TPropertyOptions;
TPropertyOption = (Database, License, LicenseMask, ConfigParameters,
                   Version, DBAlias);
```

подключены все опции, иначе будут заданы только отмеченные в опциях вопросы.

Если у вас сложилось впечатление, что с компонентом `TIBServerProperties` сложно работать, постараюсь разубедить с помощью небольшого примера.

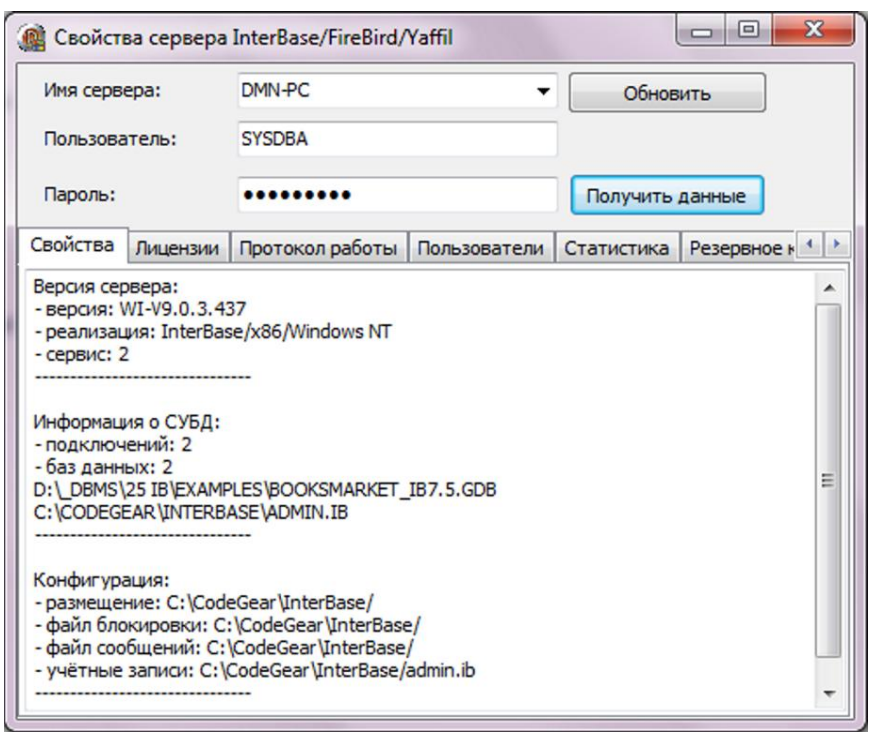


Рис. 26.3. Результаты исследования сервера InterBase 9.0

На главной форме проекта расположите героя этих страниц — компонент `IBServerProperties1`. Кроме того, нам потребуются строки ввода `edServerName`, `edUser` и `edPassword` для хранения имени хоста, пользователя и пароля и многострочный редактор `Memo1:TMemo` (в него мы станем собирать полученную информацию о сервере). Все остальное вы найдете в листинге 26.8.

Листинг 26.8. Получение параметров сервера InterBase

```
var i:integer;
begin
with IBServerProperties1 do
begin
  ServerName:=edServerName.Text;
  Protocol:=TCP;
  Params.Clear;
  Params.Add('user_name='+edUser.Text);
  Params.Add('password='+edPassword.Text);
  Active:=True;
  Memo1.Clear;
  FetchVersionInfo; //сведения о версии
     Memo1.Lines.Add('Версия сервера:');
     Memo1.Lines.Add('- версия: '+VersionInfo.ServerVersion);
     Memo1.Lines.Add('- реализация: '+VersionInfo.ServerImplementation);
     Memo1.Lines.Add('- сервис: '+IntToStr(VersionInfo.ServiceVersion));
  FetchDatabaseInfo; //информация о БД
    Memo1.Lines.Add('Информация о СУБД:');
    Memo1.Lines.Add('- подключений:'
                    +IntToStr(DataBaseInfo.NoOfAttachments));
    Memo1.Lines.Add('- баз данных:'
                    +IntToStr(DataBaseInfo.NoOfDatabases));
     for i:=0 to High(DataBaseInfo.DbName) do
           Memo1.Lines.Add(DataBaseInfo.DbName[i]);

  FetchConfigParams; //конфигурационные параметры
    Memo1.Lines.Add('');
    Memo1.Lines.Add('Конфигурация:');
    Memo1.Lines.Add('- размещение: '+ConfigParams.BaseLocation);
    Memo1.Lines.Add('- файл блокировки: '+ConfigParams.LockFileLocation);
    Memo1.Lines.Add('- файл сообщений:'
                    +ConfigParams.MessageFileLocation);
    Memo1.Lines.Add('- учетные записи:'
                     +ConfigParams.SecurityDatabaseLocation);
```

```
    for i:=0 to High(ConfigParams.ConfigFileParams) do
          Memo1.Lines.Add(ConfigParams.ConfigFileParams[i]);

  Active:=False;
end;
end;
```

ЗАМЕЧАНИЕ

Вся служебная информация о сервере InterBase хранится в системной базе данных admin.ib. Системная база располагается в каталоге сервера и содержит около 50 таблиц.

Результаты опроса иллюстрирует рис. 26.3, исследованию подверглась СУБД InterBase 9.0, развернутая на основе операционной системы Windows.

Сервис лицензирования, *TIBLicensingService*

В составе пакета сервера Interbase имеется динамическая библиотека iblicense.dll, отвечающая за управление лицензиями. Она позволяет проверять, добавлять, удалять лицензии и просматривать информацию о ключе и идентификаторе. Для получения доступа к функционалу лицензирования совсем не обязательно обращаться напрямую к названной библиотеке. Вместо этого можно воспользоваться услугами компонента `TIBLicensingService`.

Доступ к ключу и идентификатору лицензии предоставляют свойства:

```
property Key: String;
property ID: String;
```

Управляют лицензиями с помощью двух методов.

Метод

```
procedure AddLicense;
```

добавляет новую лицензию, а метод

```
procedure RemoveLicense;
```

удаляет лицензию. Того же результата можно добиться с помощью свойства

```
property Action: TLicensingAction; //по умолчанию LicenseAdd
TLicensingAction = (LicenseAdd, LicenseRemove);
```

но в этом случае следует вызывать унаследованный у класса `TIBControlService` метод `ServiceStart()`.

Пример добавления новой лицензии приведен в листинге 26.9.

Листинг 26.9. Добавление новой лицензии

```
with IBLicensingService1 do
begin
```

```
  Active:=True;
  Key:=edKey.Text;
  ID:=edID.Text;
  AddLicense;
  Active:=False;
end;
```

Коды авторизации InterBase "складируются" в корневой каталог сервера InterBase в файл ib_license.dat.

Конфигурирование сервера, *TIBConfigService*

Компонент `TIBConfigService` предоставляет программисту рычаги управления основными опциями базы данных, дублируя возможности консоли управления СУБД I**BConsole** по конфигурированию сервера (рис. 26.4).

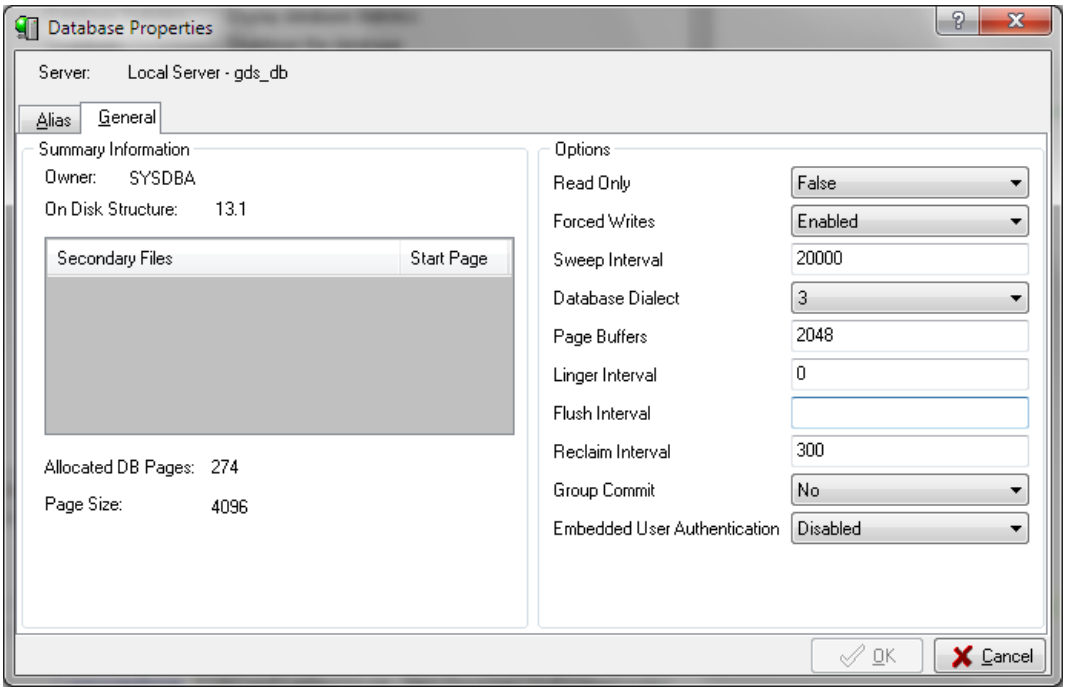


Рис. 26.4. Настройка основных свойств базы данных из консоли **IBConsole**

За подключение компонента-сервиса к интересующей БД отвечает его предок — класс `TIBCustomConfigService`.

Три свойства:

```
property Database : TIBDatabase;
property DatabaseName: String;
property Transaction : TIBTransaction;
```

предназначены для соединения с предоставляющим доступ к БД компонентом `TIBDatabase`, определением пути и имени файла БД и взаимодействия с транзакцией.

Для обеспечения надежного функционирования БД сервер Interbase способен поддерживать теневую копию базы данных.

Процедура

```
procedure ActivateShadow;
```

подключает ранее созданную теневую копии БД.

Теневая база данных ведется по принципу зеркала — в нее в реальном масштабе времени заносятся те же данные, что и в основную.

Замечание

Теневая база данных повышает надежность основной БД при случайном повреждении или удалении основного файла с данными. Однако она не может заменить архивные БД. Основная причина в том, что теневая база данных (являясь отражением основной) содержит все пользовательские ошибки и "мусор".

Процедура

```
procedure ShutdownDatabase (Options: TShutdownMode; Wait: Integer);
TShutdownMode = (Forced, DenyTransaction, DenyAttachment);
```

отключает БД от сервера.

Порядок отключения определяется параметром `Options`. Силовое отключение `Forced` отсоединит БД независимо от наличия к ней доступа через время ожидания в `Wait` секунд. Менее "агрессивный" способ — `DenyTransaction` — дождется завершения текущей транзакции. Наконец, самый деликатный подход обеспечит опция `DenyAttachment` — БД отключится только после отсоединения от нее всех пользователей.

При работе любой БД в ней накапливается "мусор" — устаревшие версии записей, появившиеся в результате редактирования и удаления данных.

Периодически производится вынос мусора:

```
procedure SweepDatabase;
```

Периодичность чистки устанавливает процедура

```
procedure SetSweepInterval (Value: Integer); // по умолчанию 20 000
```

Параметр определяет число транзакций, после завершения которых ненужные записи будут автоматически удалены:

```
procedure SetDBSqlDialect (Value: Integer);
```

Число страниц, используемых для буферизации записей, назначает процедура

```
procedure SetPageBuffers (Value: Integer);
```

Для подключения БД к серверу предназначена процедура

```
procedure BringDatabaseOnline;
```

Процедура

```
procedure SetReserveSpace (Value: Boolean);
```

увеличивает дисковое пространство, зарезервированное под размещение разных версий одних и тех же записей.

Режим записи данных в строки таблиц устанавливает процедура

```
procedure SetAsyncMode(Value: Boolean);
```

Значение `true` вводит асинхронный режим записи, `false` — синхронный.

Для перевода БД в режим только для чтения следует передать `true` в метод

```
procedure SetReadOnly(Value: Boolean);
```

Для восстановления нормального режима работы вызываем процедуру с параметром `false`.

Благодаря процедуре

```
procedure ReclaimMemory;
```

осуществляется очистка выделенной памяти от ненужных данных. Эту операцию можно автоматизировать, для чего достаточно установить периодичность очистки с помощью метода

```
procedure SetReclaimInterval(Value : Integer);
```

Настроенный по умолчанию работающий на платформе Windows сервер InterBase сразу записывает содержимое страниц памяти на жесткий диск компьютера. Однако допускается изменить ситуацию и позволить серверу дольше кэшировать данные в ОЗУ. Такой подход существенно ускорит работу СУБД, но при этом несколько повысит риск потери данных.

Для включения режима работы с кэшем обратитесь к процедуре

```
procedure FlushDatabase;
```

Метод

```
procedure SetFlushInterval(Value : Integer);
```

назначает период времени (в секундах), после которого данные из локального буфера будут отправлены на жесткий диск.

Возврат к обычному режиму работы обеспечит процедура

```
procedure DisableFlush;
```

Можно разрешить одновременно сохранять изменения, внесенные группой транзакций, для этого следует передать `true` в метод

```
procedure SetGroupCommit(Value : Boolean);
```

Для отказа от группового применения транзакции — направляем в метод `false`.

По умолчанию, после отключения от БД последнего пользователя она выгружается из памяти. Очистку памяти можно отсрочить с помощью процедуры

```
procedure SetLingerInterval(Value : Integer);
```

Метод устанавливает время ожидания в секундах, в течение которого InterBase станет ожидать подключения к БД нового пользователя.

Ведение журнала транзакций

Начиная с версии 7.5, у сервера InterBase появилась возможность вести журнал транзакций БД. Для создания журнала допустимо воспользоваться как средствами консоли `IBConsole` (рис. 26.5), так и возможностями компонента `TIBConfigService`.

Журнал (journal) состоит из одного или нескольких файлов, в которые заносятся все действия, осуществляемые с объектами БД и, в первую очередь, с записями таблиц. Благодаря этому создаются контрольные точки, позволяющие администратору при необходимости вернуть БД к состоянию на определенный момент времени. Все файлы журналов сохраняются в *журнал-архив* (journal-archive). По своей сути журнал-архив — это просто каталог, хранящий дамп с данными и файлы журналов.

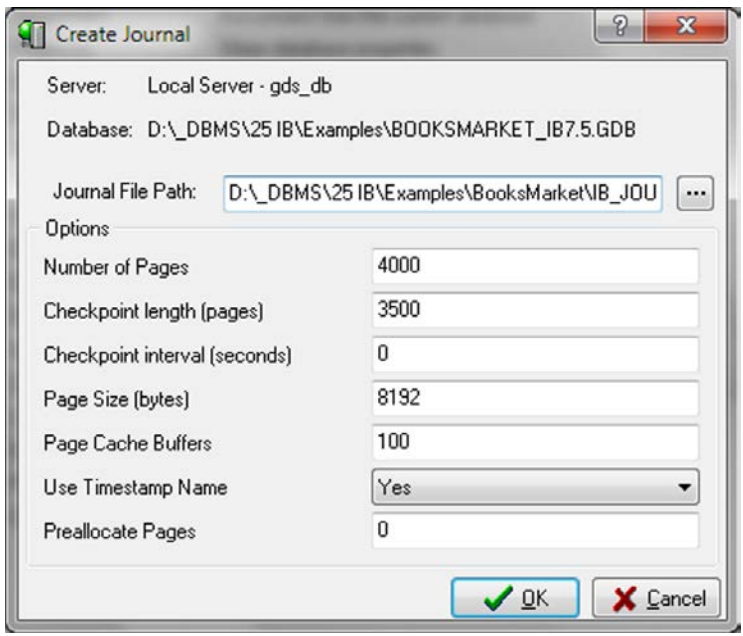


Рис. 26.5. Создание журнала БД из консоли **IBConsole**

В объектно-ориентированном представлении Delphi характеристики журнала транзакций реализуются в форме объекта `TIBJournalInformation`, интегрированного в состав компонента `TIBConfigService`. Доступ к параметрам журнала предоставляется благодаря свойству компонента

```
property JournalInformation : TIBJournalInformation; // только для чтения
```

Основные методы и свойства журнала приведены в табл. 26.1.

Таблица 26.1. *Характеристики журнала* `TIBJournalInformation`

Функция/свойство	Описание	По умолчанию
function CreateJournalAttributes : **String;**	Строка с атрибутами, применяемыми при создании и реконфигурации журнала, в состав строки входят параметры: CHECKPOINT LENGTH; CHECKPOINT INTERVAL; PAGE SIZE; PAGE CACHE; TIMESTAMP NAME или NO TIMESTAMP NAME	—
function CreateJournalLength : **String;**	Размер создаваемого журнала	—
property HasArchive : Boolean; {только для чтения}	Принимает значение true при наличии журнала-архива (создаваемого методом CreateJournalArchive)	—
property HasJournal : Boolean; {только для чтения}	Проверка факта существования журнала, если он имеется — свойство устанавливается в true	—
property CheckpointLength : Integer;	Указывает, сколько страниц должно быть записано в журнал до тех пор, пока не будет создана контрольная точка	500
property CheckpointInterval : Integer;	Определяет число секунд между созданиями контрольных точек	0
property PageLength : Integer;	Число страниц, после заполнения которых создается очередной файл журнала	500
property PageSize : Integer;	Размер страницы журнала в байтах, значение должно быть кратным X во второй степени (1024, 2048, 4096, 8192 или 16 384)	0
property PageCache : Integer;	Определяет, сколько страниц журнала должно быть кэшировано в памяти	100
property TimestampName : Boolean;	Определяет, будет ли добавлена метка времени в имя файла журнала	true
property Directory : String;	Папка, в которой будет вестись журнал	

Работа с журналом должна начинаться с создания журнала-архива. Для этого предназначен метод

```
procedure CreateJournalArchive(Directory : String);
```

в который передается путь к каталогу с файлами журналов. Процедура обращается к инструкции ISQL сервера `CREATE JOURNAL ARCHIVE`. О факте создания журнала-архива просигнализирует свойство `HasArchive` (см. табл. 26.1).

Внимание!

В многопользовательских проектах следует обеспечить совместный доступ к журналу транзакций всем клиентским приложениям. Поэтому располагайте журнал-архив в открытом для доступа сетевом ресурсе.

Для создания журнала необходимо воспользоваться процедурой

```
procedure CreateJournal;
```

предварительно не забыв в свойстве `Directory` (см. табл. 26.1) определить каталог, в котором будет вестись журнал. В результате в адрес сервера InterBase будет отправлена инструкция ISQL `CREATE JOURNAL`. Сигналом того, что журнал успешно создан, станет значение `true` в свойстве `HasJournal`.

Если журнал существует, то для получения полной информации о нем следует обратиться к методу

```
procedure GetJournalInformation;
```

Процедура

```
procedure AlterJournal;
```

изменяет настройки журнала транзакций.

Метод обращается к инструкции `ALTER JOURNAL SET` и передает в нее новые параметры журнала (которые вы должны передать в свойства экземпляра класса `TIBJournalInformation`).

Для удаления журнала вызовите метод

```
procedure DropJournal;
```

Это аналог обращения к процедуре `DROP JOURNAL`, после удаления файла журнала свойство `HasJournal` перейдет в состояние `false`.

Метод

```
procedure DropJournalArchive;
```

удаляет весь архив журналов.

Для этого процедура воспользуется инструкцией `DROP JOURNAL ARCHIVE`, свидетельством того, что архив удален, станет значение `false` в свойстве `HasArchive`.

Замечание

Для восстановления данных из журнала-архива следует воспользоваться утилитой gbak.exe, которую вы обнаружите в каталоге …\InterBase\BIN:

```
gbak -archive_recover [-until <метка_времени>] <имя_архива>
<имя_базы_данных>
```

Если опустить параметр `-until`, то БД будет возвращена к последней контрольной точке.

Протокол работы сервера, *TIBLogService*

Сервер InterBase протоколирует свою работу и сохраняет информацию об этом в файл InterBase.Log. В файле протокола содержится служебная информация о старте и останове сервера, о создании баз данных и о возникших ошибках. Для упрощения доступа к файлу протокола, в особенности, если сервер развернут на удаленной станции, в состав VCL Delphi включен компонент `TIBLogService`.

Прежде чем мы приступим к изучению `TIBLogService` и еще нескольких сервисных компонентов, отвечающих за проверку БД, сбор статистики, управление учетными записями и резервное копирование, уделим немного внимания их предку — классу `TIBControlAndQueryService` (см. рис. 26.2).

Основная задача класса `TIBControlAndQueryService` — обеспечить обмен неформатированным текстом между СУБД и сервисными компонентами.

За получение текстовых данных отвечают два очень похожих метода:

```
function GetNextLine : String;
function GetNextChunk : AnsiString;
```

Разница между ними в том, что функция `GetNextLine()` контролирует наличие в тексте символа перевода строки и запрашивает текст построчно. А функция `GetNextChunk()` нацелена на работу с порциями данных.

Свойство

```
property Eof: boolean; //только для чтения
```

позволяет контролировать факт достижения конца текста.

Обсудив особенности базового класса, вернемся к теме подраздела — получению данных из протокола сервера. Если вы загляните в модуль IBServices, в котором описаны все компоненты с линейки **InterBase Admin**, и поищите в нем объявление класса `TIBLogService`, то всего-навсего обнаружите следующие строки кода:

```
TIBLogService = class(TIBControlAndQueryService)
protected
    procedure SetServiceStartOptions; override;
end;
```

Иначе говоря, компонент `TIBLogService` на 99,9% повторяет код своего предка, не привнося в него ни одного нового свойства и метода.

Завершая разговор о компоненте `TIBLogService`, приведем фрагмент кода, демонстрирующий порядок передачи протокола в многострочный текстовый редактор `TMemo` (листинг 26.10).

Листинг 26.10. Получение протокола работы сервера

```
With IBLogService1 do
begin
   Active:=True;
```

```
   ServiceStart;
    while eof<>true do
       memo1.Lines.Add(GetNextLine);
   Active:=false;
end;
```

При повторении этого примера не забудьте вначале проинициализировать основные параметры компонента (`ServerName`, `Protocol`, `Params`).

Статистика, *TIBStatisticalService*

Сбором основной системной и статистической информации занимается компонент `TIBStatisticalService`. Для присоединения компонента к БД следует заполнить свойство

```
property DatabaseName: String;
```

Состав получаемой статистики зависит от состояния свойства

```
property Options : TStatOptions;
```

Перечень опций приведен в табл. 26.2.

Таблица 26.2. *Опции статистики* `TStatOption`

Опция	Описание
`DataPages`	Информация об использовании страниц данных в таблицах БД. По запросу будет возвращена информация о размещении основой страницы (primary pointer page); корневой индексной странице (index root page); общее число страниц с данными (data pages); числе указателей на страницы с данными (data page slots); процент заполнения страниц (average fill); распределение наполнения страниц (fill distribution)
`DbLog`	Включить в отчет информацию о протоколе БД
`HeaderPages`	Общая информация о БД, включая версию СУБД, версию ОС, дату создания БД, диалект, размер страницы, информацию о номерах транзакций, наличии теневых копий и т. д.
`IndexPages`	Сводные данные об индексах БД: названия, число уровней в индексном дереве (depth), число узлов в дереве (nodes), число страниц на нижнем уровне индекса (leaf buckets), среднее заполнение индексных страниц (fill distribution) и т. д.
`SystemRelations`	Информация о системных таблицах
`RecordVersions`	Информация о размерах и версиях записей таблиц
`StatTables`	Получение данных по определенным таблицам, указанным в свойстве `TableNames`

С точки зрения разработки программного кода, работа с компонентом `TIBStatisticalService` напоминает взаимодействие с рассмотренными ранее компонентами и вновь основывается на построчном поучении данных с помощью метода `GetNextLine()`. Листинг 26.11 демонстрирует порядок получения общей информации о СУБД с именем `dbname`.

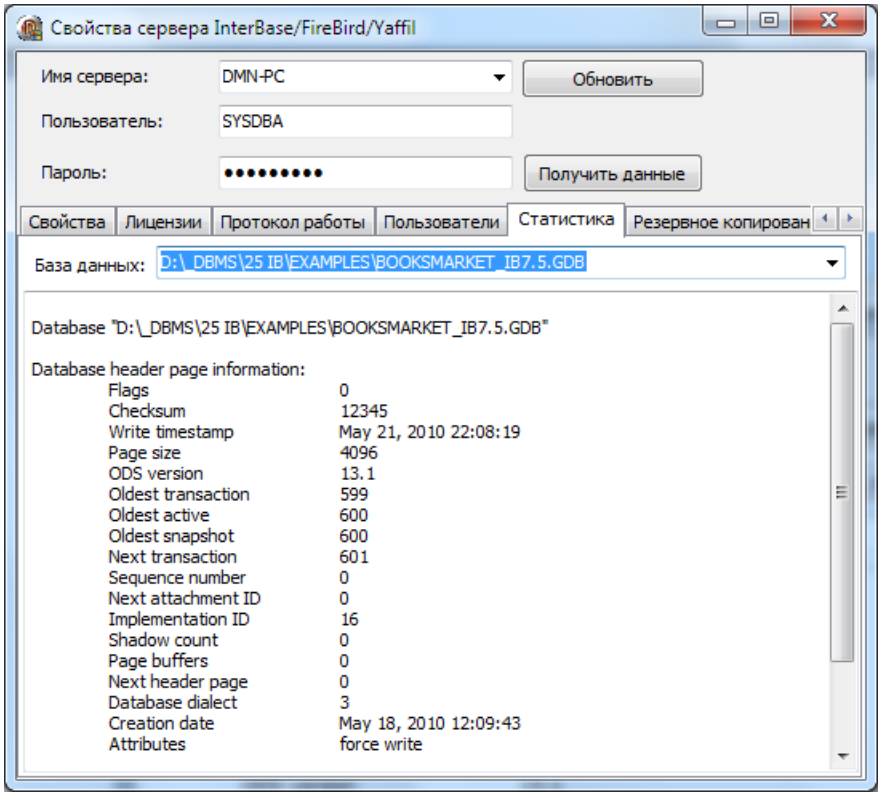


Рис. 26.6. Анализ системной БД ADMIN.IB в режиме HeaderPages

Листинг 26.11. Получение общей информации о БД

```
procedure TfrmMain.GetStatistics(const dbname: string);
begin
Memo1.Clear;
IBStatisticalService1.DatabaseName:=dbname;
 with IBStatisticalService1 do
   begin
     Active:=true;
     Options:=[HeaderPages];
     ServiceStart;
```

```
    while eof<>true do
        Memo1.Lines.Add(GetNextLine);
    Active:=false;
  end;
end;
```

Результат работы программы иллюстрирует экранный снимок (рис. 26.6). Специалист легко расшифрует все полученные данные. Например, параметр `Implementation ID` информирует об операционной системе хоста. Идентификатор может принимать значения от 1 до 20, в частности, число 16 соответствует 32-разрядным версиям Windows, а числа 19 и 20 — ОС Linux.

Проверка БД, *TBDValidationService*

Сервер InterBase обладает встроенным механизмом, позволяющим проверять состояние БД и останавливать выполнение незавершенных транзакций. Сервис проверки возложен на один из наиболее простых административных компонентов InterBase — `TIBValidationService`. Для подготовки компонента к работе следует определить имя обслуживаемой базы данных `DatabaseName` и разобраться с опциями проверки.

Особенности проверки определяются состоянием свойства:

```
property Options: TValidateOptions;
```

Доступные опции проверки приведены в табл. 26.3. Обратите внимание, что их заметно больше, чем предоставляемые по умолчанию из штатной консоли управления сервером InterBase (рис. 26.7).

***Таблица 26.3.** Опции проверки БД `TValidateOption`*

Опция	Описание
`LimboTransactions`	Возвратить список транзакций с неизвестным статусом
`CheckDB`	Проверить БД (для восстановления следует включить опцию `ValidateDB`)
`IgnoreChecksum`	При проверке игнорировать ошибки контрольной суммы
`KillShadows`	Удалить незадействованные теневые файлы
`MendDB`	Подготовить БД к резервированию
`SweepDB`	Осуществить чистку БД от “мусора”
`ValidateDB`	Восстанавливать БД
`ValidateFull`	Проверить и исправить ошибки в фрагментах записей

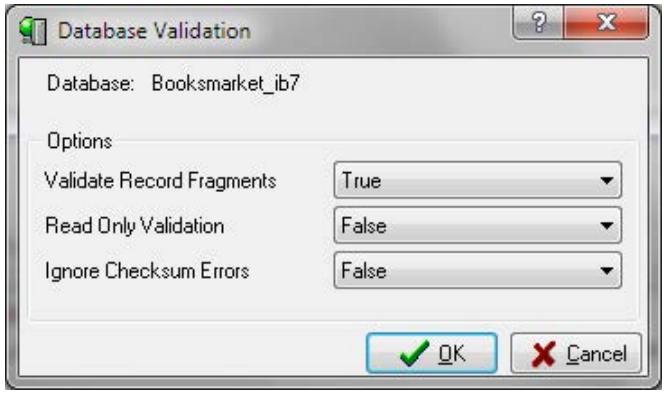


Рис. 26.7. Проверка БД из консоли **IBConsole**

Установив режим проверки БД, необходимо принять решение о том, как следует поступать с выполняющимися на момент проверки транзакциями и сообщить о решении в свойство

```
property GlobalAction: TTransactionGlobalAction;
```

Перечень возможных вариантов приведен в табл. 26.4.

Таблица 26.4. *Управление транзакциями* `TTransactionGlobalAction`

Опция	Описание
`CommitGlobal`	Зафиксировать транзакции
`RollbackGlobal`	Откатить транзакции
`RecoverTwoPhaseGlobal`	Восстановить двухфазные транзакции
`NoGlobalAction`	Не предпринимать никаких действий

Среди всех транзакций InterBase особого внимания достойна неопределенная транзакция "in limbo", которая может появиться в результате одновременной работы транзакции с двумя (и более) базами данных. Логика работы сервера требует, чтобы при завершении транзакции она была подтверждена во всех БД или ни в одной из них. Такая процедура подтверждения называется *двухфазной* (two phase commit). Проблема двухфазной процедуры в том, что подтверждения транзакций применяются последовательно, сначала в первой БД, затем во второй… Если в это время произойдет сбой, то вполне вероятно, что в каких-то БД изменения окажутся уже зафиксированными, а в каких-то нет. Как следствие, транзакция переходит в неопределенное состояние, с этого момента сервер оказывается не в состоянии принять корректное решение о подтверждении или откате транзакции.

Для получения сведений о зависших транзакциях следует вызвать метод

```
procedure FetchLimboTransactionInfo;
```

Чтобы узнать число транзакций, находящихся в неопределенном статусе, нужно обратиться к свойству:

```
property LimboTransactionInfoCount: Integer; //только для чтения
```

После этого мы сможем "пообщаться" с каждой из транзакций в статусе "in limbo", передавая ее индекс в свойство

```
property LimboTransactionInfo[Index: integer]: TLimboTransactionInfo;
```

Информация о транзакции возвращается в виде доступной только для чтения структуры (листинг 26.12).

Листинг 26.12. Класс описания транзакции `TLimboTransactionInfo`

```
type TLimboTransactionInfo = class
  public
    MultiDatabase: Boolean;      //признак доступа к нескольким БД
    ID: Integer;                 //идентификатор транзакции
    HostSite: String;            //имя хоста
    RemoteSite: String;          //имя удаленного сайта
    RemoteDatabasePath: String; //путь к удаленной БД
    State: TTransactionState;    //состояние
    Advise: TTransactionAdvise; //уведомление
    Action: TTransactionAction; //действие над транзакцией
  end;
```

Еще один способ доступа к состоящим в неопределенном статусе транзакциям обеспечивает свойство

```
property LimboTransactionInfos : TLimboTransactionInfos; {только для чтения}
```

Впрочем, результат будет такой же, как и при общении со свойством `LimboTransactionInfo`, с той лишь разницей, что мы получим доступ к массиву `TLimboTransactionInfos = array of TLimboTransactionInfo`.

Наконец, самое главное. Для устранения ошибок в неопределенных транзакциях вызываем метод

```
procedure FixLimboTransactionErrors;
```

Управление учетными записями, *TIBSecurityService*

По умолчанию во время инсталляции InterBase на сервере регистрируется всего одна учетная запись системного администратора SYSDBA с паролем "masterkey". Названная учетная запись предоставляет администратору весь спектр прав по отношению к серверу и развернутым на нем базам данных.

Замечание

Информация о зарегистрированных пользователях сервера InterBase хранится в таблице RDB$USERS системной базы данных admin.ib, а данные о ролях можно найти в таблице RDB$ROLES.

Для управления учетными записями в состав компонентов InterBase Admin введен класс TIBSecurityService.

Для получения полного списка учетных записей пользователей следует вызвать метод

```
procedure DisplayUsers;
```

Процедура возвратит некоторый набор записей, число которых можно узнать с помощью свойства

```
property UserInfoCount: Integer;
```

Вся полученная информация заносится в свойство-массив

```
property UserInfos : TUserInfos; //только для чтения
```

Для работы с отдельной записью удобнее воспользоваться свойством

```
property UserInfo[Index: Integer]: TUserInfo; //только для чтения
```

В свойство передается индекс интересующей нас строки, в ответ нам возвращается информационная структура (листинг 26.13).

Листинг 26.13. Информация о пользователе TUserInfo

```
type TUserInfo = class
 public
   UserName: String;          //логин
   FirstName: String;         //фамилия
   MiddleName: String;        //имя
   LastName: String;          //отчество
   GroupID: Integer;          //идентификатор группы
   UserID: Integer;           //идентификатор пользователя
   GroupName : String;        //имя группы
   SystemUserName : String;   //системное имя пользователя
   DefaultRole : String;      //роль по умолчанию
   Description : String;      //описание
   ActiveUser : Boolean;      //признак активности
 end;
```

Поля записи дублируются одноименными свойствами:

```
property UserName : String;
property FirstName : String;
property MiddleName : String;
property LastName : String;
```

```
property GroupID : Integer;  //по умолчанию 0
property UserID : Integer;   //по умолчанию 0
property GroupName : String;
property SystemUserName : String;
property DefaultRole : String;
property Description : String;
property ActiveUser : Boolean; //по умолчанию false
```

Для получения данных определенной учетной записи допускается обратиться к процедуре

```
procedure DisplayUser(UserName: String);
```

Единственный параметр процедуры должен содержать системное имя (логин) пользователя. В этом случае будет заполнен только один элемент свойства-массива — `UserInfo[0]`.

В листинге 26.14 продемонстрирован способ сбора информации о зарегистрированных на сервере пользователях. Для этого задействуется компонент `TListView`, который можно настроить во время визуального проектирования (установив свойство `ViewStyle` в состояние `Report` и добавив необходимое количество колонок в свойстве `Columns`) или во время выполнения программы, например в рамках события `OnShow()` формы, на которой размещен компонент.

Листинг 26.14. Подготовка компонента `TListView` к получению учетных записей

```
ListView1.ViewStyle:=vsReport;//табличный режим представления данных
ListView1.Columns.Clear;      //очистка колонок таблицы
//создаем требуемые колонки
with ListView1.Columns.Add do  Caption:='№';
with ListView1.Columns.Add do  Caption:='Логин';
with ListView1.Columns.Add do  Caption:='Имя';
with ListView1.Columns.Add do  Caption:='Отчество';
with ListView1.Columns.Add do  Caption:='Фамилия';
//…
```

Для сбора данных потребуется повторить строки листинга 26.15.

Листинг 26.15. Сбор учетных записей

```
with IBSecurityService1 do
begin
   Active:=true;
   DisplayUsers;          //запрашиваем список пользователей
   ListView1.Items.Clear; //удаляем старые данные
   for i:=0 to UserInfoCount - 1 do //просмотр всего списка
```

```
      begin
        With ListView1.Items.Add do
        begin
          Caption:=IntToStr(i+1);
          SubItems.Add(UserInfo[i].UserName);
          SubItems.Add(UserInfo[i].FirstName);
          SubItems.Add(UserInfo[i].MiddleName);
          SubItems.Add(UserInfo[i].LastName);
          SubItems.Add(UserInfo[i].GroupName);
          SubItems.Add(UserInfo[i].UserName);
          SubItems.Add(UserInfo[i].SystemUserName);
          SubItems.Add(UserInfo[i].DefaultRole);
          SubItems.Add(UserInfo[i].Description);
          if UserInfo[i].ActiveUser=true then
             SubItems.Add('Активен');
        end;
      end;
   Active:=false;
end;
```

Для добавления, редактирования и удаления учетных записей соответственно предназначены методы:

```
procedure AddUser;    //добавить учетную запись
procedure ModifyUser; //редактировать учетную запись
procedure DeleteUser; //удалить учетную запись
```

Перечисленные процедуры обращаются к инструкциям ISQL: `CREATE USER`, `ALTER USER` и `DROP USER`.

Во время добавления и модификации учетных записей не забываем про самое главное — пароль пользователя. Его нужно передать в свойство

```
property Password : String;
```

Функциональные возможности, которые предоставляет компонент для пользователя, зависят от состояния свойства

```
property SecurityAction: TSecurityAction //по умолчанию ActionAddUser
TSecurityAction = (ActionAddUser,    //разрешено добавлять учетную запись
                   ActionDeleteUser, //разрешено удалять
                   ActionModifyUser,  //разрешено редактировать
                   ActionDisplayUser);//разрешено просматривать
```

При проведении операций с учетными записями можно конкретизировать название базы данных, в интересах которой формируются полномочия пользователей. Для этого предназначено свойство

```
property UserDatabase : String;
```

Допустим, что в нашем проекте для регистрации пользователя разработана форма `frmUserLogin`. На форме размещены строки ввода (`edUserName`,

edFirstName и т. д.) и другие элементы управления, позволяющие администратору БД ввести необходимые учетные данные (рис. 26.8).

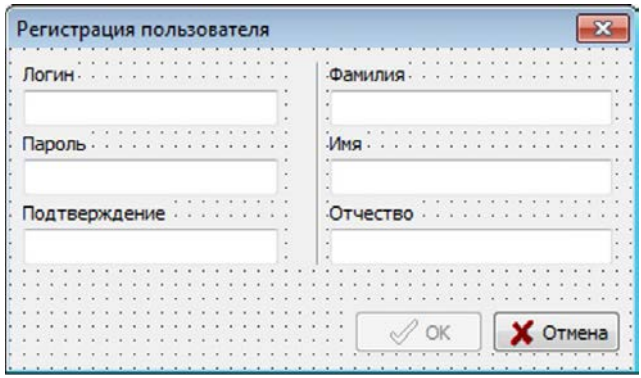


Рис. 26.8. Форма регистрации учетной записи нового пользователя

Тогда процедура добавления очередной учетной записи будет выглядеть так, как в листинге 26.16.

Листинг 26.16. Добавление новой учетной записи

```
procedure TfrmMain.acAddUserExecute(Sender: TObject);
begin
frmUserLogin:=TfrmUserLogin.Create(Application.MainForm);
if frmUserLogin.ShowModal=mrOK then
   With IBSecurityService1 do
   begin
     Active:=true;
     TRY
       UserName:=frmUserLogin.edUserName.Text;
       FirstName:=frmUserLogin.edFirstName.Text;
       MiddleName:=frmUserLogin.edMiddleName.Text;
       LastName:=frmUserLogin.edLastName.Text;
       Password:=frmUserLogin.edPassword.Text;
     AddUser;
     UpdateUsersList;
     FINALLY
     Active:=false;
     END;
   end;
frmUserLogin.Release;
end;
```

Форма регистрации вызывается из главной формы проекта и выводится на экран в модальном режиме. Пользователь вводит все необходимые данные и нажимает кнопку **OK**.

Сам по себе факт добавления нового пользователя не означает, что он приобретает абсолютную власть над БД. Политика безопасности InterBase требует, чтобы пользователь наделялся определенными правами на конкретные таблицы и действия с ними. Чтобы избавиться от рутинных операций по предоставлению каждому вновь создаваемому пользователю стандартного набора прав, в InterBase имеется возможность создавать роли (roles) с заранее подобранными параметрами.

Для того чтобы все вновь создаваемые учетные записи пользователей включались в роль по умолчанию, имя роли нужно передать в свойство

```
property DefaultRole : String;
```

Пример удаления учетной записи приведен в листинге 26.17.

Листинг 26.17. Удаление учетной записи

```
Procedure DropUser(const User :string);
begin
if MessageBox(frmMain.Handle,
              pChar('Удалить пользователя "'+s+'" ?'),
              pChar('Удаление учетной записи'),
              MB_YESNO)=idYes  then
              begin
                with IBSecurityService1 do
                begin
                 Active:=true;
                 UserName:=User;
                 DeleteUser;
                 Active:=false;
                end;
                UpdateUsersList;
              end;
end;
```

Единственный параметр процедуры — системное имя пользователя.

В завершение разговора о компоненте отметим еще два свойства, позволяющие управлять механизмом *встроенной аутентификации* (Embedded User Authentication, EUA). Особенность EUA в том, что он предоставляет возможность сохранять имя и пароль пользователя непосредственно в БД. Это достаточно удобно, в особенности, когда требуется позаботиться о переносимости БД.

Для активизации EUA следует передать значение `true` в единственный параметр процедуры

```
procedure EnableEUA(Value : Boolean);
```

Для приостановки аутентификации EUA обращаемся к процедуре

```
procedure SuspendEUA(Value : Boolean);
```

Резервное копирование и восстановление, *TIBBackupService* и *TIBRestoreService*

Нормальное функционирование базы данных зависит от очень многих факторов. Выход из строя аппаратной части сервера, сбой питания на сервере, воздействие вирусов, ошибки в ОС и прикладном ПО, действия неподготовленных пользователей, сознательное воздействие на БД злоумышленников, — вот далеко не полный список причин, из-за которых БД может быть повреждена или даже полностью уничтожена.

Разработано множество методов защиты БД, о них мы уже говорили в *главе 10*, но наиболее надежный среди них — своевременное резервное копирование данных.

Сервис резервного копирования не просто создает новую полную копию файла с БД. Алгоритм работы InterBase значительно сложнее. Во-первых, из резервной копии удаляется весь мусор (устаревшие и ненужные записи). Во-вторых, оптимизируются индексы таблиц. В-третьих, проверяется корректность данных. Кроме того, процесс резервного копирования не требует передачи БД в монопольное пользование утилите копирования и не мешает другим пользователям в это же время работать с БД.

Для того чтобы сервис резервного копирования и восстановления БД можно было широко применять в проектах Delphi, в состав InterBase Admin введены компоненты `TIBBackupService` и `TIBRestoreService`. Первый отвечает за создание резервных копий, второй — за восстановление данных из архивов. Оба класса созданы на фундаменте `TIBBackupRestoreService`.

Ключевое свойство, унаследованное сервисными компонентами от родительского класса, отвечает за визуализацию процесса создания резервной копии и восстановление БД из копии:

```
property Verbose : Boolean; \\по умолчанию False
```

Если вы имеете хотя бы небольшой опыт резервного копирования БД штатными средствами InterBase (рис. 26.9), то без особого труда разберетесь с основными свойствами компонента `TIBBackupService`.

Путь и имя файла с исходной базой данных передаются в свойство

```
property DatabaseName: String;
```

Путь и имя будущего файла с резервной копией заносятся в свойство

```
property BackupFile: TStrings;
```

Так как свойство представляет собой набор строк, в него можно передавать ссылки на несколько файлов-архивов.

На сегодняшний день уже утратившее актуальность свойство

```
property BlockingFactor: Integer;
```

устанавливает коэффициент блокирования стримера (накопителя на магнитной ленте).

Особенности создания резервной копии определяет свойство

```
property Options : TBackupOptions;
```

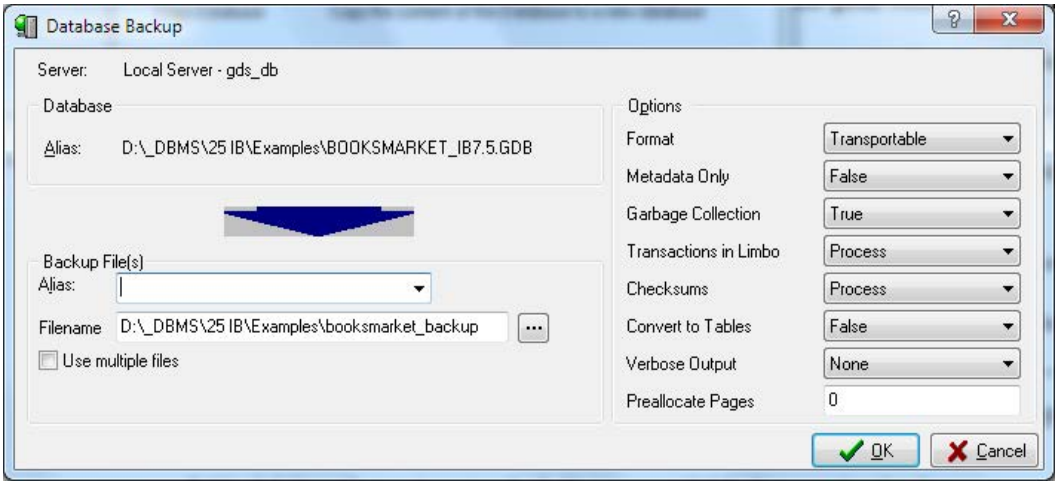


Рис. 26.9. Диалог резервного копирования консоли управления InterBase

Перечень опций приведен в табл. 26.5.

Таблица 26.5. *Опции резервирования БД* `TBackupOption`

Опция	Описание
`IgnoreChecksums`	Не проверять контрольную сумму
`IgnoreLimbo`	Игнорировать незавершенные двухфазные транзакции
`MetadataOnly`	Архивировать только метаданные. В результате получим пустую БД с таблицами, индексами, генераторами, триггерами и хранимыми процедурами, но таблицы окажутся незаполненными. Такой способ резервного копирования может пригодиться для создания пустой БД с целью ее развертывания на другом хосте
`NoGarbageCollection`	Запретить сборку мусора
`OldMetadataDesc`	Сохранять метаданные в старом формате
`NonTransportable`	Сделать копию непереносимой на другие серверы
`ConvertExtTables`	Перенести данные из внешних таблиц во внутренние

Листинг 26.18 демонстрирует порядок создания резервной копии.

Листинг 26.18. Создание резервной копии

```
Screen.Cursor:=crHourGlass;
with IBBackupService1 do
begin
```

```
  try
   Active:=True;
   Verbose:=True;
   DataBaseName:=Edit1.Text;   //имя БД
   BackupFile.Add(Edit2.Text); //файл резервной копии
   Options:=Options+[IgnoreLimbo]; //игнорировать 2-фазные транзакции
   ServiceStart;                   //старт сервиса
   ShowMessage('Операция завершена!');
  finally
   Screen.Cursor:=crDefault;
   Active:=False;
  end;
end;
```

Научившись проводить резервное копирование, рассмотрим обратную сторону медали — сервис восстановления данных из резервной копии. В визуальной библиотеке компонентов задача восстановления возложена на компонент `TIBRestoreService`.

Для восстановления базы данных следует определить путь и имя восстанавливаемых файлов:

```
property DatabaseName: TStrings;
```

и указать путь и имя исходного файла с архивом:

```
property BackupFile: TStrings;
```

Перед началом операции восстановления допускается изменить размер страниц восстанавливаемой БД:

```
property PageSize: Integer; \\по умолчанию 4096
```

Свойство может принимать значения: 1024, 2048, 4096, 8192 или 16 384.

Размер буфера в страницах БД назначается с помощью свойства

```
property PageBuffers: Integer;
```

Особенности восстановления данных определяются свойством

```
property Options : TRestoreOptions; \\по умолчанию [CreateNewDB]
```

Описание опций `TRestoreOptions` приведено в табл. 26.6.

Таблица 26.6. *Опции восстановления БД* `TRestoreOption`

Опция	Описание
`DeactivateIndexes`	По умолчанию, после восстановления таблиц и данных InterBase обновляет все индексы таблиц. В этот момент может возникнуть ошибка, например из-за наличия записей-дубликатов. Чтобы избежать сбоя во время восстановления БД, можно отключить индексы
`NoShadow`	В восстанавливаемой БД отключается ведение теневой копии данных

Таблица 26.6 *(окончание)*

Опция	Описание
`NoValidityCheck`	Не проверять корректность восстанавливаемых данных. Опцию следует активировать, если в восстанавливаемой БД планируется изменить некоторые накладываемые на данные ограничения, а в архивной копии имеются данные, не соответствующие этим ограничениям
`OneRelationAtATime`	По умолчанию во время восстановления БД сначала восстанавливаются метаданные, содержащие информацию о структуре БД, а затем данные таблиц. Если во время восстановления возникают ошибки несоответствия записей таблиц требованиям метаданных, то можно задействовать опцию `OneRelationAtATime`, в этом случае метаданные и данные каждой таблицы будут сохраняться одновременно
`Replace`	Опция разрешает перезаписать существующий файл БД, иначе восстановление будет отменено
`CreateNewDB`	Восстановить БД в новый файл
`UseAllSpace`	В настройках по умолчанию при приближении заполнения страницы БД к 80% создается новая страница данных. При включении опции `UseAllSpace` очередная страница данных создается только после 100% заполнения предыдущей страницы. Активизация этой опции полезна в БД, эксплуатируемых в режиме только для чтения

Пример восстановления БД (листинг 26.19) во многом похож на код формирования резервной копии.

Листинг 26.19. Создание нового экземпляра БД из резервной копии

```
Screen.Cursor:=crHourGlass;
with IBRestoreService1 do
begin
  try
   Active:=True;
   Verbose:=True;
   DataBaseName.add(Edit1.Text); //имя БД
   BackupFile.Add(Edit2.Text);   //файл резервной копии
   Options:= [CreateNewDB];      //создаем новый экземпляр БД
   ServiceStart;                 //запускаем сервис
   ShowMessage('Операция завершена!');
  finally
   Screen.Cursor:=crDefault;
   Active:=False;
  end;
end;
```

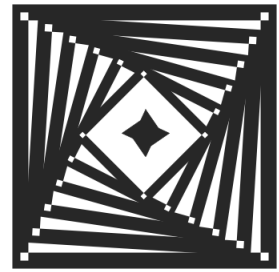
Восстановление данных из резервной копии — весьма ответственное мероприятие, поэтому во время его проведения следует руководствоваться двумя правилами:

- Во время восстановления данных из архивной копии администратору следует организовать монопольный доступ к серверу InterBase.
- Всегда восстанавливайте БД в новый файл (опция `CreateNewDB`), не затирая исходную базу. Иначе, если архив окажется некорректным, вы рискуете потерять действующие данные.

Резюме

Производительность сервера БД, надежность хранения данных, политика безопасности и многое другое зависит не только от качества написанной программистом программы, но и от действий администратора системы. Однако хороший администратор дорого стоит, поэтому не каждая компания в состоянии позволить себе содержать квалифицированного "бездельника". В подобных организациях роль администратора возлагается на одного из сотрудников, у которого и без того хватает своих должностных обязанностей, не говоря уже об изучении порядка работы с диалектом ISQL в консоли InterBase.

Именно для упрощения жизни внештатных администраторов и разработана линейка компонентов **InterBase Admin**. Они умеют создавать резервные копии и восстанавливать из них БД, получать сведения об основных параметрах сервера, проверять корректность хранимых данных, предоставлять сервис лицензирования и т. п. Самое главное достоинство административных компонентов в том, что они позволяют быстро создать приложения, помогающие обычному пользователю поддерживать работоспособность сервера БД.



Глава 27

Технология доступа к данным dbExpress

В течение долгого времени доминирующим способом доступа к базам данных в среде Delphi был механизм BDE. После почти десяти лет эксплуатации фирменного "двигателя" (примерно к началу 2000-х годов) разработчики Delphi пришли к однозначному выводу — технология BDE перестала удовлетворять потребностям современного рынка программного обеспечения. Причин тому несколько. Не хочу упорядочивать их по значимости, т. к. они равноценны.

- Изначально компания Borland (так именовался создатель Delphi в 1990-х годах) формировала BDE как максимально открытую структуру, предоставив программистам развитую палитру доброжелательных компонентов и низкоуровневый набор функций BDE API. Эту открытость оценили тысячи программистов и включили BDE в свои проекты. Первоначальная успешность BDE в итоге оказалось миной замедленного действия. Вся сложность заключалась в обеспечении обратной совместимости между очередной версией Delphi и старыми проектами. При подготовке выхода в свет новой Delphi программисты формально были готовы совершенствовать BDE, но не могли, иначе она перестала бы воспринимать предыдущие проекты. Получился замкнутый круг…
- Клиентские приложения, построенные на основе BDE, при установке на компьютеры заказчиков требовали размещения на них ядра BDE. А это несколько мегабайт библиотек и драйверов, из которых большая часть никогда не применялась.
- Разработчики новых драйверов BDE не успевали за несущимся вперед рынком СУБД. Что, в свою очередь, не позволяло компаниям оперативно переводить свои базы данных на более современные платформы СУБД.

Набив изрядное количество шишек, программисты решили несколько сместить акценты при построении очередной фирменной технологии. Изначально, еще на этапе разработки эскизов перспективной модели создатели технологии поставили себе следующие цели:

- Будущая технология обязана целиком и полностью опираться на язык SQL.
- Механизм доступа к данным необходимо сделать максимально простым и нересурсоемким.

- Технология не должна зависеть от операционной системы.
- Затраты при распространении клиентских приложений следует минимизировать.

Указанные цели были вскоре достигнуты в рамках модели доступа к данным dbExpress. Сегодня dbExpress может работать на платформе как Windows, так и Linux. Драйверы доступа к данным реализованы в виде небольших по размеру файлов библиотек. Вместе с поставкой Embarcadero RAD Studio разработчику предлагаются драйверы доступа к большинству современных СУБД (Informix, Microsoft SQL Server, Oracle, DB2, Interbase, MySQL и Blackfish SQL), но, правда, за это пришлось расплачиваться утратой ряда преимуществ. В первую очередь это отсутствие кэширования данных на клиентской стороне, и как следствие — однонаправленные курсоры, позволяющие перемещаться по записям отношения только вперед. Указанные недостатки существенно осложняют работу программиста, но при наличии определенных знаний и умений их можно преодолеть.

Соединение с сервером БД, компонент *TSQLConnection*

Наличие в проекте компонента `TSQLConnection` — обязательное условие для организации взаимодействия между клиентским приложением dbExpress и базой данных. В иерархии наследования VCL перед ним расположен класс `TCustomConnection`. Общий предок обусловливает схожесть основных родовых черт между нашим новым знакомым и такими важными классами, как `TADOConnection`, `TIBDatabase` и `TDatabase` (на сегодня уже неактуальным, появившимся на свет еще в эпоху безраздельного царствования BDE) (рис. 27.1).

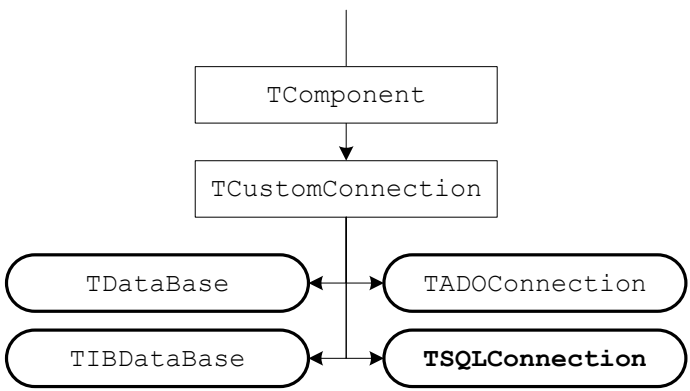


Рис. 27.1. Иерархия наследников класса `TCustomConnection`

Настройка компонента

Для обеспечения соединения клиентского приложения с сервером баз данных программисту требуется позаботиться о решении нескольких частных подзадач. В первую очередь, следует выбрать подходящий драйвер СУБД (свойства `DriverName` и `LibraryName`). Посредничество драйвера необходимо для того, чтобы наши SQL-запросы корректно транслировались в адрес сервера. Совместно с драйвером СУБД на клиентской стороне трудится файл динамической библиотеки (свойство `VendorLib`), он позволяет нашим приложениям обращаться к специфичным функциям сервера баз данных. Если библиотека базируется на модели COM, то еще потребуется указать интерфейс головной функции для конкретной модификации СУБД (свойство `GetDriverFunc`). Таким образом, после размещения компонента `TSQLConnection` в модуле данных нашего проекта понадобится безошибочно описать три свойства. Понимая, что столь бурное начало может оказаться не по плечу начинающему программисту, разработчики Delphi пошли ему навстречу и предложили упрощенный способ настройки компонента. Такой способ вхождения в соединение начинается с определения содержания свойства

```
property ConnectionName: string;
```

По умолчанию в интересах `ConnectionName` заранее заготовлено несколько вариантов имен соединений (по числу драйверов, поддерживаемых СУБД). Во время проектирования клиентского приложения имя соединения выбирают в инспекторе объектов из выпадающего списка напротив названия свойства. Создать новое соединение или перенастроить уже существующее позволяет утилита Data Explorer.

Если упрощенный способ настройки соединения вас не устраивает, то никто не запрещает все реализовать своими руками. Поговорим об этом подробнее.

Начнем с того, что файлы драйверов мирно проживают в каталоге BIN. В поставке Delphi XE вы обнаружите достаточно широкий спектр поддерживаемых СУБД (табл. 27.1). Размер большинства драйверов не превышает 500 Кбайт, что снижает размер устанавливаемого на клиентский компьютер дистрибутива. Что касается файлов клиентских библиотек, то их следует "выуживать" из набора файлов интересующего вас сервера баз данных.

ВНИМАНИЕ!

Без файлов драйвера и клиентских библиотек разработанное нами приложение при переносе на другие компьютеры утратит работоспособность, поэтому не забывайте включать перечисленные файлы в свой инсталляционный пакет!

Если мы не планируем использовать именованное соединение, то работа с компонентом начинается с выбора драйвера с помощью свойства

```
property DriverName: string;
```

Драйвер определит, какие именно библиотеки должны использоваться для обслуживания специфичной СУБД и для клиентской стороны. В случае Blackfish SQL и DataSnap драйверы СУБД компонуются статическим образом (т. е. присоединяются к исполняемому файлу).

Таблица 27.1. *Драйверы и библиотеки dbExpress*

СУБД	Библиотека	
	Драйвер СУБД	**Клиентская библиотека**
Sybase Adaptive Server Anywhere (ASA)	dbxasa.dll	dbodbc9.dll
Sybase Adaptive Server Enterprise (ASE)	dbxase.dll	libct.dll и libcs.dll
DB2	dbxdb2.dll	db2cli.dll
InterBase	dbxint.dll	gds32.dll сборка для InterBase
Firebird	dbxfb.dll	gds32.dll сборка для Firebird
Informix	dbxinf.dll	isqlb09a.dll
MSSQL	dbxmss.dll	oledb, sqlncli10.dll
MySQL	dbxmys.dll	LIBMYSQL.dll
Oracle	dbxora.dll	oci.dll
Blackfish SQL	К исполняемому файлу подключается модуль DBXBlackfishSQL	Применяются сборки .NET, в частности: Borland.Data.TDBXClientDriverLoader, Borland.Data.DbxClientDriver
DataSnap	К исполняемому файлу подключается модуль DBXDataSnap	

Замечание

При появлении очередных версий Delphi названия драйверов и библиотек могут изменяться. Поэтому для большей уверенности перед перекомпиляцией прежних проектов обязательно изучите файлы инициализации dbxdrivers.ini и dbxconnections.ini с описанием стандартных настроек для каждого из поддерживаемых серверов БД.

После назначения имени драйвера в свойстве `DriverName` компонент автоматически подключит соответствующие библиотеки к свойствам

```
property LibraryName: string;//драйвер СУБД
property VendorLib: string;//клиентская библиотека
```

Подбором драйверов и библиотек этап предварительной настройки соединения завершается, и мы переходим к описанию нюансов обслуживания конкретной базы данных. Для этого нам следует обратиться к свойству

```
property Params: TStrings;
```

В результате среда программирования вызовет редактор соединения, позволяющий назначить его параметры и протестировать их корректность. Список пара-

метров зависит от конкретной реализации сервера, но в самом общем случае нам приходится определиться с именем хоста, на котором развернут сервер, именем базы данных, именем и паролем пользователя. Перечень ключей компонент получает из файла инициализации dbxconnections.ini.

Для того чтобы компонент `TSQLConnection` загружал конфигурационные данные перед моментом инициализации соединения с СУБД, установите в `true` свойство

```
property LoadParamsOnConnect: Boolean;
```

В таком случае компонент обратится к файлу инициализации dbxconnections.ini и импортирует его данные. Если же разрабатываемое приложение намерено использовать какие-то эксклюзивные настройки, то установите свойство в `false` и загрузитесь из указанного вами файла инициализации с помощью процедуры

```
procedure LoadParamsFromIniFile( AFileName : String = '' );
```

Здесь `AFileName` — имя настроечного файла, впрочем, если параметр не передавать, то метод обратится к файлу dbxconnections.ini.

Управление соединением

Разобравшись с базовыми свойствами компонента, обсудим самое главное — процесс управления соединения. Как и положено наследнику `TCustomConnection`, главный герой этой главы обладает парой привычных методов, осуществляющих подключение и отключение компонента:

```
procedure Open; overload;
procedure Close;
```

Тот же результат достигается за счет обращения к свойству

```
property Connected: Boolean;
```

Кроме того, по состоянию `Connected` можно судить о факте соединения с сервером.

В ключах параметров указывают все тонкие настройки сервера и регистрационные данные пользователя так, как продемонстрировано в листинге 27.1.

Листинг 27.1. Настройка основных параметров соединения

```
with SQLConnection1.Params do
 begin
   Values['HostName'] :='DEVELOPER\MSSQL_DeveloperServer';
   Values['DataBase'] :='MyDatabaseName';
   Values['User_Name']:=edUserName.Text;
   Values['Password'] :=edPassword.Text;
 end;


SQLConnection1.Connected:=true;
```

При желании настроечные данные можно редактировать и во время проектирования. Для этого предназначен редактор соединения, вызываемый щелчком по свойству `Params` компонента.

С процессом установления и разрыва соединения связан классический перечень событий (табл. 27.2).

Таблица 27.2. *События, связанные с соединением и разрывом соединения*

Событие	Описание
`property BeforeConnect: TNotifyEvent;`	Генерируется перед установкой соединения
`property AfterConnect: TNotifyEvent;`	Генерируется после установки соединения
`property BeforeDisconnect: TNotifyEvent;`	Генерируется перед разрывом соединения
`property AfterDisconnect: TNotifyEvent;`	Генерируется после разрыва соединения

Если программная логика приложения требует постоянного удержания соединения с БД, даже в случае, когда нет ни одного активного открытого набора данных, то проследите, чтобы свойство `KeepConnection` оставалось в состоянии `true`. Иначе с закрытием последнего набора данных соединение будет разорвано.

В листинге 27.1 мы реализовали свой собственный способ регистрации пользователя на сервере. Если пойти стандартным путем, то для передачи в адрес СУБД имени и пароля пользователя (а также основных параметров соединения) следует использовать событие

```
property OnLogin: TSQLConnectionLoginEvent;
type TSQLConnectionLoginEvent = procedure(Database: TSQLConnection;
                                          LoginParams: TStrings) of object;
```

Замечание

В череде событий компонента `TSQLConnection` событие `OnLogin()` генерируется сразу после `BeforeConnect()`. Событие будет вызвано только при условии, что свойство `LoginPrompt` установлено в состояние `true`.

О текущем состоянии соединения можно судить по свойству

```
property ConnectionState: TConnectionState;
```

Возможные варианты состояний приведены в табл. 27.3.

Таблица 27.3. *Возможные значения* `TConnectionState`

Состояние	Описание
`csStateClosed`	Соединение отсутствует
`csStateOpen`	Компонент соединен с БД

Таблица 27.3 *(окончание)*

Состояние	Описание
csStateConnecting	Процесс соединения инициирован, но еще не завершен
csStateExecuting	Компонент отправил в адрес СУБД инструкцию SQL, и она выполняется
csStateFetching	Компонент получает информацию от сервера
csStateDisconnecting	Момент разрыва соединения, но оно еще не завершено

Создание БД

Хотя основное предназначение `TSQLConnection` заключается в установлении соединения с БД, при необходимости компонент готов и на "неординарные" поступки. Листинг 27.2 демонстрирует порядок создания БД для платформы Blackfish SQL.

Листинг 27.2. Создание БД средствами `TSQLConnection` на платформе Blackfish SQL

```
with SQLConnection1 do
begin
  DriverName:= 'BlackfishSQL';
      with Params do
      begin
        Add('drivername=BLACKFISHSQL');
        Add('Password=masterkey');
        Add('user_name=sysdba');
        Add('port=2508');
        Add('create=true');
        Add('readonlydb=false');
        Add('HostName=localhost');
        Add('Database=D:\booksmarket.jds');
      end;
      LoginPrompt:= False;
      Connected:= True;
end;
```

Создание подключения в Data Explorer

Для управления соединениями с БД в состав Delphi входит утилита Data Explorer, она вызывается из главного меню Delphi: **View | Data Explorer**. Рассмотрим порядок работы с утилитой на примере подключения к БД, развернутой на базе сервера MySQL.

Замечание

Популярную бесплатную СУБД MySQL вы можете скачать с сайта **http://www.mysql.ru/**. На момент написания книги на страничке MySQL была опубликована альфа-версия MySQL 6.0. Это вполне работоспособная пробная версия сервера, но программисту Delphi лучше остановить свой выбор на MySQL версии 5.0 или 5.1.

Допустим, что на базе сервера MySQL 5.1 развернута демонстрационная БД книжного магазина. Для создания пустой схемы в командной строке утилиты MySQL Command Line Client достаточно ввести одну строку кода (рис. 27.2):

```
CREATE DATABASE BOOKSMARKET;
```

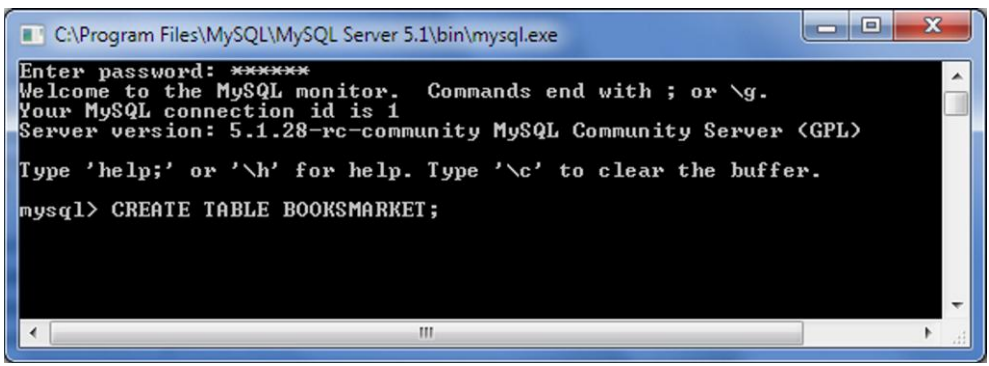


Рис. 27.2. Утилита MySQL Command Line Client

В окне управления **Data Explorer** среды Delphi вы обнаружите иерархию узлов с названиями СУБД, поддерживаемых технологией dbExpress. Найдите узел **MYSQL** и, щелкнув по узлу правой кнопкой мыши, вызовите его контекстное меню (рис. 27.3). Нас интересует элемент меню **Add New Connection** (Добавить новое соединение). В появившемся на экране окне в строке ввода **Connection Name** (Название соединения) введите подходящее название (например, я выбрал имя `MYSQL_BOOKSMARKET`) и нажмите кнопку **OK**.

В результате утилита Data Explorer создаст узел соединения с параметрами по умолчанию. Нам потребуется внести некоторые изменения в предложенные системой настройки (рис. 27.4). Для этой цели вызываем всплывающее меню только что созданного узла `MYSQL_BOOKSMARKET` и выбираем пункт меню **Modify Connection** (Модифицировать соединение).

В ответ на это действие на дисплее компьютера отобразится окно диалога, позволяющего исправить в параметры соединения (рис. 27.5).

Нам потребуется осуществить следующие операции:

- В строке ввода **Server Name** (Имя сервера) определяем имя сетевой станции, на которой развернут сервер MySQL. Если сервер установлен на локальном компьютере, то вводим слово "localhost".
- В строке **Database Name** (Название БД) указываем имя интересующей нас БД.

- В строке **User Name** (Имя пользователя) должно оказаться имя администратора СУБД (для MySQL это имя "root").
- В строке **Password** (Пароль) указываем пароль администратора.

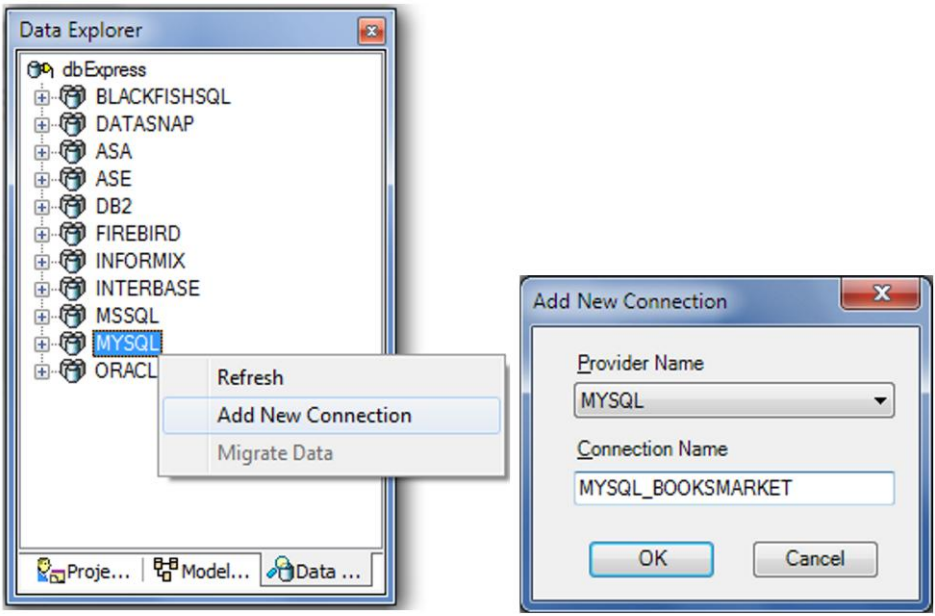


Рис. 27.3. Создаем новое соединение в проводнике данных Data Explorer

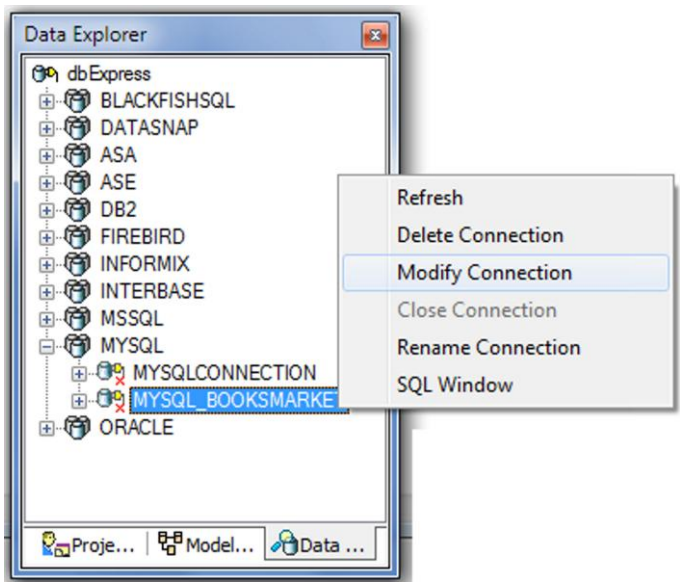


Рис. 27.4. Вызов диалога настройки соединения в Data Explorer

Нажав кнопку **Test Connection** (Тест соединения), убедитесь в корректности параметров соединения, только после этого, щелкнув по кнопке **OK**, закройте окно настройки соединения.

С этого момента в списке доступных соединений `ConnectionName` компонента `TSQLConnection` появится новая строка — `MYSQL_BOOKSMARKET`.

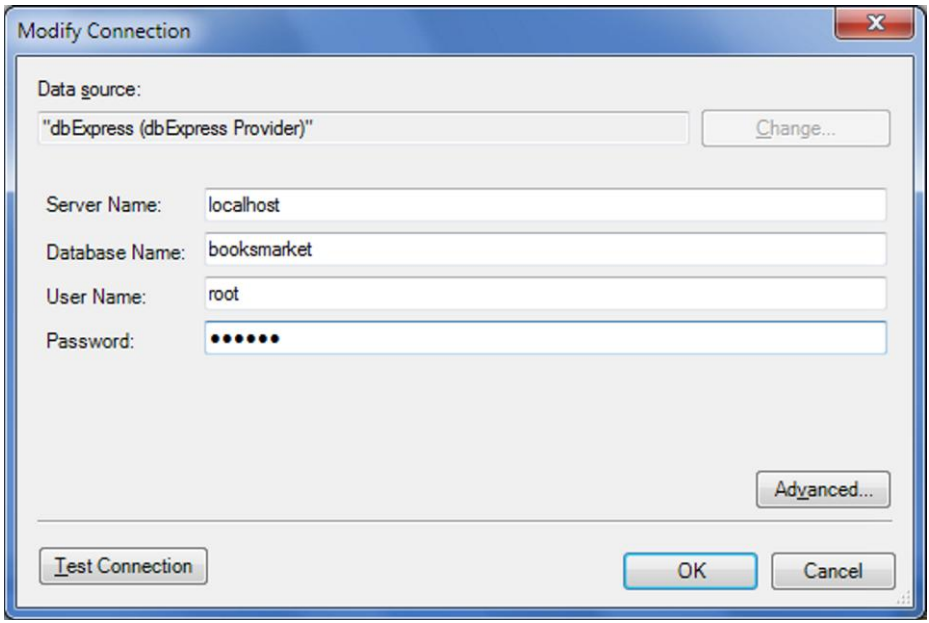


Рис. 27.5. Окно настройки соединения

Пример подключения

Компьютерная сеть современного развивающегося предприятия редко бывает статичной, она совершенствуется вместе с предприятием. В сеть вводят новые хосты и домены, предоставляют новые точки доступа, меняют оборудование и программное обеспечение. В таких условиях нельзя исключить изменений в настройках СУБД, например, переноса сервера баз данных на другой компьютер или перенастройки номера сетевого порта. Обстановка предъявляет повышенные требования к клиентским приложениям, нам нужно разрабатывать программное обеспечение, способное легко адаптироваться к изменившейся сетевой обстановке. Поэтому предложу вашему вниманию заготовку кода, демонстрирующую универсальный подход к процессу подключения к базе данных клиентского приложения dbExpress. В данном примере в качестве целевого сервера БД выбрана СУБД MySQL, однако это не догма и предложенный код можно взять за основу при работе с любыми СУБД.

Для реализации задуманного нам потребуется создать новый проект, в который помимо главной формы (`frmMain : TfrmMain`) войдут модуль данных (`frmDM : TDataModule`) и форма подключения к БД и регистрации пользователя (`frmUserLogin : TfrmUserLogin`). Сразу удалите модуль данных `frmDM` и форму регистрации `frmUserLogin` из перечня автоматически создаваемых форм проекта. Для этого выберите пункт меню **Project | Options** и в новом окне, на вкладке **Forms**, переместите названные формы в список **Available Forms**.

Дайте команду на сохранение проекта. В процессе сохранения изменим имена программных модулей: у главной формы `frmMain` программный модуль получит название main.pas; у формы регистрации `frmUserLogin` — `UserLogin.pas`; у модуля данных `frmDM` — `DM.pas`; головной файл проекта предлагаю назвать `BMforMySQL.dproj`.

В модуле данных `frmDM` расположите всего один компонент — главное действующее лицо этой главы — `SQLConnection1: TSQLConnection`. В форме регистрации `frmUserLogin` нам придется разместить больше элементов управления (рис. 27.6):

- комбинированный список `cbHostName: TComboBox`, предназначенный для хранения списка сетевых компьютеров;
- строку ввода `edPort: TEdit` для указания номера порта, по которому будет организовано сетевое соединение (по умолчанию MySQL настроен на порт с номером 3306);
- строку ввода `edDataBase:TEdit` для ввода названия БД;

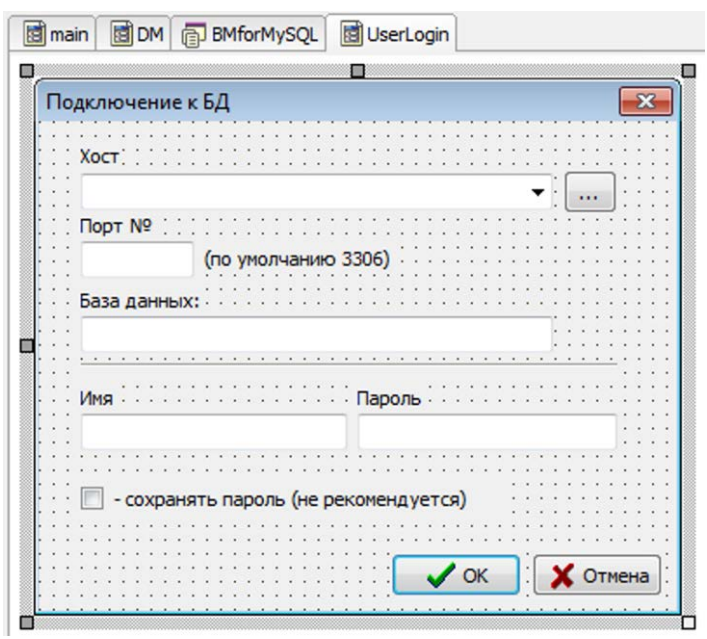


Рис. 27.6. Форма подключения к БД и регистрации пользователя

- строки ввода `edUserName: TEdit` и `edPassword: TEdit` для ввода имени и пароля пользователя;
- компонент-флажок `chbSavePassword: TCheckBox`, который будет позволять пользователю сохранять пароль;
- кнопки `btnOK: TBitBtn` и `btnCancel: TBitBtn`, свойству `Kind` кнопки `btnOK` присвойте значение `bkOk` — нажатие на эту кнопку свидетельствует о том, что пользователь ввел все данные и хочет осуществить соединение с БД.

В секции частных объявлений формы регистрации `frmUserLogin` объявите константу `section_name`:

```
private
const section_name='booksmarket_for_mysql';
```

Это название секции системного реестра, в котором я рассчитываю сохранить параметры соединения нашего проекта БД книжного магазина с СУБД MySQL.

Подключите к форме ссылку на модуль данных и модуль работы с реестром:

```
uses dm, registry;
```

Переходим к событию вывода формы на экран `OnShow()`. При его выполнении наше приложение постарается прочитать из системного реестра значения для инициализации свойств компонентов формы (листинг 27.3).

Листинг 27.3. Получение данных о параметрах соединения из системного реестра

```
procedure TfrmUserLogin.FormShow(Sender: TObject);
begin
  with TRegIniFile.Create('Software') do
  begin
   cbHostName.Text:= ReadString(section_name, 'hostname', 'localhost');
   edPort.Text:= ReadString(section_name, 'port', '3306');
   edDataBase.Text:= ReadString(section_name,'database', 'booksmarket');
   edUserName.Text:= ReadString(section_name, 'user','root');
   chbSavePassword.Checked:= ReadBool(section_name, 'savepass', false);
   edPassword.Text:= ReadString(section_name, 'password','');
  Free;
  end;
end;
```

Опишем событие закрытия формы `OnClose()`. Именно в момент прекращения работы формы параметры соединения сохраняются в системном реестре (листинг 27.4).

Листинг 27.4. Сохранение параметров соединения в системном реестре

```
procedure TfrmUserLogin.FormClose(Sender: TObject;
                                  var Action: TCloseAction);
```

```
begin
  with TRegIniFile.Create('Software') do
  begin
   WriteString(section_name, 'hostname', cbHostName.Text);
   WriteInteger(section_name,'port',StrToIntDef(edPort.Text, 3306));
   WriteString(section_name, 'database', edDataBase.Text);
   WriteString(section_name, 'user', edUserName.Text);
   WriteBool(section_name,   'savepass', chbSavePassword.Checked);
   if chbSavePassword.Checked then
      WriteString(section_name,'password', edPassword.Text)
    else
      WriteString(section_name,'password', '');
  Free;
  end;
end;
```

Выбрав имя хоста, номер порта, путь к файлу БД и введя регистрационные данные, пользователь нажимает кнопку `btnOK`. В ответ на это мы создаем модуль данных (листинг 27.5).

Листинг 27.5. Создание модуля данных

```
procedure TfrmUserLogin.btnOKClick(Sender: TObject);
begin
  Application.CreateForm(TfrmDM, frmDM);
end;
```

Переходим к модулю данных `frmDM`. В строке `uses` упомянем три модуля (`DBXCommon`, `Windows`, `UserLogin`) и опишем событие `OnCreate()`, возникающее в момент создания модуля. Код примера приведен в листинге 27.6.

Листинг 27.6. Событие создания модуля данных

```
procedure TfrmDM.DataModuleCreate(Sender: TObject);
begin
With SQLConnection1 do
begin
  LoginPrompt:=false;
  DriverName:='MYSQL';
  with Params do
  begin
   Clear;
   Add('hostname='+frmUserLogin.cbHostName.Text);
```

```
    Add('user_name='+frmUserLogin.edUserName.Text);
    Add('password='+frmUserLogin.edPassword.Text);
    Add('database='+frmUserLogin.edDataBase.Text);
    Add('port='+frmUserLogin.edPort.Text);
    Add('create=false');
  end;

  TRY
  Open; {предпринимаем попытку соединения}
   EXCEPT {Ошибка соединения}
       on err:TDBXError do
         MessageBoxEx(0,pWideChar(err.Message),
                      'Ошибка',MB_ICONEXCLAMATION,MB_OK);
       else raise;
   END;
end;
end;
```

В этом событии мы описываем параметры соединения и делаем попытку подключения к БД. Если попытка заканчивается неудачей — генерируется исключительная ситуация.

Чтобы завершить пример, осталось сделать самое главное — внести изменения в головной модуль проекта (листинг 27.7). Для вызова на экран листинга головного модуля воспользуйтесь пунктом меню **Project | View Source**.

Листинг 27.7. Исходный код головного модуля проекта

```
program BMforMySQL;

uses
  Forms, Windows,
  main in 'main.pas' {frmMain},
  DM in 'DM.pas' {frmDM: TDataModule},
  UserLogin in 'UserLogin.pas' {frmUserLogin};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  {самой первой создаем форму регистрации}
  frmUserLogin:=TfrmUserLogin.Create(application);
```

```
   if (frmUserLogin.ShowModal=idOK) and
        (frmDM.SQLConnection1.Connected=true) then
      begin {Если регистрация прошла успешно}
         frmUserLogin.Release;
         Application.CreateForm(TfrmMain, frmMain);{главная форма}
         {Здесь создаются другие формы проекта
          …}
         Application.Run; {старт приложения}
       end
       else
       begin {соединение с БД не осуществлено}
        if frmDM<>nil then frmDM.Free;
        frmUserLogin.Release;
        Application.Terminate;
       end;
end.
```

Главная идея приведенного кода заключается в том, что создание главной формы проекта (и всех остальных форм) происходит только при условии успешного соединения с требуемой БД, иначе работа приложения прекращается.

Управление подчиненными наборами данных

Компонент `TSQLConnection` способен влиять на подключенные к нему компоненты-наборы данных. Весь массив этих элементов управления хранится в свойстве

```
property DataSets[Index: Integer]: TCustomSQLDataSet;
```

Число компонентов, использующих соединение, мы выясним благодаря свойству

```
property DataSetCount: Integer;
```

Для закрытия всех открытых наборов данных достаточно вызвать процедуру

```
procedure CloseDataSets;
```

Управление транзакциями

Прежде чем попытаться управлять транзакциями, необходимо убедиться, что целевая СУБД их поддерживает. Для этого предназначено свойство

```
property TransactionsSupported: LongBool;
```

Значение `true` свидетельствует о том, что в один и тот же момент времени сервер способен выполнять, по крайней мере, одну транзакцию. Если же одновременно могут быть запущены несколько транзакций, то об этом просигнализирует свойство

```
property MultipleTransactionsSupported: LongBool;
```

Для старта новой транзакции обратитесь к процедуре

```
procedure StartTransaction(TransDesc: TTransactionDesc);
```

Процедура

```
procedure Commit(TransDesc: TTransactionDesc);
```

завершит транзакцию и сохранит результаты работы.

Откат транзакции осуществит процедура

```
procedure Rollback(TransDesc: TTransactionDesc);
```

В качестве параметра во все три метода передается одна и та же запись: `TTransactionDesc`.

Эта структура позволяет идентифицировать транзакцию и определять ее основные характеристики (листинги 27.8 и 27.9).

Листинг 27.8. Уровни изоляции транзакции

```
type TTransIsolationLevel = (xilDIRTYREAD,
                             xilREADCOMMITTED,
                             xilREPEATABLEREAD,
                             xilCUSTOM);
```

Листинг 27.9. Описание транзакции `TTransactionDesc`

```
type  TTransactionDesc = packed record
     TransactionID    : LongWord; //уникальный идентификатор транзакции
     GlobalID         : LongWord;//глобальный идентификатор для ORACLE
     IsolationLevel   : TTransIsolationLevel;//уровень изоляции
     CustomIsolation  : LongWord; //пользовательский уровень изоляции
end;
```

Система заполняет поля с идентификаторами транзакций сразу после старта транзакции, поэтому на долю программиста остается только настроить уровень изоляции.

Во время выполнения транзакции свойство

```
property InTransaction: Boolean;
```

переходит в состояние `true`.

Выполнение SQL-инструкций

При острой необходимости компонент `TSQLConnection` вполне способен самостоятельно отправить в адрес сервера команду SQL. В простейшем случае (когда команда не содержит параметров) воспользуемся методом

```
function ExecuteDirect(const SQL: string): Integer;
```

При корректном выполнении метода он возвратит нулевое значение. В противном случае функция передаст нам код ошибки. Если же SQL-команда включает параметры и, более того, возвращает какое-то результирующее множество, то выбираем одну из перегружаемых версий метода:

```
function Execute(const SQL: string, Params: TParams): Integer; overload;
function Execute(const SQL: string, Params: TParams,
                 var ResultSet: Object): Integer; overload;
```

Ограничение числа выполняющихся инструкций

Одновременно при одном соединении может выполняться ограниченное число инструкций (в том числе возвращаться открытых наборов данных). Информацией такого рода с нами может поделиться свойство

```
property MaxStmtsPerConn: LongWord; //только для чтения
```

Это ограничение определяется настройками сервера БД.

О том, сколько в данный момент времени выполняется инструкций, расскажет свойство

```
property ActiveStatements: LongWord;
```

Вполне естественно, что значение не может превысить максимальный предел — `MaxStmtsPerConn`. Поэтому для большей надежности перед отправкой очередной команды в адрес сервера не мешает опросить эти свойства и узнать, готова ли СУБД уделить нам пару-другую миллисекунд своего драгоценного времени (листинг 27.10).

Листинг 27.10. Проверка готовности СУБД к выполнению SQL-запроса

```
if SQLConnection1.ActiveStatements <= SQLConnection1.MaxStmtsPerConn
then SQLQuery1.Open else
MessageDlg('Сервер занят!'+#13+'Попробуйте повторить операцию позднее',
           mtWarning, [mbOK] , 0);
```

При достижении предельного значения компонент `TSQLConnection` способен открыть еще одно соединение с СУБД, но для этого свойство

```
property AutoClone: Boolean; //по умолчанию true
```

должно быть установлено в `true`. В противном случае обязанности по клонированию соединения ложатся на программиста, благо в его распоряжении имеется метод

```
function CloneConnection(): TSQLConnection;
```

Информирование о БД

Ряд методов компонента специализируется на получении информации об имеющихся в распоряжении базы данных объектах (таблицах, процедурах, индек-

сах и т. п.). Перечень этих процедур приведен в табл. 27.4. У некоторых из перечисленных в таблице процедур имеются перегружаемые версии, работающие в кодировке UNICODE.

Таблица 27.4. *Информационные методы компонента* `TSQLConnection`

Метод	Описание
procedure GetTableNames(List: TStrings; SystemTables: Boolean = False);	Метод предоставит список имеющихся в БД таблиц. Перечень таблиц будет передан в параметр `List`. Если список должен содержать информацию о системных таблицах, то установите в `true` второй параметр процедуры (см. свойство `TableScope`)
procedure GetFieldNames(**const** TableName: **String**; List: TStrings);	Процедура построит список полей таблицы `TableName` и направит его в параметр `List`
procedure GetIndexNames(**const** TableName: string; List: TStrings);	Список индексов таблицы `TableName` передается в параметр `List`
procedure GetProcedureNames(List: TStrings); overload;	Метод построит список хранимых процедур
procedure GetProcedureNames(const PackageName: string; List: TStrings); overload;	Только для СУБД Oracle. Процедура возвратит перечень хранимых процедур библиотеки с именем `PackageName`
procedure GetProcedureNames(const PackageName: string; const SchemaName: string; List: TStrings); overload;	
procedure GetPackageNames(List: TStrings);	Только для СУБД Oracle. Процедура вернет список библиотек для текущей БД
procedure GetProcedureParams(ProcedureName: string; List: TList); overload;	Получение информации о параметрах хранимой процедуры
procedure GetProcedureParams(ProcedureName: string; PackageName: string; List: TList); overload;	Получение информации о параметрах хранимой процедуры Oracle
procedure GetProcedureParams(ProcedureName: string; PackageName: string;SchemaName: string; List: TList); overload;	

Совместно с методом `GetTableNames()` трудится свойство

```
property TableScope: TTableScopes; //по умолчанию tsTable и tsView
```

```
type TableScope = (tsSynonym,
                   tsSysTable,{системные таблицы}
                   tsTable, {обычные таблицы}
                   tsView); {представления}
```

оно определяет тип таблиц, который может увидеть компонент.

Аутентификация пользователя в DataSnap

При построении трехуровневых приложений баз данных по технологии DataSnap сервер приложений dbExpress имеет возможность проверять подлинность клиента. Для этого необходимо соблюсти два условия:

- Сетевой обмен между сервером и клиентом должен осуществляться с помощью протокола HTTP.
- Сервер приложений должен использовать компонент `TDSHTTPServiceAuthenticationManager`, который и осуществляет проверку.

В таком случае пароль и имя пользователя клиентской программы должны передаваться при посредничестве свойств `DsAuthPassword` и `DSAuthUser`.

Мониторинг работы программы, *TSQLMonitor*

Зачастую доводка "сырой" версии приложения до кондиции занимает больше времени, чем собственно программирование. Эти слова подтвердит любой программист, проведший не одну ночь за отлавливанием "багов" в исходном коде. Однако какой бы ни был уровень профессионализма разработчика программного обеспечения, как бы тщательно не тестировалась программа, но после того, как она попадет в руки конечного пользователя, вновь появляется поток отказов. Причем интенсивность отказов и сбоев в работе приложения состоит в обратной зависимости от степени подготовленности оператора. Как устранять ошибки, если уровень пользователя таков, что он не в состоянии даже приблизительно описать, что он делал с приложением в момент сбоя?

Разработчики Delphi не хуже нас знают о проблемах программистов и всеми силами стараются помочь создавать эффективные и надежные программные продукты. Один из примеров — компонент `TSQLMonitor`, позволяющий осуществлять мониторинг информационного потока между клиентским приложением и сервером баз данных.

Компонент-монитор прост и удобен в работе. Он присоединяется к БД с помощью свойства

```
property SQLConnection: TSQLConnection;
```

Вся трассировочная информация накапливается в списке строк

```
property TraceList: TWideStrings;
```

Число хранимых сообщений можно ограничить, установив предельное значение в свойстве

```
property MaxTraceCount: Integer;//по умолчанию -1
```

По умолчанию все ограничения сняты (свойству присвоено значение –1).

Собранную в TraceList информацию целесообразно сохранять в текстовый файл, для этого в свойстве

```
property FileName: string;
```

указывается путь и имя файла с протоколом мониторинга. Допускается сразу сбрасывать протокол работы на станцию администратора сети, что позволит последнему оперативно вмешиваться в работу оператора.

Для того чтобы результаты мониторинга сохранялись автоматически, устанавливаем в true свойство

```
property AutoSave: Boolean;//по умолчанию false
```

Для запуска мониторинга обращаемся к свойству

```
property Active: Boolean;
```

Значение true активизирует компонент.

В простейшем случае для ведения протокола работы клиентского приложения dbExpress достаточно подключить SQL-монитор к соединению TSQLConnection и написать три строки кода (листинг 27.11).

Листинг 27.11. Организация автоматического SQL-мониторинга

```
SQLMonitor1.FileName:='work.log';
SQLMonitor1.AutoSave:=True;
SQLMonitor1.Active:=True;
```

Более тонкую настройку мониторинга можно произвести в коде обработки событий компонента. Событий у TSQLMonitor всего два.

Первое происходит перед тем, как сообщение попадет в список TraceList:

```
property OnTrace: TTraceEvent;

TTraceEvent = procedure(Sender: TObject;
                        TraceInfo: TDBXTraceInfo;
                        var LogTrace: Boolean) of object;
```

В этой процедуре можно перехватить и исследовать сообщение. Смысловое содержание окажется доступным благодаря параметру TraceInfo, это запись, в которой имеется четыре свойства (табл. 27.5). Последний параметр события LogTrace разрешает или запрещает регистрацию сообщения в свойстве TraceList.

***Таблица 27.5.** Свойства структуры TDBXTraceInfo*

Свойство	Описание
`property Message: UnicodeString;`	Текстовое сообщение
`property TraceFlag: TDBXTraceFlag;`	Флаги трассировки (None, Prepare, Execute, Error, Command, Connect, Transact, Blob, Misc, Vendor, Parameter , Reader, DriverLoad, MetaData, Driver, Custom) позволяют идентифицировать проводимые в данный момент операции

Таблица 27.5 *(окончание)*

Свойство	Описание
property TraceLevel: TDBXTraceLevel;	Уровень трассировки может принимать следующие значения: None, Debug, Info, Warn, Error, Fatal
property CustomCategory: UnicodeString;	Дополнительная классификация сообщения

Второе событие просто информационное:

```
property OnLogTrace: TTraceLogEvent;
TTraceLogEvent = procedure(Sender: TObject;
                           TraceInfo: TDBXTraceInfo) of object;
```

Инициируется только после помещения в свойство TraceList нового сообщения.

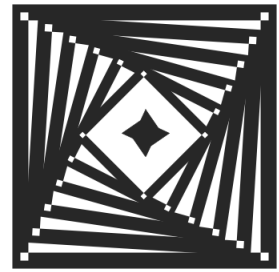
ВНИМАНИЕ!

Если вы планируете воспользоваться помощью события OnTrace() или OnLogTrace(), сделайте так, чтобы в строке uses модуля упоминались файлы DBXCommon и DBCommonTypes.

Резюме

Технология dbExpress — это универсальный механизм, позволяющий приложениям получать доступ к базам данных, работающим под управлением большинства современных клиент-серверных СУБД. Для обеспечения работоспособности клиентского приложения на клиентском компьютере в состав инсталляционного пакета необходимо включать библиотеки, выбор которых зависит от целевой СУБД.

Соединение с БД организуется с помощью компонента TSQLConnection. Помимо обеспечения соединения, названный компонент позволяет управлять транзакциями, взаимодействовать с подчиненными наборами данных и получать системную информацию об объектах БД.



ГЛАВА 28

Наборы данных dbExpress

В коллекции элементов управления, специализирующихся на обслуживании наборов данных dbExpress, мы обнаружим шесть компонентов (рис. 28.1). В первую очередь это классические для Delphi компоненты: таблица (`TSQLTable`), запрос (`TSQLQuery`) и хранимая процедура (`TSQLStoredProc`), еще универсальный элемент — `TSQLDataSet`, способный просматривать таблицы, выполнять запросы и обращаться к хранимым процедурам. Затем следует относительно новый компонент (`TSQLServerMethod`), специализирующийся на вызове методов сервера DataSnap. Все перечисленные классы описаны в модуле SqlExpr.

Картину дополняет компонент `TSimpleDataSet`, в отличие от своих коллег этот набор данных "проживает" в модуле SimpleDS. Появление `TSimpleDataSet` среди компонентов-наборов данных dbExpress объясняется тем, что все наследники `TCustomSQLDataSet` предназначены для обслуживания однонаправленных курсоров и не способны кэшировать данные. Простой набор данных несколько сглаживает остроту проблемы однонаправленных курсоров, поскольку специализируется на хранении записей в локальном буфере.

Ключевые методы и свойства всех компонентов dbExpress определяются в абстрактных классах `TDataSet`, `TWideDataSet` и `TCustomSQLDataSet`. С универсальным набором данных (олицетворяемым классом `TDataSet`) мы давно знакомы и про его беспрецедентный "родительский вклад" в общую идеологию построения дочерних компонентов нам известно практически все. Смысловая нагрузка, возложенная на `TWideDataSet`, невелика, на его уровне реализуется связь с интерфейсом `IProviderSupport2`.[1] Указанный класс введен сравнительно недавно (начиная с Delphi 2007), а в прежних версиях за `TDataSet` сразу следовал базовый класс наборов данных технологии dbExpress — `TCustomSQLDataSet`.

Замечание

Компонент `TSQLServerMethod` предназначен для вызова методов сервера DataSnap из клиентского приложения. Более подробно с этим компонентом мы познакомимся в *главе 31*.

[1] Напомню, что интерфейс `IProviderSupport` реализован на уровне `TDataSet`.

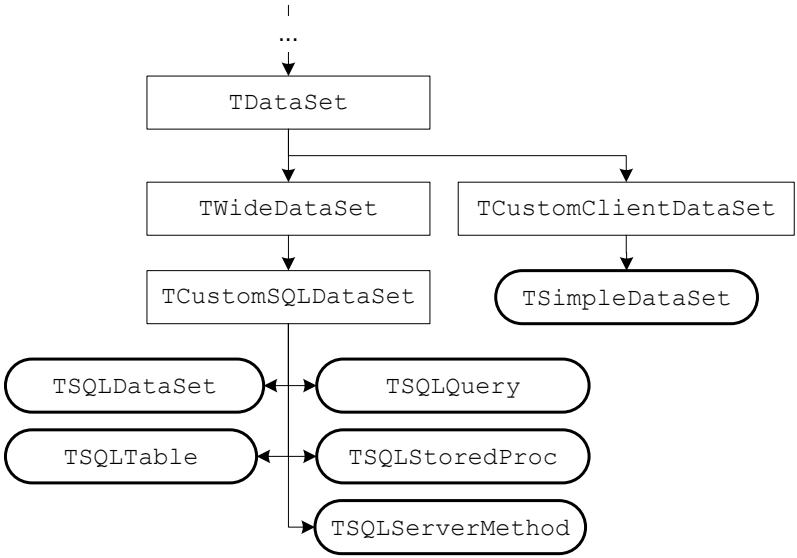


Рис. 28.1. Компоненты-наборы данных dbExpress

Базовый класс *TCustomSQLDataSet*

Разговор о `TCustomSQLDataSet` начнем с констатации факта, что в задачи базового класса для наборов dbExpress входит не столько развитие функционала, заложенного в родоначальнике всех наборов данных `TDataSet`, сколько его ограничение... Появление ярко выраженных черт "реакционера" у класса объясняется тем, что технология dbExpress недолюбливает механизм кэширования данных. Как следствие `TCustomSQLDataSet` проявил склонность к "ретроградству" и ввел множество ограничений:

- Однонаправленные курсоры существенно затрудняют навигацию по строкам отношения. Наборам данных dbExpress позволено двигаться только вперед, последовательно перебирая запись за записью. А если возникает необходимость вернуться к предыдущей строке, то приходится отправляться к первой строке отношения и вновь перебирать записи. Таким образом, для движения по набору данных у нас остается только два метода: `First()` и `Next()`.
- Из-за отсутствия механизма буферизации данных у компонентов dbExpress нет сервисных возможностей, нуждающихся в кэшировании данных, в первую очередь, фильтрации и поиска данных в полях подстановки (lookup fields).
- Модификация данных должна осуществляться только с помощью инструкций структурированного языка запросов `INSERT`, `UPDATE` и `DELETE`.
- С компонентами dbExpress не могут сотрудничать такие популярные в приложениях БД элементы управления, как сетки `TDBGrid` и `TDBCtrlGrid`; компоненты, работающие с подстановочными полями `TDBLookupListBox` и `TDBLookupComboBox`.

Оставшиеся после "секвестра" свойства и методы наборов данных dbExpress практически полностью повторяют свойства и методы своих коллег — компонентов dbGo и Interbase. Нам остается лишь заострить свое внимание на ряде особенностей.

Независимо от класса компонента работа с набором данных dbExpress начинается с подключения элемента управления к базе данных:

```
property SQLConnection: TSQLConnection;
```

В проектах dbExpress базу данных олицетворяет компонент `TSQLConnection` (о возможностях которого мы уже говорили в *главе 27*).

Формирование инструкций SQL

Вся система доступа к данным в dbExpress построена на основе SQL-запросов, поэтому с теоретической точки зрения, разработчикам технологии достаточно было создать один-единственный компонент, умеющий разговаривать на SQL. Это и есть упомянутый ранее `TSQLDataSet`. Вместе с тем, компания Embarcadero традиционно поддерживает линейку классических для Delphi и привычных для программистов компонентов — таблицу, запрос и хранимую процедуру. Все они общаются с СУБД командами SQL, описываемыми в свойстве

```
property CommandText: string;
```

Указанное свойство опубликовано лишь у одного компонента `TSQLDataSet`, остальные компоненты прячут его от программистов. Текст команды SQL находится в прямой зависимости от состояния свойства:

```
property CommandType: TSQLCommandType;
type TSQLCommandType = (ctQuery, ctTable, ctStoredProc);
```

Если тип команды `ctQuery`, то из этого следует, что в свойстве `CommandText` должна быть описана полная инструкция SQL, базирующаяся на ключевых словах `SELECT`, `INSERT`, `UPDATE`, `DELETE` и т. п. Если в свойстве `CommandType` установлено значение `ctStoredProc`, то в `CommandText` достаточно передать имя выполняемой хранимой процедуры. Последний вариант — `ctTable` означает, что в `CommandText` компонент рассчитывает обнаружить имя таблицы, с которой он и будет дальше работать.

Кроме свойства, определяющего тип команды, в классе имеется уточняющее свойство:

```
property DbxCommandType: UnicodeString;
```

которое заполняется типами команд после подключения к компоненту-соединению.

Свойство

```
property SortFieldNames: string;
```

позволяет задать порядок сортировки строк в результирующем наборе данных.

По содержимому свойства будет формироваться инструкция `ORDER BY` в запросе `SELECT`. Если сортировка осуществляется по двум или большему числу полей, то названия полей должны разделяться точкой с запятой.

Получение системной информации

В классе TCustomSQLDataSet заложен важный функционал, позволяющий потомкам класса получать от СУБД достаточно полную системную информацию о БД и ее основных объектах. Для того чтобы указать серверу БД, какая именно системная информация нам необходима, обращаемся к процедуре

```
procedure SetSchemaInfo(SchemaType: TSchemaType;
                        SchemaObjectName: string;
                        SchemaPattern: string;
                        PackageName: string = );
```

Здесь параметр SchemaType определяет тип запрашиваемых метаданных (табл. 28.1); SchemaObjectName — имя таблицы или хранимой процедуры в служебной информации, в которой мы нуждаемся; SchemaPattern — маска, фильтрующая результаты, она может включать трафаретные символы "%" (несколько символов) и "_" (один символ); PackageName — имя пакета Oracle.

Таблица 28.1. *Тип метаданных, элементы перечисления* TSchemaType

Значение	Описание
stNoSchema	Выход из режима сбора метаданных
stTables	Таблицы и представления
stSysTables	Системные таблицы
stProcedures	Хранимые процедуры
stColumns	Столбцы таблицы
stProcedureParams	Параметры процедуры
stIndexes	Индексы
stPackages	Пакеты БД; применяется только совместно с СУБД Oracle
stUserNames	Имена пользователей

Допустим, нам необходимо собрать информацию о таблицах и представлениях БД. Чтобы получить системную информацию такого рода, потребуется помощь любого потомка класса TCustomSQLDataSet и компонента, в котором мы отобразим полученные метаданные.

В моем примере на роль набора данных dbExpress был назначен компонент SQLDataSet1: TSQLDataSet, а собранные данные помещаются в список ListView1:TListView. Процесс получения метаданных стартует в момент вывода на экран формы frmSchema (листинг 28.1).

Листинг 28.1. Сбор информации о составе таблиц БД

```
procedure TfrmSchema.FormShow(Sender: TObject);
var i:Integer;
```

```
begin
ListView1.Items.BeginUpdate;
ListView1.Clear;
   ListView1.Columns.Clear;
ListView1.ViewStyle:=vsReport;
 with frmDM.SQLDataSet1 do
 begin
   SetSchemaInfo(stTables,'','','');
   Open;
   {получаем информацию о количестве полей в наборе данных
    и подготавливаем колонки ListView1 к приему данных}
   for i:=0 to Fields.Count - 1 do
      With ListView1.Columns.Add do
           begin
            AutoSize:=True;
            Caption:=Fields[i].DisplayLabel;
           end;
   If BOF=False then First; //переход к первой строке набора
   while EOF=false do {перебираем строки набора}
   begin
       with ListView1.Items.Add do {добавляем новый элемент TListItem}
         begin
          Caption:=Fields[0].AsString;
           for i:=0 to Fields.Count - 1 do
              if Fields[i].IsNull=false then
                 Subitems.Add(Fields[i].AsString);
         end;
    Next; //переход к очередной строке набора
   end;
   SetSchemaInfo(stNoSchema,'','','');  //завершаем работу с метаданными
 end;
ListView1.Items.EndUpdate;
end;
```

При желании можно улучшить пример, дополнив его возможностью сбора системной информации о таблицах БД. Для этого вам потребуется еще один список `ListView2:TListView`.

ВНИМАНИЕ!

Завершая сбор системной информации, необходимо перевести набор данных в режим обычной работы: вновь вызвать метод `SetSchemaInfo()`, но на этот раз с аргументом `stNoSchema`.

Пользователь выбирает одну из строк списка `ListView1` с именем интересующей его таблицы. В ответ на это генерируется событие `OnChange()` списка. В этом событии мы заполним второй список данными о составе полей (листинг 28.2).

Листинг 28.2. Сбор информации о полях таблицы

```
procedure TfrmSchema.ListView1Change(Sender: TObject;
                                     Item: TListItem; Change: TItemChange);
    var TableName:String;
        i:integer;
        cr:TCursor;
    begin
    if (Item=nil) then Exit; //если нет выбранного элемента — выход

    cr:=Screen.Cursor;
    Screen.Cursor:=crHourgLass;
    TableName:=Item.SubItems[1]; //здесь находится имя таблицы

    ListView2.Items.BeginUpdate;
    ListView2.Clear;
    ListView2.Columns.Clear;
    ListView2.ViewStyle:=vsReport;

     with frmDM.SQLDataSet1 do
     begin
       if Active then Close;
       SetSchemaInfo(stColumns,TableName,'','');
       Open;

        if ListView2.Columns.Count=0 then //если названия колонок пусты
            for i:=0 to Fields.Count - 1 do //собираем данные о колонках
                With ListView2.Columns.Add do
                      Caption:=Fields[i].DisplayLabel;

       If BOF=False then First; //переход к первой записи
       while eof=false do {перебираем строки набора}
       begin
           with ListView2.Items.Add do {добавляем новый элемент TListItem}
             begin
              Caption:=Fields[0].AsString;
               for i:=0 to Fields.Count - 1 do
                  if Fields[i].IsNull=false then
                     Subitems.Add(Fields[i].AsString);
             end;
        next; //переход к очередной записи
       end;

       SetSchemaInfo(stNoSchema,'','','');  //завершаем работу с метаданными
     end;
    ListView2.Items.EndUpdate;
    Screen.Cursor:=cr;
    end;
```

Приведенный код во многом повторяет предыдущий листинг. Наиболее важное изменение коснулось состава параметров метода `SetSchemaInfo()`. Для получения метаданных о таблице, выбранной пользователем, в первый параметр передается значение `stColumns`, а во второй — имя таблицы. Результаты нашей работы отражены на рис. 28.2, где приведена подробная информация о составе таблицы поставок `DELIVERIES` из БД "Книжный магазин", развернутой на сервере MySQL.

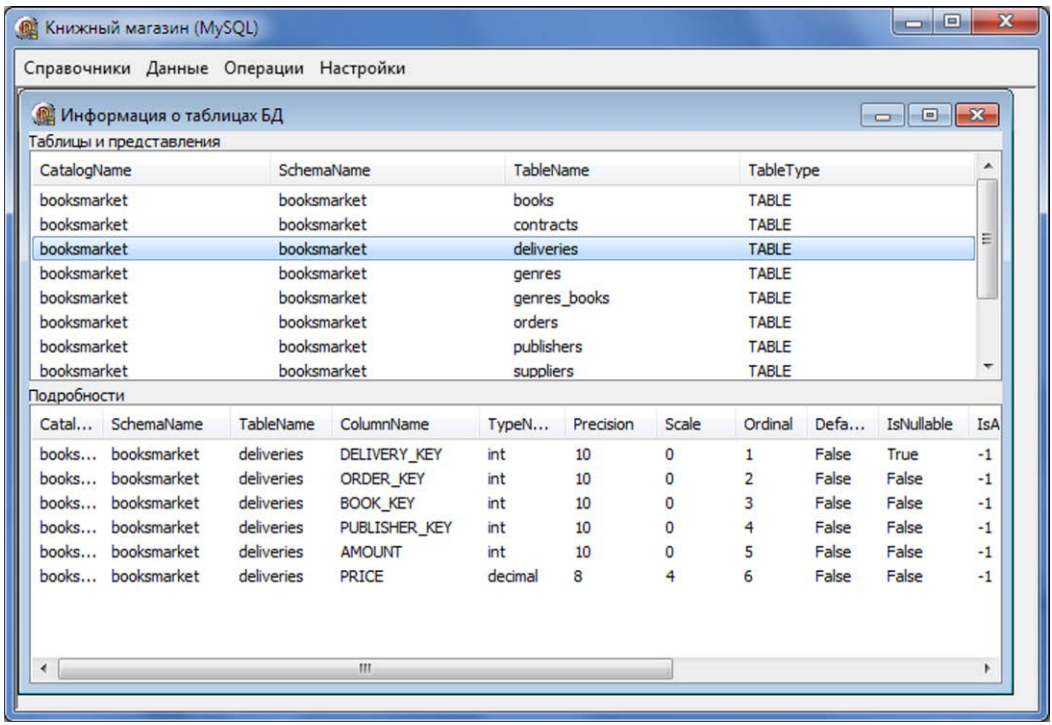


Рис. 28.2. Сбор системной информации о таблицах БД

Поступление в клиентское приложение системной информации далеко не всегда обусловлено явным обращением к функции `SetSchemaInfo()`. Без метаданных не обойтись и в обычном режиме работы. Отправляя в адрес клиентского приложения строки таблиц, сервер, кроме собственно данных, может передавать некоторый дополнительный объем служебной информации. Это пояснения к подключенным индексам, состав и доменные ограничения полей таблиц, перечень параметров хранимых процедур и другие важные характеристики объектов БД. Если вы твердо уверены, что указанная информация клиентскому приложению не нужна, то откажитесь от метаданных, отключив свойство

```
property GetMetadata: Boolean;
```

Однако отказ от потока "излишней" информации может сослужить плохую службу, т. к. без служебных данных компоненты dbExpress могут вообще утратить работоспособность.

Набор данных dbExpress, компонент *TSQLDataSet*

Абстрактному классу `TCustomSQLDataSet` наиболее близок по духу компонент `TSQLDataSet`. Фактически он объединяет в себе функциональные возможности триумвирата компонентов — таблицы, запроса и хранимой процедуры. Благодаря такой самодостаточности компонента `TSQLDataSet` необходимость использования в наших проектах `TSQLTable`, `TSQLQuery` и `TSQLStoredProc` при построении любого проекта dbExpress можно поставить под сомнение.

ВНИМАНИЕ!

Еще раз отметим ключевую особенность всех потомков класса `TCustomSQLDataSet` — однонаправленные наборы данных. Это означает, что компоненты не кэшируют записи в локальном буфере. Поэтому перебор записей в результирующем наборе может осуществляться только вперед, от первой к последней. Ко всему прочему, такие компоненты не в состоянии фильтровать полученную информацию, организовывать синхронный просмотр и поиск. Существуют ограничения по отображению полученных с помощью потомков `TCustomSQLDataSet` данных, в частности, для этого нельзя использовать такие популярные элементы управления, как `TDBGrid` и `TDBCtrlGrid`.

Работа с компонентом начинается с подключения его к БД с помощью свойства `SQLConnection`.

Наличие свойства `CommandText` позволяет нам применять `TSQLDataSet` в роли любого из перечисленных компонентов, щелчок по свойству в инспекторе объектов вызывает редактор запросов (свойство `CommandType=ctQuery`), который окажет помощь при проектировании простейших команд. Возможности редактора далеко не безграничны, но, по крайней мере, он позволит начинающему разработчику не перепутать названия таблиц и полей (рис. 28.3).

Если команда SQL основана на директиве `SELECT`, то для выполнения команды используют метод `Open()`, унаследованный от далекого предка `TDataSet`. Во всех остальных случаях вызываем метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Передав в параметр `ExecDirect` значение `true`, мы просигнализируем СУБД, что наша команда не нуждается в предварительной подготовке.

В остальном компонент не сильно отличается от его коллег со страничек dbGo и InterBase. Это подтверждает листинг 28.3, демонстрирующий вставку новой записи с помощью `TSQLDataSet`.

Листинг 28.3. Вставка новой записи в таблицу `ORDERS`

```
With SQLDataSet1 do
begin
 CommandType := ctQuery;
 ParamCheck:=true;
 CommandText:= 'INSERT INTO ORDERS (CONTRACT_KEY, ORDERNUM, ORDERDATE)
                          VALUES (:CONTRACT_KEY, :ORDERNUM, :ORDERDATE)';
```

```
  Params[0].AsInteger := 18;
  Params[1].AsInteger := 45;
  Params[1].AsString := '01/04/2011';
  ExecSQL;
end;
```

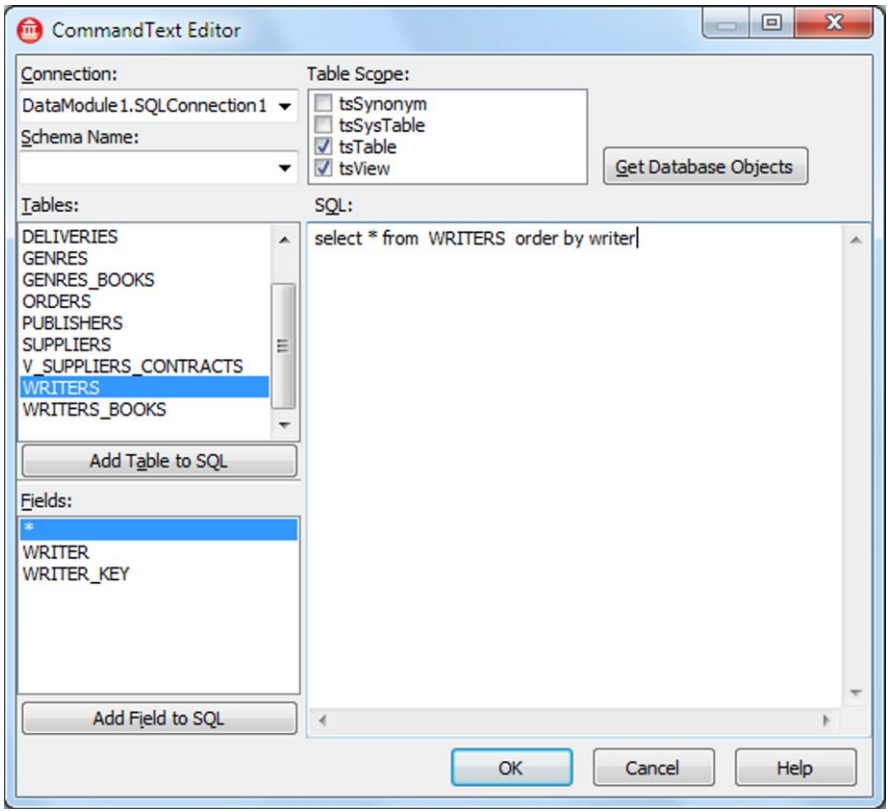


Рис. 28.3. Редактор свойства `CommandText`

В соответствии с классическими традициями наборов данных Delphi доступ к перечню параметров обеспечивает свойство

```
property Params: TParams;
```

Для того чтобы компонент самостоятельно регенерировал список параметров (имена, типы данных и т. д.), при вводе или любом изменении SQL-команды в поле `CommandText` убедитесь, что свойство

```
property ParamCheck: Boolean;
```

установлено в состояние `true`.

Внимание!

При работе с хранимыми процедурами старайтесь объявлять параметры в той же последовательности, в какой они описаны на серверной стороне.

Большинство событий компонента TSQLDataSet получены в наследство (правда, в весьма ограниченном виде) от класса TDataSet. Компонент способен реагировать на следующие события: подключение к таблице BeforeOpen() и AfterOpen(); отключение от таблицы BeforeClose() и AfterClose(); перемещение по записям BeforeScroll() и AfterScroll(); обновление данных BeforeRefresh() и AfterRefresh(); работа с вычисляемым полем OnCalcFields().

Кроме того, у компонента TSQLDataSet имеется четыре собственных обработчика событий. Все они предназначены для анализа инструкций SQL, отправляемых на выполнение. Исполнение команды INSERT можно отследить с помощью события

```
property ParseInsertSql: TParseInsertSqlEvent;
TParseInsertSqlEvent = procedure(var FieldNames: TWideStrings;
                                 SQL: UnicodeString;
                                 var BindAllFields: Boolean;
                                 var TableName: UnicodeString) of object;
```

Здесь FieldNames — имена полей, задействованных при выполнении инструкции, SQL — текст инструкции; BindAllFields — задействовать все поля; TableName — имя таблицы.

Команды SELECT, UPDATE и DELETE обслуживаются событиями:

```
property ParseSelectSql: TParseSqlEvent;
property ParseUpdateSql: TParseSqlEvent;
property ParseDeleteSql: TParseSqlEvent;
TParseSqlEvent = procedure(var FieldNames: TWideStrings;
            SQL: UnicodeString; var TableName: UnicodeString) of object;
```

Особенности обслуживания BLOB-полей

При обслуживании данных, хранящихся в полях BLOB, имеет смысл познакомиться со свойством

```
property MaxBlobSize: Integer; //по умолчанию 0
```

которое накладывает ограничение на максимальное число байтов, загружаемых из любого BLOB-поля БД в буфер памяти. По умолчанию свойство принимает нулевое значение, т. е. предельное значение настраивается в соответствии с установками соединения с СУБД. Если программист не планирует ограничивать максимальное значение обслуживаемых BLOB-данных, то он передаст в это свойство –1.

Таблица *TSQLTable*

К сожалению, по сравнению с компонентами-коллегами (TADOTable, TIBTable, TTable) функционал таблицы dbExpress сильно сокращен. По большому счету, компонент TSQLTable способен только получать данные из таблиц, все остальные сервисные возможности остаются за гранью возможностей таблицы.

Работа с компонентом начинается с подключения к соединению с помощью свойства SQLConnection.

Вторым шагом должен стать выбор целевой таблицы, ее имя передается в свойство

```
property TableName: string;
```

Компонент открывают стандартным способом: с помощью метода `Open()` или переводом свойства `Active` в состояние `true`.

Редактировать, добавлять и удалять запись компонент не способен. За одним маленьким исключением: метод

```
procedure DeleteRecords;
```

удалит все записи из таблицы.

Если у таблицы имеются индексы сортировки, то для их подключения можно воспользоваться свойствами

```
property IndexName: string;//имя индекса
property IndexFieldNames: string;//имена полей индекса
```

Если между двумя таблицами имеется отношение "главная таблица — подчиненная таблица", то для настройки подчиненного компонента пригодятся свойства

```
property MasterSource: TDataSource;//главный источник данных
property MasterFields: string; //поле для связи
```

При разработке пользовательского интерфейса приложения нужно учитывать, что компонент работает с однонаправленными курсорами, поэтому таблица dbExpress не способна функционировать совместно с сеткой `TDBGrid`. Поэтому для отображения данных из таблицы приходится идти на некоторые хитрости. Допустим, что доступ к таблице поставщиков обеспечивает компонент `tblSuppliers:TSQLTable`, тогда, чтобы собрать все записи из таблицы можно воспользоваться листингом 28.4.

Листинг 28.4. Получение данных из таблицы `TSQLTable`

```
ListBox1.Clear;
  with frmDM.tblSUPPLIERS do
    begin
    If Active=false then Open;
      First;
      while eof<>true do
      begin
        ListBox1.Items.AddObject(FieldByName('SUPPLIER').AsString,
                  TObject(FieldByName('SUPPLIER_KEY').AsInteger));
        Next;
      end;
      close;
    end;
```

В предложенном вашему вниманию примере мы получаем все данные из таблицы поставщиков и сохраняем их в компоненте-списке `ListVox1:TListBox`.

Запрос *TSQLQuery*

Компонент-запрос специализируется на отправке серверу инструкции на языке SQL. Работа с компонентом не сильно отличается от действий с однотипными компонентами из других технологий доступа к данным. После подключения к БД с помощью свойства `SQLConnection` следует определить текст запроса

```
property SQL: TWideStrings;
```

Запрос можно редактировать как во время проектирования БД, так и при выполнении программы. Если запрос содержит параметры, то доступ к ним осуществляется с помощью свойства

```
property Params:TParams;
```

Для того чтобы параметры компонента самостоятельно подстраивались под все изменения в тексте запроса, убедитесь, что свойство

```
property ParamCheck;
```

установлено в `true`.

Если запрос основан на инструкции `SELECT` и предназначен для вывода набора записей, то для получения строк используем метод `Open()` так, как предложено в листинге 28.5.

Листинг 28.5. Построение запроса в компоненте `TSQLQuery`

```
With SQLQuery1 do
Begin
 SQL.Clear;
 SQL.Add('SELECT * FROM BOOKS ORDER BY BOOK');
 SQL.Open;
End;
```

Если запрос работает с инструкциями `INSERT`, `UPDATE` и `DELETE`, то для отправки SQL-запроса к серверу применяют метод `ExecSQL()`.

Хранимая процедура *TSQLStoredProc*

Хранимая процедура `TSQLStoredProc` специализируется на вызове процедур, заранее подготовленных на стороне сервера. При подготовке к работе компонент следует соединить с базой данных, поэтому в процессе создания проекта самым первым шагом программиста должно быть подключение к `TSQLConnection` с помощью свойства `SQLConnection`.

Имя хранимой процедуры передается в свойство

```
property StoredProcName: string;
```

Если свойство `ParamCheck` компонента находится в состоянии `true`, то при изменении имени процедуры компонент самостоятельно обращается к базе данных и собирает сведения о параметрах процедуры и заносит их в свойство `Params`.

При работе с СУБД Oracle потребуется свойство

```
property PackageName: string;
```

Это имя пакета, в котором сервер хранит интересующую нас процедуру.

Для принудительной предварительной подготовки процедуры к выполнению на стороне сервера можно вызвать функцию

```
function PrepareStatement(
                        var RecordsAffected: Integer): TCustomSQLDataSet;
```

Впрочем, разработчики Delphi XE рекомендуют обращаться к свойству `Prepared`, гарантирующему, что сервер не станет без всякой надобности перекомпилировать уже давно готовую к работе процедуру.

Если процедура не основана на инструкции `SELECT` (не возвращает результирующего набора данных или возвращает результаты в виде выходных параметров), то для ее вызова обращаемся к методу

```
function ExecProc: Integer;
```

В противном случае нужно использовать метод `Open()` либо установить в `true` свойство `Active`.

Одной из существенных особенностей технологии dbExpress является то, что построенная на основе инструкции `SELECT` хранимая процедура может возвращать несколько наборов данных. В этом случае записи поступают по порциям — в виде нескольких подмножеств `TCustomSQLDataSet`. Для доступа к очередной порции данных пригодится метод

```
function NextRecordSet: TCustomSQLDataSet;
```

Обращение к методу вынуждает хранимую процедуру передать во временный набор данных небольшое подмножество строк, повторный вызов передаст следующую порцию и так далее, пока не будет достигнут конец набора. Как только данные закончатся, метод возвратит `nil`.

Листинг 28.6 демонстрирует порядок получения данных с помощью `NextRecordSet()` из БД, развернутой на базе Microsoft SQL Server.

Листинг 28.6. Получение набора данных с помощью `TSQLStoredProc`

```
var DS: TCustomSQLDataSet;
begin
If SQLStoredProc1.Active then SQLStoredProc1.close;
SQLStoredProc1.ParamByName('@ParamName').AsString:=Edit1.Text;
SQLStoredProc1.Open;
  DS:= SQLStoredProc1; {забираем первое подмножество данных}
  while DS <> nil do {цикл обращения к очередному подмножеству записей}
  begin
    DS.First; {переход к первой записи}
    while DS.Eof=false do {цикл перебора строк в подмножестве}
    begin
      {… операции с записью}
```

```
      DS.Next;{переход к следующей записи}
    end;
    if DS <> SQLStoredProc1
        then DS.Free; {все строки в подмножестве обслужены}
    DS := SQLStoredProc1.NextRecordSet; {забираем следующее подмножество}
  end;
SQLStoredProc1.Close; {завершаем работу с хранимой процедурой}
end;
```

ЗАМЕЧАНИЕ

Одна из неприятных сторон технологии dbExpress — отсутствие кэширования запрашиваемых данных на клиентской стороне. В связи с этим получение полного результирующего набора записей осуществляется поэтапно (небольшими порциями). Для обращения к очередному подмножеству данных требуется вызов метода `NextRecordSet()`.

Простой набор данных *TSimpleDataSet*

Несмотря на наличие в имени класса слова "simple", что в переводе с английского означает "простой", компонент `TSimpleDataSet` не такой уж и простак. Чтобы убедиться в этом, найдите объявление класса `TSimpleDataSet` в модуле SimpleDS. Изучение первых строк описания компонента (листинг 28.7) позволит нам сделать несколько важных выводов.

Листинг 28.7. Фрагмент листинга объявления набора данных `TSimpleDataSet`

```
type TSimpleDataSet = class(TCustomClientDataSet)
  private
    FConnection: TSQLConnection;    //внешний объект соединения
    FInternalConnection: TSQLConnection; //внутренний объект соединения
    FDataSet: TInternalSQLDataSet; //встроенный набор данных dbExpress
    FProvider: TDataSetProvider;    //объект-провайдер
  Protected
…
```

Во-первых, наш новый знакомый является прямым потомком класса `TCustomClientDataSet` (который был подробно рассмотрен в *главе 21*). Другими словами, компонент представляет собой полноценный клиентский набор данных, способный кэшировать записи, полученные из таблиц БД. В свою очередь, это означает, что с помощью `TSimpleDataSet` можно снизить остроту основной проблемы технологии dbExpress и начать работать на клиентской стороне с однонаправленным курсором так, будто это полноценный двунаправленный курсор.

Во-вторых, компонент вооружен двумя полями типа `TSQLConnection`, что позволяет ему подключаться к БД двумя способами. Обычный способ заключается в разделении с другими компонентами `TSQL…` одного общего соединения `TSQLConnection`. Альтернативный способ позволяет подключаться к БД абсолютно самостоятельно, с помощью своего собственного внутреннего экземпляра `TSQLConnection`. Вторым способом без особой необходимости злоупотреблять не следует, поскольку все серверы баз данных обычно ограничивают число допустимых соединений, ведь чем больше открытых соединений, тем дороже оказывается лицензия на программное обеспечение СУБД.

В-третьих, наличие поля `TDataSetProvider` говорит о том, что компонент инкапсулирует возможности провайдера для получения и передачи данных в интересах тонких клиентов многоуровневых приложениях баз данных.

Наконец, в-четвертых, поле `TInternalSQLDataSet` обеспечивает полную совместимость с технологией доступа к данным dbExpress.

Замечание

Компонент `TSimpleDataSet` призван решать проблему кэширования данных на клиентской стороне. Объединив в себе лучшие черты классов `TCustomClientDataSet` и `TSQLDataSet`, компонент устраняет большинство недостатков наборов данных dbExpress.

Порядок работы с компонентом таков. С помощью свойства `Connection` присоединяем компонент к БД. Затем обращаемся к встроенному объекту — набору данных `TInternalSQLDataSet`:

```
property DataSet: TInternalSQLDataSet;
```

Свойство `DataSet` предоставляет доступ к рассмотренным в начале главы свойствам `CommandType` и `CommandText` встроенного набора данных. Благодаря им мы определяем порядок обращения к таблицам, представлениям или хранимым процедурам БД.

Продемонстрируем особенности работы с простым набором данных, дополнив начатый ранее проект демонстрационной базы данных для сервера MySQL справочником издательств. Для этого разместите в модуле данных (`frmDM:TDataModule`) два новых компонента:

- Простой набор данных `TSimpleDataSet`. Переименуйте компонент в `sdsPublishers`.
- Источник данных `TDataSource`. Переименуйте компонент в `dsPublishers`.

Теперь добавьте к проекту новую дочернюю форму (я ее назову `frmPublishers`) и разместите на ней сетку `TDBGrid`. Найдите событие `OnShow()` формы издательств и внесите в него строки из листинга 28.8.

Листинг 28.8. Пример использования `TSimpleDataSet` в проектах dbExpress

```
uses DM,DB;
…
procedure TfrmPublishers.FormShow(Sender: TObject);
```

```
begin
With frmDM.sdsPublishers do
begin
  Close;
  Connection:=frmDM.SQLConnection1;
  DataSet.CommandType:=ctTable;
  DataSet.CommandText:='PUBLISHERS';
  Open;
end;
  frmDM.dsPublishers.DataSet:=frmDM.sdsPublishers;
  DBGrid1.DataSource:=frmDM.dsPublishers;
end;
```

За счет способности компонента `TSimpleDataSet` буферизовать данные мы приобрели не только возможность отображать содержимое таблицы в сетке `TDBGrid`, но и теперь сможем редактировать строки в кэше компонента и отправлять их в БД. Как это делать, мы уже обсуждали в *главе 21*.

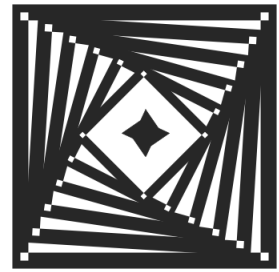
Резюме

Библиотека dbExpress — основное средство построения проектов БД Delphi XE, специализирующихся на быстром просмотре информации на серверах SQL. Самая важная особенность dbExpress — отсутствие кэширования данных на стороне клиента.

Отрицательная сторона однонаправленных курсоров — существенные сервисные ограничения наборов данных dbExpress (сложность фильтрации, поиска, невозможность работать с сетками `TDBGrid` и `TCtrlGrid`). Решение задач, связанных с редактированием данных, разработка сложных клиент-серверных или многоуровневых приложений, обеспечивающих серьезную работу с данными с помощью dbExpress, требует от программиста больше временных затрат, чем например, в InterBase.

Достоинство технологии — простота реализации и низкая ресурсоемкость приложений.

Основным союзником компонентов dbExpress является простой набор данных `TSimpleDataSet`, благодаря которому существенно снижается острота проблемы односторонних курсоров и упрощается проектирование клиентских приложений БД.



Глава 29

Интерфейс приложения и компоненты Data Access

Более десятка визуальных элементов управления среды программирования Delphi предназначены для решения вопросов отображения содержимого отдельных полей или коллекции строк из набора данных. Многие из них обучены редактировать данные, а кое-кто даже способен управлять процессом ввода/удаления данных из таблиц. Почти все рассматриваемые в этой главе компоненты размещены на странице **Data Controls** палитры компонентов и называются *элементами управления для работы с данными* (Data-aware Components, DAC), за исключением источника данных `TDataSource`, который находится на страничке **Data Access** и играет роль невидимого посредника между компонентами-наборами данных и компонентами DAC.

Чтобы с самых первых строк порадовать читателя, напомню, что благодаря компонентам DAC, мы, не написав ни одной строки кода, сможем строить несложные приложения баз данных. Кто об этом забыл, вернитесь к первым страницам *главы 17*.

Источник данных — компонент *TDataSource*

Невизуальный элемент управления `TDataSource` является потомком класса `TComponent`. Компонент — "минималист", у него всего четыре опубликованных свойства, пара методов и три обработчика событий. Но, несмотря на такую скромность, источник данных справляется с двумя важными задачами.

Во-первых, компонент `TDataSource` незаменим при организации взаимодействия между компонентом доступа к набору данных (элементы `TADOTable`, `TSQLQuery`, `TIBStoredProc` и т. п.) и элементами управления данными (`TDBNavigator`, `TDBEdit`, `TDBGrid`). Источник берет на себя ответственность за передачу информации в компоненты отображения данных и возвращает назад осуществленные пользователем изменения (рис. 29.1). Благодаря такой исключительной

способности вопрос отображения и редактирования данных, связанных с `TDataSource`, становится настолько тривиальным, что иногда ловишь себя на мысли, что разрабатывать приложения БД становится скучновато. Но заранее не расслабляйтесь — назло Delphi мы все равно найдем себе трудности.

Во-вторых, компонент `DataSource` помогает организовать связь между главным и подчиненным наборами данных. Этот вопрос мы уже обсуждали в *главе 23* (см. листинг 23.11), тогда мы научились устанавливать отношение "один ко многим" между двумя таблицами.

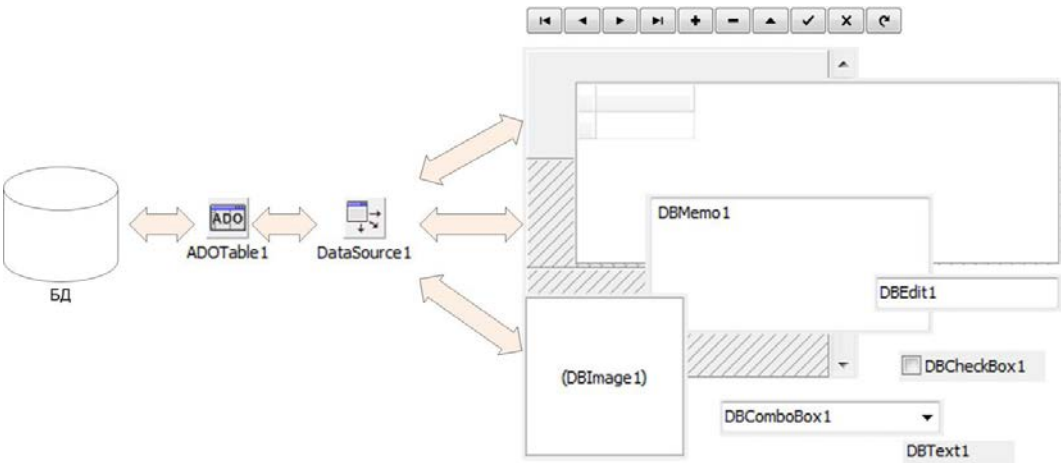


Рис. 29.1. Место компонента `TDataSource` в приложении БД

Ключевым свойством компонента по праву считается

```
property DataSet: TDataSet;
```

благодаря которому и поддерживается связь с набором данных.

Свойство доступно как в период разработки приложения, так и во время выполнения. Во время разработки инспектор объектов предоставит программисту список из всех доступных в приложении потомков `TDataSet` (таблиц, запросов, хранимых процедур), нам лишь останется выбрать нужный из них.

Для проверки, связан ли в текущий момент источник данных с набором `DataSet`, предназначена функция

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

Контроль состояния набора данных, к которому в данный момент присоединен `DataSource`, осуществляет с помощью свойства

```
property State: TDataSetState; //только для чтения
```

Это свойство нам уже встречалось в *главе 18*.

Оставшиеся два свойства имеют прямое отношение к элементам отображения данных. Первое приспособлено для их централизованного включения/отключения:

```
property Enabled: Boolean;
```

Второе свойство (в состоянии `true`) обеспечивает автоматический перевод в режим редактирования связанного с `DataSource` компонента-набора данных:

```
property AutoEdit: Boolean;
```

Такой переход осуществляется при внесении пользователем изменений в значения полей в элементах отображения данных. Еще один способ перехода в редактирование сводится к вызову метода

```
procedure Edit;
```

Нелегкая судьба посредника между данными и элементами управления данными обрекает компонент `TDataSource` на вечные хлопоты, это в полной мере отражается в характере обрабатываемых событий. При любом изменении в данных (например, при переходе от одной записи к другой или в момент модификации записи) вызывается обработчик события:

```
property OnDataChange: TDataChangeEvent;
type TDataChangeEvent = procedure(Sender: TObject;
                                  Field: TField) of object;
```

Здесь параметр `Field` указывает, в каком из полей произошли изменения. Но если одновременно модифицировалось два и более поля, параметр окажется пустым — `nil`.

При каждом изменении состояния связанного набора данных происходит событие

```
property OnStateChange: TNotifyEvent;
```

Перед передачей в базу модификаций, сделанных пользователем, вызывается обработчик

```
property OnUpdateData: TNotifyEvent;
```

который может использоваться для проверки правильности данных перед их сохранением.

Общие черты компонентов отображения данных

Практически все компоненты отображения данных построены на базе стандартных элементов управления и являются их логическим развитием для работы с данными, хранящимися в реляционных таблицах. Вполне естественно, что для обеспечения работы с БД компоненты приобрели новые черты, к которым, в первую очередь, относится технология соединения с набором данных. Следуя традициям объектно-ориентированного программирования, механизм соединения спрятан от нас в исходных кодах модулей `DB`, `DBCtrls`, `DBGrids` и т. д. На нашу долю остается совсем немногое, разве что "потереть кувшин с джинном" и указать, к какому набору данных мы планируем подключиться. Соединение реализуется при прямом посредничестве элемента управления `TDataSource`. Поэтому первой и наиболее существенной объединяющей чертой всех элементов управления, предназначенных для работы с данными, является не что иное, как источник данных. Для

подключения к источнику потребуется передать имя необходимого компонента `TDataSource` в свойство

```
property DataSource: TDataSource;
```

Компоненты отображения данных, специализирующиеся на предоставлении доступа к отдельному полю набора данных (`TDBText`, `TDBEdit`, `TDBMemo` и т. д.), для подключения к столбцу потребуют передать его имя в свойство

```
property DataField: String;
```

Если соединение осуществилось без проблем, то в свойстве

```
property Field: TField; //только для чтения
```

мы обнаружим ссылку на соответствующий обслуживаемому столбцу таблицы экземпляр объекта-поля `TField`.

Все компоненты, способные не просто отображать, но еще и редактировать данные, переводятся в режим только для чтения хрестоматийным свойством

```
property ReadOnly: Boolean;
```

На этом разговор, посвященный общим чертам представителей линейки **Data Controls**, закончен, перейдем к обсуждению индивидуальных особенностей каждого элемента управления.

Сетка базы данных — компонент *TDBGrid*

Компонент `TDBGrid` — наверное, один из наиболее востребованных элементов управления. Его основное преимущество над всеми своими коллегами заключается в том, что сетка способна вывести на экран сразу несколько строк и столбцов набора данных, а все остальные компоненты (за исключением `TDBCtrlGrid`) ориентированы на работу только с текущей записью и одним-единственным полем. Родословная `TDBGrid` берет начало от обычной сетки (класс `TCustomGrid`) . А новшества, связанные с работой с базами данных, введены в следующем по иерархии наследования классе — `TCustomDBGrid`. Непосредственно класс `TDBGrid` просто опубликовал методы и свойства своих предшественников, а нового ничего не вносил.

Каждая подключенная к активному набору данных сетка знает, содержимое каких именно полей ей доверено обслуживать. Этой информацией сетка способна поделиться и с нами. Свойство

```
property Fields[Index: Integer]: TField; //только для чтения
```

предоставляет доступ ко всем объектам `TField` набора данных, к которому присоединена сетка. Общее число полей доступно благодаря свойству

```
property FieldCount: Integer; //только для чтения
```

Ссылку на поле, соответствующее выделенной в сетке ячейке, мы обнаружим в свойстве

```
property SelectedField: TField;
```

Индекс выбранной колонки находится в свойстве

```
property SelectedIndex: Integer;
```

Обращаю внимание, что свойства SelectedField и SelectedIndex не только информационные, но и доступны для записи. С их помощью вы получаете возможность выбирать требуемую колонку из текста программы.

Некоторые особенности поведения сетки определяются свойством

```
property Options: TDBGridOptions;
TDBGridOptions = set of TDBGridOption;
```

Описание опций приведено в табл. 29.1.

Таблица 29.1. *Опции сетки* `TDBGridOption`

Значение	Описание
dgEditing	Сетка допускает редактирование своего содержимого, но при условии, что в состав опций не входит значение dgRowSelect
dgAlwaysShowEditor	Ячейки сетки постоянно готовы к редактированию своих данных, пользователю даже нет необходимости нажимать клавиши <Enter> или <F2>
dgTitles	Самая верхняя строка сетки отводится для показа заголовков колонок
dgIndicator	Напротив текущей записи появляется указатель
dgColumnResize	Разрешается перестановка местами и изменение ширины колонок
dgColLines	Колонки сетки разделяются линией
dgRowLines	Строки сетки разделяются линией
dgTabs	Нажатие клавиш <Tab> и <Shift>+<Tab> позволяет перемещаться между ячейками сетки
dgRowSelect	Пользователь теряет возможность выбрать отдельную ячейку сетки. Вместо этого в сетке выделяется вся строка целиком. Включение в состав опций значения dgRowSelect запрещает редактирование данных, отображаемых в сетке
dgAlwaysShowSelection	Сетка продолжает подкрашивать другим цветом выбранные пользователем ячейки, даже если она сама потеряла фокус ввода
dgConfirmDelete	Перед удалением строки из сетки (пользователь нажал клавиши <Ctrl>+<Delete>) вызывается диалоговое окно, предлагающее подтвердить эту операцию
dgCancelOnExit	Автоматически отменяет несохраненные изменения в данных при потере сеткой фокуса ввода
dgMultiSelect	Допускается одновременный выбор более одной строки сетки
dgTitleClick	Сетка реагирует на щелчок по заголовкам колонок
dgTitleHotTrack	При появлении указателя мыши над заголовком колонки, заголовок подсвечивается

Одновременный выбор нескольких строк

Если в опциях сетки включены флаги dgMultiSelect и dgRowSelect, разрешающие одновременный выбор двух и более строк, то массив отмеченных пользователем строк окажется в свойстве компонента TDBGrid:

```
property SelectedRows: TBookmarkList; //только для чтения
```

Свойство предоставляет доступ к специальному объекту, экземпляру класса TBookmarkList, хранящему список закладок (с закладками мы уже встречались во время изучения класса TDataSet).

Общее число строк, выбранных пользователем в сетке, окажется в свойстве

```
Property Count : integer;
```

Доступ к отдельному элементу списка закладок обеспечивается благодаря свойству

```
property Items[Index: Integer]: TBookmark;
```

Параметр Index определяет порядковый номер строки в массиве.

Обратную задачу (по содержимому закладки выяснить ее индекс в массиве) решает метод

```
function IndexOf(const Item: TBookmark): Integer;
```

Если в списке закладок элемент Item обнаружен не был, то функция вернет значение -1. Аналогичного результата можно достичь, обратившись к методу

```
function Find(const Item: TBookmark; var Index: Integer): Boolean;
```

Функция проверяет, входит ли закладка Item в список, если закладка обнаружена, то метод порадует нас значением true. Номер закладки в массиве записывается в формальный параметр Index.

Для проверки, входит ли текущая запись в перечень отмеченных пользователем строк, следует обратиться к свойству списка закладок

```
property CurrentRowSelected: Boolean;
```

Если строка выделена, то свойство вернет true, в обратном случае false. Поле CurrentRowSelected несет не только информационную нагрузку, с помощью этого свойства мы можем помечать строки программным образом.

Обновление списка закладок с удалением поврежденных элементов (например, когда запись физически удалена из набора данных, а ссылающаяся на нее закладка все еще "болтается" в памяти) реализуется методом

```
function Refresh: Boolean;
```

Очистку всего списка закладок осуществит метод

```
procedure Clear;
```

В результате вызова процедуры со всех строк сетки снимается признак выделения и инициируется процедура перерисовки ее клиентской области.

И, наконец, последний и самый опасный метод класса TBookmarkList. Он обеспечивает физическое удаление выбранных в сетке записей из связанного с ней набора данных TDataSet:

```
procedure Delete;
```

Колонки сетки

По умолчанию в момент подключения компонента TDBGrid к активному набору данных в сетке создаются колонки. Количество колонок определяется числом доступных полей набора данных — каждому полю сопоставляется своя колонка. Порядок следования колонок в точности соответствует порядку столбцов, определенных в наборе данных. Если колонки создавались автоматически, то их называют *динамическими*. Вместе с тем, существует и другой способ создания колонок. Программист обладает правом определить перечень требуемых ему колонок, как во время проектирования (для этого в Delphi реализован специальный редактор), так и непосредственно в коде приложения. Колонки, созданные "вручную", именуют *статическими*.

Доступ ко всем колонкам компонента TDBGrid предоставляется с помощью свойства

```
property Columns: TDBGridColumns;
```

которое направит нас к коллекции колонок элемента управления.

Коллекция колонок — класс *TDBGridColumns*

Класс TDBGridColumns называют коллекцией колонок. поскольку он является непосредственным наследником коллекции TCollection — специализированного класса, ориентированного на хранение объектов различного типа.

Общее число колонок, находящихся в контейнере, узнается благодаря свойству

```
property Count: Integer; //только для чтения
```

Доступ к отдельному элементу коллекции реализуется уже знакомым нам способом. Необходимо указать порядковый номер элемента в списке:

```
property Items[Index: Integer]: TColumn default;
```

Возможные значения Index ограничены диапазоном от 0 до Count-1. Приемлемы два способа вызова колонки (с обращением и без обращения к свойству items), их иллюстрирует листинг 29.1.

Листинг 29.1. Доступ к экземпляру колонки

```
var Clmn : TColumn;
begin
Clmn := DBGrid1.Columns.Items[0]; //обращаемся к первой колонке
Clmn := DBGrid1.Columns[DBGrid1.Columns.Count-1];{обращаемся к последней колонке}
```

Как уже упоминалось ранее, колонки могут создаваться автоматически системой, так и руками разработчика. Для выяснения способа создания предназначено специальное свойство коллекции колонок:

```
type TDBGridColumnsState = (csDefault, csCustomized);
property State: TDBGridColumnsState;
```

Значение csDefault свидетельствует о том, что коллекция заполнилась динамически, csCustomized — здесь поработал программист.

Ссылка на компонент-сетку, которой принадлежат колонки, находится в свойстве

```
property Grid: TCustomDBGrid; //только для чтения
```

Методы класса TDBGridColumns, в первую очередь, нацелены на манипуляции с колонками, содержащимися в коллекции. Создание новой колонки обеспечивается функцией

```
function Add: TColumn;
```

Новый экземпляр TColumn добавляется в конец списка.

Два метода решают вопросы сохранения информации об атрибутах колонок в файл или память:

```
procedure SaveToFile(const Filename: string);  //сохранить в файл
procedure SaveToStream(S: TStream);      //сохранить в память
```

Задача загрузки данных из файла или памяти решается с помощью другой пары процедур:

```
procedure LoadFromFile(const Filename: string); //загрузить из файла
procedure LoadFromStream(S: TStream);     //загрузить из памяти
```

Благодаря этим методам мы имеем возможность проектировать приложения, в которых пользователи смогут настраивать внешний вид, состав и последовательность колонок сетки на свой вкус. В листинге 29.2 как раз и предложено такое решение.

Листинг 29.2. Загрузка колонок из файла

```
procedure TForm1.FormShow(Sender: TObject);
var UserName : array[0..24]  of char;
    W        : Longword;
    FilePath, FileName : string;
begin
GetUserName(UserName,W); {узнаем имя пользователя}
 FilePath:=ExtractFilePath(Application.ExeName);{узнаем путь к приложению}
 {формируем имя файла с настройками колонок =
 "путь к приложению"+ "имя пользователя" + ".clm"}
 FileName:=FilePath+StrPas(UserName)+'.clm';
 If FileExists(FileName)=True {если такой файл существует}
   then DBGrid1.Columns.LoadFromFile(FileName); {загружаем пользовательские
настройки}
end;
```

Имя пользователя мы выясняем с помощью системной функции GetUserName(). Затем формируем имя файла настроек внешнего вида колонок. Оно состоит из пути к каталогу приложения, имени пользователя и расширения .clm. После проверки

факта существования этого файла загружаем данные. При закрытии формы решаем обратную задачу — сохраняем настройки пользователя при закрытии приложения (листинг 29.3).

Листинг 29.3. Сохранение колонок в файл

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var UserName : array[0..24]  of char;
    W        : Longword;
    FilePath, FileName : string;
begin
 GetUserName(UserName,W);
 FilePath:=ExtractFilePath(Application.ExeName);
 FileName:=FilePath+ StrPas(UserName)+'.clm';
 DBGrid1.Columns.SaveToFile(FileName);{сохраняем данные в файл}
end;
```

Возврат настроек колонок сетки в исходное состояние (состояние по умолчанию) обеспечит процедура

```
procedure RestoreDefaults;
```

Метод

```
procedure RebuildColumns;
```

полностью обновляет колонки в коллекции. При вызове этой процедуры все существовавшие ранее колонки уничтожаются. Новые колонки формируются только в том случае, если сетка подключена к открытому набору данных.

Колонка — класс *TColumn*

Экземпляр колонки реализован на базе класса `TColumn` и содержится в коллекции колонок сетки — свойстве `Columns`. Колонка располагает обширным набором свойств и методов, к наиболее важным из которых стоит отнести свойства, отвечающие за взаимодействие с полем набора данных.

Колонка представляет интересы соответствующего столбца набора данных. Поэтому она обладает знаниями о физическом имени этого поля:

```
property FieldName: String;
```

Более того, в колонке хранится ссылка на соответствующий ей объект-поле:

```
property Field: TField;
```

При необходимости допустимо запретить редактирование отдельной колонки сетки:

```
property ReadOnly: Boolean;
```

В табл. 29.2 приведен ряд свойств, влияющих на порядок отображения и редактирования данных колонки.

Таблица 29.2. *Форматирование текста в колонке*

Свойство	Описание
`property Alignment: TAlignment;`	Порядок выравнивания текстовых данных
`property Color: TColor;`	Фоновый цвет колонки
`property Font: TFont;`	Шрифт
`property Visible: Boolean;`	Видимость колонки
`property Width: Integer;`	Ширина колонки в пикселах

Каждая колонка снабжена заголовком — ячейкой, размещенной в самом верхнем ряду фиксированной области сетки. Для настройки внешнего вида заголовка необходимо обратиться к свойству

```
property Title: TColumnTitle;
```

В листинге 29.4 продемонстрирован порядок настройки заголовка нулевой колонки сетки.

Листинг 29.4. Изменение параметров колонки

```
Table1.Columns[0].Title.Font.Size:=12;
Table1.Columns[0].Title.Color:=clYellow;
```

Замечание

Для централизованного изменения шрифта заголовков всех колонок предназначено свойство сетки `TitleFont`.

Для выяснения, какие из атрибутов колонки отличаются от атрибутов, присваиваемых колонке по умолчанию, следует обратиться к свойству

```
property AssignedValues: TColumnValues; //только для чтения
```

Здесь возвращаемое значение `TColumnValues` является множеством значений `TColumnValue`:

```
type TColumnValue = (cvColor, cvWidth, cvFont, cvAlignment, cvReadOnly,
                     cvTitleColor, cvTitleCaption, cvTitleAlignment,
                     cvTitleFont, cvImeMode, cvImeName);
```

Если свойство возвратит значения `[cvColor, cvWidth]`, значит, цвет фона и ширина колонки отличаются от первоначальных. Если результирующее множество окажется пустым, то внешний вид колонки не изменялся. Как восстановить атрибуты по умолчанию? Для этого предназначено несколько функций, начинающихся со слова `Default` (табл. 29.3).

Таблица 29.3. *Настройки сетки по умолчанию*

Метод	Описание
`function DefaultAlignment: TAlignment;`	Выравнивание текста в столбце по умолчанию

Таблица 29.3 *(окончание)*

Метод	Описание
`function DefaultColor: TColor;`	Цвет по умолчанию (`clWindow`)
`function DefaultFont: TFont;`	Параметры шрифта по умолчанию
`function DefaultReadOnly: Boolean;`	Было ли разрешено редактирование данных в колонке
`function DefaultWidth: Integer;`	Ширина столбца (по умолчанию 64 пиксела)

Для полного возврата к настройкам по умолчанию вызовите процедуру

```
procedure RestoreDefaults;
```

Метод

```
procedure Assign(Source: TPersistent); override;
```

копирует колонку `Source` в текущую колонку.

По умолчанию в заголовке колонки отображается содержимое свойства `DisplayLabel` объекта `TField`, к которому подключена эта колонка. Для изменения значения по умолчанию воспользуйтесь свойством

```
property DisplayName: String;
```

Если колонка ассоциирует обычное текстовое поле, предназначенное для хранения сравнительно небольшого перечня данных, то целесообразно объединить возможные варианты допустимых значений в единый список и обеспечить пользователя механизмом выбора нужного значения из этого списка. Для решения такой задачи столбец `TColumn` обладает свойством

```
property PickList: TStrings;
```

Свойство доступно и в инспекторе объектов, и во время выполнения приложения. В листинге 29.5 продемонстрирован способ заполнения справочного списка аббревиатурами названий дней недели для динамически создающейся колонки.

Листинг 29.5. Подготовка перечня значений `PickList`

```
procedure TForm1.Table1AfterOpen(DataSet: TDataSet);
var i : Integer;
begin
For i :=0 to DBGrid1.Columns.Count-1 do
    if DBGrid1.Columns[i].FieldName='WEEKDAY' then
    begin
    With DBGrid1.Columns[i].PickList do
        begin
        Add('Пн.');
        Add('Вт.');
        …
```

```
        Add('Вск.');
        end;
      break;
      end;
end;
```

После открытия набора данных мы выполняем перебор всех колонок сетки, чтобы найти колонку, отображающую данные из поля `WEEKDAY`. Выяснив индекс этой колонки, мы заполняем ее сокращенными названиями дней недели.

Внимание!

При вызове метода `RestoreDefaults()` для восстановления атрибутов колонки по умолчанию очищается содержимое `PickList`.

Программист обладает возможностью ограничить число строк, выпадающих из справочного листа, при помощи свойства

```
property DropDownRows: Cardinal;
```

При выборе пользователем ячейки, обладающей справочным листом, в правой ее части появляется кнопка, щелчок по которой инициирует выпадение списка. Колонка разрешает определить порядок показа этой кнопки. Для этого реализовано свойство:

```
type TColumnButtonStyle = (cbsAuto, cbsEllipsis, cbsNone);
property ButtonStyle: TColumnButtonStyle;
```

По умолчанию свойство установлено в значение `cbsAuto` — автоматическое включение кнопки при наличии в колонке справочного листа или если колонка связана с полем синхронного просмотра. При выборе значения `cbsEllipsis` с ячейкой ассоциируется кнопка с многоточием на поверхности, щелчок по кнопке вызывает событие `OnEditButtonClick()`. Значение `cbsNone` — отказ от показа кнопки.

Ряд свойств колонки нацелен на обслуживание полей, состоящих из нескольких дочерних полей. Признак того, что колонка взаимодействует с полями такого типа — значение `true`, возвращаемое информационным свойством

```
property Expandable: Boolean;//только для чтения
```

Чтобы развернуть составную колонку и отобразить все дочерние поля, установите в `true` свойство

```
property Expanded: Boolean;
```

Обратная операция скроет дочерние поля.

Количество непосредственных предков колонки возвратит метод

```
function Depth: Integer;
```

Если родительских колонок нет — ответом станет нулевое значение.

Внимание!

Для того чтобы сетка была способна увидеть составные поля, у соответствующего ей набора данных должно быть включено свойство `ObjectView`.

Если родительская колонка все-таки существует, то ссылка вносится в свойство
```
property ParentColumn: TColumn;
```
Если составная колонка видима (`visible=true`) и не отображает дочерних колонок (`expanded=false`), то в свойстве
```
property Showing: Boolean;
```
окажется значение `false`.

С колонкой может быть связано свое собственное всплывающее меню
```
property PopupMenu: TPopupMenu;
```
Колонка знает, какой сетке она принадлежит:
```
property Grid: TCustomDBGrid;
```
Если с колонкой сетки связан вложенный набор данных, то, вызвав метод компонента `TDBGrid`,
```
procedure ShowPopupEditor(Column: TColumn;
          X: Integer = Low(Integer); Y: Integer = Low(Integer)); dynamic;
```
мы сможем показать его в отдельном окне. Здесь `Column` — ссылка на колонку; `X` и `Y` — экранные координаты левого верхнего угла всплывающего окна.

Обработка событий

В табл. 29.4 приведены шесть событий компонента `TDBGrid`, обеспечивающих взаимодействие пользователя и элемента управления.

Таблица 29.4. *События сетки* `TDBGrid`

Событие	Описание
`property OnColEnter: TNotifyEvent;`	Событие генерируется в момент получения фокуса ввода одной из колонок сетки
`property OnColExit: TNotifyEvent;`	Одна из колонок сетки теряет фокус ввода

`property OnCellClick: TDBGridClickEvent;`
 	
`type TDBGridClickEvent = procedure (Column: TColumn) of object;`
	Событие вызывается при щелчке кнопкой мыши по ячейке сетки, здесь `Column` — колонка, которой принадлежит данная ячейка

`property OnTitleClick: TDBGridClickEvent;`
 	
`type TDBGridClickEvent = procedure (Column: TColumn) of object;`
	Щелчок мышью по заголовку колонки

Таблица 29.4 *(окончание)*

Событие	Описание

`property OnColumnMoved: TMovedEvent;`
 	
`type TMovedEvent = procedure ( Sender: TObject; FromIndex, ToIndex: Longint) of object;`
	Событие вызывается после перемещения колонки из позиции `FromIndex` в позицию `ToIndex`

`property OnEditButtonClick: TNotifyEvent;`	Щелчок по кнопке внутри ячейки сетки

События прорисовки данных

Если вам по душе тоскливые ряды безликих цифр, то можете пропустить пару-другую страничек. Но если вы стремитесь, чтобы даже самый скучный бухгалтерский отчет радовал глаз не меньше, чем рождественская елка, давайте поговорим о способах оформления данных в сетке.

Изначально внешний вид сетки назначается свойством

```
property DrawingStyle: TGridDrawingStyle;
```

Возможных вариантов не столь много:

```
TGridDrawingStyle = (gdsClassic, //классический стиль
                     gdsThemed,  //стиль определяется темой Windows
                     gdsGradient);//градиентный
```

При выборе градиентного режима отображения сетка разрешает нам подобрать наиболее подходящие цвета начала и завершения заливки:

```
property GradientStartColor: TColor;
property GradientEndColor: TColor;
```

Свойство `DrawingStyle` в большей степени определяет особенности вывода границ и заголовков столбцов сетки и практически не затрагивает сами данные.

В компоненте `TDBGrid` реализован обработчик события, способный внести в унылый мир отображаемых в сетке строк и столбцов существенное разнообразие. Событие возникает в тот момент, когда сетка собирается перерисовать свою ячейку:

```
property OnDrawColumnCell: TDrawColumnCellEvent;

TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect;
                                  DataCol: Integer; Column: TColumn;
                                  State: TGridDrawState) of object;
```

Здесь `Rect` — прямоугольные координаты ячейки в клиентской области сетки; `DataCol` — индекс колонки в массиве колонок, которой принадлежит ячейка; `Column` — объект (колонка) и `State` — текущее состояние ячейки. Последний аргумент представляет собой множество

```
type TGridDrawState = set of (gdSelected, gdFocused, gdFixed);
```

в котором значение gdSelected свидетельствует о том, что ячейка выбрана; gdFocused — ячейка в фокусе ввода и gdFixed — ячейка находится в фиксированной области сетки.

Имея в своем распоряжении такой совершенный инструмент и используя возможности канвы сетки, можно создать шедевр любой сложности. Первым делом, информируем компонент о том, что теперь процессом перерисовки его содержимого руководим мы. Для этого переводим в false свойство

```
property DefaultDrawing: Boolean;//по умолчанию true
```

По умолчанию свойство установлено в true, т. е. перерисовка ячеек сетки является внутренним делом сетки.

А теперь рассмотрим небольшой пример (листинг 29.6). Сетка DBGrid1 отображает данные из таблицы стран Country, входящей в состав демонстрационной базы данных dbdemos.mdb. Таблица содержит некоторые данные о государствах Северной и Южной Америки. В состав данных входят название страны (колонка NAME), столицы (CAPITAL), континента (CONTINENT), площадь (AREA) и население (POPULATION).

Листинг 29.6. Управление графическим выводом в сетке

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
                          const Rect: TRect; DataCol: Integer;
                          Column: TColumn; State: TGridDrawState);
var s : string;
begin
with DBGrid1.Canvas do
begin
{определяем цвет выбранной пользователем ячейки}
If (gdSelected in State) then Brush.Color:=clHighlight
                         else Brush.Color:=clWindow;
FillRect(Rect);      {заливаем ячейку выбранным цветом}
{колонки с текстовыми данными выводим с обычным форматированием}
If (Column.Field.DataType=ftWideString) OR
   (Column.Field.DataType=ftString)
then {если поле текстовое, то осуществляем обычный вывод}
    TextRect(Rect,Rect.Left,Rect.Top,Column.Field.Value)
else
  if Column.Field.FieldName='AREA' then {площадь государства}
  begin
  //выбираем цвет шрифта в зависимости от размера страны
    If  Column.Field.Value<1000000 then
        Font.Color:=clBlack        else
    If (Column.Field.Value>1000000) and (Column.Field.Value<2000000) then
        Font.Color:=clBlue else
        Font.Color:=clRed;
```

```
    TextRect(Rect,Rect.Left, Rect.Top, Column.Field.AsString);
  end else
  if Column.Field.FieldName='population' then {колонка с численностью населения}
  begin
    If Column.Field.Value<50000000 then Font.Color:=clBlack
                                   else Font.Color:=clRed;
   {определяем форматирование для числа с применением разделителей тысяч}
   s:=Format('%.0n',[Column.Field.AsFloat]);
   {выравниваем текст по правому краю}
   TextRect(Rect,Rect.Right-TextWidth(S),Rect.Top, S);
  end
end;
end;
```

У сетки есть еще одно событие, приспособленное для внесения изменений в процесс перерисовки ячейки:

```
property OnDrawDataCell: TDrawDataCellEvent;
```

Но разработчики Delphi оставили его исключительно для обеспечения обратной совместимости и настоятельно рекомендуют в новых проектах им не пользоваться.

Статический текст — компонент *TDBText*

Элемент управления `TDBText` предназначен только для отображения текущего значения поля и не имеет абсолютно никаких навыков редактирования связанных с ним данных. Компонент является прямым потомком класса `TCustomLabel`, от которого также берет начало обычная текстовая метка `TLabel`. Для подключения компонента к набору данных достаточно выполнить стандартные операции определения источника данных (свойство `DataSource`) и указать, какое именно поле нас интересует (свойство `DataField`).

Для обеспечения автоподстройки размеров компонента при выводе текста с различным количеством символов установите в `true` свойство

```
property AutoSize: Boolean;
```

Если текстовые данные настолько необъятны, что никак не желают помещаться в одну строку, следует перевести в `true` свойство

```
property WordWrap: Boolean;//по умолчанию false
```

В этом случае элемент управления самым "чудесным" образом вспомнит про свои навыки вывода информации в несколько строк.

Строка ввода — компонент *TDBEdit*

Компонент класса `TDBEdit` представляет собой строку ввода, способную не только отображать, но и редактировать текущее значение поля. Предок элемента управления — класс `TCustomMaskEdit` (кстати, на его же основе построена строка ввода с маской `TMaskEdit`).

Если вы имеете опыт программирования в Delphi, то я уверен, что все базовые свойства и методы строки ввода вам уже известны. Кроме классических для всех элементов управления, работающих с БД, свойств: `DataSource` и `DataField` при работе с `TDBEdit` вам могут пригодиться еще несколько свойств и методов.

Доступ к тексту предоставляет свойство

```
property Text: TMaskedText;
```

Порядок назначения маски ввода мы уже обсуждали ранее, в частности, спецификаторы маски вы найдете в табл. 19.5.

Запрет на редактирование текста накладывает свойство

```
property ReadOnly: Boolean;
```

при установке его в `true`.

Установить предел на максимальное число символов, вводимых в компонент, можно с помощью свойства

```
property MaxLength: Integer;
```

По умолчанию в свойстве установлено нулевое значение, т. е. длина строки не ограничена.

Если планируется применение потомка класса `TCustomEdit` для ввода конфиденциальной информации (например, пароль доступа) и пользователь программного обеспечения не хочет знакомить с ними случайного прохожего, заглянувшего в монитор, то для скрытия вводимого текста рекомендуется воспользоваться свойством

```
property PasswordChar: Char;
```

Если значение свойства отличается от символа с кодом ноль (`#0`), то при отображении содержимого строки реальные символы будут подменяться символом из `PasswordChar`.

Из достаточно большого перечня событий компонента `TDBEdit` для программиста БД крайне полезно событие, возникающее в момент нажатия клавиши клавиатуры:

```
property OnKeyPress: TKeyPressEvent;
TKeyPressEvent = procedure (Sender: TObject; var Key: Char) of object;
```

Благодаря параметру `Key` (в котором передается код выбранного пользователем символа) событие позволяет создавать "передовую линию обороны", обеспечивающую корректность данных, передаваемых в БД. Допустим, что строка ввода предназначена для ввода в поле семизначного номера телефона, тогда можно предложить следующий вариант обработки события (листинг 29.7).

Листинг 29.7. Ограничение на ввод данных в компонент `TDBEdit`

```
procedure TForm1.DBEdit1KeyPress(Sender: TObject; var Key: Char);
begin
if ((KEY<'0') or (KEY>'9') or (Length(DBEdit1.Text)>=7) ) and (KEY<>#8) then
{ #8 — код клавиши Backspace}
begin
    Key:=#0;
    if (Length(DBEdit1.Text)>=7)=false then
    ShowMessage('Допускаются только цифры!')
    else
    ShowMessage('Номер больше 6 знаков!')
end;
end;
```

Многострочный редактор — *TDBMemo*

Основой многострочного текстового редактора служит абстрактный класс `TCustomMemo`. Непосредственно элемент управления `TDBMemo` спроектирован для работы с текстовыми полями больших размеров или с BLOB-полями, специализирующимися на хранении неформатированных текстовых данных.

Многострочный редактор представляет собой симбиоз оконного элемента управления и класса `TStrings`, инкапсулированного в компонент и доступного благодаря свойству

```
property Lines: TStrings;
```

Именно свойство `Lines` и позволяет хранить многострочные текстовые данные.

Если элемент управления `TDBMemo` подключен к BLOB-полю, то программист может определить порядок загрузки данных из таблицы в компонент при помощи свойства

```
property AutoDisplay: Boolean; //по умолчанию true
```

Если оно установлено в `true`, то обновление строк в компоненте осуществляется автоматически после каждого изменения данных в поле, например, при переходе к следующей строке в наборе данных. Если в свойстве содержится значение `false`, то вместо реальных данных компонент будет отображать название поля. Отказ от автоматического режима загрузки BLOB-данных в редактор `TDBMemo` несколько ускоряет процесс вывода данных, т. к. мы освобождаемся от необходимости обработки достаточно большого массива информации при открытии набора данных и перемещении по его записям. Но в последнем случае каждый раз при необходимости загрузить данные в элемент управления, потребуется вызов метода

```
procedure LoadMemo; virtual;
```

Во всем остальном элемент управления практически повторяет своего коллегу — стандартный текстовый редактор `TMemo`.

Редактор расширенного формата — *TDBRichEdit*

Создав многострочный текстовый редактор, позволяющий обрабатывать текстовые данные в полях BLOB, разработчики Delphi не смогли остановиться и "по инерции" решили аналогичную задачу для текстовых данных, представленных в *расширенном текстовом формате* (Rich-text Format, RTF). Тем более что для этого была создана превосходная база — класс `TCustomRichEdit`.

Кроме работы с форматированным текстом, никаких принципиальных особенностей у `TDBRichEdit` нет. Если вы применяли родственный компонент `TRichEdit`, то с легкостью справитесь и с его более "продвинутым" коллегой.

Изображение — компонент *TDBImage*

Элемент `TDBImage` специализируется на обработке графических данных, содержащихся в BLOB-полях. Несмотря на сильную схожесть с обычным компонентом `TImage`, у них разные предки. Если предтечей классического `TImage` служит класс `TGraphicControl`, то ближайший родственник нашего нового знакомого — `TCustomControl`. Однако, по крайней мере, для сохранения видимости преемственности в классе `TDBImage` вновь реализованы практически все одноименные с `TImage` свойства и методы для работы с изображениями. И точно так же интегрировано ключевое свойство

```
property Picture: TPicture;
```

инкапсулирующее "золотой" минимум возможностей для работы с растровой графикой и метафайлами.

Замечание

Класс `TPicture` служит классическим примером инкапсуляции и полиморфизма. Экземпляр класса представляет собой универсальное хранилище изображения, способное содержать популярные графические объекты: `TIcon`, `TBitmap`, `TMetafile` и `TJPEGImage`. Более того, `TPicture` превосходно справляется с решением вопросов по вызову методов, характерных для конкретного потомка `TGraphic`, тем самым упрощая работу программиста.

Для ускорения работы с BLOB-полями в элементе управления объявлено свойство

```
property AutoDisplay: Boolean;
```

задача которого аналогична функции одноименного поля компонента `TDBMemo` — включение/отключение автозагрузки данных из поля. Если свойство установлено в `false`, то для инициализации процесса загрузки изображения в компонент вызывают метод

```
procedure LoadPicture;
```

Также упомяну о существовании свойств `Center` и `Stretch`, соответственно центрующих и растягивающих изображение.

Список — *TDBListBox*

Список `TDBListBox` — ближайший родственник обычного списка `TListBox`. Элемент управления позволяет пользователю передавать в поле таблицы одно из заранее подготовленных текстовых значений. Перечень допустимых значений хранится в свойстве

```
property Items : TStrings;
```

Заполнение списка строк может осуществляться как в период разработки проекта во встроенном редакторе, так и во время работы приложения. В последнем случае все определяется знаниями программиста. Он может научить элемент управления загружать справочные данные из текстового файла, из таблицы или вводить их вручную — кому как больше нравится. Единственное условие для этого — знание возможностей класса `TStrings` (табл. 29.5).

***Таблица 29.5.** Ключевые свойства и методы класса `TStrings`*

Событие	Описание
`property Count : Integer;`	Число строк в списке
`property Strings[Index: Integer]: string;`	Доступ к элементу списка
`property Objects[Index: Integer]: TObject;`	Доступ к объекту списка
`procedure SaveToStream(Stream: TStream);`	Сохранение списка в поток
`procedure LoadFromStream(Stream: TStream);`	Загрузка списка из потока
`procedure SaveToFile( const FileName: string);`	Сохранение в файл
`procedure LoadFromFile( const FileName: string);`	Извлечение из файла
`function Add(const S: string): Integer;`	Добавляет в конец набора строку `S` и возвращает ее порядковый номер
`function AddObject(const S: string; AObject: TObject): Integer;`	Добавляет в конец набора строку `S` и ассоциирует с данной строкой объект, возвращает порядковый номер

Других, принципиально важных свойств и методов в классе `TDBListBox` нет, так что остановимся на сказанном.

Комбинированный список — *TDBComboBox*

Комбинированный список выбора `TDBComboBox` обладает всеми чертами классического списка выбора `TComboBox`. Компонент (точно так же, как и рассмотренный ранее список `TDBListBox`) призван помочь пользователю выбрать определенное значение из заранее подготовленного списка и передать его в поле таблицы.

Работа с компонентами `TDBComboBox` и `TDBListBox` практически ничем не отличается. Текущее значение поля, связанного с компонентом, отображается в строке редактирования элемента управления. Список значений содержится в свойстве `Items` (см. табл. 29.5), индекс текущего значения — в `ItemIndex`.

Группа переключателей — *TDBRadioGroup*

Компонент `TDBRadioGroup` реализует группу переключателей, состояние которых определяется значением текущего поля присоединенного набора данных. Одновременно может быть отмечена только одна из кнопок группы. При выборе пользователем кнопки, связанное с ним значение передается в базу данных. Ближайший родственник рассматриваемого элемента управления — обычный компонент `TRadioGroup`.

Список заголовков кнопок определяется содержимым свойства `Items`:

```
property Items: TStrings;
```

Перечень возможных значений в свою очередь определяется свойством

```
property Values: TStrings;
```

Оба свойства доступны во время как разработки, так и выполнения программы (см. табл. 29.5). Отдельно подчеркну, что число строк в обоих списках должно быть одинаковым.

Внимание!

Список значений `Values` не запрещается оставлять и пустым. В этом случае в базу станут передаваться определенные в свойстве `Items` заголовки переключателей.

Текущее значение связанного поля мы можем выяснить из свойства

```
property Value: String;
```

Индекс выбранной кнопки в группе находится в свойстве

```
property ItemIndex: Integer;
```

Флажок — *TDBCheckBox*

Элемент управления TDBCheckBox позволяет пользователю обслуживать поля логического типа — выбирать или отказываться от выбора какого-то значения. Хотя флажок нацелен на взаимодействие с полями данных типа Boolean, его также можно научить передавать в базу текстовые значения. Для этого предназначены свойства

```
property ValueChecked: String;  //значение на включение флажка
property ValueUnchecked: String;//значение на отключение флажка
```

по умолчанию содержащие текст "True" и "False" соответственно. Теперь, если в присоединенном к компоненту текстовом поле базы данных обнаруживается значение True, то этот факт найдет свое отражение — в виде "галочки" на поверхности элемента. Значение False, наоборот, прогонит всех "галок". Компонент способен отслеживать одновременно несколько значений, но тогда их нужно разделять точкой с запятой:

```
DBCheckBox1.ValueChecked := 'Да;Вкл';
DBCheckBox1.ValueUnchecked := 'Нет;Выкл';
```

В заключение упомяну два свойства (унаследованные от класса TCustomCheckBox), позволяющие как проверять текущее состояние элемента управления, так и включать/отключать флажок:

```
property Checked: Boolean;
```

и

```
type TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
property State: TCheckBoxState;
```

Причем последнее свойство позволяет контролировать и третье неактивное состояние, возвращая значение cbGrayed.

Компонент — *TDBCtrlGrid*

Класс TDBCtrlGrid — единственный компонент страницы **Data Controls** среды программирования Delphi, название которого не поддается дословному переводу. Если за корень названия принять слово "grid" (англ. сетка) и на этом основании сделать вывод о функциональном назначении компонента, то можно серьезно ошибиться, т. к. сетку элемент управления напоминает весьма отдаленно. Косвенным доказательством этому утверждению послужит иерархия наследования — непосредственный предок компонента класс TWinControl, не имеющий о сетках VCL даже поверхностного представления. Корректнее говорить о TDBCtrlGrid, как о наборе динамически создаваемых панелей. Как и все элементы управления, предназначенные для работы с набором данных, рассматриваемый компонент требует подключения к источнику данных DataSource. После открытия связанного с ним набора данных компонент самостоятельно создает несколько панелей. Каждая па-

нель ассоциируется с отдельной записью набора. А теперь главная черта TDBCtrlGrid — этот элемент управления может являться владельцем других компонентов, предназначенных для работы с данными, и не только их. Например, в период разработки проекта на поверхности TDBCtrlGrid программист имеет право разместить строки ввода TDBEdit и поясняющие текстовые метки TLabel. После запуска приложения наша "сетка" самым "чудесным" образом размножит доверенные ей элементы управления (рис. 29.2).

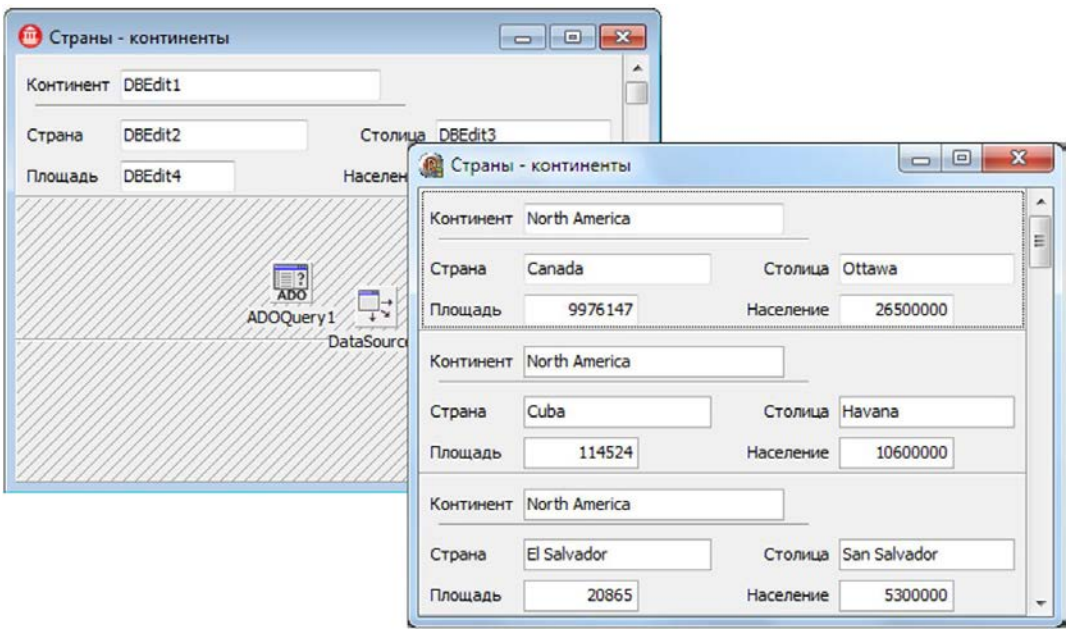


Рис. 29.2. Компонент TDBCtrlGrid во время проектирования и в работе

Порядок следования записей в компоненте TDBCtrlGrid определен свойством

```
property Orientation: TDBCtrlGridOrientation;
type TDBCtrlGridOrientation = (goVertical, goHorizontal);
```

По умолчанию свойство принимает значение goVertical, что соответствует последовательности сверху – вниз. В этом случае сетка снабжается вертикальной полосой прокрутки. Второе возможное состояние goHorizontal укажет компоненту о необходимости располагать записи слева направо и включит горизонтальную полосу прокрутки.

По умолчанию панели компонента размещаются в одном столбце. Для изменения этой традиции требуется настроить свойство

```
property ColCount: Integer;
```

Количество рядов, видимых в клиентской области элемента управления, определяется свойством

```
property RowCount: Integer;
```

За геометрические размеры панелей отвечают свойства

```
property PanelHeight: Integer;
property PanelWidth: Integer;
```

Общее число одновременно видимых панелей доступно из свойства

```
property PanelCount: Integer; //только для чтения
```

Свойство попросту умножает значения свойств `ColCount` и `RowCount`.

Индекс панели текущей записи хранится в свойстве

```
property PanelIndex: Integer;
```

Это же свойство можно применять при необходимости переместиться к панели с определенным номером. Запомните, что `PanelIndex` — всего лишь номер панели среди панелей, расположившихся в видимой клиентской области сетки, а не индекс записи в наборе. Диапазон допустимых значений `PanelIndex` находится в пределах от нуля (первая видимая панель) до `PanelCount-1` (последняя видимая панель).

Имитацию фокуса ввода для выбранной панели обеспечивает свойство

```
property ShowFocus: Boolean;
```

Если оно включено, то визуально запись выделяется прямоугольником.

Выбранную запись можно также подчеркнуть отдельным цветом:

```
property SelectedColor: TColor;
```

Еще более тонкие возможности по определению особенностей вывода панели на экран предоставляет событие

```
property OnPaintPanel: TPaintPanelEvent;
type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid;
                                  Index: Integer) of object;
```

Для рисования на поверхности панели воспользуйтесь ее свойством `Canvas`. Параметр `Index` информирует программиста, какую именно панель в данный момент выводит на экран обработчик события.

Если свойство

```
property AllowDelete: Boolean; //по умолчанию True
```

установлено в `true`, пользователь может удалить текущую запись, нажав комбинацию клавиш <Ctrl>+<Del>.

Для вставки новой записи достаточно нажать комбинацию <Ctrl>+<Ins>, но при одном условии, что свойство

```
property AllowInsert: Boolean; //по умолчанию True
```

также должно находиться в состоянии `true`.

Если свойство

```
property EditMode: Boolean;
```

содержит значение `true`, то перевод компонента в режим редактирования осуществляется автоматически при нажатии клавиши <F2>.

Для выполнения операции, определенной параметром `Key`, вызовите процедуру

```
procedure DoKey(Key: TDBCtrlGridKey);
type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab, gkLeft,
gkRight, gkUp, gkDown, gkScrollUp, gkScrollDown, gkPageUp, gkPageDown, gkHome,
gkEnd, gkInsert, gkAppend, gkDelete, gkCancel);
```

Это перемещение по панелям, перевод в режим редактирования, вставки, удаления, отмены.

ВНИМАНИЕ!

Существует ряд ограничений, препятствующих размещению на `TDBCtrlGrid` элементов управления. К числу недопустимых относятся `TDBGrid`, `TDBCtrlGrid`, `TDBLookupListBox`, `TDBLookupComboBox`, `TDBChart`. Признаком лояльности служит наличие стиля `csReplicatable` в свойстве `ControlStyle` компонента, размещаемого на панели. Этот стиль указывает, что компонент может копироваться для каждой панели. Допустимо, но не желательно размещение элементов, потребляющих много системных ресурсов: `TDBMemo`, `TDBRichEdit`, `TDBImage`.

Поля подстановки

Механизм синхронного просмотра, благодаря которому мы приобретаем возможность подставлять в основной набор данных поля из справочной таблицы, кроме сетки `TDBGrid` поддерживают еще два компонента: `TDBLookupListBox` (список) и `TDBLookupComboBox` (комбинированный список). Общим предком изучаемых элементов управления служит абстрактный класс `TDBLookupControl`. Именно в нем реализован ключевой алгоритм, позволяющий компоненту взаимодействовать со справочным набором данных.

ЗАМЕЧАНИЕ

Если понятие "поле подстановки" вам не знакомо, то рекомендую вернуться назад и пролистать *главу 19*. Один из разделов названной главы посвящен подстановочным полям (полям синхронного просмотра данных) и имеет непосредственное отношение к излагаемому здесь материалу.

Для осуществления синхронного просмотра программисту требуется организовать взаимодействие между основным и справочным наборами данных и указать имена внешнего, первичного ключа и название поля подстановки.

Соединение с основным набором данных производится в два действия. С помощью свойств

```
property DataSource: TDataSource;//источник данных базового набора данных
property DataField: String;      //имя внешнего ключа
```

требуется определить источник данных и имя поля, являющегося внешним ключом для связи со справочной таблицей.

После того как компонент соединится с базовой таблицей, на первый план выходит задача подключения элемента управления к справочному набору данных. Для этого нам понадобится последовательно заполнить три свойства:

```
property ListSource: TDataSource; //источник данных справочного набора
property KeyField: String;        //ключевое поле справочного набора
property ListField: String;       //имя подстановочного поля (полей)
```

Прежде всего, необходимо присоединить компонент ко второму источнику данных ListSource (который, в свою очередь, связан со справочным набором данных). После этого действия мы получим возможность определить имена первичного ключа (KeyField) и подстановочного поля (ListField). В ListField могут содержаться имена нескольких полей, но их названия требуется разделять точкой с запятой. В этом случае в одной строке компонента будут выведены значения этих полей одновременно.

ВНИМАНИЕ!

Настройка пяти перечисленных свойств — необходимое и одновременно достаточное условие для успешного функционирования потомков изучаемого класса TDBLookupListBox и TDBLookupComboBox.

Теперь поговорим о некоторых тонкостях класса TDBLookupControl. Весь перечень подстановочных полей кэшируется в списке:

```
property ListFields: TList;
```

Данное свойство недоступно в инспекторе объектов, однако может сослужить хорошую службу в тех ситуациях, когда программист осуществляет динамическое подключение компонента во время выполнения программы. Тогда перед назначением свойства ListField имеет смысл проверять наличие поля с таким названием в списке ListFields.

Если в свойстве ListField определено несколько полей, то нам может пригодиться свойство

```
property ListFieldIndex: Integer;//по умолчанию 0
```

Оно укажет, какое из полей (включенных в строку ListField) будет использоваться при поиске. Если мы отображаем несколько полей в компоненте TDBLookupComboBox, то свойство ListFieldIndex определит значение в области строки ввода.

Для проверки допустимости редактирования данных вызовите метод компонента

```
function CanModify: Boolean;
```

Для явного запрета установите в true свойство

```
property ReadOnly: Boolean;
```

Если поле базового набора данных не обязательно для заполнения и допускает пустые значения, то для ввода неопределенного значения NULL удобно назначить "быструю" клавишу с помощью свойства

```
property NullValueKey: TShortCut;
```

Список подстановки — *TDBLookupListBox*

Задача элемента управления заключается в обеспечении связи основного набора данных со справочной таблицей. Значения из справочного набора данных представляются в виде списка. Выбирая ту или иную строку в элементе управления, пользователь осуществляет подстановку выбранной записи в основной набор.

Значение выбранной строки всегда можно получить, обратившись к свойству

```
property SelectedItem: String;
```

Если ни одна из строк не выбрана, то свойство окажется пустым.

Число видимых строк в компоненте определяется свойством

```
property RowCount: Integer;
```

Комбинированный список подстановки — *TDBLookupComboBox*

Элемент управления решает ту же задачу, что и компонент `TDBLookupListBox`.

Выбранное в комбинированном списке значение доступно в свойстве

```
property Text: String; //только для чтения
```

Метод

```
procedure DropDown;
```

принудительно извлекает выпадающий список. В момент показа списка возникает событие

```
property OnDropDown: TNotifyEvent;
```

Пока список виден, свойство

```
property ListVisible: Boolean;//только для чтения
```

возвращает значение `true`.

Для скрытия выпадающего списка применяйте метод

```
procedure CloseUp(Accept: Boolean);
```

При этом возникает другое событие

```
property OnCloseUp: TNotifyEvent;
```

Три свойства определяют особенности выпадающего списка:

```
property DropDownAlign: TDropDownAlign;//выравнивание
property DropDownRows: Integer; //число видимых строк в списке
property DropDownWidth: Integer;//ширина списка в пикселах
```

Навигатор — *TDBNavigator*

Навигатор по набору данных создан на основе класса `TCustomPanel` и визуально представляет собой группу кнопок (рис. 29.3). Компонент инкапсулирует в себе "золотой" минимум методов, достаточных для создания простейшего приложения, работающего с таблицей базы данных. Сразу после подключения к источнику данных при помощи свойства `DataSource` элемент управления сможет обеспечить перемещение по записям в наборе, вставку, редактирование и удаление записи. Достаточно щелкнуть по соответствующей кнопке. Другими словами, навигатор — отличный подарок для лентяя, т. к. позволяет создать элементарное приложение без единой строки кода.

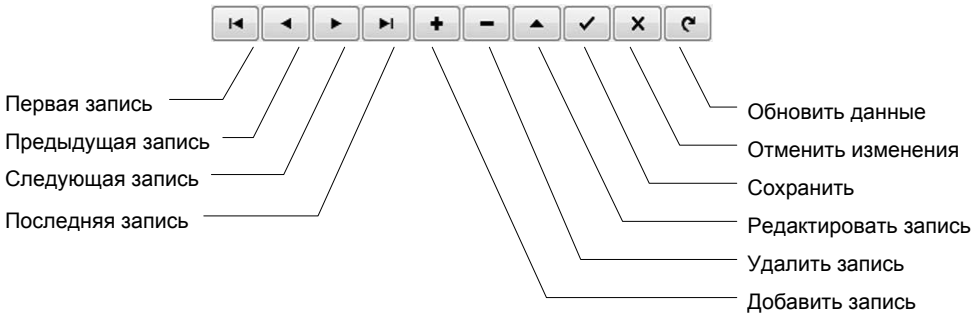


Рис. 29.3. Внешний вид навигатора

По умолчанию компонент содержит 10 кнопок, однако по желанию программиста вполне можно отказаться от показа ненужных элементов. Для этого следует настроить свойство

```
property VisibleButtons: TButtonSet;
```

```
type TButtonSet = set of TNavigateBtn;
```

```
type TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert,
                     nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);
```

Полагаю, что в особых комментариях тип данных `TNavigateBtn` не нуждается: `nbFirst` — кнопка "Первая запись", `nbPrior` — "Предыдущая запись" и т. д.

Если вы не собираетесь предоставить пользователю возможность удалять запись, то спрячьте кнопку удаления `nbDelete`:

```
DBNavigator1.VisibleButtons:=DBNavigator1.VisibleButtons-[nbDelete];
```

Как и практически все визуальные элементы управления VCL, навигатор обладает элементарным обработчиком события `OnClick()`. Правда, несколько усовершенствованным (компонент способен различать, по какой из кнопок был произведен щелчок):

```
property OnClick: ENavClick;
```

```
type ENavClick = procedure (Sender: TObject;
                            Button: TNavigateBtn) of object;
```

Благодаря параметру `Button`, можно узнать, какая из кнопок выступила инициатором события.

Перед выполнением операции, сопоставленной с кнопкой, возникает событие

```
property BeforeAction: ENavClick;
```

В навигаторе реализован дополнительный "рубеж обороны" по предотвращению несанкционированного удаления записи. Установив в `true` свойство

```
property ConfirmDelete: Boolean;
```

мы заставим компонент выводить сообщение, требующее подтвердить операцию удаления.

Для придания кнопкам модного плоского вида переведите в `true` свойство

```
property Flat: Boolean;
```

Несмотря на то, что `TDBNavigator` представляет собой монолитный компонент, для каждой своей кнопки он умеет отображать самостоятельную всплывающую подсказку. Для назначения текстов подсказок реализован набор строк

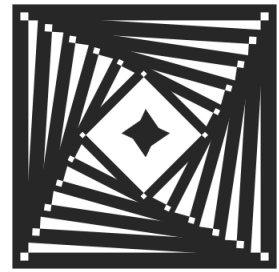
```
property Hints: TStrings;
```

При заполнении свойства данными требуется соблюдать последовательность строк: первая строка подсказки соответствует кнопке "Первая запись", вторая — "Предыдущая запись", последняя — "Обновить данные" (точно в том порядке, как размещены кнопки компонента). В противном случае подсказки перепутаются.

Резюме

Изученные в этой главе компоненты могут применяться во всех приложениях баз данных независимо от организации доступа к данным, будь то Interbase, ADO или устаревший BDE. Есть некоторые ограничения для технологии dbExpress. Из-за неспособности компонентов dbExpress кэшировать данные, совместно с ними нельзя применять сетки `TDBGrid` и `TCtrlGrid`.

Однако у компонентов DAC есть соперники, которые способны и на более серьезные подвиги. Не удивляйтесь, это не какие-то сверхсовременные компоненты из состава последней версии Delphi, а обычные элементы управления. Не верите? Тогда переверните эту страницу — вас ждет следующая глава.



Глава 30

Нестандартные решения для стандартных компонентов

Зачастую при разработке пользовательского интерфейса приложений баз данных, программисты стараются отказаться от услуг классических компонентов, работающих с данными (`TDBEdit`, `TDBGrid` и т. п.). Причин тому несколько. Это и желание сократить размерность результирующего кода (как-никак мы уклоняемся от подключения дополнительных библиотек), и стремление более строго контролировать процесс обслуживания данных. Но в первую очередь — необходимость решения каких-либо специфических задач, обслуживание особенных данных и построение нестандартного интерфейса приложения.

Создавая универсальные средства работы с данными, компания Embarcadero не смогла предусмотреть всех тонкостей и нюансов. Впрочем, в этом и не было особой необходимости, ведь открытая архитектура библиотеки визуальных компонентов позволяет программистам влиять на поведение практически любого компонента.

Эта глава посвящена вопросам обслуживания данных с помощью компонентов, "непривычных" для проектов БД: `TListBox`, `TComboBox`, `TCheckListBox`, `TListView`, `TStringGrid` и `TTreeView`. Общая особенность работы с перечисленными элементами управления заключается в способе загрузки данных из таблиц БД, ведь, в отличие от классических `TDBGrid`, `TDBText` и т. п., у стандартных компонентов попросту отсутствуют свойства и методы, обеспечивающие связь с набором данных.

Компоненты-списки

Плеяда элементов управления, списков, к которым относятся `TListBox`, `TComboBox`, `TComboBoxEx`, `TCheckListBox` и ряд других компонентов, специализируется на хранении набора текстовых строк. Независимо от особенностей реализации каждого из классов, "сердцем" этих компонентов является свойство

```
property Items: TStrings;
```

предоставляющее доступ к списку обслуживаемых компонентом строк.

Доступ к отдельной строке списка (элементу) осуществляется по ее порядковому номеру — индексу.

Простейшим способом добавления в список очередной строки будет обращение к методам `Add()` и `Insert()`, но как быть, если кроме текста с элементом списка требуется связать дополнительные данные, допустим, значение первичного ключа записи? В этом случае лучше всего воспользоваться процедурой

```
procedure AddItem(Item: string; AObject: Object);
```

Первый параметр метода — текст добавляемого элемента, а второй — объект, ассоциируемый с этим элементом. Благодаря аргументу `AObject` мы получаем возможность прикрепить к каждой строке списка дополнительную информацию, представляемую в виде отдельного объекта `TObject`.

При необходимости компонент-потомок класса `TCustomListBox` позволит нам сменить данные `AData`, связанные с элементом, идентифицируемым индексом `Index`

```
procedure SetItemData(Index: Integer; AData: LongInt);
```

Допустим, что мы планируем загрузить в компонент `ComboBox1` данные из справочной таблицы авторов нашей демонстрационной БД. Таблица включает два поля: первичный ключ `WRITER_KEY` и имя автора `WRITER`. Доступ к данным производим с помощью запроса `SQLQuery1`, инициализацией которого мы займемся в момент создания формы проекта (листинг 30.1).

Листинг 30.1. Подготовка запроса к получению данных

```
with SQLQuery1 do
begin
  SQL.Clear;
  SQL.Add('SELECT * FROM WRITERS ORDER BY WRITER')
  Prepare;
end;
```

Нам необходимо загрузить все данные из таблицы в список строк. Для этого мы последовательно перебираем все записи, возвращенные в результате выполнения запроса `SQLQuery1`. Информация о каждой строке передается в `ComboBox1` с помощью метода `AddObject()` (листинг 30.2).

Листинг 30.2. Заполнение комбинированного списка записями из таблицы

```
With SQLQuery1 do
begin
Open;
 while EOF=false do
  begin
    ComboBox1.Items.AddObject(FieldByName('WRITER').AsString,
                    TObject(FieldByName('WRITER_KEY').AsInteger));
```

```
    Next;
  end;
 close;
end;
```

Обратите внимание на то, что, кроме текстовых данных, каждая строка списка получает объект `TObject`, содержащий значение первичного ключа.

Решение обратной задачи (позволяющей выяснить значение первичного ключа для определенного элемента списка) предложено в листинге 30.3.

Листинг 30.3. Считывание значения первичного ключа из комбинированного списка

```
Function GetPrimaryKey(ItemIndex:Integer):Integer;
begin
  If (ItemIndex>-1) AND (ItemIndex< ComboBox1.Items.Count) then
      Result:=INTEGER(ComboBox1.Items.Objects[ItemIndex])
  Else Result:=-1;
end;
```

Подобный код можно внедрить в событие `OnChange()` комбинированного списка `TComboBox`, в событие `OnClickCheck()` компонента `TCheckListBox` и в событие `OnClick()` обычного списка `TListBox`.

Теперь переключим наше внимание на элемент управления `TCheckListBox`. Компонент отличается одной очень полезной чертой — он позволяет пользователю отмечать "галочкой" понравившиеся ему строки. Благодаря этой особенности можно создавать неординарный пользовательский интерфейс в приложениях баз данных, в которых встречается отношение многие ко многим. Хороший тому пример имеется в нашей демонстрационной базе данных книжного магазина. Здесь отношение "один ко многим" встречается дважды. Во-первых, между книгами (таблица `BOOKS`) и жанрами произведений (таблица `GENRES`). Если вы ненадолго вернетесь к ER-модели БД (см. рис. 6.7), то вспомните, что для решения этой проблемы мы ввели таблицу-коммутатор `GENRES_WRITERS`. Во-вторых, между произведением (таблица `BOOKS`) и авторами (таблица `WEITERS`). Поэтому у нас имеется еще один коммутатор — таблицы `WRITERS_BOOKS`.

Для заполнения компонента `CheckListBox1` данными из таблицы используем уже отработанную схему — пробегаем по всем записям таблицы жанров и передаем их названия и значения первичных ключей в элемент управления (листинг 30.4).

Листинг 30.4. Заполнение компонента `TCheckListBox` данными

```
With SQLQuery1 do
begin
 SQL.Clear;
```

```
 SQL.Add('SELECT GENRE, GENRE_KEY FROM GENRES ORDER BY GENRE');
 Open;
 while EOF=false do
  begin
    CheckListBox1.Items.AddObject(FieldByName('GENRE').AsString,
                        TObject(FieldByName('GENRE_KEY').AsInteger));
    Next;
  end;
 Close;
end;
```

После того как пользователь получил возможность работать с жанрами произведения, предлагаю решить еще одну полезную задачу — передать в БД книжного магазина информацию о новой книге. Допустим, что в магазин поступил бессмертный роман Ильи Ильфа и Евгения Петрова "12 стульев". В нашем распоряжении имеется готовая форма `frmBook` для заполнения информации о новой книге (рис. 30.1). Название книги оператор вводит в строку `Edit1`, авторы книги и их ключи заносятся в список `ListBox1`, жанр произведения отмечается в компоненте `CheckListBox1`. Заметьте, что мы максимально усложнили себе задачу: у книги есть два автора и жанр произведения классифицируется как роман, сатира и юмор. Налицо два отношения "многие ко многим".

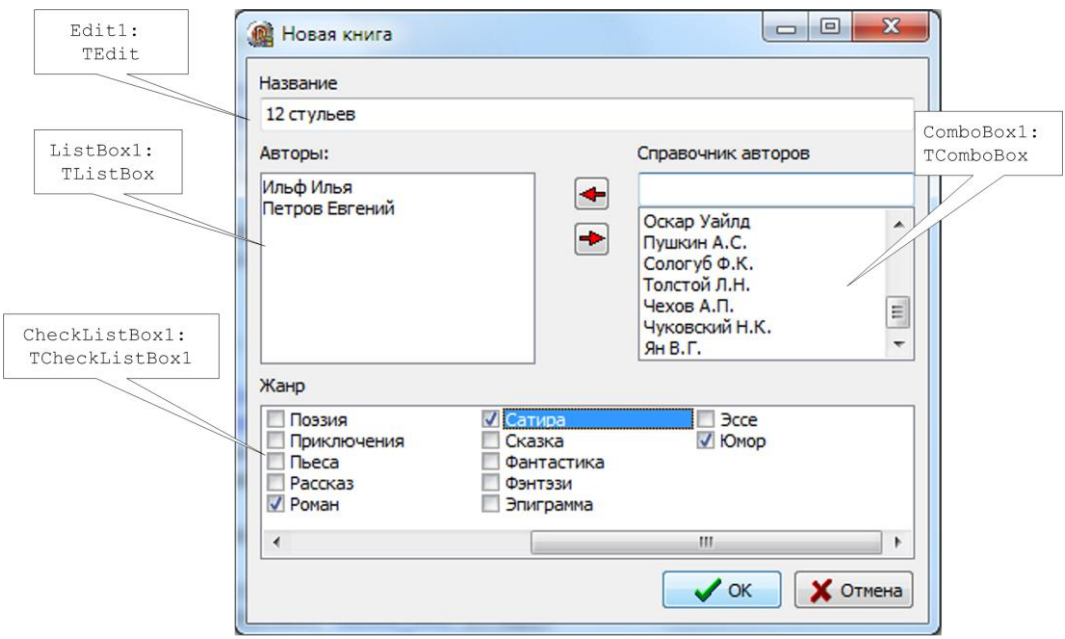


Рис. 30.1. Форма ввода данных о новой книге

Что должно произойти при нажатии кнопки **OK**?

1. Название книги передается в поле `TITLE` таблицы `BOOKS`.
2. Мы должны узнать значение первичного ключа `BOOK_KEY` новой записи.
3. Данные об авторах помещаются в таблицу `WRITERS_BOOKS`.
4. Данные о жанре направляются в таблицу `GENRES_BOOKS`.

Все перечисленные операции следует осуществить в рамках одной транзакции, приведенной в листинге 30.5.

Листинг 30.5. Транзакция сохранения записи с новой книгой

```
var TransDesc: TTransactionDesc;
    i,BOOK_KEY,KEY:Integer;
…
{Старт транзакции}
TransDesc.IsolationLevel:=xilREPEATABLEREAD;
DataModule1.SQLConnection1.StartTransaction(TransDesc);

TRY

With SQLQuery1 do
begin
 SQL.Clear;
 {1- добавляем в BOOKS запись о названии книги}
 SQL.Add('INSERT INTO BOOKS (TITLE)');
 SQL.Add('VALUES ('+#39+TRIM(frmBook.Edit1.Text)+#39+')');
 ExecSQL;
 {2 - узнаем первичный ключ новой записи}
 SQL.Clear;
 SQL.Add('SELECT BOOK_KEY FROM BOOKS');
 SQL.Add('WHERE TITLE='+#39+TRIM(frmBook.Edit1.Text)+#39);
 Open;
 BOOK_KEY:=FieldByName('BOOK_KEY').AsInteger;
 Close;
{3- данные об авторах передаем в таблицу WRITERS_BOOKS}
 SQL.Clear;
 SQL.Add('INSERT INTO WRITERS_BOOKS');
 SQL.Add('(WRITER_KEY,BOOK_KEY) VALUES (:WRITER_KEY,:BOOK_KEY)');
 Params.ParamValues['BOOK_KEY']:=BOOK_KEY;
   {перебираем все строки в ListBox1 и узнаем первичные ключи авторов}
  for i:=0 to frmBook.ListBox1.Count - 1 do
        begin
         KEY:=INTEGER(frmBook.ListBox1.Items.Objects[i]);
```

```
        Params.ParamValues['WRITER_KEY']:=KEY;
        ExecSQL;
       end;
 {4- данные о жанрах передаем в таблицу GENRES_BOOKS}
      SQL.Clear;
      SQL.Add('INSERT INTO GENRES_BOOKS');
      SQL.Add('(GENRE_KEY,BOOK_KEY) VALUES (:GENRE_KEY,:BOOK_KEY)');
      Params.ParamValues['BOOK_KEY']:=BOOK_KEY;
     {ищем отмеченные "галочками " жанры}
     for i:=0  to frmBook.CheckListBox1.Count - 1 do
       if frmBook.CheckListBox1.Checked[i] then
         begin
          KEY:=INTEGER(frmBook.CheckListBox1.Items.Objects[i]);
          Params.ParamValues['GENRE_KEY']:=KEY;
          ExecSQL;
         end;
  end;
  DataModule1.SQLConnection1.Commit(TransDesc);
 EXCEPT
  {откат транзакции в случае ошибок}
  DataModule1.SQLConnection1.Rollback(TransDesc);
 END;
```

Предложенный пример можно реализовать не только на динамическом SQL, но и с помощью хранимых процедур или нескольких компонентов-запросов. Все зависит от возможностей СУБД и знаний программиста.

Компонент *TListView*

Компоненты-списки, подобные `TListBox`, вполне пригодны для работы с одной, может быть, двумя колонками текстовых данных, но что делать, если таких колонок несколько? Ответ таков — воспользоваться услугами списка с расширенными возможностями — компонента `TListView`.

Проведем небольшую экскурсию по его свойствам и методам. В первую очередь нас интересует особенности работы с `TListView` в табличном режиме работы. Для этого переводим его свойство

```
property ViewStyle: TViewStyle;
type  TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport);
```

в состояние `vsReport`.

С этого момента компонент `TListView` приступит к выводу информации в виде многоколоночного отчета.

Перечень всех колонок программист должен либо определить заранее, во время разработки проекта в свойстве-контейнере

```
property Columns: TListColumns;
```

либо сформировать колонки на лету, обращаясь к методу

```
function Add: TListColumn;
```

Пример создания колонок во время выполнения приложения приведен в листинге 30.6.

Листинг 30.6. Создание колонок в компоненте `TListView`

```
var i : integer;
    ListColumn : TListColumn;
begin
ListView1.ViewStyle:=vsReport;
ListView1.Columns.Clear;
with SQLTable1 do
begin
  SQLConnection:=SQLConnection1;
  TableName:='ORDERS';
  Open;
  {Открыв таблицу, создаем колонки по числу полей таблицы}
     for i:=0 to FieldCount-1 do
     begin
      ListColumn:=ListView1.Columns.Add;
      ListColumn.Caption:= Fields.Fields[i].DisplayName;
     end;
   Close;
end;
end;
```

После подготовки колонок мы можем переходить к заполнению компонента данными. Напомню, что отдельный элемент списка описывается классом `TListItem`, а весь перечень элементов компонент `TListView` хранит в свойстве

```
property Items: TListItems;
```

Элемент списка `TListItem` может похвастаться широким перечнем свойств и методов, но сейчас я упомяну лишь основные из них. Заголовок элемента вносится в свойство

```
property Caption: string;
```

Каждый элемент может обладать набором строк

```
property SubItems: TStrings;
```

Строки из набора `SubItems` выводятся в колонках компонента `ListView`, если свойство `ViewStyle` последнего установлено в `vsReport`.

Индекс элемента в списке находится в свойстве

```
property Index : Integer;
```

С любым элементом списка допускается связать любые данные, для этого предназначен указатель

```
property Data: Pointer;
```

Для удаления элемента воспользуйтесь методом

```
procedure Delete;
```

Коллекция `TListItems` снабжена перечнем методов, позволяющих добавлять, редактировать и удалять элементы. Для того чтобы добавить в расширенный список новый элемент, воспользуйтесь методом

```
function Add: TListItem;
```

Для вставки элемента `TListItem` в определенное место списка служит метод

```
function Insert(Index: Integer): TListItem;
```

Новый элемент помещается в позицию, указанную в `Index`.

Метод

```
procedure Delete(Index: Integer);
```

удаляет элемент из списка.

Для очистки всего перечня элементов используйте метод

```
procedure Clear;
```

Листинг 30.7 содержит продолжение примера использования компонента `TListView` на службе БД.

Листинг 30.7. Заполнение `TListView` данными

```
var i:integer;
    LI:TListItem;
begin
with SQLTable1 do
begin
Open;
  while eof<>true do //перебираем записи таблицы
   begin
   LI:=ListView1.Items.Add; //добавляем новый элемент
   {заполняем элемент информацией}
   LI.Caption:=Table1.Fields.Fields[0].AsString;
   for i:=0 to  Table1.FieldCount-1 do
        LI.SubItems.Add(Table1.Fields.Fields[i].AsString);
   Next; //переходим к следующей записи
  end;
 Close;
end;
end;
```

Сетка, компонент *TStringGrid*

Простейшие задачки загрузки в компонент текстовых данных мы уже "щелкаем как орехи". Но как быть, когда с компонентом нужно связать структурированную информацию, содержащую не банальные "текст + ключ", а что-то более изысканное, например, ячейку расписания занятий студенческой группы в вузе? Кто-то уже побывал студентом, кто-то грызет гранит науки сейчас, кому-то это предстоит в будущем. Но каждый знает, что основу жизнедеятельности студента составляет расписание. Ячейка расписания проинформирует нас о дате и времени занятия, названии дисциплины, виде и порядковом номере занятия, аудитории, в которой оно будет проведено, и о многом другом, но для примера, пожалуй, хватит. Мы не станем сейчас обсуждать особенности организации подобных данных, распределение полей по таблицам, связи между этими таблицами… Будем считать, что это вопрос решенный, и все перечисленные информационные элементы поступают нам в результате выполнения какого-то запроса.

Для реализации столь грандиозных планов не обойтись без своего собственного типа данных — записи `TCell`, которая содержит все данные, подлежащие хранению (листинг 30.8).

Листинг 30.8. Фрагмент структуры данных `TCell` — ячейки занятия

```
type pCell =^TCell;
     TCell=PACKED RECORD
     Lesson_Cod : String[4]; {код дисциплины}
     Faculty_Name, Lesson_Name : String[30];{название кафедры, дисциплины}
     Num : Integer;          {номер занятия}
      {значения ключевых полей и другие данные
end;
```

Напомню о существовании у компонента `TStringGrid` свойства

```
property Objects [ACol, ARow: Integer]: TObject;
```

позволяющего связывать с отдельной ячейкой сетки любой объект. В нашем случае, с каждой из ячеек сетки будет ассоциирован указатель на запись `Cell : pCell`. Для заполнения сетки данными в цикле `while..do` перебираем все строки результирующего набора данных, полученного от запроса `SQLQuery1` (листинг 30.9).

Листинг 30.9. Заполнение сетки `TStringGrid` данными

```
Var  Cell : pCell;
     aCol, aRow : integer;
begin
…
```

```
With SQLQuery1 do
 While Eof<>true do {пока не достигнут конец набора}
 Begin
   Cell:=New(pCell);{распределяем память под структуру pCell}
   {заполняем поля структуры}
   Cell.Lesson_Cod:=FieldByName('Lesson_Cod').AsString;
   Cell.Faculty_Name:=FieldByName('Faculty_Name').AsString;
  {и так далее, заполняем все поля записи}

   aCol:=…; {по какому-то правилу назначаем координаты очередной ячейки}
   aRow:=…;
   SG.Objects[aCol,aRow]:=TObject(Cell);  {связываем с ячейкой объект}
  Next; {переход к следующей записи}
End;
```

Идея фрагмента кода из листинга 30.9 проста. Каждая полученная строка данных сохраняется в памяти в формате структуры `TCell`. Для этих целей мы вызываем функцию `New()`. Получив указатель на область памяти, заполняем соответствующие поля структуры и связываем указатель с ячейкой сетки, воспользовавшись ее свойством `Objects`.

Вывод данных в ячейках сетки производится в событии `OnDrawCell()` компонента `TStringGrid`. Все остальное определяется только вашим воображением и знаниями функций графической библиотеки GDI (листинг 30.10).

Листинг 30.10. Прорисовка ячеек сетки `TStringGrid`

```
procedure TfrmDayTable.StringGrid1DrawCell(Sender: TObject;
                                    ACol, ARow: Integer;
                                    Rect: TRect; State: TGridDrawState);
var Cell:pCell;
    Rct:TRect;
begin
  Cell:=pCell(SG.Objects[aCol,aRow]); //указатель на данные ячейки
  With StringGrid1.Canvas do
  Begin
   Brush.Color:=clWindow;
   FillRect(Rect);
   TextOut(Rect.Left+1,Rect.Top+1, Cell^.Lesson_Cod); {код дисциплины}
   //и так далее
  end;
end;
```

На рис. 30.2 приведен экранный снимок фрагмента расписания на основе сетки `TStringGrid`.

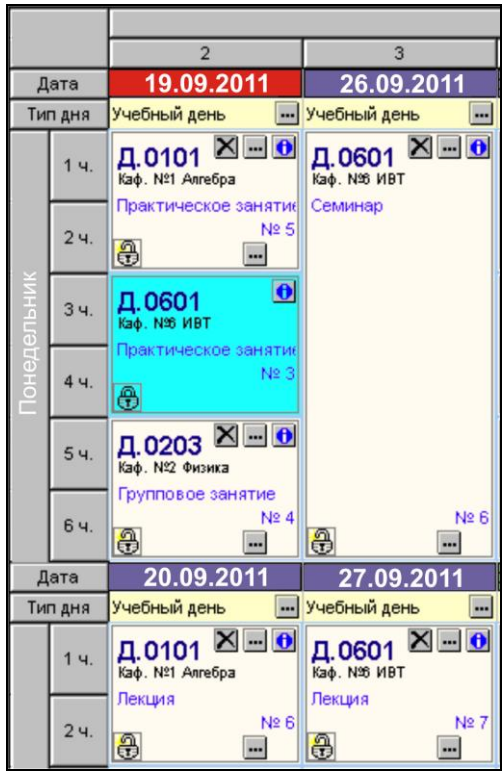


Рис. 30.2. Расписание в компоненте `TStringGrid`

Завершая работу с сеткой, необходимо позаботиться об очистке всех связанных с ней данных, здесь нам поможет процедура `Dispose()`. Следует перебрать все ячейки компонента, у которых есть ссылки на внешние данные (листинг 30.11).

Листинг 30.11. Удаление связанных с сеткой объектов

```
for aCol:=1 to StringGrid1.ColCount-1 do
     for aRow:=1 to StringGrid1.RowCount-1 do
       begin
         if StringGrid1.Objects[aCol, aRow])<>nil then
            Dispose(StringGrid1.Objects[aCol, aRow]);
        end;
```

Нам пришлось здорово потрудиться, но мы научили компонент `TStringGrid` решать задачу повышенной сложности — работать со структурированными данными.

Иерархические данные

В своей повседневной деятельности мы довольно часто сталкиваемся с информацией, представленной в иерархической форме. Это, например, организационная структура предприятия, штатный состав отдела, иерархия наследования классов VCL, различного рода технологические цепочки. По сравнению с однообразными сетками структурированные в виде дерева данные более наглядны и удобны для конечного пользователя. Рассмотрим основные способы хранения иерархически организованных данных в реляционных таблицах и особенности представления данных такого рода в элементах управления Delphi (рис. 30.3).

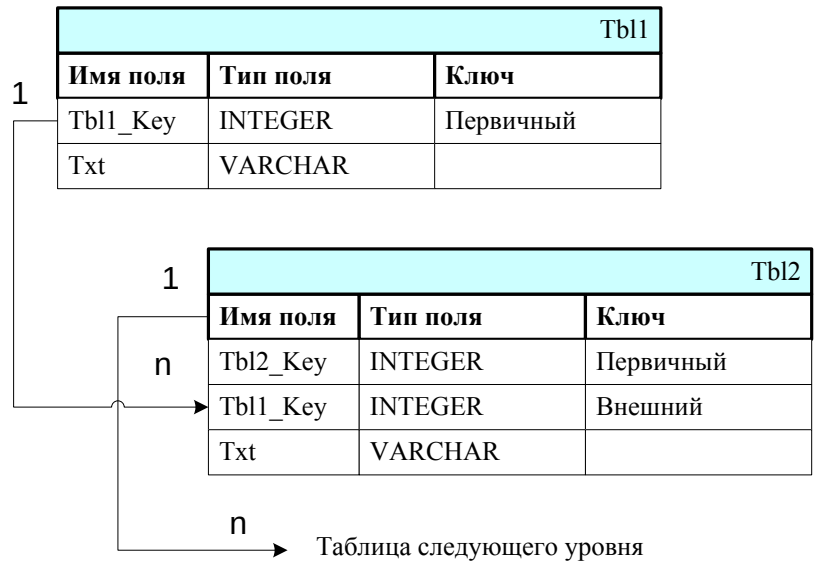


Рис. 30.3. Вариант объединения таблиц для хранения иерархически организованных данных

С точки зрения различия способов хранения иерархических данных в двумерных реляционных таблицах следует выделить два направления. Первое — объединение данных в таблицы, связанные отношением "главная-подчиненная". Второе — работа с таблицей, рекурсивно ссылающейся на саму себя.

Многотабличное представление иерархических данных

Первая идея проста в реализации и в программном сопровождении. Создается цепочка связанных таблиц (рис. 30.3). Узлы иерархической структуры каждого уровня хранятся в соответствующих таблицах: самого верхнего уровня — в главной таблице `Tbl1`, следующего — в `Tbl2` и т. д.

Для сбора данных строится простейший объединяющий запрос вида

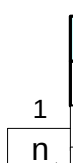
```
SELECT * FROM Tbl1
LEFT OUTER JOIN Tbl2 ON Tbl2.Tbl1_Key= Tbl1.Tbl1_Key
```

Методы вставки, редактирования и удаления строк также не вызовут особой сложности.

К сожалению, многотабличный способ хранения иерархий подходит лишь для использования в простейших проектах, описывающих элементарные статичные структуры. Как только встает вопрос о реконфигурировании данных, например (см. рис. 5.10) о выводе записи "Транспортный цех" из подчинения узла "Управление" и передачи в узел "Структура", мы сразу столкнемся с определенными трудностями. Только представьте себе, сколько придется сочинить кода, чтобы перенести запись со всеми подчиненными ей строками с одного уровня на другой (читай из одной таблицы в другую). При этом нужно умудриться сохранить все связи "один ко многим" между записями всех таблиц. Сложность программной реализации подобной задачи прямо пропорциональна количеству задействованных таблиц. Другой недостаток многотабличных структур связан с ограничением на число уровней вложения иерархической структуры — сколько таблиц, столько и уровней. Нужно добавить новый — будь добр, генерируй очередную таблицу.

Рекурсивная связь

Древообразные структуры можно описать и в пределах одной-единственной таблицы. В этом случае таблица связывается сама с собой — внешний ключ ссылается на первичный (родительский) ключ той же самой таблицы. Таким образом, в отличие от многотабличного представления иерархических данных, мы сможем создавать цепочки взаимного подчинения узлов практически неограниченной длины. Кроме того, проявив немного сноровки, мы позволим пользователю свободно перемещать узлы внутри иерархии и легко менять владельца узла.



TREE		
Имя поля	**Тип поля**	**Ключ**
TREE_KEY	INTEGER	Первичный
PAR_KEY	INTEGER	Внешний
TXT	CHAR	
IND	INTEGER	

Рис. 30.4. Структура таблицы с поддержкой рекурсии

На рис. 30.4 представлена структура таблицы такого рода. Здесь `TREE_KEY` — первичный ключ, `PAR_KEY` — внешний ключ, `TXT` — текстовое описание. Последнее поле `IND` пригодится нам только в случае, если программная логика приложения

будет допускать сортировку элементов и переподчинение элементов дерева, например, перемещение узла самого нижнего уровня наверх, с подчинением ему всей остальной иерархии. При этом в поле необходимо заносить порядковый номер записи в иерархии. Это утверждение я поясню позднее, а пока познакомимся со способом представления иерархических структур в компоненте `TTreeView`.

Компонент достаточно сложен и в библиотеке компонентов занимает обособленное место. "Сердцем" `TTreeView` по праву считается свойство

```
Property Items: TTreeNodes;
```

Это не что иное, как хранилище всех узлов. Каждый узел описывается экземпляром класса `TTreeNode`. Узел в состоянии владеть дочерними узлами, и сам может быть подчиненным по отношению к своему родительскому узлу.

Назначение узла — хранение данных. Для этого в экземпляре класса `TTreeNode` объявлены два свойства:

```
property Text: string;
property Data: Pointer;
```

Свойство `Text` специализируется на обслуживании текстовой информации. Это строка, которая будет отображаться в качестве заголовка узла. Указатель `Data` — универсальное свойство, оно способно связать с узлом любой объект (число, файл, картинку).

Узел описывается дополнительной характеристикой — уровнем. Корневой узел иерархии размещен на нулевом уровне, первый его потомок — на первом, потомок потомка — на втором... Чтобы узнать, на каком уровне иерархии находится узел, нужно обратиться к свойству

```
property Level: Integer;
```

Порядковый номер узла в коллекции описывается его абсолютным индексом

```
property AbsoluteIndex : Integer; //только для чтения
```

Отсчет начинается с нуля. Операции добавления, удаления или перемещения узлов внутри коллекции автоматически переиндексируют все узлы дерева.

Порядковый номер элемента в списке родительского узла находится в свойстве

```
property Index: Longint; //только для чтения
```

Предусмотрено несколько способов добавления нового узла в коллекцию (табл. 30.1).

Таблица 30.1. *Методы добавления нового узла в дерево* `TTreeView`

Функция	Описание
`function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Узел добавляется самым последним на уровне узла `Node`
`function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer) : TTreeNode;`	Узел добавляется первым, на уровне узла `Node`
`function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer) : TTreeNode;`	Узел добавляется в качестве последнего дочернего узла

Таблица 30.1 *(окончание)*

Функция	Описание
`function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Узел добавляется в качестве первого дочернего узла. Остальные дочерние узлы сдвигаются вниз
`function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Новый узел вставляется перед `Node`
`function InsertNode(Node, Sibling: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;`	Узел `Sibling` вставляется перед `Node` на том же уровне

Разработаем небольшой проект, демонстрирующий особенности построения приложений иерархических баз данных. Создайте новую таблицу в соответствии со структурой, изображенной на рис. 30.4 (для своего примера я воспользовался услугами БД Access), и сохраните ее под именем `TREE`.

Начнем новый проект Delphi. Он содержит одну главную форму `frmMain` и модуль данных `DataModule1` (рис. 30.5). Разместите на модуле данных пять компонентов `TADOQuery`. Компонент `qSelect` отвечает за отправку запроса на сбор данных, `qEditText` — редактирование текста, `qDelete` — удаление строки, `qInsert` — вставку новой строки. В паре с компонентом `qInsert` работает `qGetKey`, он возвратит значение первичного ключа только что добавленного элемента.

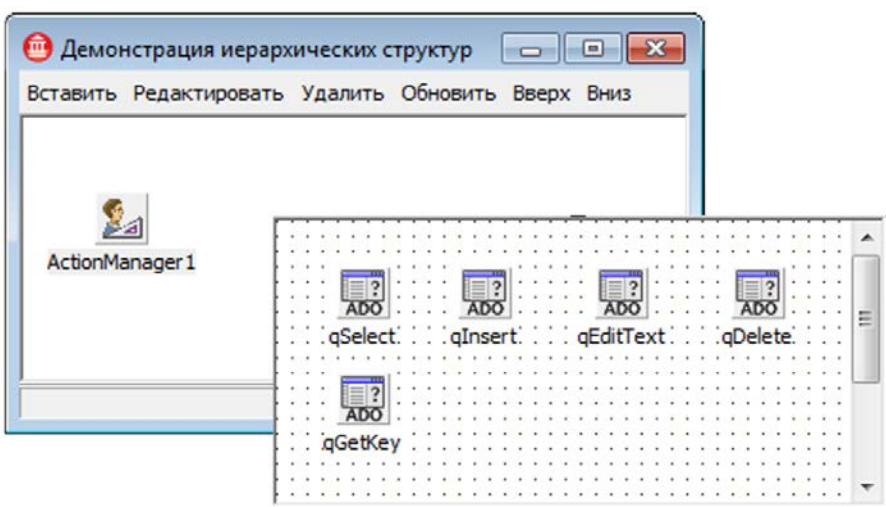


Рис. 30.5. Форма и модуль данных проекта

На поверхности главной формы разместите дерево — компонент `TTreeView`. Для вызова тех или иных команд приспособим менеджер `TActionManager`.

Пока он содержит четыре команды:

- `acUpdate` — команда на обновление содержимого дерева.
- `acInsert` — вставка новой записи.
- `acEdit` — редактирование записи.
- `acDelete` — удаление узла.

Инициализация проекта

Предлагаю инициализировать проект в момент создания модуля данных, так мы решим две задачи:

- настроим строку соединения с базой данных для всех `TADOQuery`;
- опишем SQL-инструкции для каждого из запросов (листинг 30.12).

Листинг 30.12. Подготовка инструкций SQL

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
var i:integer;
    s: string;
begin
s:=ExtractFilePath(Application.exename)+'tree.accdb';//путь к БД
{подключим все компоненты к файлу БД}
 for i:=0 to ComponentCount-1 do
    if Components[i] is TADOQuery then
       with Components[i] as TADOQuery do
          begin
          SQL.Clear;
          ParamCheck:=True;
          ConnectionString:='Provider=Microsoft.ACE.OLEDB.12.0;
               Data Source='+s+';Persist Security Info=False';
          end;
//инструкция получения данных
 qSelect.sql.add('SELECT * FROM TREE ORDER BY IND');
//инструкция вставки записи
 qInsert.sql.add('INSERT INTO TREE (TXT, PAR_KEY,IND) ');
 qInsert.sql.add('VALUES (:TXT,:PAR_KEY,:IND)');
//инструкция получения значения первичного ключа последней записи
 qGetKey.sql.add('SELECT MAX(Tree_Key) AS NEWKEY FROM TREE');
//инструкция редактирования записи
 qEditText.sql.add('UPDATE TREE SET TXT=:TXT WHERE TREE_KEY=:TREE_KEY');
//инструкция удаления записи
 qDelete.sql.add('DELETE FROM TREE WHERE TREE_KEY=:TREE_KEY');
end;
```

Текст инструкций SQL достаточно прост и не должен вызывать у читателя недоумения. Впрочем, если что-то непонятно, вернитесь к главам, посвященным языку SQL. По ходу примера к этой процедуре будет добавлено еще несколько строк, но пока модуль данных нам не нужен.

Новая запись

В нашем примере процесс вставки в таблицу новой записи разбит на две составные части — процедуры `InsertNode()` и `acInsertExecute()`.

Функция `InsertNode()` добавляет в таблицу новую запись, вызывая соответствующие инструкции SQL (листинг 30.13). Ее мы упомянем в разделе публичных объявлений модуля данных. Функция включает три параметра: `ParKey` — родительский ключ узла-владельца, `Index` — порядковый индекс элемента в дереве и текст метки узла — `Txt`. Функция возвращает значение первичного ключа добавленной записи.

Листинг 30.13. Функция добавления новой записи в таблицу

```
function TDataModule1.InsertNode(const ParKey, Index: integer;
                                 const Txt: string): Integer;
begin
  with qInsert.Parameters do
  begin
    ParamValues['PAR_KEY']:=ParKey;
    ParamValues['IND']:=Index;
    ParamValues['TXT']:=Trim(Txt);
    qInsert.ExecSQL;
  end;
  with qGetKey do
  begin
   Open;
   Result:=qGetKey.FieldByName('NewKey').AsInteger;
   Close;
  end;
end;
```

В теле процедуры мы последовательно обращаемся к запросам `qInsert` и `qGetKey`. С помощью первого компонента мы передаем в таблицу новые данные, второй компонент возвращает значение первичного ключа новой записи.

По команде `acInsert : TAction` генерируется событие `OnExecute()`. Из кода события вызывается функция `InsertNode()`, и в элементе управления `TreeView1` создается новый узел `TTreeNode` (листинг 30.14).

Листинг 30.14. Вставка нового узла в дерево

```
procedure TfrmMain.acInsertExecute(Sender: TObject);
var txt:string;      //текст узла
    Node:TTreeNode; //ссылка на новый узел
    ParKey, ItemIndex :integer; //ключ и индекс узла
    Obj:^INTEGER; //ссылка на объект
begin
{запрашиваем название новой записи-узла}
if InputQuery('Новая запись', '', txt)=true then
   begin
     if Assigned(TreeView1.Selected) then //если выбран узел
        begin
          {владельцем станет выбранный узел}
          ParKey:=Integer(TreeView1.Selected.Data^);
          {узнали значение первичного ключа узла владельца,
           это значение будет передано во внешний ключ нового узла}
          ItemIndex:=TreeView1.Items.Count+1;
        end else
        begin   {владельца нет, создаем узел нулевого уровня}
          ParKey:=-1;
          {раз нет владельца, то родительский ключ -1}
          ItemIndex:=0;
        end;
    {резервируем область памяти для хранения значения ключа}
     New(Obj);
     {добавляем запись в таблицу (см. листинг 30.13)}
     Obj^:=DataModule1.InsertNode(ParKey, ItemIndex, Txt);
    {теперь в Obj хранится указатель на значение первичного ключа}
    {создаем узел в дереве}
     Node:=TreeView1.Items.AddChildObject(TreeView1.Selected,Txt,Obj);
     {выбираем узел}
     Node.Selected:=True;
   end;
end;
```

В нашем примере процесс добавления новой записи включает три этапа:

1. Запрос у пользователя текста новой записи.
2. Выяснение, какой из уже имеющихся узлов станет владельцем будущей записи, или же будет создан узел самого верхнего уровня.
3. Собственно вставка новой записи и создание соответствующего ей элемента `TTreeNode`.

Сбор данных

Процедура заполнения компонента TreeView1 данными из таблицы станет основой нашей программы. Мы должны позаботиться не только о передаче данных из строк таблицы в соответствующие узлы иерархической структуры, но еще и о корректном построении дерева с сохранением всех связей между элементами (листинг 30.15).

Листинг 30.15. Получение записей из таблицы и построение дерева

```
procedure TfrmMain.UpdateTree;
   {вложенная процедура для поиска родительского узла по ключу Key}
   Function FindParentNode(Key:integer):TTreeNode;
    var i:integer;
    begin
    Result:=nil;
    {перебираем все узлы дерева, пока не встретим узел с нужным ключом}
    for i:=0 to TreeView1.Items.Count-1 do
        if Integer(TreeView1.Items.Item[i].Data^)=Key then
        begin
        Result:=TreeView1.Items.Item[i]; {запрашиваемый узел найден}
        Break; {прерываем цикл}
        end;
    end;

var Obj:^INTEGER;
    Node:TTreeNode;
begin
 TreeView1.Items.BeginUpdate; //начинаем строить дерево
 ClearTree; //очистка дерева и памяти (см. листинг 30.16)

  with DataModule1.qSelect do
  begin //получение данных из таблицы
   Open; //отправляем запрос
     while eof<>true do
     begin //перебираем все записи из набора данных
       New(Obj);
       Obj^:=FieldByName('Tree_Key').AsInteger; {значение ключа}
       {по ключу Par_Key ищем родительский узел}
       Node:=FindParentNode(FieldByName('Par_Key').AsInteger);
       {создаем экземпляр TTreeNode и связываем его с данными Obj}
```

```
        TreeView1.Items.AddChildObject(Node,
                                  FieldByName('Txt').AsString,Obj);
        Next; //к следующей записи набора данных
      end;
    Close; //закрываем набор данных
  end;
 TreeView1.Items.EndUpdate; //построение дерева завершено
end;
```

В цикле `while..do` проводится перебор всех записей, возвращенных в результате выполнения запроса `qSelect`. Метод `New()` резервирует четыре байта памяти для хранения значения первичного ключа записи. Позже ссылка на этот участок памяти будет передана соответствующему узлу `TTreeNode`. Вложенная процедура `FindParentNode()` среди элементов дерева ищет узел с ключом `Key`. Этот узел и станет родительским для очередной записи. Используя метод `AddChildObject()`, создаем новый узел и передаем ему указатель на данные `Obj`. Без комментариев у нас осталась процедура `ClearTree()`, поговорим о ней подробнее.

Очистка данных

Перед завершением работы программы и перед любым обновлением иерархической структуры в компоненте `TTreeView` мы обязаны позаботиться об освобождении памяти, отведенной под хранение дополнительных данных, связанных с узлами дерева. Для этого наш проект необходимо дополнить процедурой `ClearTree()` (листинг 30.16).

Листинг 30.16. Очистка дерева

```
procedure TfrmMain.ClearTree;
var i:integer;
begin
for i:=0 to TreeView1.Items.Count-1 do
     if TreeView1.Items.Item[i].Data<>nil then
            Dispose(TreeView1.Items.Item[i].Data);
TreeView1.Items.Clear;
end;
```

В цикле `for..do` осуществляется перебор всех узлов дерева, для каждого элемента вызывается метод `Dispose()`, очищающий участок памяти, на который ссылается свойство `Data` узла.

Процедура очистки дерева понадобится нам еще в одном месте проекта — ее следует вызывать в момент закрытия главной формы (листинг 30.17).

Листинг 30.17. Закрываем форму с деревом

```
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
ClearTree;
end;
```

Редактирование записи

Для редактирования текста узла воспользуемся событием `OnEdited()` элемента управления `TTreeView`. Напомню, что это событие генерируется после того, как пользователь внесет изменения в текст узла дерева (листинг 30.18).

Листинг 30.18. Редактирование названия узла дерева

```
procedure TfrmMain.TreeView1Edited(Sender: TObject; Node: TTreeNode;
  var S: String);
begin
 with DataModule1.qEditText.Parameters do
 begin
  ParamValues['Txt']:=S;
  ParamValues['Tree_Key']:=INTEGER(Node.Data^);
  DataModule1.qEditText.ExecSQL;
 end;
end;
```

Новый текст содержится в параметре `S` события. Это значение (воспользовавшись услугами компонента `qEditText`) мы и отправим в таблицу.

Для удобства пользователя введем еще один, альтернативный способ редактирования текста узла, воспользовавшись услугами команды `acEdit` (листинг 30.19).

Листинг 30.19. Альтернативный способ редактирования названия узла

```
procedure TfrmMain.acEditExecute(Sender: TObject);
var Node:TTreeNode;
    s:string;
begin
Node:=TreeView1.Selected; {этот узел выбран пользователем}
s:=Node.Text; {узнаем старое название узла}
if InputQuery('Редактируем узел', 'Новое название', s)=true then
    with DataModule1.qEditText do
     begin
       Parameters.ParamValues['Txt']:=S;
       Parameters.ParamValues['Tree_Key']:=INTEGER(Node.Data^);
```

```
      ExecSQL;   {отправляем команду на редактирование}
      Node.Text:=s; {меняем название}
    end;
end;
```

Удаление записи

В иерархических структурах даже процедура удаления ненужной записи не выглядит банально. Основная загвоздка в том, что перед уничтожением узла мы должны позаботиться об удалении из БД всех подчиненных ему элементов (листинг 30.20).

Листинг 30.20. Удаление узла дерева

```
procedure TfrmMain.acDeleteExecute(Sender: TObject);

    Procedure DeleteRecord(Node:TTreeNode);
    {Вложенная процедура удаления строки записи и данных узла}
    var Obj:^INTEGER;
    begin
         Obj:=Node.Data;
         DataModule1.qDelete.Parameters.ParamValues['Tree_Key']:=Obj^;
         DataModule1.qDelete.ExecSQL;
         Dispose(Obj);
    end;

var Node:TTreeNode;
begin
If MessageBox(Handle,PChar('Запись:'+#13+TreeView1.Selected.Text),
PChar('Удаление'),MB_YESNO+MB_ICONQUESTION)=idYes then
  begin
      {пока есть подчиненные записи — вызываем процедуру удаления}
      while TreeView1.Selected.HasChildren=true do
      begin
         Node:=TreeView1.Selected.GetLastChild;
         while Node.HasChildren=true do Node:=Node.GetLastChild;
         DeleteRecord(Node);
      end;
 {после удаления из таблицы всех подчиненных записей
  удаляем родительский узел}
  DeleteRecord(TreeView1.Selected);
  TreeView1.Selected.Delete;
  end;
end;
```

Получив от пользователя подтверждение на удаление узла, мы выясняем, есть ли у него дочерние элементы. Если это так, то удаляем узлы и связанные с ними данные, начиная с самого последнего узла.

Сортировка узлов

Логика работы программы должна позволять изменять очередность следования дочерних узлов одной группы — узлов, принадлежащих одному и тому же родителю. Вот и настал черед обсудить, как организуется сортировка узлов в древовидной структуре. Напомню, что разрабатывая таблицу `TblTree`, специально для этих целей мы зарезервировали в ней отдельное поле `Ind`, предназначенное для хранения числа, определяющего порядок следования узлов в дереве.

Первое наше действие — решение задачи управления данными, хранящимися в этом поле из кода программы. Для этого разместите на модуле данных очередной компонент `TADOQuery` и переименуйте его в `qEditIndex`. Как и все остальные компоненты `TADOQuery` в рассматриваемом примере, проинициализируем новый элемент управления в самом конце события `OnCreate()` модуля данных (листинг 30.21).

Листинг 30.21. Доработка кода создания модуля данных

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
var i:integer;
    s: string;
begin
s:=ExtractFilePath(Application.exename)+'tree.accdb';
…
//НОВАЯ СТРОКА!!!
qEditIndex.sql.add('UPDATE TREE SET Ind=:Ind Where Tree_Key=:Tree_Key');
end;
```

С этого момента благодаря компоненту `qEditIndex` мы получаем возможность изменять значение в поле `ItemIndex` записи, идентифицируемой первичным ключом `TblTree_Key`.

Завершив подготовку компонента `qEditIndex`, переходим к главной форме проекта. Дополните менеджер команд `ActionManager1` новой командой `acUp` с заголовком "Поднять узел" (листинг 30.22). По этой команде выбранный узел дерева `TTreeNode` переместится на одну ступеньку вверх. Команда на подъем узла наверх должна сохранять активность до тех пор, пока над перемещаемым элементом имеется, по крайней мере, один узел, иначе его просто некуда двигать — он достиг своего предела на "карьерной лестнице".

Листинг 30.22. Перевод команды acUp в актуальное состояние

```
procedure TfrmMain.acUpUpdate(Sender: TObject);
begin
acUp.Enabled:=(TreeView1.Selected<>nil)
                and
               (TreeView1.Selected.getPrevSibling<>nil);
end;
```

Для перевода узла на одну позицию вверх потребуется помощь метода `MoveTo()`. В нем укажем, с каким узлом мы будем меняться местами, и режим обмена (`naInsert` — вставка) (листинг 30.23).

Листинг 30.23. Перемещение выбранного узла на одну позицию вверх

```
procedure TfrmMain.acUpExecute(Sender: TObject);
 var CurrentNode, BeforeNode : TTreeNode;
begin
 CurrentNode:=TreeView1.Selected; //выбранный пользователем узел

 {узел, с которым мы меняемся местами}
 BeforeNode:=CurrentNode.GetPrevChild(CurrentNode);
 CurrentNode.MoveTo(BeforeNode,naInsert); //перемещаем узел

{узлы сами пересчитали свои абсолютные индексы,
теперь вносим новые значения в таблицу}
with DataModule1.qEditIndex.Parameters do
begin
  ParamValues['Ind']:=CurrentNode.AbsoluteIndex;
  ParamValues['Tree_Key']:=INTEGER(CurrentNode.Data^);
  DataModule1.qEditIndex.ExecSQL;

  ParamValues['Ind']:=BeforeNode.AbsoluteIndex;
  ParamValues['Tree_Key']:=INTEGER(BeforeNode.Data^);
  DataModule1.qEditIndex.ExecSQL;
end;
end;
```

Подняв узел, нужно позаботиться об изменении содержимого поля `ItemIndex`, хранящего очередность элементов дерева в таблице. Для этого мы дважды обращаемся к запросу `qEditIndex` и изменяем данные для узлов `CurrentNode` и `BeforeNode` — передаем в таблицу значения их абсолютных индексов.

Идея опускания узла на уровень вниз аналогична только что рассмотренной, поэтому на ней останавливаться не станем и сразу перейдем к вопросу переподчинения.

Переподчинение узлов

Пожалуй, с точки зрения программирования, при работе с иерархическими данными вопрос переподчинения узлов наиболее сложный (и соответственно интересный). На первый взгляд, решение задачи элементарно — меняем значение родительского ключа `Par_Key` записи на новое, и узел переподчинен. Именно в этом месте и скрывается подводный камень. Сразу поясню это утверждение на примере. Допустим, что в нашей таблице хранится организационная структура какого-то вымышленного предприятия. Значения записей таблицы, соответствующие этой структуре, приведены в табл. 30.2.

Таблица 30.2. *Значения записей в таблице БД*

TREE_KEY	TXT	PAR_KEY	IND
1	Структура	NULL	1
2	Управление	1	2
3	Дирекция	2	3
4	Генеральный директор	3	4
5	Замдиректора	3	5
6	Гл. инженер	3	6
7	Отдел кадров	2	7
8	Транспортный цех	2	8
9	Начальник цеха	8	9
10	Производство	1	10
11	Склады	1	11
12	Склад расходных материалов	11	12

При заполнении таблицы мною специально внесена логическая ошибка — транспортный цех (`Par_Key=2`) подчинен управлению предприятием (`TblTree_Key=2`). Здравый смысл подсказывает, что этому узлу здесь не место. Допустим, что мы переподчинили запись "Транспортный цех" узлу "Производство", просто поменяв значение поля `Par_Key` с 2 на 10 (табл. 30.3).

Таблица 30.3. *Переподчинение записей в таблице БД*

TREE_KEY	TXT	PAR_KEY	IND
1	Структура	NULL	1
2	Управление	1	2
…	…	…	…
8	Транспортный цех	10	8
9	Начальник цеха	8	9
10	Производство	1	10
…	…	…	…

Что из этого получится? При попытке обновить данные в дереве `TreeView1` в адрес базы данных (см. листинг 30.12) отправляется запрос

```
SELECT * FROM TREE ORDER BY IND
```

требующий возвратить набор данных, упорядоченный по содержимому поля `Ind`. В результирующем наборе запись "Транспортный цех" (`Ind=8`) *следует перед своим новым родительским узлом* "Производство" (`Ind=10`). Соответственно при построении дерева в компоненте `TreeView1` (см. листинг 30.15) мы столкнемся с проблемой: начав обслуживать запись "Транспортный цех", процедура не найдет для нее родительского узла, ведь до записи "Производство" мы пока не добрались, и соответствующий ей узел еще не добавлен в дерево…

Внимание!

При переподчинении узла, кроме замены значения в поле внешнего ключа, необходимо обновить данные в поле сортировки, чтобы упорядочить следование записей в результирующем наборе — *родительские узлы должны обрабатываться ранее, чем дочерние*. Чем больше "расстояние" между старым и новым родительским узлом, тем больше записей придется обработать.

На модуле данных поместите очередной компонент `TADOQuery` и переименуйте его в `qEditParKey`. К процедуре обработки события `OnCreate()` модуля данных `DataModule1` добавьте пару новых строк (листинг 30.24).

Листинг 30.24. Доработка кода создания модуля данных

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
…
//НОВЫЕ СТРОКИ
qEditParKey.sql.add('UPDATE TREE);
qEditParKey.sql.add('SET Par_Key=:Par_Key WHERE Tree_Key=:Tree_Key');
end;
```

Изучив инструкцию SQL, вы поймете, что компонент `qEditParKey` будет отвечать за изменение значения родительского ключа `Par_Key` записи.

С точки зрения интерфейсной части проекта наиболее удобным способом переподчинения узлов дерева будет обычное перетаскивание узла мышью. Для этого в секции частных объявлений формы опубликуйте переменную `DropNode`, в ней мы сохраним ссылку на перетаскиваемый узел:

```
private
    DropNode : TTreeNode;
```

Убедитесь, что у компонента `TreeView1` отключен автоматический режим перетаскивания, — свойство `DragMode` установлено в состояние `dmManual`. Сигналом для начала операции drag-and-drop послужит событие `OnMouseMove()` компонента `TreeView1` (листинг 30.25).

Листинг 30.25. Инициализация операции перетаскивания узла

```
procedure TfrmMain.TreeView1MouseMove(Sender: TObject;
                                      Shift: TShiftState;  X, Y: Integer);
var Node:TTreeNode;
begin
Node:=TreeView1.GetNodeAt(X, Y); //этот узел выбран пользователем
 if Assigned(Node) and (ssLeft in SHIFT) then
    TreeView1.BeginDrag(false,10); //начало перетаскивания
end;
```

Мы проверяем наличие узла в точке с координатами `X` и `Y`. Если таковой имеется, и, кроме того, пользователь удерживает в нажатом состоянии левую кнопку мыши, инициируем перетаскивание — вызываем метод `BeginDrag()`.

Теперь последовательно обрабатываем череду событий `OnStartDrag()`, `OnDragOver()` и `OnDragDrop()`. В первом событии мы лишь передадим в ранее объявленную глобальную переменную `DragNode` — ссылку на перетягиваемый узел (листинг 30.26).

Листинг 30.26. Сохранение ссылки на перетаскиваемый узел

```
procedure TfrmMain.TreeView1StartDrag(Sender: TObject;
                                      var DragObject: TDragObject);
begin
DragNode:=TreeView1.Selected;
end;
```

Задача события `OnDragOver()` — проверка допустимости операции перетаскивания (листинг 30.27).

Листинг 30.27. Проверка допустимости операции перетаскивания

```
procedure TfrmMain.TreeView1DragOver(Sender, Source: TObject;
                X, Y: Integer; State: TDragState; var Accept: Boolean);
var SenderNode:TTreeNode;
begin
SenderNode:=TreeView1.GetNodeAt(X, Y); //узнаем новый родительский узел
//проверяем корректность смены владельца узла
Accept:=(Assigned(SenderNode)) {есть получатель}
        AND
        NOT (SenderNode=DragNode) {попытка подчинить самому себе}
        AND
        NOT (SenderNode.HasAsParent(DragNode)); {родительский узел нельзя подчинить дочернему узлу}
end;
```

Здесь мы контролируем, чтобы пользователь не переподчинил узел самому себе, и запрещаем подчинение родительского узла своему дочернему узлу.

Все наши труды завершаются масштабным событием `OnDragDrop()`, которое решает три задачи: перемещает узел, изменяет содержимое поля `Par_Key` записи и сохраняет новую очередность следования узлов (листинг 30.28).

Листинг 30.28. Подчинение узла новому владельцу и сохранение изменений

```
procedure TfrmMain.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var SenderNode, Node:TTreeNode;
begin
SenderNode:=TreeView1.GetNodeAt(x, y); //узнаем новый родительский узел

if Assigned(SenderNode) and
  (MessageBox(Application.Handle,
   PChar('Подчинить "'+DragNode.Text+'"'+#13+' узлу "'+
          SenderNode.Text +'"?'),
   PChar('Перемещение узла'), MB_YESNO+MB_ICONQUESTION) = ID_Yes)
then
  begin
    screen.Cursor:=crHourgLass;
    //1 перемещаем узел в дереве
    DragNode.MoveTo(SenderNode,naAddChild);
    //2 обновляем внешний ключ записи
    with DataModule1.qEditParKey do
    begin
      ParamValues['Par_Key']:=INTEGER(SenderNode.Data^);
```

```
      ParamValues['Tree_Key']:=INTEGER(DragNode.Data^);
      ExecSQL;
    end;
    //3 обновляем значения в полях сортировки для всех узлов дерева!
    Node:=TreeView1.Items.GetFirstNode;
    repeat
      with DataModule1.qEditIndex do
      begin
        ParamValues['Ind']:=Node.AbsoluteIndex;
        ParamValues['Tree_Key']:=INTEGER(Node.Data^);
        ExecSQL;
      end;
      Node:=Node.GetNext;
    until NOT Assigned(Node);
    screen.Cursor:=crDefault;
  end;
end;
```

Исключительно ради упрощения примера вопрос сохранения в таблице порядка следования узлов решен "в лоб" — мы перебираем все узлы от "А" до "Я" и передаем в поле `ItemIndex` таблицы их новый абсолютный индекс. Это решение вполне подойдет для сравнительно небольших таблиц, ну, скажем, до 100–200 записей. Но если речь идет о тысяче узлов, то нужно позаботиться об улучшении алгоритма.

Наша работа завершена, на рис. 30.6 представлен экранный снимок работающей программы с иерархической структурой.

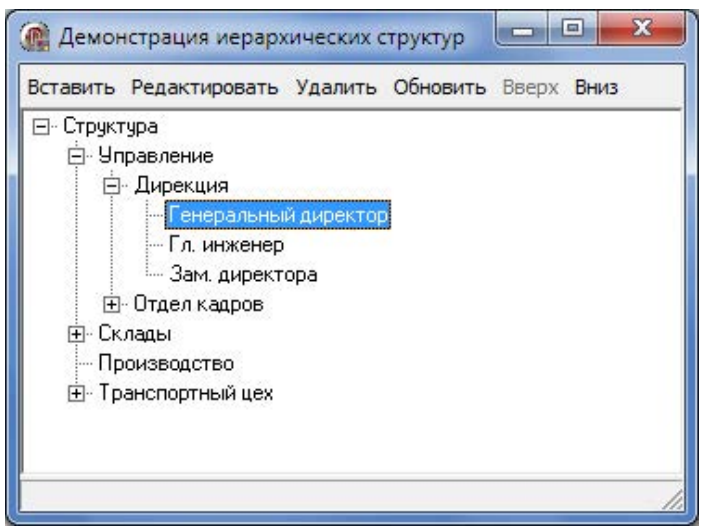
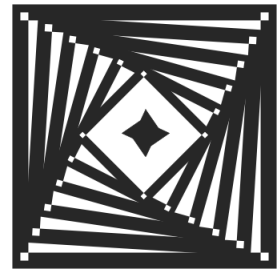


Рис. 30.6. Экранный снимок представления иерархических данных

Резюме

При разработке пользовательского интерфейса приложений баз данных не останавливайтесь только на компонентах со страницы **Data Controls** палитры компонентов. Всегда помните о существовании десятков других элементов управления, способных решать задачи по обслуживанию данных не менее эффективно.

Рассмотренные в этой главе приемы особенно актуальны для приложений баз данных, работающих в режиме однонаправленного курсора, например dbExpress. В подобных приложениях имеются существенные ограничения на перемещение по записям, которые полностью снимаются в описанных здесь примерах.



Глава 31

Многоуровневые БД на основе DataSnap

В *главе 3* уже упоминалось о существовании весьма неординарного подхода к разработке клиент-серверных проектов БД. Его идея заключается во внедрении между сервером и клиентом дополнительного программного уровня — своеобразного интеллектуального моста между удаленными клиентскими приложениями и СУБД.

Мост выступает не просто в качестве банального посредника, перекачивающего битовые потоки из БД к клиенту и обратно. Основная задача промежуточного приложения — облегчить работу клиента, предоставив ему максимально возможный перечень услуг, вплоть до того, что клиент превращается просто в средство просмотра данных (тонкого клиента), а вся программная логика сосредотачивается в приложении-посреднике.

Промежуточный уровень выступает не только помощником клиентского приложения, но и оказывает существенную поддержку серверу БД, перекладывая на себя часть его задач (например, поддержку бизнес-правил, упорядочивание данных, фильтрацию и поиск, обслуживание вычисляемых полей и т. п.). В ситуации, когда СУБД обслуживает большое количество клиентов, любая помощь, снимающая часть вычислительной нагрузки, не окажется лишней.

Обсуждая плюсы многоуровневых проектов, отметим, что наличие дополнительного логического звена в цепи "БД – клиент" существенно упрощает сопровождение системы в целом. Благодаря тому что в тонких клиентах присутствует минимум программной логики (нет кода — нет ошибок), разработчик ПО промежуточного уровня практически не отвлекается на выпуск обновлений для пользовательских компьютеров. Есть возможность сосредоточить основное внимание на совершенствовании промежуточного ПО.

В современных версиях Delphi предусмотрено несколько вариантов создания многоуровневых проектов, мы рассмотрим самую перспективную на сегодня технологию — DataSnap.

Замечание

Корпорация Embarcadero далеко не одинока в своем стремлении обеспечить прозрачный межсетевой обмен для осуществления распределенных вычислений и обработки данных. В корпорации Microsoft давно эксплуатируется программная архитектура распределенного COM (Distributed Component Object Model, DCOM), а сейчас набирает обороты .NET. С Microsoft давно соперничает консорциум Object Management Group (OMG), который разработал и активно продвигает архитектуру распределенных вычислений, получившую название Common Object Request Broker Architecture (CORBA).

Архитектура трехзвенного проекта БД DataSnap

Разработчик многоуровневого приложения БД на базе технологии DataSnap столкнется с тремя его элементами (рис. 31.1):

- сервером СУБД;
- сервером приложений;
- клиентскими приложениями.

В качестве сервера СУБД может использоваться практически любой современный сервер баз данных, будь то фирменные разработки Embarcadero — Interbase и Blackfish SQL или продукты других ведущих компаний.

Сервер приложений DataSnap представляет собой отдельный модуль, реализованный в виде обычного приложения для Windows, консольного приложения или службы Windows. Модуль сервера приложений может разворачиваться на том же хосте, на котором установлена СУБД, или на станции, имеющей сетевое подключение к хосту с СУБД. В любом случае, в современных версиях Delphi передачу вызовов между сервером СУБД и сервером приложений DataSnap целесообразно осуществлять с помощью компонента `TSQLConnection`. Это наш давний знакомый, подробно рассмотренный в *главе 27*.

Концептуально сервер приложений состоит из двух взаимодействующих модулей: модуля контейнера сервера и модуля методов сервера. Хотя оба построены на платформе обычного для БД модуля данных `TDataModule`, между ними есть существенная разница. Контейнер сервера `TServerContainer` — весьма неординарный класс. Если при создании проекта вы укажете, что сервер должен представлять собой консоль или классическое приложение с формой VCL, то мастер построения сервера DataSnap создаст `TServerContainer` на основе обычного модуля данных `TDataModule`. Но если мы в самом начале проектирования объявим, что создаем службу Windows, то в цепочку наследования контейнера попадет еще один класс — `TService` (рис. 31.2).

Контейнер сервера вместе с располагаемыми на нем компонентами отвечает за взаимодействие с удаленными клиентскими приложениями, для этого задействуется стек протоколов TCP/IP или более высокоуровневый протокол HTTP. При использо-

вании протокола передачи гипертекста кроме собственно организации обмена данными мы приобретаем дополнительные возможности по аутентификации пользователя. Выбор того или иного протокола обмена в первую очередь определяется условиями функционирования нашей БД. Если все клиентские приложения и сервер приложений сосредоточены в рамках локальной сети предприятия, то стоит остановить свой выбор на транспорте TCP/IP. Если в качестве опорной сети задействуются необъятные просторы Интернета, то ради повышения безопасности используем всю мощь HTTP и возможности компонента `TDSHTTPServiceAuthenticationManager` (явно претендующего на попадание в книгу рекордов Гиннеса в номинации "за самое длинное название"). Компонент, название которого занимает полстроки, служит для проверки подлинности клиента. При смешанном характере доступа к ресурсам БД технология DataSnap допускает создание приложений, поддерживающих оба протокола одновременно.

По сравнению с контейнером, модуль методов сервера существенно дальше продвинулся по иерархии VCL — он впитал в себя генетические черты еще трех промежуточных классов. Основная задача многочисленных родственников класса `TServerMethods` — упростить динамический вызов методов.

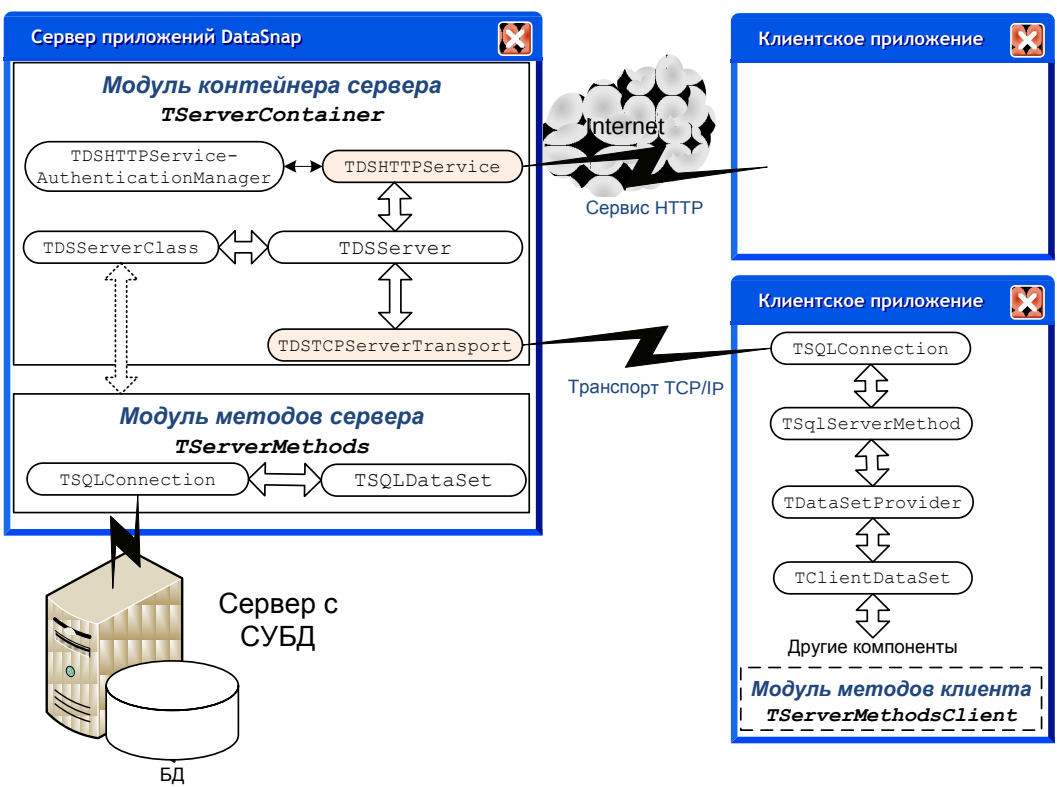


Рис. 31.1. Состав многоуровневого приложения БД на базе DataSnap в Delphi 2010/XE

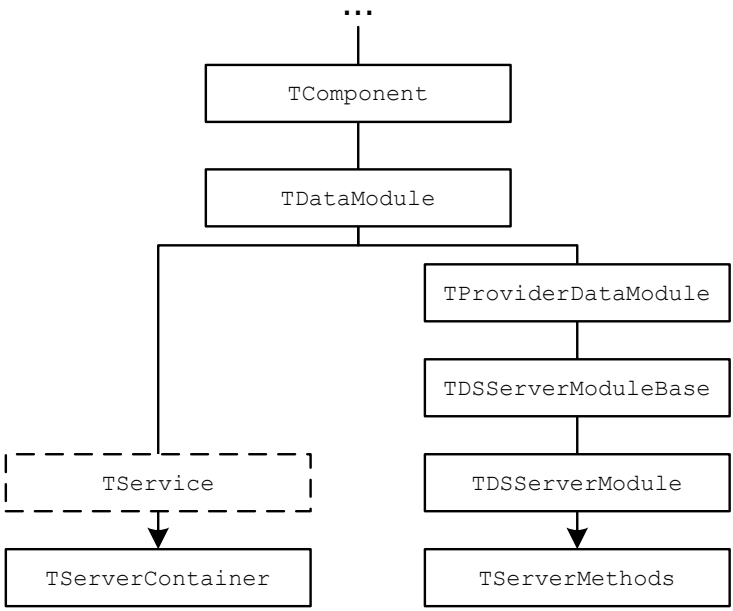


Рис. 31.2. Иерархия построения модулей `TServerContainer` и `TServerMethods`

Модуль методов сервера решает две основные задачи. Во-первых, он служит обычной площадкой для невизуальных компонентов, в частности, на нем располагают уже упоминавшийся ранее компонент-соединение `TSQLConnection` (отвечающий за подключение сервера приложений к БД) и компоненты-наборы данных dbExpress. Во-вторых, в исходном коде модуля объявляются и описываются дополнительные методы сервера, которые могут быть предоставлены в распоряжение клиента, например, сервер в состоянии помочь клиентам осуществлять сложные математические расчеты. Таким образом, модуль методов сервера представляет собой логическое ядро сервера приложений DataSnap.

Нам осталось дать общую характеристику третьего ингредиента проекта баз данных DataSnap в Delphi 2010 и Delphi XE — клиентского приложения. Для того чтобы клиентское приложение получило возможность обращаться к методам сервера, у клиента создается свой собственный модуль методов (см. рис. 31.1), выступающий логическим проводником идей модуля методов сервера. Если в модуле сервера объявляется функция с условным названием `MyFunction()`, то в модуле методов клиента должна быть объявлена одноименная функция с аналогичным перечнем параметров. Клиентское приложение обращается к своему методу, даже не подозревая, что технология DataSnap позаботится перенаправить вызов к серверу приложений. Сервер обработает запрос клиента и возвратит результат обратно в вызывающую функцию клиента. Сразу заострю ваше внимание на том, что совпадение заголовков методов сервера и клиента ни в коем случае не говорит об идентичности реализации этих методов. Код метода сервера и метода клиента различаются на все сто процентов, но разговор об этом мы также ненадолго отложим, ведь

вводная часть немного затянулась, и нам уже давно пора приступить к обсуждению компонентов проекта DataSnap.

ЗАМЕЧАНИЕ

При построении клиента DataSnap могут использоваться два различных подхода: клиент на базе компонента `TSqlServerMethod` и клиент на основе провайдера `TDSProvidrConnection`. Второй способ предназначен для упрощения перевода старых приложений на новую версию DataSnap.

Сервер *TDSServer*

Основу сервера приложений составляет компонент `TDSServer`, — своего рода "диспетчерский центр" всего сервера. С одной стороны, компонент отвечает за организацию связи между сервером DataSnap и его клиентами, для этого компонент управляет информационными потоками TCP/IP и HTTP. С другой стороны, `TDSServer` несет ответственность за предоставление в распоряжение клиентов методов сервера, для этого он обеспечивает пересылку запросов клиентов в модуль методов сервера.

Класс описан в модуле DSServer, а соответствующий ему компонент вы обнаружите на страничке **DataSnap Server** палитры компонентов Delphi. Во время разработки проекта компонент следует разместить в модуле контейнера сервера `TServerContainer`.

Несмотря на столь ответственный пост, `TDSServer` весьма скромен. На страничке **Properties** инспектора объектов опубликовано всего четыре свойства. Самое главное из них:

```
property AutoStart: Boolean; //по умолчанию true
```

В состоянии `true` это свойство обеспечит автоматический старт сервера сразу после создания экземпляра класса. Кроме того, для запуска и останова сервера могут применяться методы

```
procedure Start;
procedure Stop;
```

О факте успешного запуска можно судить по состоянию свойства

```
property Started: Boolean;
```

Второе интересующее нас свойство

```
property HideDSAdmin: Boolean; //по умолчанию false
```

определяет степень доверия сервера приложений к подключенным к нему клиентам. По умолчанию сервер разрешает клиентам обращаться к встроенным в него методам администрирования `DSAdmin`, но при желании компонент может и "повредничать", установив в свойство значение `false`. Как следствие, клиенты DataSnap окажутся на "голодном пайке". Встроенных методов `DSAdmin` всего дюжина, они предназначены для низкоуровневого управления сервера DataSnap (табл. 31.1).

Таблица 31.1. *Методы* `TDSAdminMethods`

Название	Описание
`CreateServerClasses`	Создание класса сервера
`CreateServerMethods`	Создание метода сервера
`FindClasses`	Поиск классов
`FindMethods`	Поиск методов
`FindPackages`	Поиск программных пакетов
`GetPlatformName`	Получение имени платформы
`GetServerClasses`	Получение классов сервера
`GetServerMethods`	Получение перечня методов сервера
`DropServerClasses`	Удаление класса сервера
`DropServerMethods`	Удаление методов сервера
`GetDatabaseConnectionProperties`	Свойства соединения с БД

Основное назначение всех событий, имеющихся в распоряжении компонента `TDSServer`, — информирование о поведении клиентских приложений. В момент подключения к серверу очередного клиента генерируется событие

```
property OnConnect: TDSConnectEvent;
```

```
TDSConnectEvent = procedure(DSConnectEventObject: TDSConnectEventObject) of
object;
```

С отключением клиента также связано отдельное событие

```
property OnDisconnect: TDSConnectEvent;
```

Обратите внимание на тот факт, что оба события возвращают ссылку на экземпляр класса `TDSConnectEventObject`. В недрах этого объекта скрывается весьма полезная информация об особенностях соединения с клиентом (свойство `ConnectProperties`) и характеристики транспортного канала между сервером и клиентом (свойство `ChannelInfo`).

Для ведения протокола работы сервера нам понадобится осуществить ряд подготовительных мероприятий. В первую очередь объявим глобальную файловую переменную

```
var LogFile:Textfile;
```

которая станет ссылаться на текстовый файл с информацией о работе сервера.

Подключение к файлу протокола производим в момент создания модуля контейнера так, как это продемонстрировано в листинге 31.1.

Листинг 31.1. Связывание файловой переменной с файлом протокола

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
begin
  AssignFile(LogFile, 'datasnap.log');
```

```
  if FileExists('datasnap.log')=true then
     Append(LogFile) else
     Rewrite(LogFile);
end;
```

В момент завершения работы сервера (листинг 31.2) разрываем связь с файловой переменной.

Листинг 31.2. Освобождение файловой переменной

```
procedure TServerContainer1.DataModuleDestroy(Sender: TObject);
begin
CloseFile(LogFile); //закрываем файл
end;
```

В строках листинга 31.3 отражено самое главное — код обработки события, возникающего в момент подключения к серверу очередного клиента.

Листинг 31.3. Вносим в протокол данные о соединениях

```
procedure TServerContainer1.DSServer1Connect(
                          DSConnectEventObject: TDSConnectEventObject);
var i:integer;
begin
  WriteLn(LogFile, DateTimeToStr(now)+
                  ' соединение '+DSConnectEventObject.ChannelInfo.Info);
  WriteLn(LogFile,'================================');
  i:=DSConnectEventObject.ConnectProperties.Count;
  for i:=0 to DSConnectEventObject.ConnectProperties.Count- 1 do
       WriteLn(LogFile,
               DSConnectEventObject.ConnectProperties.Properties[i]);
 WriteLn(LogFile,'================================');
end;
```

На рис. 31.3 представлен экранный снимок блокнота с открытым в нем файлом протокола. Как видите, мы получили достаточно много полезной информации об IP-адресе клиента, номере порта и других технических характеристиках соединения.

После того как клиентское приложение обращается к какому-либо методу сервера, на сервере возникает событие

```
property OnPrepare: TDSPrepareEvent;
TDSPrepareEvent = procedure (DSPrepareEventObject: TDSPrepareEventObject) of
object;
```

В этом обработчике события программист может получить данные о подготовке к выполнению метода. Как минимум, мы сможем выяснить имя вызываемого метода `MethodAlias`.

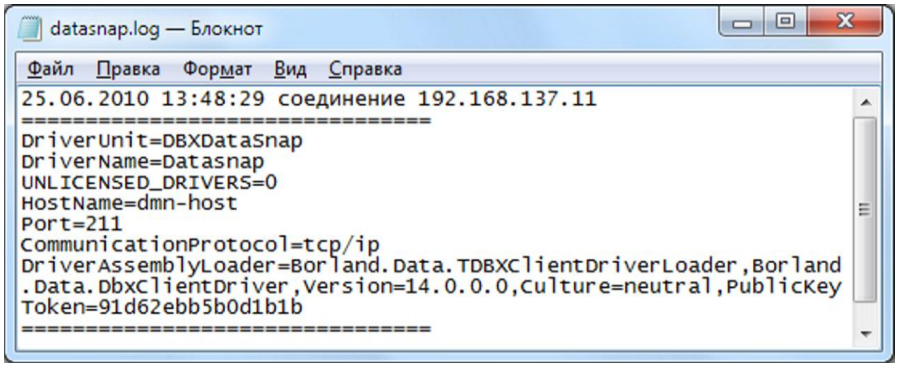


Рис. 31.3. Информация о подключении клиента к серверу DataSnap

Заключительное событие

```
property OnError: TDSErrorEvent;
TDSErrorEvent = procedure (
                    DSErrorEventObject: TDSErrorEventObject) of object;
```

генерируется при возникновении на сервере исключительных ситуаций.

Если вы нуждаетесь в очень подробной трассировке работы сервера, то обязательно обратите внимание на событие

```
property OnTrace: TDBXTraceEvent;
TDBXTraceEvent = function (TraceInfo: TDBXTraceInfo): CBRType of object;
```

Основной параметр события — запись `TDBXTraceInfo`. Она содержит четыре поля, которые мы уже рассмотрели при изучении компонента `TSQLMonitor` (см. *главу 28*).

Внимание!

Если вы планируете воспользоваться помощью события `OnTrace()`, проверьте, чтобы к строке `uses` модуля были подключены файлы `DBXCommon` и `DBCommonTypes`.

Для вывода результатов трассировки можно задействовать листинг 31.4.

Листинг 31.4. Трассировка

```
function TServerContainer1.DSServer1Trace(
                                    TraceInfo: TDBXTraceInfo): CBRType;
begin
  WriteLn(LogFile, 'Трассировка '+TraceInfo.CustomCategory);
  WriteLn(LogFile, #9+TraceInfo.Message);
  Result:=cbrUSEDEF; //действие по умолчанию
end;
```

В результате вся дополнительная информация будет направляться в файл протокола.

Класс сервера *TDSServerClass*

В проектах DataSnap компонент `TDSServerClass` несет прямую ответственность за предоставление опубликованных на сервере методов клиентским приложениям. Компонент располагается в контейнере сервера и подключается к компоненту `TDSServer` с помощью свойства

```
property Server;
```

Еще одно свойство, которому следует уделить внимание, определяет особенности жизненного цикла экземпляра класса `TDSServerClass`

```
property LifeCycle; //по умолчанию Session
```

Существуют три варианта настройки жизненного цикла компонента: `Session`, `Server` и `Invocation`. По умолчанию свойство устанавливается в состояние `Session`, т. е. в момент подключения к серверу клиентского приложения специально для него формируется отдельный объект `TDSServerClass`. Созданный объект обслуживает клиента на протяжении всей сессии и уничтожается сразу после отключения клиентского приложения от сервера. Перевод жизненного цикла в режим `Server` принципиально поменяет работу приложения, сервер будет обладать одним-единственным экземпляром `TDSServerClass`. И этот экземпляр постарается обслуживать все входящие подключения. Наконец, состояние `Invocation` может пригодиться в тех случаях, когда вы планируете, чтобы экземпляр объекта `TDSServerClass` появлялся на свет для обслуживания каждого обращения к методу сервера и выгружался из памяти, сразу после выполнения метода. Таким образом, каждый очередной вызов метода обслуживается новым экземпляром `TDSServerClass`.

Замечание

Для разработки приложений баз данных наиболее предпочтителен сессионный режим. Принцип "отдельный клиент — отдельная сессия" позволяет одновременно обслуживать несколько входящих заявок и упрощает контроль работы клиента на протяжении всей сессии.

Наиболее примечательное событие компонента `TDSServerClass`

```
property OnGetClass: TDSGetClassEvent;
TDSGetClassEvent = procedure (DSServerClass: TDSServerClass;
                              var PersistentClass: TPersistentClass) of object;
```

подготавливает появление на свет очередного экземпляра класса `TDSServerClass`, определяя особенности жизненного цикла экземпляра класса. Для того чтобы упростить работу программиста, в Delphi 2010 предусмотрен мастер создания проекта сервера DataSnap. Мастер самостоятельно добавит в код обработчика события `OnGetClass()` всего одну строку (листинг 31.5), указывающую на то, что создаваемый экземпляр класса должен повторять настройки компонента, расположенного в контейнере сервера.

Листинг 31.5. Создание экземпляра класса `TDSServerClass`

```
procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TServerMethods1;
end;
```

Если вы работаете в Delphi 2009, то код, приведенный в листинге 31.5, следует обязательно написать вручную.

Процесс создания и уничтожения объекта `TDSServerClass` сопровождает пара событий:

```
property OnCreateInstance: TDSCreateInstanceEvent;
property OnDestroyInstance: TDSDestroyInstanceEvent;
```

Их можно использовать, например, для подсчета числа обращений к методам.

Заключительное событие

```
property OnPrepare: TDSPrepareEvent;
```

позволяет управлять процессом подготовки метода сервера к выполнению.

Обмен данными между клиентом и сервером, компоненты *TDSTCPServerTransport* и *TDSHTTPService*

При построении транспортных каналов между клиентами и сервером приложений Delphi предлагает разработчику сделать выбор из двух вариантов обмена данными:

- низкоуровневое соединение на основе стека протоколов TCP/IP с применением компонента `TDSTCPServerTransport`;
- высокоуровневый сервис на базе протокола HTTP с применением компонента `TDSHTTPService`.

Соединение на основе стека TCP/IP наиболее универсальное. На сегодняшний день данный протокол доминирует как в локальных, так и в глобальных сетях, поэтому приложение, разработанное на основе компонента `TDSTCPServerTransport`, сохранит работоспособность как в сети предприятия, так и на просторах Интернета. Однако следует понимать, что, как только ваш трафик выйдет за пределы родной сети предприятия, он станет предметом потенциальных атак со стороны множества недоброжелателей. Снизить риск несанкционированного доступа к данным позволяет организация межсетевого обмена через высокоуровневый протокол передачи гипертекста. Протокол HTTP также базируется на стеке TCP/IP, но обладает более развитыми возможностями по обеспечению безопасности передачи данных.

Поддержка сервиса HTTP осуществляется силами компонента `TDSHTTPService`, при необходимости повысить безопасность на помощь можно призвать компонент `TDSHTTPServiceAuthenticationManager`.

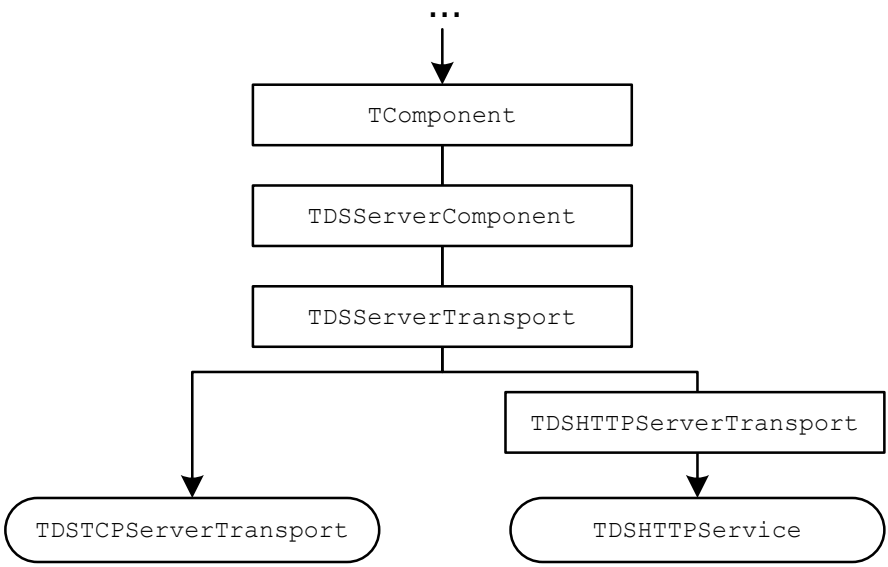


Рис. 31.4. Иерархия наследования компонентов `TDSTCPServerTransport` и `TDSHTTPService`

При построении компонентов, ответственных за организацию транспортного канала, разработчики Delphi использовали общую цепочку наследования VCL (рис. 31.4). Благодаря наличию одинаковых предков, оба компонента приобрели несколько схожих черт. Во-первых, это свойство, с помощью которого компоненты подключаются к серверу `TDSServer`:

```
property Server;
```

Во-вторых, коллекция фильтров, определяющих особенности процесса передачи потока байтов:

```
property Filters;
```

С Delphi поставляется фильтр ZLibCompression, позволяющий сжимать поток данных между сервером и клиентом. Кроме того, можно создавать свои собственные транспортные фильтры. Индивидуальные черты компонентов приведены в табл. 31.2 и 31.3.

***Таблица 31.2.** Параметры соединения `TDSTCPServerTransport`*

Свойство	Назначение
`property Port: Integer;`	Номер порта, к которому подключается клиент. По умолчанию за DataSnap закреплен порт 211

Таблица 31.2 *(окончание)*

Свойство	Назначение
property MaxThreads: Integer;	Максимальное число одновременно поддерживаемых потоков. По умолчанию ограничения отсутствуют — свойство установлено в 0
property PoolSize: Integer;	Размер пула соединений. По умолчанию равно 10. Свойство должно быть настроено до старта сервера
property BufferKBSize;	Размер буфера (в килобайтах) для кэширования операций чтения и записи. По умолчанию равен 32

Компонент `TDSTCPServerTransport` не обладает обработчиками событий.

Таблица 31.3. *Параметры соединения* `TDSHTTPService`

Свойство	Назначение
property HttpPort: Word;	Номер прослушиваемого порта. По умолчанию за HTTP закреплен порт 80. Если на компьютере уже используется служба HTTP, то номер порта следует изменить
property Active: Boolean;	Включение и отключение соединения
property RESTContext: String	Определяет URL-адрес для вызова сервера в качестве службы REST[1]
property DSHostname: String;	Альтернатива свойству Server. Обеспечивает способ подключения к серверу DataSnap по имени
property DSPort: Integer;	Используется только совместно со свойством `DSHostname`. Определяет номер порта
property AuthenticationManager: TDSHTTPServiceAuthenticationManager;	Свойство для подключения менеджера аутентификации, обеспечивающего проверку подлинности информации, поступающей на сервер
property ServerSoftware: String;	Свойство информирует о программном обеспечении сервера, доступно только для чтения

[1] Репрезентативная передача состояния (Representational State Transfer, REST) — архитектурный стиль, позволяющий клиентам обращаться к ресурсам сервера.

На вооружении компонента TDSHTTPService имеется пять обработчиков событий. Из них четыре обслуживают стиль доступа REST и поэтому для относительно несложных многоуровневых приложений БД существенного интереса не представляют. Оставшееся событие:

```
property Trace: TDSHTTPServiceTraceEvent;
TDSHTTPServiceTraceEvent = procedure(Sender: TObject;
                                     AContext: TDSHTTPContext;
                                     ARequest: TDSHTTPRequest;
                                     AResponse: TDSHTTPResponse) of object;
```

специализируется на трассировке работы компонента. Оно может пригодиться, если обмен между клиентом и сервером осуществляется в формате HTTP.

Аутентификация, *TDSHTTPServiceAuthenticationManager*

При организации сетевого обмена с помощью протокола HTTP в сервере DataSnap имеет смысл использовать компонент TDSHTTPServiceAuthenticationManager, позволяющий проверять подлинность передаваемых данных.

Чтобы пройти процедуру проверки подлинности, клиентское приложение должно передать в адрес сервера имя и пароль пользователя. Для этого предназначены свойства DSAuthUser и DsAuthPassword соединителя TSQLConnection.

За организацию проверки отвечает событие

```
property HTTPAuthenticate: TDSHTTPAuthenticationEvent;
```

Во время проверки кроме имени и пароля пользователя, можно проконтролировать протокол и контекст:

```
TDSHTTPAuthenticationEvent = procedure(Sender: TObject;
                                       const Protocol: UnicodeString;
                                       const Context: UnicodeString;
                                       const User: UnicodeString;
                                       const Password: UnicodeString;
                                       var valid: boolean) of object;
```

В перечне параметров события основную роль играет переменная valid, передав в нее true, мы укажем, что проверка подлинности прошла успешно. Иначе в адрес клиента будет отправлено сообщение об ошибке аутентификации — HTTP/1.1 201 Unauthorized.

Метод сервера *TSqlServerMethod*

Компонент TSQLServerMethod предназначен для вызова методов сервера DataSnap и возвращения полученных результатов в клиентское приложение. Осо-

бенность компонента в том, что он построен на основе набора данных `TCustomSQLDataSet` и, как следствие, обладает всеми необходимыми навыками общения с наборами данных.

Компонент подключается к БД точно так же, как и все наборы данных dbExpress — с помощью свойства

```
property SQLConnection:TSQLConnection;
```

Затем следует указать имя интересующего нас метода в свойстве

```
property ServerMethodName: UnicodeString;
```

В предоставляемом свойством списке мы обнаружим не только объявленные нами методы, но и:

- перечень административных методов `DSAdmin`;
- методы предоставления метаданных `DSMetadata`;
- методы интерфейса `IAppServer`, необходимые для обеспечения совместимости с устаревшим парком приложений DataSnap.

Входные и выходные параметры передаются с помощью свойства

```
property Params:TParams;
```

Заполнив необходимые параметры, даем команду на выполнение метода

```
procedure ExecuteMethod;
```

Своих собственных обработчиков событий у компонента нет, а имеющиеся события унаследованы от классов-предков.

Пример проекта DataSnap

После того как мы с вами получили все необходимые теоретические знания для самостоятельной разработки проекта DataSnap, перейдем к практике. В качестве целевой СУБД предлагаю выбрать MySQL, тем более что на основе этой СУБД ранее мы уже подготовили базу данных книжного магазина. Предлагаемый вашему вниманию пример достаточно универсален, поэтому вместо MySQL SQL можете воспользоваться любой другой предпочтительной для вас СУБД, от обычной настольной, до многопользовательской.

Нашу работу начнем с создания сервера приложений DataSnap. Для этого выбираем пункт меню **File | Other** и в окне **New Items** находим пиктограмму **DataSnap Server** (рис. 31.5). Щелчок по кнопке **OK** активизирует деятельность мастера по созданию сервера приложений.

На экране вашего компьютера отобразится окно **New DataSnap Server**, позволяющее настроить базовые параметры проекта (рис. 31.6). Начнем с определения типа приложения. Delphi предлагает три варианта будущего сервера: классическое приложение VCL, консольное приложение и сервис Windows. К несчастью, варианты обычного приложения VCL или консоли не вполне подходят для реализации наших планов. Причина в следующем — сервер приложений должен находиться в постоянной готовности к обслуживанию запросов клиентов. Однако если сервер реализован в формате обычного приложения, то для его запуска (даже если прило-

жение установлено в автозагрузку) необходим вход пользователя в систему Windows, а это не всегда возможно. Поэтому в приложениях баз данных единственным верным выбором становится разработка сервиса — Service Application. После соответствующей настройки сервисы Windows стартуют автоматически, вместе с запуском системы, и в их работу не требуется никакого внешнего вмешательства.

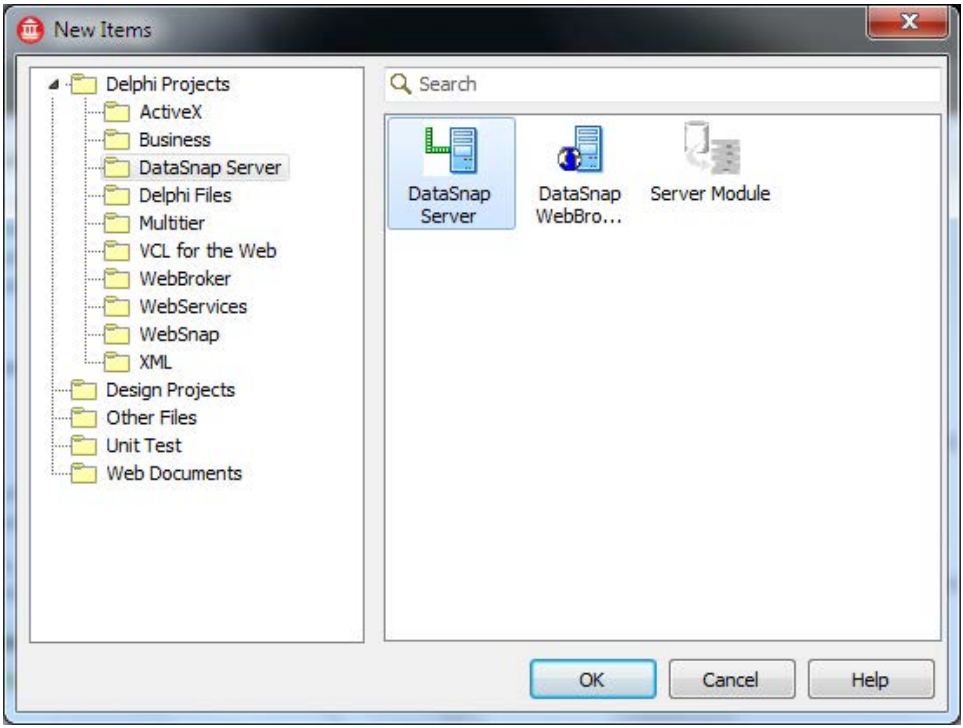


Рис. 31.5. Вызов мастера проекта **DataSnap Server**

Внимание!

Если вы начинающий программист, то свои первые шаги в DataSnap лучше сделайте в проекте VCL Forms, это значительно упростит отладку проекта. Но еще раз подчеркну, что полноценный сервер приложений баз данных DataSnap должен быть реализован только в качестве сервиса Windows.

Второй элемент настройки определяет выбор коммуникационного протокола, обеспечивающего взаимодействие между сервером и клиентами DataSnap. В примере мы воспользуемся всеми предложенными мастером услугами и включим поддержку как TCP/IP, так и HTTP. Постараемся не забыть и про поддержку аутентификации, поставив "галочку" рядом со словом **Authentication**.

На заключительном этапе нам следует настроить порядок создания класса методов сервера. Мастер предлагает выбрать один из трех отправных классов: `TDSServerModule`, `TDataModule` или `TPersistent`. Наилучшее решение установлено

по умолчанию — `TDSServerModule`. Согласимся с ним, и тогда в дальнейшем у нас не возникнет никаких проблем с динамическим вызовом методов сервера.

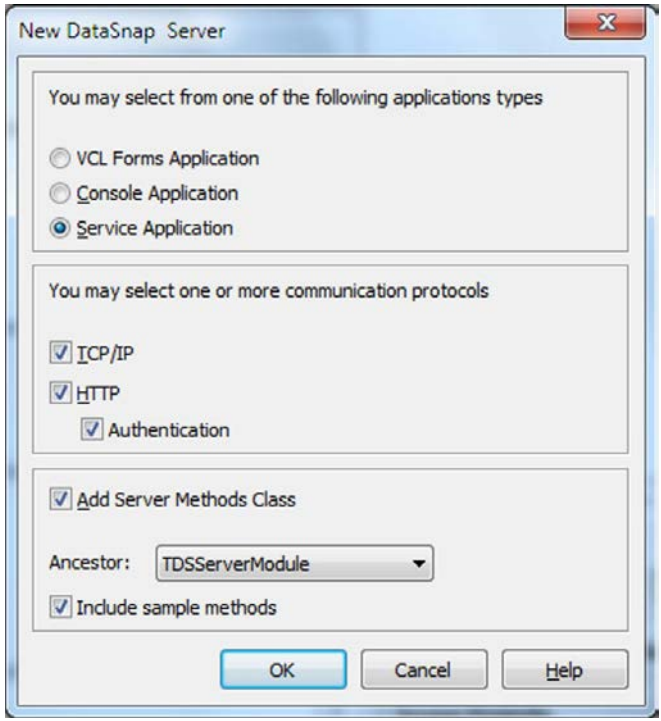


Рис. 31.6. Настройка параметров сервера приложений

Кроме того, мастер советует нам создать один демонстрационный метод сервера (об этом напоминает надпись **Include sample methods**). Мы не станем возражать против такой помощи, метод-пример поможет нам быстрее понять порядок объявления методов сервера.

Щелчок по кнопке **OK** приведет к созданию проекта сервера приложений DataSnap со следующим перечнем модулей:

- Модуль методов сервера `ServerMethods1` и соответствующий ему программный модуль `ServerMethodsUnit1.pas`.
- Контейнер сервера `ServerContainer1` с модулем `ServerContainerUnit1.pas`. Заметьте, что мастер подготовил к работе все компоненты, необходимые для создания сервера (рис. 31.7).
- Головной модуль будущего сервиса — `Project1`.

Сохраните проект в отдельном каталоге, пока не переименовывая модули, единственное исключение сделаем для имени головного файла проекта — назовем его `DataSnapService.dproj`.

Изучите код модуля методов `ServerMethodsUnit1.pas`, приведенный в листинге 31.6.

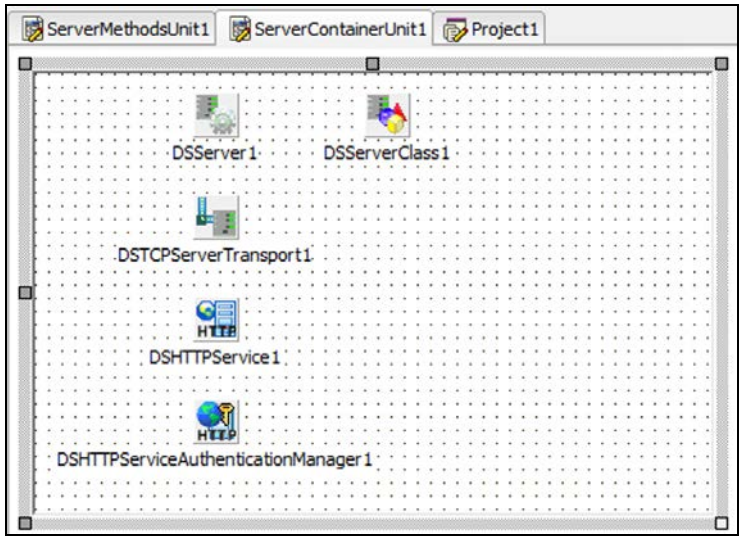


Рис. 31.7. Модуль контейнера сервера DataSnap

Листинг 31.6. Код модуля методов `ServerMethodsUnit1`

```
unit ServerMethodsUnit1;
interface

uses
  SysUtils, Classes, DSServer;

type
  TServerMethods1 = class(TDSServerModule)
  private
    { Private declarations }
  public
    { Public declarations }
    function EchoString(Value: string): string;//демонстрационный метод
  end;

implementation

{$R *.dfm}

function TServerMethods1.EchoString(Value: string): string;
begin
```

```
  Result := Value; //метод работает в качестве эхо
end;
end.
```

В код модуля с нашего согласия мастер включил демонстрационный метод `EchoString()`. Задача метода заключается в "передразнивании" клиентского приложения — метод возвращает клиенту DataSnap, полученную от него же текстовую строку.

Модуль контейнера сервера `ServerContainerUnit1.pas` имеет существенно большее наполнение (листинг 31.7), но также не содержит никаких "подводных камней". Самое первое, на что следует обратить внимание, — контейнер `TServerContainer1` создается на основе класса `TService`. Это опорный класс для разработки служб Windows.

Листинг 31.7. Модуль контейнера сервера `ServerContainerUnit1`

```
unit ServerContainerUnit1;

interface
uses
  SysUtils, Classes, SvcMgr, DSTCPServerTransport,  DSHTTPCommon, DSHTTP,
  DSServer, DSCommonServer;

type
  TServerContainer1 = class(TService) {контейнер на основе класса-службы}
    DSServer1: TDSServer;
    DSTCPServerTransport1: TDSTCPServerTransport;
    DSHTTPService1: TDSHTTPService;
    DSHTTPServiceAuthenticationManager1:
                          TDSHTTPServiceAuthenticationManager;
    DSServerClass1: TDSServerClass;
    procedure DSServerClass1GetClass(DSServerClass: TDSServerClass;
                           var PersistentClass: TPersistentClass);
    procedure ServiceStart(Sender: TService; var Started: Boolean);
  private
    { Private declarations }
  protected
    function DoStop: Boolean; override;     {остановка службы}
    function DoPause: Boolean; override;    {приостановка службы}
    function DoContinue: Boolean; override;{продолжение выполнения}
    procedure DoInterrogate; override;      {состояние службы}
```

```
  public
    function GetServiceController: TServiceController; override;
  end;

var ServerContainer1: TServerContainer1;

implementation
uses Windows, ServerMethodsUnit1;

{$R *.dfm}

procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TServerMethods1;
end;
procedure ServiceController(CtrlCode: DWord); stdcall;
begin
  ServerContainer1.Controller(CtrlCode);
end;
function TServerContainer1.GetServiceController: TServiceController;
begin
  Result := ServiceController;
end;

function TServerContainer1.DoContinue: Boolean;//продолжить
begin
  Result := inherited;
  DSServer1.Start;
end;
procedure TServerContainer1.DoInterrogate; //состояние службы
begin
  inherited;
end;
function TServerContainer1.DoPause: Boolean; //приостановка службы
begin
  DSServer1.Stop;
  Result := inherited;
end;
function TServerContainer1.DoStop: Boolean; //остановка службы
begin
```

```
  DSServer1.Stop;
  Result := inherited;
end;
procedure TServerContainer1.ServiceStart(Sender: TService;
                              var Started: Boolean); //старт службы
begin
  DSServer1.Start;
end;
end.
```

Защищенные методы `ServiceStart()`, `DoStop()`, `DoPause()`, `DoContinue()` и `DoInterrogate()` определяют поведение службы во время запуска, остановки, приостановки, при возобновлении выполнения после приостановки или при получении запроса о состоянии службы. Единственный опубликованный метод `GetServiceController()` возвращает экземпляр контроллера. Кроме того, уже знакомый нам метод `DSServerClass1GetClass()` отвечает за создание новых экземпляров класса `TServerMethods1`, которые будут появляться на свет для обслуживания каждого входящего клиентского подключения.

Перейдите к модулю контейнера `ServerContainer1` и откройте страничку его свойств в инспекторе объектов. При создании сервиса *менеджер системных служб* (Service Control Manager, SCM) присвоит ему точно такое же физическое имя, как и название модуля контейнера. Согласитесь, что имя "`ServerContainer1`" не вполне раскрывает суть создаваемой нами службы, поэтому переименуем название модуля контейнера:

```
Name:='DSService';
```

ВНИМАНИЕ!

Если сервер приложений DataSnap создается в виде службы Windows, то физическое имя службы окажется точно таким же, как и имя контейнера.

Свойство `DisplayName` определяет название сервиса, отображаемое в колонке "Имя" консоли управления службами. Предлагаю остановиться на названии `DataSnapDemoService`. Проверьте, чтобы свойство `StartType` было установлено в режим `stAuto`, это обеспечит автоматический запуск сервиса.

Одновременно нажав клавиши <Ctrl>+<F9>, откомпилируйте проект.

Регистрация службы

Теперь нам необходимо зарегистрировать полученный EXE-файл в качестве службы Windows. Для регистрации сервиса в операционной системе, владеющее службой приложение должно быть запущено из командной строки с ключом `/INSTALL`, например:

```
C:\DataSnapService\DataSnapService.exe /install
```

Проверьте, чтобы сервис появился в окне служб вашего компьютера (рис. 31.8).

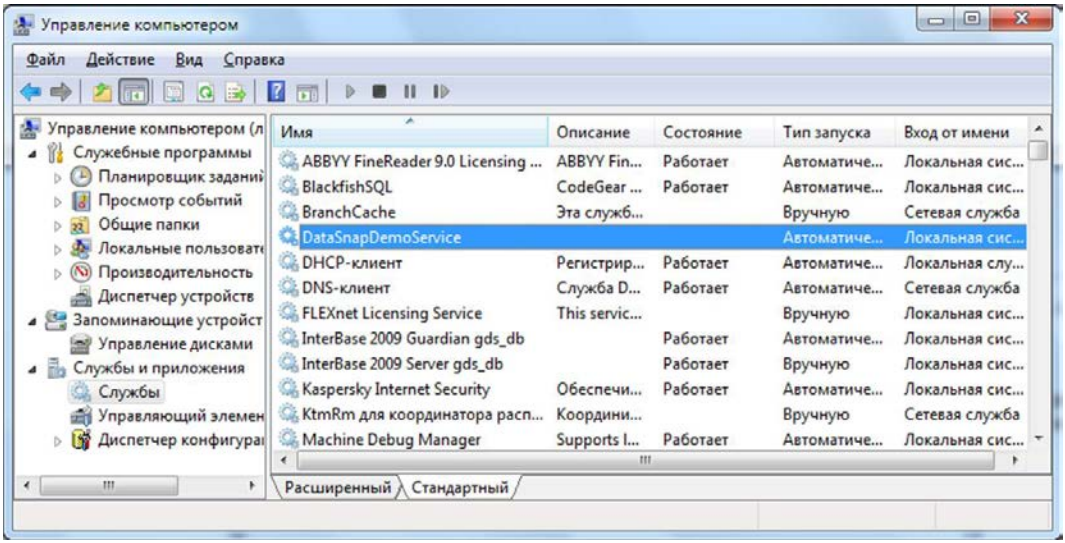


Рис. 31.8. Служба `DataSnapDemoService` в консоли управления служб Windows 7

Для снятия с регистрации применяют ключ `/UNINSTALL`. Процесс установки сервиса сопровождается выводом уведомляющего сообщения. Для отказа от показа окна уведомления используйте ключ `/SILENT`.

Подготовка клиентского приложения

Создайте обычный проект VCL и сразу добавьте к нему модуль данных `TDataModule`. Расположите в модуле данных компонент `SQLConnection1:TSQLConnection`. Сохраните проект в отдельном каталоге под любым именем.

Выберите компонент `SQLConnection1`. Установите его свойство `Driver` в состояние `DataSnap`. Свойству `CommunicationProtocol` присвойте значение TCP/IP и обязательно проверьте номер порта `Port`, по умолчанию он должен принимать значение `211`. В свойстве `HostName` укажите имя или IP-адрес хоста, на котором развернута служба DataSnap. Если вы отлаживаете сервер и клиент на одном компьютере, то воспользуйтесь аббревиатурой `localhost`. Отключите диалог регистрации пользователя, установив в `false` свойство `LoginPrompt`. Проверьте корректность подключения к службе сервера приложений, временно установив в состояние `true` свойство `Connected`. Убедившись в готовности `SQLConnection1` сотрудничать с разработанной нами службой DataSnap, верните свойство `Connected` в исходное состояние.

Щелкните правой кнопкой мыши по компоненту `SQLConnection1`. В контекстном меню компонента найдите пункт **Generate DataSnap client classes**. Щелчок по этому элементу меню заставит Delphi сгенерировать модуль методов клиента (листинг 31.8).

Листинг 31.8. Автоматически созданный класс `TServerMethods1Client`

```
type
  TServerMethods1Client = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FEchoStringCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
                       AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function EchoString(Value: string): string;
  end;
```

Среди методов класса `TServerMethods1Client` для нас самой большой интерес представляет метод `EchoString()`. Это не что иное, как способ вызова объявленного на сервере DataSnap одноименного метода (он станет работать "эхом" — возвращать клиенту его же текстовый запрос). Вновь сохраните проект, во время сохранения переименуйте автоматически сгенерированный модуль методов клиента в `MethodsClient.pas`.

Воспользовавшись пунктом меню **File | Use Unit…**, подключите к главной форме проекта модуль методов клиента MethodsClient.pas. Затем разместите на главной форме клиента строку ввода `Edit1` и метку `Label1`, перейдите к событию `OnChange()` строки ввода и напишите там строки кода из листинга 31.9.

Листинг 31.9. Вызов демонстрационного метода сервера

```
procedure TForm1.Edit1Change(Sender: TObject);
var SMC:TServerMethods1Client;
begin
 SMC:=TServerMethods1Client.Create(
                 DataModule1.SQLConnection1.DBXConnection);
 Label1.Caption:=ServerMethods1Client.EchoString(Edit1.Text);
 SMC.Free;
end;
```

Обратитесь к модулю данных. В момент его создания нам следует подключиться к службе сервера DataSnap. В простейшем случае это будет выглядеть так, как показано в листинге 31.10.

Листинг 31.10. Активизируем соединение

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
begin
SQLConnection1.Open;
end;
```

Клиент готов к работе. Запустите проект и протестируйте работу метода `EchoString()`. Теперь переходим к самому интересному — приспособим наш сервер DataSnap к работе с базой данных.

Подключение сервера приложений к БД

Прежде чем мы приступим к программированию, необходимо остановить сервис сервера DataSnap. Для этого воспользуйтесь услугами консоли управления компьютером (см. рис. 31.8). Найдите службу среди перечня сервисов компьютера и остановите сервис.

ЗАМЕЧАНИЕ

Для запуска и остановки службы программисту может оказаться удобнее работать не с консолью управления службами, а с входящей в состав Windows утилитой командной строки net.exe. Для останова службы следует набрать команду `NET STOP имя_сервиса`, для запуска `NET START имя_сервиса`.

Как я и обещал ранее, сейчас мы приступим к превращению сервера DataSnap в сервер приложений многоуровневого проекта БД на основе MySQL. Для этого откроем модуль методов сервера и разместим на нем два компонента:

- Соединитель `SQLConnection1:TSQLConnection`. Задача компонента — обеспечение подключения сервера приложений DataSnap к серверу баз данных MySQL. Для этого в свойстве `ConnectionName` выберите соединение `MYSQL_BOOKSMARKET`. Оно было создано в *главе 27*, посвященной изучению компонента `TSQLConnection` (см. рис. 27.3). При желании вы можете создать любое другое подключение, например на базе сервера BlackfishSQL. В составе инсталляции Delphi имеется демонстрационная база данных employee.jds. Проверьте наличие регистрационных данных пользователя в параметрах компонента `Params` и отключите свойство `LoginPrompt`, вызывающего диалог регистрации. Проверьте факт успешного подключения компонента к базе данных, для этого активизируйте свойство `Connected`. Убедившись, что все в порядке, отключите компонент от БД.
- Набор данных `SQLDataSet1: TSQLDataSet`. Задача набора данных — подключение к таблице БД. Воспользовавшись свойством `SQLConnection`, подключите компонент к соединению. Установите свойство `CommandType` в состояние `ctTable`. В списке `CommandText` выберите любую таблицу из состава базы данных, я остановил свой выбор на таблице `WRITERS`.

Перейдите в раздел реализации модуля методов. В нем есть пока единственный демонстрационный метод `EchoString()`. Мы с вами расширим возможности сервера и объявим метод, позволяющий возвратить клиенту все записи таблицы `WRITERS`. Для этого в секции публичных объявлений класса `TServerMethods1` опишем заголовок функции `WritersDataSet` (листинг 31.11). В раздел реализации метода нам предстоит ввести всего две строки, которые позволят методу возвратить вызывающему клиентскому приложению набор данных `TDataSet`.

Листинг 31.11. Объявление нового метода сервера приложений

```
type
  TServerMethods1 = class(TDSServerModule)
    ...
  public
    { Public declarations }
    Function EchoString(Value: string): string;
    Function WritersDataSet:TDataSet; //новый метод сервера
  end;
    ...
Function TServerMethods1.WritersDataSet: TDataSet;
begin
 SQLDataSet1.Open;
 Result:=SQLDataSet1;
end;
```

После столь "непосильных" трудов вновь откомпилируйте проект сервера DataSnap и запустите его в качестве службы Windows.

Получение данных клиентским приложением

Преобразуем наше обычное клиентское приложение DataSnap в клиента трехуровневого приложения базы данных.

Сначала предлагаю посмотреть, каким изменениям подвергнется клиентский модуль методов после его повторной регенерации. Для этого вновь вызовем контекстное меню компоненту `SQLConnection1` и щелкнем по элементу **Generate DataSnap client classes** (генерировать модуль методов клиента). Открыв модуль MethodsClient.pas, вы увидите, что он дополнился новым методом `WritersDataSet()`, возвращающим набор данных из таблицы авторов. Как видите, клиент даже в период разработки достаточно легко отслеживает появление новых методов на сервере DataSnap.

Замечание

В распоряжении программиста Delphi имеется специальный компонент, который позволяет очень просто обращаться ко всем имеющимся в распоряжении сервера методам, он называется `TSqlServerMethod`.

Чтобы клиент приобрел возможность работать с БД, на помощь к уже имеющемуся в модуле данных соединению `SQLConnection1` необходимо пригласить следующий перечень компонентов (рис. 31.9):

- Метод сервера `SqlServerMethod1: TSqlServerMethod`. Данный компонент позволит обратиться ко всем методам, объявленным на сервере. С помощью свойства `SQLConnection` подключаемся к компоненту `SQLConnection1`. Весь перечень доступных методов окажется в свойстве `ServerMethdName`, найдите в этом списке метод `WritersDataSet`.
- Провайдер `DataSetProvider1: TDataSetProvider`. С помощью свойства `DataSet` подключите провайдер к компоненту `SqlServerMethod1`.
- Клиентский набор данных `ClientDataSet1:TClientDataSet`. Воспользовавшись свойством `ProviderName`, присоединяем компонент к провайдеру `DataSetProvider1`.
- Источник данных `DataSource1: TDataSource`. Источник состыкуем с компонентом `ClientDataSet1`.

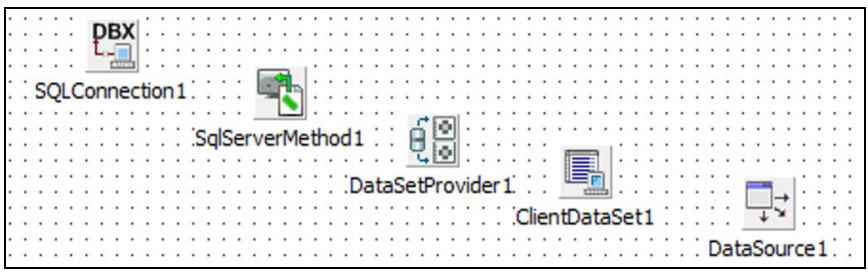


Рис. 31.9. Модуль данных клиентского приложения DataSnap с доступом только для чтения

Все остальное — дело техники. В простейшем случае размещаем на форме клиента сетку `TDBGrid` и подключаем компонент к источнику данных `DataSource1` — тонкий клиент, способный получать данные, готов. Заметьте, что программировать нам почти не пришлось, правда, нужно не забыть активизировать подключение и клиентский набор данных во время старта клиента (листинг 31.12).

Листинг 31.12. Активизация клиентского набора данных

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
begin
 SQLConnection1.Open;
 ClientDataSet1.Open;
end;
```

К сожалению, полученный набор данных доступен только для чтения, однако это не беда и прямо сейчас мы найдем способ исправить ситуацию.

Реализация на сервере метода вставки новой записи

Научим сервер DataSnap редактировать данные по требованию пользователей. Допустим, нам необходимо реализовать метод вставки новой записи в таблицу авторов. Для достижения поставленной цели нам потребуется помощь компонента-запроса `TSQLQuery`.

Остановите службу сервера DataSnap. Откройте в Delphi проект сервера и разместите запрос `SQLQuery1` в модуле методов `ServerMethods1`. Проинициализируем компонент в момент создания модуля (листинг 31.13), для этого нам понадобится обработчик события `OnCreate()`.

Листинг 31.13. Инициализация SQL-команды вставки новой записи

```
procedure TServerMethods1.DSServerModuleCreate(Sender: TObject);
begin
with SQLQuery1 do
begin
  SqlConnection:=SQLConnection1;
  SQL.Clear;
  SQL.Add('INSERT INTO WRITERS');
  SQL.Add('(WRITER) VALUES (:WRITER)');
end;
end;
```

Мы разработали текст инструкции SQL, позволяющий добавлять в таблицу `WRITERS` новую строку. Теперь нам необходимо подготовить очередной метод сервера. Я его назову `WritersInsert` (листинг 31.14).

Листинг 31.14. Метод сервера вставки записи в таблицу `WRITERS`

```
type
  TServerMethods1 = class(TDSServerModule)
…
    Function WritersDataSet:TDataSet;
    Procedure WritersInsert(S:String); //метод добавления новой записи
  end;
…
```

```
Procedure TServerMethods1.WritersInsert(Writer: String);
begin
  SQLQuery1.Params[0].Value:= Writer;
  SQLQuery1.ExecSQL();
end;
```

Метод сервера получит от клиента фамилию нового автора и передаст ее в таблицу. Откомпилируйте проект и вновь запустите службу сервера DataSnap.

Доступ к методу вставки записи на стороне клиента

Для того чтобы клиент получил доступ к методу вставки новой записи, перенесите в модуль данных клиента очередной компонент `TSqlServerMethod`, допустим, что он получил имя `SqlServerMethod2`. Подключите компонент `SqlServerMethod2` к соединителю `SQLConnection1`. Раскройте свойство `ServerMethodName` и найдите в списке метод `TServerMethods1.WritersInsert`. Обратите внимание на то, что в перечне параметров компонента `Params` автоматически появился параметр `Writer`.

Теперь, чтобы воспользоваться услугами метода сервера, нам достаточно написать пару строк кода (листинг 31.15).

Листинг 31.15. Обращение к методу вставки новой записи со стороны клиента

```
SqlServerMethod2.Params.ParamByName('Writer').Value:=Edit1.Text;
SqlServerMethod2.ExecuteMethod;
```

Архитектура DataSnap, совместимая со старыми клиентскими приложениями

Рассмотренная в этой главе архитектура DataSnap появилась в Delphi 2010 и рекомендована к применению в современных трехуровневых проектах БД. Но что делать со старыми проектами DataSnap? Переписать их заново? Для упрощения рефакторинга старых проектов и обеспечения совместимости новых серверов приложений со старыми клиентами можно воспользоваться незначительно модифицированной архитектурой DataSnap (рис. 31.10).

К счастью, для того чтобы новый сервер DataSnap стал доступным для клиентских приложений, разработанных в предыдущих версиях Delphi, не придется проводить бессонных ночей за компьютером. Вместо этого, следует снабдить каждый из наборов данных сервера своим провайдером — компонентом `TDataSetProvider`.

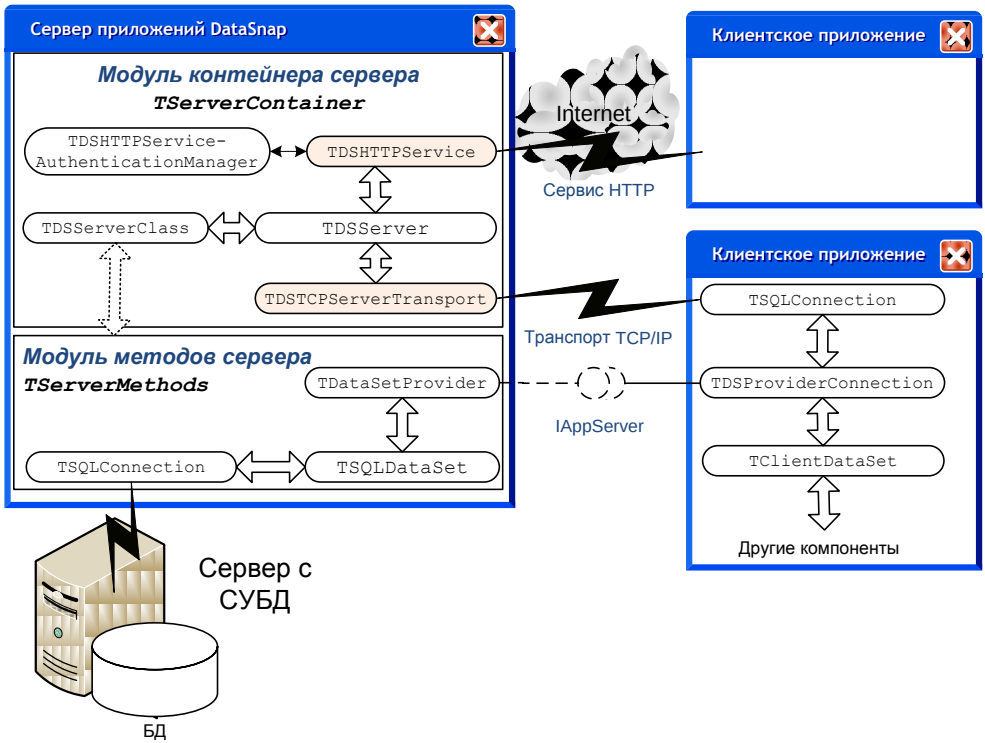


Рис. 31.10. Архитектура проекта DataSnap Delphi 2010/XE, совместимая со старыми клиентами

В нашем примере подобная операция делается в два счета. Останавливаем сервис сервера, в модуль методов сервера добавляем провайдер набора данных `DataSetProvider1:TDataSetProvider`. С помощью свойства `DataSet` подключаем компонент к единственному набору данных `SQLDataSet1`. Осталось откомпилировать проект и перезапустить сервис. Кто не уложился за минуту?

Старым клиентам DataSnap был неведом компонент `TDSSqlMethod`, поэтому вы не найдете его в клиенте, изображенном на рис. 31.10. Вместо него Embarcadero предлагает применять компонент `TDSProviderConnection`, который позволит клиентскому приложению получить доступ к набору данных сервера.

Внимание!

Ключевая особенность старых версий многоуровневых приложений — широкое применение интерфейса `IAppServer` для доступа клиента к серверу.

Интерфейс *IAPPServer*

Интерфейс `IAppServer` реализуется на стороне сервера приложений и служит точкой опоры для всей технологии взаимодействия с клиентскими модулями.

В листинге 31.16 приведен фрагмент модуля midas.pas, в котором содержится объявление интерфейса сервера приложений.

Листинг 31.16. Объявление интерфейса IAppServer

```
type IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function  AS_ApplyUpdates(const ProviderName: WideString;
                        Delta: OleVariant;
                    MaxErrors: Integer; out ErrorCount: Integer;
                    var OwnerData: OleVariant): OleVariant; safecall;
    function  AS_GetRecords(const ProviderName: WideString;
                          Count: Integer;
                          out RecsOut: Integer;
                      Options: Integer; const CommandText: WideString;
                      var Params: OleVariant;
                      var OwnerData: OleVariant): OleVariant; safecall;
    function  AS_DataRequest(const ProviderName: WideString;
                          Data: OleVariant): OleVariant; safecall;
    function  AS_GetProviderNames: OleVariant; safecall;
    function  AS_GetParams(const ProviderName: WideString;
                      var OwnerData: OleVariant): OleVariant; safecall;
    function  AS_RowRequest(const ProviderName: WideString;
                          Row: OleVariant; RequestType: Integer;
                      var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString;
                       const CommandText: WideString;
                       var Params: OleVariant;
                       var OwnerData: OleVariant); safecall;
  end;
```

Рассмотрим методы интерфейса в порядке их объявления.

- Первый по счету метод AS_ApplyUpdates() — передает все полученные от клиента изменения набора данных к БД, определяемой провайдером ProviderName. Информация об изменениях находится в параметре Delta. Аргумент MaxErrors устанавливает порог ошибок, при превышении которого операция отменяется. Выходной параметр ErrorCount возвращает количество ошибок, возникших во время операции. Последний параметр OwnerData позволяет передавать дополнительную служебную информацию.
- Функция AS_GetRecords() — возвращает пакет со строками данных от БД, связанной с провайдером ProviderName. Число запрашиваемых строк задается параметром Count, если значение равно –1 , возвращаются все строки; 0 — получаем только метаданные; больше нуля — возвращается требуемое число строк. Выходной параметр RecsOut возвращает реальное число полученных строк. Опции находятся в параметре Options (см. табл. 31.4). В CommandText передает-

ся инструкция SQL или имена таблицы, хранимой процедуры, из которых мы получаем данные. Названный параметр активизируется, только если в составе опций имеется флаг `poAllowCommandText`. Массив параметров передается через аргумент `Params`.

- Метод `AS_DataRequest()` — отвечает за генерацию события `OnAS_DataRequest()` у компонента провайдера (например, потомка `TCustomProvider`).
- Метод `AS_GetProviderNames()` — возвращает список с именами доступных провайдеров.
- Метод `AS_GetParams()` — направляет провайдеру текущие параметры набора данных.
- Метод `AS_RowRequest()` — возвращает строку набора данных `Row` от провайдера `ProviderName`. Параметр `RequestType` информирует о типе возвращаемых данных.
- Заключительный метод `AS_Execute()` — отправляет на выполнение запрос или хранимую процедуру, указанную в параметре `CommandText`. Имеющиеся параметры передаются через `Params`.

Провайдер набора данных, компонент *TDataSetProvider*

Двусторонний обмен между наборами данных сервера и клиента обеспечивают компоненты класса `TDataSetProvider`. Во время проектирования компоненты-провайдеры размещаются в модуле методов сервера приложений, количество провайдеров обычно соответствует числу компонентов наборов данных (`TSQLDataSet`, `TSQLTable`, `TSQLQuery`). Во время выполнения программы провайдер организует информационный канал между конкретным набором данных сервера и используемым в составе программы клиента DataSnap компонентом `TDSProviderConnection`, транслируя между ними пакеты данных с записями и служебной информацией.

ВНИМАНИЕ!

Использование провайдеров наборов данных `TDataSetProvider` и `TDSProviderConnection` в проектах DataSnap целесообразно лишь в том случае, когда клиентское приложение должно работать на основе интерфейса `IAppServer`.

В идеале расположенному на стороне клиента набору данных (компоненту `TClientDataSet`) хотелось бы обладать возможностью напрямую работать с набором данных сервера приложений DataSnap (компонентом `TSQLDataSet`). Если бы это было возможно, то компоненты сервера и клиента обменивались данными через понятный друг другу набор методов универсального набора данных `TDataSet`. К сожалению, подобное общение в старых проектах DataSnap было невозможно. Для того чтобы клиентский набор данных прежних версий Delphi смог "увидеть" набор данных сервера приложений, компонентам приходится воспользоваться ус-

лугами более низкоуровневого интерфейса IAppServer. С некоторой степенью упрощения можно сказать, что высокоуровневые методы TDataSet инкапсулируются в более низкоуровневые методы интерфейса сервера приложений IAppServer. Роль "переводчиков" высокоуровневых методов TDataSet в методы интерфейса IAppServer возлагаются на провайдеров. На стороне сервера это компоненты TDataSetProvider, а на стороне клиента — TDSProviderConnection.

В простейшем случае единственное, что необходимо сделать после размещения компонента на удаленном модуле данных сервера приложений, — настроить свойство

```
property DataSet: TDataSet;
```

подключив провайдер к определенному набору данных. Все остальные задачи по обмену данными между клиентом и сервером провайдер в состоянии решить самостоятельно.

Замечание

Для того чтобы клиентский набор данных TClientDataSet получил доступ к провайдеру, ему следует воспользоваться поддержкой компонента TDSProviderConnection.

Если вам не по душе "самостоятельность" провайдера TDataSetProvider, можно внести некоторые поправки в его деятельность.

Для того чтобы клиентский набор данных мог анализировать ограничения, наложенные на подключаемый к нему набор данных, провайдер передает эти метаданные клиентскому приложению:

```
property Constraints: Boolean; //по умолчанию true
```

Чтобы воспрепятствовать отправке информации об ограничениях, отключите это свойство, присвоив ему значение false.

Если свойство

```
property Exported: Boolean; //по умолчанию true
```

установлено в состояние true, то клиентский набор данных приобретает возможность взаимодействовать с сервером с помощью интерфейса IAppServer. Этот интерфейс необходим в ситуации, когда клиент должен обладать возможностью обновлять данные на сервере, поэтому договоримся никогда не изменять присвоенное по умолчанию значение свойства Exported.

Порядок реакции провайдера на изменения данных на клиентской стороне определяет свойство

```
property ResolveToDataSet: Boolean; //по умолчанию false
```

По умолчанию (состояние false) провайдер не проявляет интереса к потоку данных от клиента, транслируя все изменения непосредственно в таблицы базы данных. Но, если сервер приложений должен включиться в процесс обработки данных с целью поддержки бизнес-логики БД (по большому счету именно для этого он и создается), включайте свойство в true. В результате связанный с провайдером компонент-набор данных сможет реагировать на возвращаемые от клиентского приложения данные, для этого у него активизируются соответствующие обработчики событий.

Реакция компонента на совместный доступ к одной и той же записи нескольких пользователей устанавливается свойством

```
property UpdateMode: TUpdateMode; //по умолчанию upWhereAll
где: TUpdateMode = (upWhereAll, upWhereChanged, upWhereKeyOnly);
```

С точки зрения устранения коллизий наилучшим режимом считается `upWhereAll`. После того как первый пользователь даст команду на сохранение записи, второй пользователь получит уведомление, что запись была изменена. Режим `upWhereChanged` допускает одновременное редактирование разных полей одной и той же записи, но противится одновременному доступу к одному и тому же полю. Наконец, режим `upWhereKeyOnly` исповедует простое правило — в БД сохранятся правки, внесенные самым последним пользователем, а остальные изменения будут аннулированы.

Дополнительные особенности поведения провайдера определяет свойство

```
property Options: TProviderOptions;
```

Перечень опций приведен в табл. 31.4. По умолчанию включена только опция `poUseQuoteChar`, все остальные опции отключены для обеспечения политики безопасности и экономии сетевого трафика.

Таблица 31.4. *Опции провайдера* `TProviderOptions`

Опция	Описание
`poFetchBlobsOnDemand`	Разрешает передачу в клиентское приложение BLOB-данных только по специальному требованию. Если указанная опция включена, то обслуживаемый провайдером клиентский набор данных `TClientDataset` сможет загрузить BLOB-данные только после явного обращения к методу `FetchBlobs()` или установив в состояние `true` свойство `FetchOnDemand`
`poFetchDetailsOnDemand`	По умолчанию, при запросе клиентом родительской таблицы провайдер включает в пакет данных записи подчиненных таблиц, связанных с ней ссылочной целостностью. Для снижения сетевого трафика и ускорения работы приложений от этой услуги можно отказаться, включив опцию `poFetchDetailsOnDemand`. В этом случае передача в клиентский набор записей подчиненных таблиц будет осуществляться только по явному запросу клиента (см. свойство `FetchOnDemand` и метод `FetchDetails` компонента `TClientDataSet`)
`poIncFieldProps`	Включает в пакет данных информацию о ключевых свойствах полей набора данных (`Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, `DisplayValues`)

Таблица 31.4 *(окончание)*

Опция	Описание
poCascadeDeletes	При удалении на стороне клиента главной записи родительской таблицы сервер имеет право осуществить каскадное удаление всех подчиненных записей
poCascadeUpdates	При изменении главной записи в родительской таблице сервер обновит все подчиненные записи
poReadOnly	Переводит набор данных в режим "только для чтения"
poAllowMultiRecordUpdates	По умолчанию изменения, затрагивающие несколько записей, применяются последовательно. Включение опции разрешает вносить изменения одновременно во все записи
poDisableInserts	Запрещает клиенту проводить операции по вставке новых записей
poDisableEdits	Запрещает клиенту редактировать записи
poDisableDeletes	Запрещает клиенту удалять записи
poNoReset	По умолчанию перед отправкой пакета данных клиентскому набору данных сервер включает все записи, начиная с первой. Включение опции разрешает серверу передавать данные, начиная с текущей. Опция влияет на работу метода AS_GetRecords интерфейса IAppServer
poAutoRefresh	Разрешает автоматическое обновление записей на стороне клиента в случае изменения данных на сервере
poPropogateChanges	Все изменения, осуществленные в обработчиках событий BeforeUpdateRecord() или AfterUpdateRecord() на стороне клиента после их физического сохранения в БД, будут вновь перенаправлены клиенту, для того чтобы он перевел свои данные в актуальное состояние
poAllowCommandText	Разрешает серверу приложений вносить изменения в поступающие от клиента команды SQL
poRetainServerOrder	Запрещает клиенту изменять порядок сортировки записей
poUseQuoteChar	Определяет, что символ двойной кавычки ("), используемый провайдером SQL, будет также применяться в запросах SQL, созданных провайдером TDataSetProvider

Компонент-провайдер обладает достаточно большим перечнем обработчиков событий (табл. 31.5). Обратите внимание, что подавляющее большинство событий предназначено для обслуживания методов интерфейса `IAppServer`.

Таблица 31.5. *События компонента* `TDataSetProvider`

№	Событие	Описание
Для сохранения изменений в наборе данных клиент вызвал метод `AS_ApplyUpdates` интерфейса `IAppServer` (в клиентском наборе данных `TClientDataSet` метод инкапсулирован функцией `ApplyUpdates`). Все накопленные на клиентской стороне изменения направлены провайдеру		
1	**property** `BeforeApplyUpdates: TRemoteEvent;` где `TRemoteEvent = procedure (Sender: TObject; var OwnerData: OleVariant) of object;`	Событие вызывается перед началом сохранения в БД изменений, полученных от клиента
2	**property** `OnUpdateData: TProviderDataEvent;` где `TProviderDataEvent = procedure (Sender: TObject; DataSet: TCustomClientDataSet) of object;`	Возникает перед тем, как провайдер приступит к сохранению данных, полученных от клиентского приложения
3	**property** `AfterApplyUpdates: TRemoteEvent;`	Генерируется после сохранения в БД данных, полученных от клиента
4	**property** `OnUpdateError: TResolverErrorEvent;` где `TResolverErrorEvent = procedure (Sender: TObject; DataSet: TCustomClientDataSet; E: EUpdateError; UpdateKind: TUpdateKind; var Response: TResolverResponse) of object;`	Событие генерируется при возникновении ошибки во время сохранения изменений
Обновление отдельной записи на сервере		
1	**property** `BeforeUpdateRecord: TBeforeUpdateRecordEvent;` где `TBeforeUpdateRecordEvent = procedure (Sender: TObject; SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean) of object;`	Вызывается перед началом применения обновлений к отдельной записи

Таблица 31.5 *(продолжение)*

№	Событие	Описание
Обновление отдельной записи на сервере		
2	**property** AfterUpdateRecord: TAfterUpdateRecordEvent; где TAfterUpdateRecordEvent = **procedure** (Sender: TObject; SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind) of object;	Событие генерируется после успешного обновления отдельной записи
Клиент обратился к методу As_Execute интерфейса IAppServer для выполнения запроса SQL		
1	**property** BeforeExecute: TRemoteEvent;	Вызывается перед выполнением команды SQL на сервере
2	**property** AfterExecute: TRemoteEvent;	Возникает после завершения выполнения команды SQL на сервере
Реакция провайдера на процесс формирования пакета служебной информации в адрес клиентского набора данных. Для получения параметров клиент должен воспользоваться методом AS_GetParams() интерфейса IAppServer, названный метод инкапсулирован в GetParams() компонента TClientDataSet		
1	**property** BeforeGetParams: TRemoteEvent;	Событие возникает перед началом формирования набора служебной информации и отправке их клиенту
2	property AfterGetParams: TRemoteEvent;	Генерируется по окончании отправки служебной информации клиенту
Провайдер обновляет текущую запись в клиентском наборе данных. Группа событий сопровождает обращение к методу AS_RowRequest интерфейса IAppServer. Метод вызывается автоматически		
1	**property** BeforeRowRequest: TRemoteEvent;	Генерируется перед началом обновления провайдером текущей записи в клиентском наборе данных
2	**property** AfterRowRequest: TRemoteEvent;	Вызывается после завершения обновления провайдером текущей записи в клиентском наборе данных

Таблица 31.5 *(окончание)*

№	Событие	Описание
Провайдер получает данные из таблиц БД и готовит их к отправке к клиенту		
1	`property BeforeGetRecords: TRemoteEvent;`	Вызывается перед началом формирования пакета данных для отправки его клиенту

2	`property OnGetData: TProviderDataEvent;`
	где
	`TProviderDataEvent = procedure (Sender: TObject; DataSet: TCustomClientDataSet) of object;`
		Событие возникает после получения данных из таблиц БД

3	`property AfterGetRecords: TRemoteEvent;`	Генерируется по окончании формирования пакета данных для отправки клиенту
Прочие события		

1	`property OnDataRequest: TDataRequestEvent;`
	где
	`TDataRequestEvent = function (Sender: TObject; Input: OleVariant): OleVariant of object;`
		Возникает при поступлении от клиента заявки на получение данных. Для этого клиент обращается к методу `AS_DataRequest` интерфейса `IAppServer`

2	`property OnGetDataSetProperties: TGetDSProps;`
	где
	`TGetDSProps = procedure (Sender: TObject; DataSet: TDataSet; out Properties: OleVariant) of object;`
		Генерируется, если провайдер добавляет дополнительную информацию в пакет данных, предназначенный для отправки клиенту

3	`property OnGetTableName: TGetTableNameEvent;`
	где
	`TGetTableNameEvent = procedure (Sender: TObject; DataSet: TDataSet; var TableName: WideString) of object;`
		Провайдер получает имя таблицы, данные которой подлежат обновлению

Подключение к провайдеру набора данных, компонент *TDSProviderConnection*

До выхода Delphi 2010 в трехуровневых проектах баз данных на основе технологии DataSnap компонент `TDSProviderConnection` применялся в клиентских при-

ложениях. Назначение компонента — предоставление для клиента DataSnap прозрачного канала на основе `IAppServer` к используемым в модуле методов сервера провайдерам наборов данных — компонентам класса `TDataSetProvider`.

Работа с компонентом начинается с подключения к соединению с помощью свойства

```
property SQLConnection: TSQLConnection;
```

Получив доступ к серверу DataSnap, компонент приобретает возможность взаимодействия с модулем методов сервера. Имя класса обязательно должно быть передано в свойство

```
property ServerClassName: string;
```

Провайдер активизируется автоматически одновременно с активизацией связанного с компонентом соединителя `TSQLConnection`. О факте успешного подключения можно судить благодаря свойству

```
property Connected:boolean;
```

После настройки компонента он сможет выступать в роли провайдера, предоставляя используемым в клиентском приложении компонентам `TClientDataSet` доступ к наборам данных сервера. Для этого клиентский набор данных с помощью свойства `RemoteServer` подключается к провайдеру `TDSProviderConnection` и выбирает в свойстве `ProviderName` имя интересующего его провайдера данных на стороне сервера.

Клиентское приложение БД на основе *IAppServer*

Предлагаю разработать еще одно клиентское приложение DataSnap, работающее на основе интерфейса `IAppServer`. Нам потребуется несколько модифицировать цепочку компонентов доступа к данным. На первой позиции по-прежнему остается соединитель `TSQLConnection`, вы уже умеете самостоятельно подключать его к серверу DataSnap, поэтому мы не станем повторяться. А дальше нас ждут изменения (рис. 31.11):

- Разместите в модуле данных компонент `DSProviderConnection1: TDSProviderConnection`. Подключаем провайдер к компоненту-соединителю, для чего обращаемся к свойству `SQLConnection`. Далее в свойстве `ServerClassName` нам следует ввести название класса, на основе которого на сервере DataSnap был создан модуль методов. В нашем примере класс назывался `TServerMethods1`.
- Клиентский набор — компонент `ClientDataSet1 : TClientDataSet`. Устанавливаем связь с удаленным сервером, для этого в свойство `RemoteServer` передаем имя провайдера услуг DataSnap — `DSProviderConnection1`. Теперь переходим к свойству `ProviderName`, в выпадающем списке вы найдете ссылку всего на одного провайдера — компонент `DataSetProvider1`, размещенный на стороне сервера.

- Источник данных `DataSource1: TDataSource` подключаем к набору данных `ClientDataSet1`.

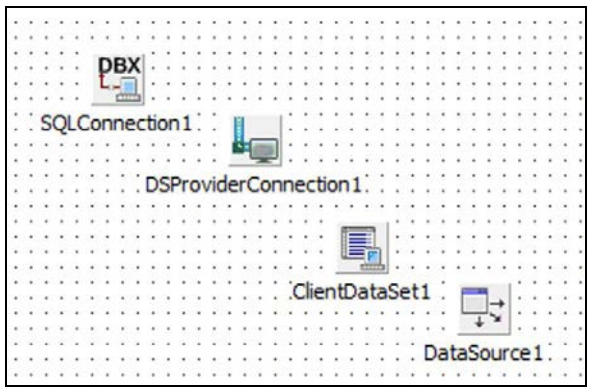


Рис. 31.11. Модуль данных клиентского приложения DataSnap с полным доступом

На главной форме клиентского приложения располагаем сетку `TDBGrid` и навигатор `TDBNavigator`. Подключаем компоненты к источнику данных `DataSource1`.

Позаботимся об активизации соединения во время старта приложения, для этого во время создания модуля данных открываем соединение `SQLConnection1.Open` и клиентский набор данных `ClientDataSet1.Open`.

Так как наше приложение претендует на право не только просматривать, но и редактировать данные, то обязательно вспоминаем о методе `ApplayUpdates()`, имеющемся в распоряжении компонента `ClientDataSet1`. Благодаря `ApplayUpdates()` все правки из локального кэша клиентского набора данных отправятся на сервер. Обратиться к методу следует в момент прекращения работы приложения (листинг 31.17).

Листинг 31.17. Сохраняем данные

```
procedure TDataModule1.DataModuleDestroy(Sender: TObject);
begin
if ClientDataSet1.ChangeCount>0 then
    ClientDataSet1.ApplyUpdates(0);
end;
```

Механизм обратного вызова

В технологии DataSnap поддерживается механизм обратного вызова, благодаря которому сервер сможет ненадолго поменяться местами с клиентом и приобрести

право обратиться к клиентскому методу. Для того чтобы это произошло необходимо выполнить одно обязательное условие, — сервер должен получить от клиента ссылку на его функцию обратного вызова.

При реализации обратного вызова взаимодействие между сервером и клиентом осуществляется в текстовом формате обмена данными, основанном на JavaScript (JavaScript Object Notation, JSON). Данные передаются в виде массива, каждая из ячеек которого представляет собой пару: имя элемента и его значение. В Delphi формат JSON описан в модуле DBXJSON, который следует подключить как к серверу, так и к клиенту DataSnap.

Ссылка на функцию обратного вызова передается в адрес сервера в виде типизированного параметра `TDBXCallback`. Для этого в модуле методов клиента объявляется класс-потомок `TDBXCallback`, в котором переписывается метод `Execute()`. Средством взаимодействия между сервером и клиентом служит единственный аргумент метода `Arg`, — структура `TJSONValue`, позволяющая представать массив пар JSON. Реализация метода `Execute()` может выглядеть так, как показано в листинге 31.18.

Листинг 31.18. Объявление `TCallback` на стороне клиента

```
type
    TCallback = class(TDBXCallback)
  public
    Function Execute(const Arg:TJSONValue): TJSONValue; override;
end;
…
function TCallback.Execute(const Arg: TJSONValue): TJSONValue;
var s:string;
begin
  Result:=TJSONValue(Arg.Clone);
  s:=TJSONObject(Result).Get(0).JsonString.ToString
                        +#13+TJSONObject(Result).Get(0).JSONValue.Value;
  ShowMessage(s);
end;
```

Теперь перейдем к серверу DataSnap. Допустим, что сервер объявляет метод, позволяющий суммировать два числа и поддерживающий возможность обратного вызова. Тогда заголовок метода сервера будет выглядеть следующим образом:

```
Function GetSum(X,Y:Integer; Callback:TDBXCallback):integer;
```

Здесь, кроме собственно подлежащих суммированию аргументов, вы обнаружите параметр обратного вызова `Callback`. Реализацию метода иллюстрирует листинг 31.19.

Листинг 31.19. Реализация метода сервера с возможностью вызова клиента

```
function TServerMethods1.GetSum(X, Y: Integer;
                                 Callback: TDBXCallback): integer;
var JSONPair:TJSONPair;
    JSONObject:TJSONObject;
begin
 Result:=X+Y;
 JSONPair:=TJSONPair.Create('Метод выполнен в ',DateTimeToStr(Now));

 JSONObject:=TJSONObject.Create;
 JSONObject.AddPair(JSONPair);

 Callback.Execute(JSONObject);
end;
```

Помимо выполнения операции сложения сервер формирует пару JSON, в которую заносит информацию о дате и времени выполнения метода. И, благодаря функции обратного вызова `Callback.Execute()`, инициирует выполнение этой функции у клиента.

Из-за необходимости передавать ссылку на функцию обратного вызова несколько усложняется порядок обращения из клиентского приложения к методу сервера (листинг 31.20).

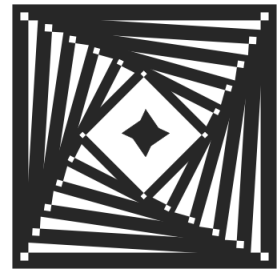
Листинг 31.20. Обращение к методу сервера со стороны клиента

```
var x,y,z:integer;
    SMC:TServerMethods1Client;
    Callback:TCallback;
begin
  x:=StrToInt(Edit1.Text);
  y:=StrToInt(Edit2.Text);
  ServerMethods1Client:=
  TServerMethods1Client.Create(DataModule1.SQLConnection1.DBXConnection);

  Callback:=TCallback.Create;

  z:=ServerMethods1Client.GetSum(x,y,Callback);
  Label1.Caption:=IntToStr(z);

  Callback.Free;
  ServerMethods1Client.Free;
end;
```

Резюме

Многоуровневые приложения предназначены для организации взаимодействия между сервером БД и удаленными тонкими клиентами. За построение моста между сервером и клиентами отвечает сервер приложений. Основная задача сервера — облегчить работу клиента за счет предоставления ему максимально возможного перечня услуг, вплоть до превращения клиента в средство просмотра данных (тонкого клиента) и сосредоточения на сервере всей программной логики.

Компания Embarcadero не только поддерживает существующие модели трехуровневых БД, но и самостоятельно их развивает. Так, очередным нововведением Delphi 2010/XE стала не имеющая аналогов в других системах разработки обновленная технология DataSnap. Основное достоинство новой DataSnap — избавление от зависимости от COM, что позволило создавать более простые и производительные приложения.



Глава 32

Управление службой сервера приложений DataSnap

При выборе формата сервера приложений DataSnap для проекта БД нам пришлось отказаться от услуг консольного проекта и обычного приложения VCL и отдать предпочтение службе Windows. Причина такого решения кроется в особенностях построения приложений баз данных DataSnap и в правилах функционирования операционной системы Windows.

Сервер приложений DataSnap обычно развертывается на том же хосте, на котором исполняется СУБД, либо на отдельном хосте, имеющем высокоскоростной доступ к серверу баз данных. И в том, и в другом случае предполагается, что сервер DataSnap должен находиться в постоянной готовности к обслуживанию входящих запросов от многочисленных клиентов. Это возможно только при условии, что сервер приложений реализован в виде службы, а не в качестве классического приложения.

Службы Windows стартуют одновременно с загрузкой операционной системы. Это очень важно, ведь для запуска обычного процесса (консоли или стандартного приложения), пусть даже он установлен в автозагрузку, обязателен вход пользователя в систему. Согласитесь, что вряд ли будет пользоваться успехом программный продукт, который придется запускать вручную при каждой перезагрузке компьютера.

Как правило, службы не предоставляют пользователю никакого графического интерфейса. Какой в нем смысл, если служба выполняется на сервере, у которого зачастую просто выключен дисплей. Но с другой стороны, как управлять службой, у которой нет пользовательского интерфейса? Что делать, если на предприятии переконфигурируют сеть, поменяют размещение сервера баз данных, сменят IP-адреса и DNS-имена хостов, изменят пароли? В таких условиях предложенный в предыдущей главе пример службы сервера приложений утратит работоспособность. В первую очередь сервер приложений потеряет контакт с сервером баз данных, ведь ради упрощения примера мы практически не уделили внимания особенностям управления сервисом Windows. Далее мы устраним этот недостаток, создав приложение для управления службой DataSnap.

Менеджер управления службами

В Windows работа служб складывается из взаимодействия трех элементов: самой службы, *менеджера управления службами* (Service Control Manager, SCM) и *управляющего приложения* (Service Control Program, SCP).

При загрузке операционной системы Windows менеджер SCM стартует одним из первых. Сразу после инициализации SCM берет в свои руки бразды правления службами — управляет процессом загрузки всех автоматически подключаемых служб. Во время работы операционной системы менеджер передает команды от управляющих программ (например, консоли управления сервисами) к имеющимся в его подчинении службам.

Перечислим основные команды:

- инсталляция и деинсталляция служб;
- запуск, приостановка, возобновление выполнения или полная остановка службы;
- анализ состояния и изменение конфигурационных характеристик службы.

Менеджер SCM отвечает за конфигурационную базу данных служб системы, а всю информацию о службах, зарегистрированных в операционной системе, менеджер прячет в системном реестре Windows.

Для организации работы со службами компьютера первым шагом программиста станет получение идентификатора менеджера служб. С этой задачей превосходно справляется функция

```
Function OpenSCManager(lpMachineName, lpDatabaseName : pAnsiChar;
                       dwDesiredAccess : cardinal) : THandle;
```

Здесь `lpMachineName` — указатель на сетевое имя интересующего нас компьютера. Сетевое имя станции должно начинаться с двух наклонных черт: `\\NetComputer`. Если идет речь о нашей собственной станции, то в этот параметр передаем неопределенный указатель `nil`. Второй параметр — `lpDatabaseName` представляет собой указатель на строку с именем базы данных менеджера управления службами. На данный момент для инициализации этого параметра предусмотрена одна-единственная константа `SERVICES_ACTIVE_DATABASE`, и поэтому функция не "обидится", если в ее второй параметр также будет направлен `nil`. Последний параметр — `dwDesiredAccess` определяет наши права доступа. Наиболее грозные возможности предоставит константа `SC_MANAGER_ALL_ACCESS` — это права администратора компьютера. При успешном выполнении функция возвращает дескриптор менеджера служб.

После того как необходимость в услугах менеджера служб отпадет, обязательно освободите SCM:

```
Function CloseServiceHandle(hSCObject : THandle) : LongBool;
```

Единственный параметр метода — дескриптор менеджера SCM.

Работа со службой

Для доступа к установленному в операционной системе сервису необходимо получить его дескриптор с помощью функции

```
Function OpenService (SCManager : THandle;
                      ServiceName : pAnsiChar;
                      DesiredAccess : Cardinal): THandle;
```

Здесь `SCManager` — полученный усилиями функции `OpenSCManager()` дескриптор менеджера служб; `ServiceName` — указатель на текстовую строку с именем сервиса и `DesiredAccess` — желательные права доступа к сервису; наибольшие возможности предоставляет права администратора `SC_MANAGER_ALL_ACCESS`. При удачном стечении обстоятельств функция вернет дескриптор затребованной нами службы.

Для запуска службы из приложения SCP используют функцию

```
Fucntion StartService(Service : THandle;
                      NumServiceArgs : Cardinal;
                      var ServiceArgVectors : pAnsiChar) : LongBool;
```

Первый параметр функции определяет указатель, интересующего нас сервиса, второй параметр `NumServiceArgs` — число строк в массиве аргументов, на который указывает третий по счету параметр `ServiceArgVectors`. В простейшем случае, как продемонстрировано в листинге 32.1, для запуска сервиса нам понадобится только его дескриптор.

Листинг 32.1. Пример запуска службы

```
var hSCM, hSrv : THandle;
    SStatus:_SERVICE_STATUS;
    args: pAnsiChar;
const SrvName='DSService'; //название сервиса
begin
 hSCM:=OpenSCManager(nil,nil,SC_MANAGER_ALL_ACCESS);
 hSrv:=OpenService(hSCM,SrvName,SC_MANAGER_ALL_ACCESS);
 if hSrv>0 then StartService(hSrv, 0, args);
 CloseServiceHandle(hSrv);
 CloseServiceHandle(hSCM);
end;
```

Обладая дескриптором сервиса, приложение вправе выдать команду на его приостановку, возобновление и завершение работы. Для этого применяется функция

```
Function ControlService(Service : THandle;
                        Control : Cardinal;
                        var Service_Status : _SERVICE_STATUS) : LongBool;
```

Первый параметр метода — дескриптор сервиса. Параметр `Control` определяет поведение сервиса (табл. 32.1). Последний выходной параметр `Service_Status` — указатель на объявленную в модуле `WinSvc` структуру `_SERVICE_STATUS`, хранящую основные характеристики службы. При корректном завершении функция возвратит значение `true`.

Таблица 32.1. *Константы управления сервисом*

Константа	Описание
SERVICE_CONTROL_STOP	Остановка сервиса
SERVICE_CONTROL_PAUSE	Временная остановка сервиса
SERVICE_CONTROL_CONTINUE	Продолжение работы сервиса после временной установки
SERVICE_CONTROL_INTERROGATE	Запрос текущего состояния сервиса

Для определения текущего состояния службы можно воспользоваться услугами функции

```
Function QueryServiceStatus(Service : THandle;
                        var Service_Status : _SERVICE_STATUS) : LongBool;
```

На вход функции подается дескриптор исследуемой службы, а вся информация о ней возвращается в параметр `Service_Status`. Это уже встречавшаяся нам структура, включающая семь полей (листинг 32.2).

Листинг 32.2. Состав структуры `_Service_Status`

```
type _SERVICE_STATUS = record
  dwServiceType: DWORD;       //тип исполняемого модуля сервиса
  dwCurrentState: DWORD;      //текущее состояние сервиса
  dwControlsAccepted: DWORD; //управляющие коды сервиса
  dwWin32ExitCode: DWORD;     //код ошибки старта/останова сервиса
  dwServiceSpecificExitCode: DWORD;//спецификатор ошибки
  dwCheckPoint: DWORD;     //значение контрольной точки, необходимое для
                  //того, чтобы служба могла уведомлять о своей работе
  dwWaitHint: DWORD;       //время, необходимое для изменения состояния
end;
```

Пример управляющего приложения SCP

Продолжим работу над сервером приложений DataSnap, начатую в *главе 31*. Перед нами стоит задача разработать управляющее приложение SCP (Service Control Program), позволяющее настраивать основные параметры службы сервера приложений в трехуровневом проекте баз данных.

Создаем обычный проект VCL Forms, содержащий единственную форму. Переименуйте форму в `frmDataSnapOptions`. При сохранении проекта соответствующий форме программный модуль назовем `DataSnapOptions.pas`. Название для всего проекта можете выбрать самостоятельно.

Если вы повторяли пример подключения приложения к серверу MySQL из *главы 27*, то некоторые элементы управления проекта окажутся вам знакомы (рис. 32.1). Основная идея управления сервисом такова. При необходимости перенастроить сервис администратор запускает приложение. После внесения изменения все новые параметры передаются в системный реестр Windows, откуда их и считывает служба DataSnap.

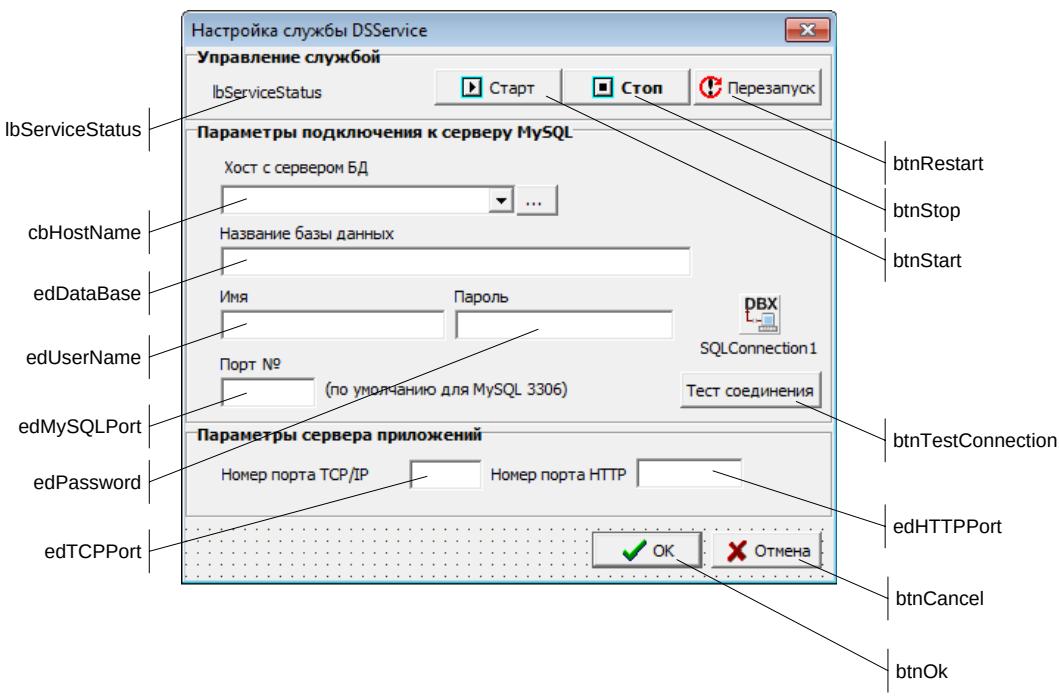


Рис. 32.1. Состав управляющего приложения

Напомню назначение компонентов формы. На верхней панели группировки **Управление службой** находятся элементы, позволяющие управлять службой сервера приложений.

- Метка `lbServiceStatus:TLabel` предназначена для отображения статуса службы (выполняется или остановлена).
- На кнопки `btnStart`, `btnStop` и `btnRestart` возложены задачи запуска, остановки и рестарта сервиса.
- На центральной панели группировки **Параметры подключения к серверу MySQL** располагаются элементы управления, определяющие основные пара-

метры подключения службы сервера приложений DataSnap к серверу баз данных MySQL.

- Комбинированный список `cbHostName:TComboBox` хранит список доступных сетевых хостов.
- Строка ввода `edDataBase:TEdit` указывает путь к файлу с базой данных.
- Строки ввода `edUserName` и `edPassword` предназначены для ввода имени и пароля доступа к серверу баз данных.
- Строка ввода `edMySQLPort` определяет номер коммуникационного порта сервера. Заметим, что по умолчанию MySQL прослушивает порт № 3306.

Нижняя панель группировки **Параметры сервера приложений** управляет параметрами соединения службы с клиентскими приложениями.

- Строка ввода `edTCPPort` отвечает за номер порта TCP.
- Строка ввода `edHTTPPort` определяет номер порта HTTP.

Программирование начнем с объявления нескольких констант и подключения модулей, которые понадобятся во время проектирования (листинг 32.3).

Листинг 32.3. Объявление констант и подключение модулей

```
var
  frmDataSnapOptions: TfrmDataSnapOptions;
const
     ServiceName='DSService';            //физическое имя службы
     section_name='DataSnap_booksmarket';//имя секции в реестре
implementation
uses registry, WinSvc, DBXCommon; //необходимые для работы модули
{$R *.dfm}
//…
```

Обратите внимание на константу `ServiceName`. В ней хранится физическое имя сервиса, позволю себе напомнить, что в случае сервиса DataSnap это имя соответствует названию модуля контейнера сервера. В предыдущей главе во время проектирования службы DataSnap присвоенное по умолчанию имя `ServerContainer1` мы поменяли на `DDService`. Если вы этого не сделали, то имя вашего сервиса может оказаться другим.

Первая процедура, которая появится в нашем приложении, предназначена для получения информации о состоянии службы. По большому счету нас интересует единственный факт — запущена ли служба (листинг 32.4).

Листинг 32.4. Информирование о состоянии сервиса

```
procedure TfrmDataSnapOptions.ServiceInfo;
var hSCM,hService:THandle;
    SS:_SERVICE_STATUS;
```

```
begin
  hSCM:=OpenSCManager(nil,SERVICES_ACTIVE_DATABASE,SC_MANAGER_ALL_ACCESS);
  try
  hService:=OpenService(hSCM,ServiceName,SC_MANAGER_ALL_ACCESS);
    QueryServiceStatus(hService,SS);
    if (SS.dwCurrentState=SERVICE_RUNNING) then {сервис выполняется}
      begin
        lbServiceStatus.Caption:='Служба выполняется';
        btnStart.Down:=true;
      end else
      begin
        lbServiceStatus.Caption:='Служба остановлена';
        btnStop.Down:=true;
      end;
  finally
  CloseServiceHandle(hService);
  CloseServiceHandle(hSCM);
  end;
end;
```

При выполнении процедуры мы получаем дескриптор менеджеру управления службами SCM, затем узнаем дескриптор интересующего нас сервиса. С помощью функции `QueryServiceStatus()` заполняем структуру `SS` сведениями о статусе службы, а затем изучаем поле `dwCurrentState`. В зависимости от того, выполняется служба или нет, изменяем состояние некоторых элементов управления формы.

В момент вывода формы на экран проверяем состояние службы, считываем конфигурационные данные из реестра и передаем их в элементы управления (листинг 32.5).

Листинг 32.5. Вывод на экран окна управления службой

```
procedure TfrmDataSnapOptions.FormShow(Sender: TObject);
begin
ServiceInfo; //узнаем текущее состояние службы
  with TRegIniFile.Create('Software') do{ Читаем данные из реестра}
  begin
   cbHostName.Text := ReadString(section_name, 'hostname', 'localhost');
   edMySQLPort.Text := ReadString(section_name, 'port', '3306');
   edDataBase.Text :=ReadString(section_name,'database','booksmarket');
   edUserName.Text := ReadString(section_name, 'user','root');
   edPassword.Text := ReadString(section_name, 'password','');
   edTCPPort.Text:=ReadString(section_name, 'TCPPort','211');
```

```
    edHTTPPort.Text:=ReadString(section_name, 'HTTPPort','80');
  Free;
  end;
end;
```

Еще одна процедура позволит запускать и останавливать службу сервера DataSnap (листинг 32.6). У процедуры всего один параметр, для старта сервиса в него передается `true`, для останова — `false`.

Листинг 32.6. Процедура запуска/останова сервиса

```
procedure TfrmDataSnapOptions.ControlDSService(Start: boolean);
var hSCM, hService : THandle;
    SS:_SERVICE_STATUS;
    args:pChar;
begin
hSCM:=OpenSCManager(nil,SERVICES_ACTIVE_DATABASE,SC_MANAGER_ALL_ACCESS);
try
  hService:=OpenService(hSCM,ServiceName,SC_MANAGER_ALL_ACCESS);
  if Start=true then StartService(hService,0,args) //старт службы
    else
    ControlService(hService,SERVICE_CONTROL_STOP,SS); //остановка
finally
  CloseServiceHandle(hService); //освободим дескрипторы
  CloseServiceHandle(hSCM);
  Sleep(1000);                  //даем системе время "подумать"
end;
end;
```

Процедура запуска/останова службы вызывается щелчком по кнопке `btnStart` (старт службы) и `btnStop` (остановка службы). Успешность выполнения команды мы проверяем с помощью процедуры `ServiceInfo` (листинг 32.7).

Листинг 32.7. Кнопка `btnStart` запускает службу

```
procedure TfrmDataSnapOptions.btnStartClick(Sender: TObject);
begin
 ControlDSService(true); //старт службы
 ServiceInfo;            //проверка статуса службы
end;

procedure TfrmDataSnapOptions.btnStopClick(Sender: TObject);
begin
```

```
 ControlDSService(false); //остановка службы
 ServiceInfo;             //проверка статуса службы
end;
```

После того как администратор введет параметры, требуемые для установки соединения, необходимо проверить их корректность. Для этого следует щелкнуть по кнопке `btnTestConnection` (листинг 32.8).

Листинг 32.8. Тестирование параметров соединения

```
procedure TfrmDataSnapOptions.btnTestConnectionClick(Sender: TObject);
begin
With SQLConnection1 do
begin
  LoginPrompt:=false;
  DriverName:='MySQL'; //заполняем параметры
  with Params do
  begin
    Clear;
    Add('hostname='+cbHostName.Text);
    Add('user_name='+edUserName.Text);
    Add('password='+edPassword.Text);
    Add('database='+edDataBase.Text);
    Add('port='+ edMySQLPort.Text);
    Add('create=false');
  end;
//необходимые для подключения к серверу БД параметры заполнены
  TRY
  Connected:=true; //попытка соединения
   MessageBoxEx(0,
               pWideChar('Параметры соединения корректны!'),
               pWideChar('Тест успешен!'),
               MB_ICONINFORMATION,MB_OK);

  btnOK.Enabled:=True; //включаем ряд элементов управления
  btnRestart.Enabled:=True;
  Connected:=false; //разрыв соединения
   EXCEPT
       on err:TDBXError do
         MessageBoxEx(0,pWideChar(err.Message),
                     'Ошибка соединения!',MB_ICONEXCLAMATION,MB_OK);
```

```
      else raise;
   END;
  end;
end;
```

Для проверки параметров предпринимается попытка подключения к БД. Ее результаты отображаются на экране, это может быть сообщение о корректности параметров, либо окно с уведомлением об ошибке.

Если параметры соединения корректны, то их следует сохранить в системном реестре (листинг 32.9), откуда их считает сервис во время перезапуска.

Листинг 32.9. Сохранение параметров соединения в реестре

```
procedure TfrmDataSnapOptions.SaveRegistry;
begin
  with TRegIniFile.Create('Software') do
  begin
   WriteString(section_name, 'hostname', cbHostName.Text);
   WriteInteger(section_name,'port',StrToIntDef(edMySQLPort.Text, 3306));
   WriteString(section_name, 'database', edDataBase.Text);
   WriteString(section_name, 'user', edUserName.Text);
   WriteString(section_name,'password', edPassword.Text);
   WriteInteger(section_name, 'TCPPort',StrToIntDef(edTCPPort.Text,211));
   WriteInteger(section_name,'HTTPPort',StrToIntDef(edHTTPPort.Text,80));
  Free;
  end;
end;
```

Для перезапуска службы с новыми параметрами администратору достаточно щелкнуть по кнопке `btnRestart` (листинг 32.10).

Листинг 32.10. Перезапуск службы

```
procedure TfrmDataSnapOptions.btnRestartClick(Sender: TObject);
begin
if MessageBox(Self.Handle,
            pWideChar('Перезапустить службу с новыми параметрами?'),
            pWideChar('Управление службой'),
            MB_YESNO+MB_ICONQUESTION)=idYes then
    begin
      SaveRegistry;   //сохраняем параметры в реестре
      btnStop.Click;  //остановка службы
      btnStart.Click; //старт службы
    end;
end;
```

Таким образом, мы создали управляющее приложение, позволяющее конфигурировать службу сервера приложений. Нам осталось научить сервис получать данные из реестра.

Доработка сервиса DataSnap

Для того чтобы сервис DataSnap, созданный в *главе 31*, получил возможность читать содержимое реестра, необходимо внести несколько незначительных доработок в модуль методов сервера `ServerMethodsUnit1` и в модуль контейнера сервера `ServerContainerUnit1`.

Если сервис в данный момент выполняется, то не забудьте его остановить. После этого вносим в программный код три исправления.

Во-первых, в модуле контейнера сервера объявляем ряд переменных, в которые запишутся параметры соединения из системного реестра и константу `section_name`, хранящую имя целевой секции реестра (листинг 32.11).

Листинг 32.11. Дополнительные параметры и константы модуля контейнера

```
unit ServerContainerUnit1;
//…
var
  DSService: TDSService;
  HostName,MySQLPort,DataBase,UserName,Password:string;
  TCPPort,HTTPPort:word;

const section_name='DataSnap_booksmarket';
implementation
uses Windows, registry, ServerMethodsUnit1;
```

Обратите внимание, что к контейнеру `ServerContainerUnit1` подключены дополнительные модули, необходимые для работы с реестром.

Во-вторых, в момент, предшествующий запуску службы, считываем параметры соединения из реестра (листинг 32.12).

Листинг 32.12. Чтение данных из реестра в момент старта сервиса

```
procedure TDSService.ServiceStart(Sender: TService;
                                  var Started: Boolean);
begin
  with TRegIniFile.Create('Software') do
  begin
   HostName := ReadString(section_name, 'hostname', 'localhost');
   MySQLPort := ReadString(section_name, 'port', '3306');
```

```
   DataBase := ReadString(section_name, 'database', 'booksmarket');
   UserName := ReadString(section_name, 'user','root');
   Password := ReadString(section_name, 'password','');
   TCPPort  :=ReadInteger(section_name, 'TCPPort',211);
   HTTPPort :=ReadInteger(section_name, 'HTTPPort',80);
   Free;
  end;
  DSTCPServerTransport1.Port:=TCPPort;
  DSHTTPService1.HttpPort:=HTTPPort;

DSServer1.Start; {старт службы}
end;
```

В-третьих, описываем событие создания модуля методов `ServerMethodsUnit1`, которое подключает службу к серверу БД (листинг 32.13).

Листинг 32.13. Инициализация соединения с сервером MySQL

```
procedure TServerMethods1.DSServerModuleCreate(Sender: TObject);
begin
With SQLConnection1 do
begin
  LoginPrompt:=false;
  DriverName:='MySQL';
  with Params do
  begin
  Clear;
   Add('hostname='+HostName);
   Add('user_name='+UserName);
   Add('password='+Password);
   Add('database='+DataBase);
   Add('port='+MySQLPort);
   Add('create=false');
  end;

  TRY
  Connected:=true;
   EXCEPT
       on err:TDBXError do
         MessageBoxEx(0,pWideChar(err.Message),
                      'Ошибка',MB_ICONEXCLAMATION,MB_OK);
```

```
      else raise;
   END;
end;
//…
end;
```

Откомпилируйте код и вновь запустите службу на выполнение. С этого момента сервис приобрел необходимую гибкость, и администратор его сможет легко перенастроить.

Создание модуля панели управления

Панель управления предоставляет доступ к глобальным настройкам операционной системы, ее системным компонентам и обслуживаемым устройствам. Для этого пользователю необходимо найти интересующий его значок, например клавиатуры, и дважды по нему щелкнуть левой кнопкой мыши. В результате инициализируется специализированная динамическая библиотека, называемая *модулем панели управления* (Control Panel Module, CPM). Модуль (его часто называют *апплетом*) управляет единственной подпрограммой, обеспечивая пользовательский контроль за параметрами настройки эксплуатационных режимов отдельного системного компонента.

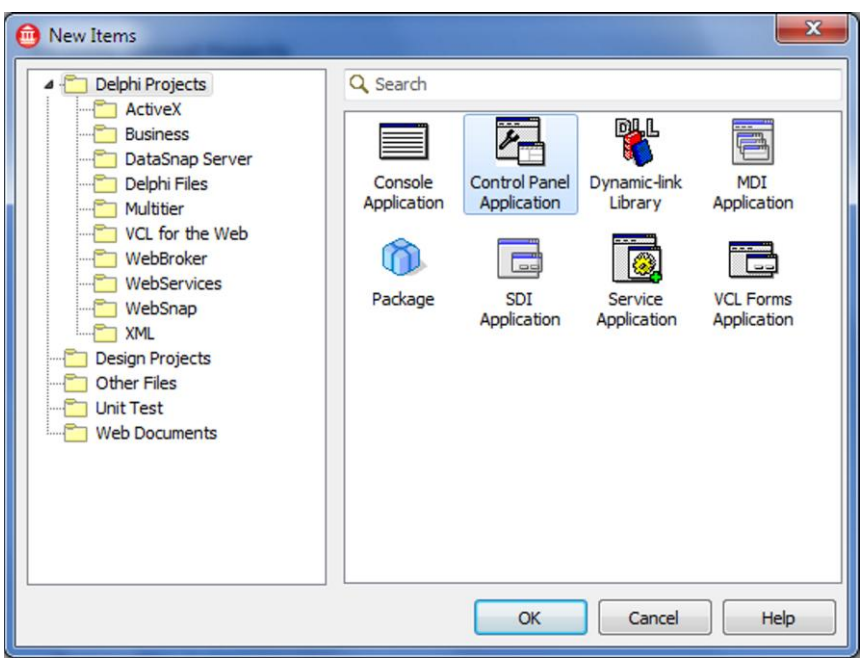


Рис. 32.2. Создание апплета панели управления Windows

Операционная система не против того, чтобы в панели управления располагались ссылки на CPM программных продуктов сторонних производителей. Предлагаю воспользоваться доброжелательностью Windows и внедрить в панель управления апплет, позволяющий управлять сервисом нашего сервера приложений DataSnap.

Выберите пункт меню **File | New | Other**, в окне **New Items** (Новые элементы) найдите пиктограмму **Control Panel Application** и щелкните по кнопке **OK** (рис. 32.2).

В результате в редакторе Delphi раскроется модуль апплета. Обратите внимание на то, что экземпляр класса `TAppletModule` является прямым наследником обычного модуля данных `TDataModule`. Это подтверждает листинг 32.14, содержащий исходный код модуля апплета `AppletModule1`.

Листинг 32.14. Исходный код модуля апплета

```
unit AppletModule1;

interface
uses Windows, Messages, SysUtils, Classes, Graphics, CtlPanel;

type
  TAppletModule1AppletModule = class(TAppletModule)
  private
  { private declarations }
  protected
  { protected declarations }
  public
  { public declarations }
  end;

var AppletModule1AppletModule: TAppletModule1AppletModule;

implementation
{$R *.DFM}
end.
```

Сохраним проект CPM. Имена программных модулей не имеют существенного значения, за единственным исключением: договоримся, что главный файл проекта получит название dsapplet.dproj.

Добавим к проекту разработанную в начале этой главы форму управления опциями службы `frmDataSnapOptions`. Обращаемся к элементу меню **Project | Add to Project** (Проект | Добавить к проекту) и в появившемся диалоге выбора файла находим и подключаем к проекту программный модуль `DataSnapOptions.pas`.

В соответствии с настройками по умолчанию подключенная нами форма подлежит автоматическому созданию в момент инициализации апплета CPM, но в нашем случае в этом нет никакой необходимости. Для устранения данного недостатка откроем головной модуль проекта (пункт меню **Project | View Source**). В коде головного модуля библиотеки нам следует отменить автоматическое создание формы `frmDataSnapOptions`. С этой целью берем в комментарии или просто удаляем строку кода, создающую форму `TfrmDataSnapOptions` (листинг 32.15).

Листинг 32.15. Исходный код головного модуля библиотеки

```
library dsapplet;

uses
  CtlPanel,
  AppletModule1 in 'AppletModule1.pas'
                          {AppletModule1AppletModule: TAppletModule},
  DataSnapOptions in 'DataSnapOptions.pas' {frmDataSnapOptions};

exports CPlApplet;
{$R *.RES}

{$E cpl}
begin
 Application.Initialize;
 Application.CreateForm(TAppletModule1AppletModule,
                        AppletModule1AppletModule);
{!!!}
{Application.CreateForm(TfrmDataSnapOptions, frmDataSnapOptions);<-- удалите
или превратите эту строку в комментарий}
 Application.Run;
end.
```

Отройте в редакторе кода модуль `AppletModule1`. В перечне событий найдите событие `OnActivate()`, оно генерируется, когда пользователь дважды щелкает по пиктограмме апплета на панели управления. Это самый подходящий момент времени для создания формы настроек службы (листинг 32.16).

Листинг 32.16. Обработка события активизации апплета

```
procedure TAppletModule1AppletModule.AppletModuleActivate
                                      (Sender: TObject; Data: Integer);
begin
with TfrmDataSnapOptions.CreateParented(Application.ControlPanelHandle) do
```

```
 begin
  ShowModal;
  Release;
 end;
end;
```

Найдите свойство `Caption` модуля `AppletModule1`. В тексте данного свойства будет содержаться заголовок апплета:

```
Caption:= 'Управление службой DataSnap';
```

Одновременно нажав клавиши <Ctrl>+<F9>, откомпилируйте проект. В результате появится файл CPL с названием dsapplet.cpl, нам осталось лишь зарегистрировать его в панели управления компьютером.

Для того чтобы апплет стал доступным в панели управления компьютером, файл *.cpl достаточно скопировать в системный каталог System32.

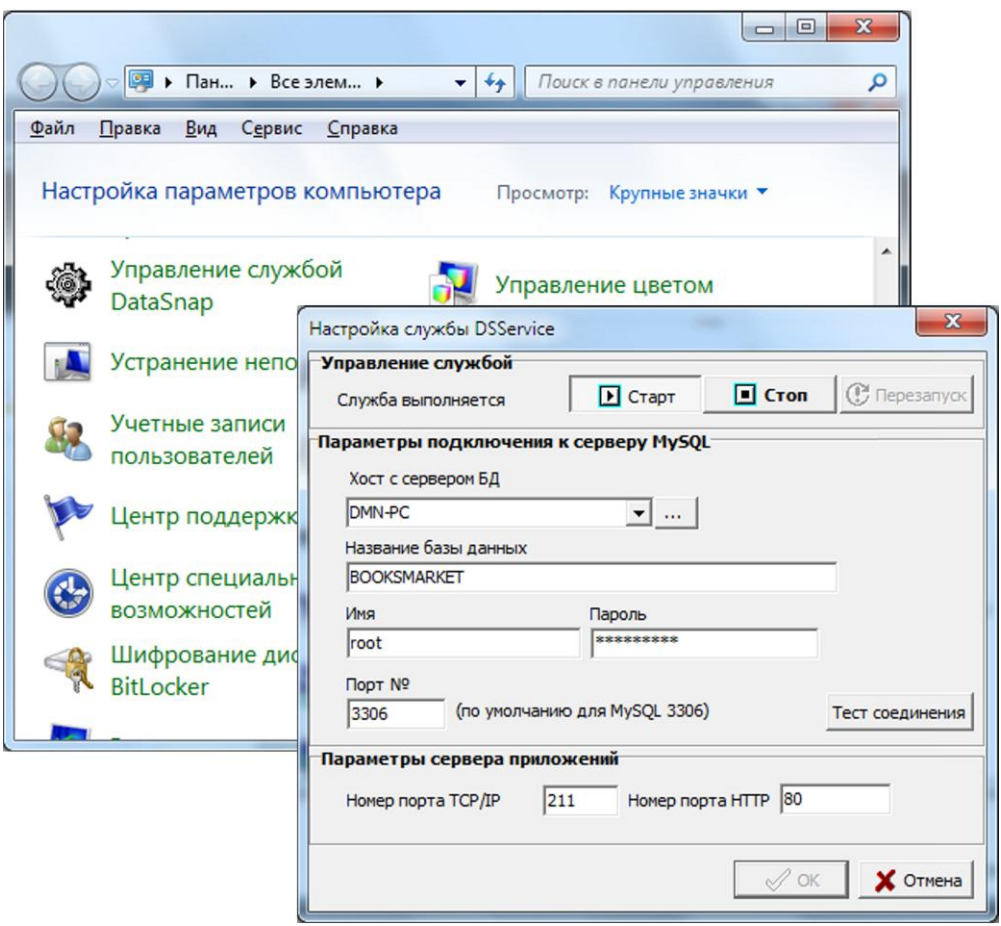


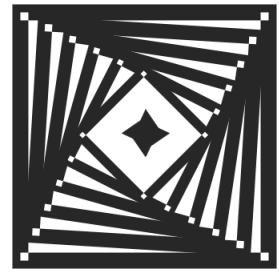
Рис. 32.3. Модуль панели управления службы DSService

На рис. 32.3 приведен экранный снимок разработанного нами модуля панели управления, позволяющего настраивать основные параметры службы сервера приложений DataSnap.

Резюме

Появившаяся в Delphi 2010/XE обновленная версия технологии DataSnap позволяет создавать эффективные трехуровневые проекты БД и обладает многими достоинствами:

- клиентские приложения не нуждаются в драйверах БД, им достаточно только набора библиотек DataSnap;
- сервер приложений способен предоставить клиенту набор данных `TDataSet`;
- появление сервера приложений позволяет расширить возможности по реализации бизнес-правил БД;
- упрощены вопросы работы в сети;
- возможна реализация процедуры аутентификации клиента.
- Теперь мы добавили еще одно преимущество — существенно упростили настройку службы DataSnap.



Глава 33

Отчеты Rave Reports

Несмотря на наличие развитых средств представления данных на современных устройствах отображения (дисплеи, проекторы, электронные доски), человеку значительно удобнее работать с бумажными носителями информации. Поэтому одна из важных сторон разработки приложений баз данных — подготовка разнообразных отчетов для печати. Для обычного пользователя внешняя сторона печати документов выглядит весьма прозаично: щелкнув по кнопке печати документа, можно расслабиться за чашечкой кофе, изредка подкидывая в принтер очередную порцию бумаги. Романтики здесь не больше, чем в ежедневной поездке в метро. Для программиста, все обстоит значительно сложнее, его обязанность — сделать все необходимое чтобы пользователь не узнал, какие нешуточные "страсти" кипят в недрах приложения баз данных.

Эта глава открывает раздел, посвященный способам разработки отчетов для БД. Здесь мы уделим внимание используемому со времен Delphi 7 популярному продукту компании Nevrona Designs — Rave Reports. Начнем с изучения назначения компонентов Rave Reports и порядка их применения в проектах баз данных Delphi.

Обзор компонентов Rave Reports

Приложение Rave Reports является средой разработки визуальных отчетов, состоящей из редактора отчетов и набора компонентов, которые вы обнаружите на страничке **Rave** палитры компонентов Delphi.

Нетрудно догадаться, что связующим звеном между отчетом Rave Reports и проектом баз данных Delphi вправе выступать любой компонент, построенный на основе универсального набора данных `TDataSet`. Это может быть таблица, запрос, хранимая процедура, а в многоуровневых приложениях DataSnap — метод сервера. Единственное условие — готовность набора данных по первому требованию компонентов Rave Reports предоставить доступ к записям таблиц БД.

Соединение *TRvCustomConnection*

Для взаимодействия с потомками класса `TDataSet` в состав Rave Repots включен "специалист по переговорам" посредник — `TRvCustomConnection` (рис. 33.1). Ключевое свойство компонента:

```
property DataSet: TDataSet;
```

С помощью `DataSet` компонент присоединяется к целевому набору данных.

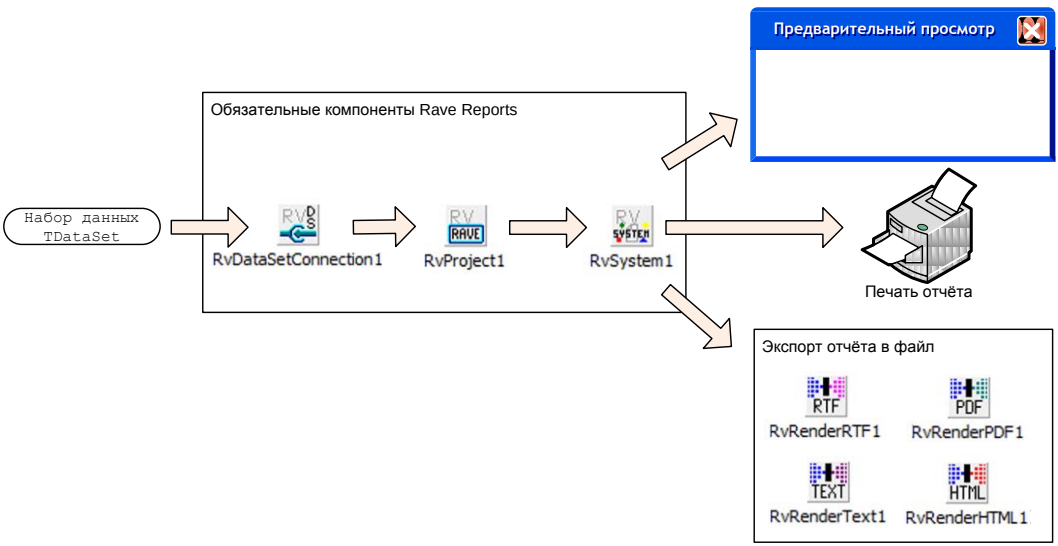


Рис. 33.1. Место компонентов Rave Reports в проекте БД

Заметим, что на странице **Rave** палитры компонентов Delphi имеются еще три компонента, предназначенные для обеспечения взаимодействия между отчетом и данными: `TRvCustomConnection`, `TRvTableConnection` и `TRvQueryConnection`. Два последних компонента давно устарели и сохранили свое присутствие на палитре компонентов только ради обеспечения обратной совместимости с проектами баз данных, написанными на основе устаревшего механизма BDE. Компонент `TRvCustomConnection` специализируется на получении информации из источников, отличных от БД. Таким образом, из всего перечня компонентов `TRv…Connection` для современного разработчика баз данных основной интерес представляет лишь `TRvDataSetConnection`.

Проект *TRvProject*

Базовым элементом отчета выступает компонент `TRvProject`. Хотя на представленной схеме (см. рис. 33.1) он следует сразу за соединением `TRvCustomConnection`,

не ищите в инспекторе объектов свойство, с помощью которого проект TRvProject должен подключиться к соединению. Подключение выполняется иначе. Во время проектирования отчета двойной щелчок по компоненту вызывает встроенный редактор, в котором мы выберем источник данных и сохраним настройки о нем во внешнем файле с расширением *.rav. Файл RAV будет содержать не только информацию о подключении, но и все параметры отчета. Указанный файл подключается к компоненту TRvProject с помощью свойства

```
property ProjectFile: string;
```

Файлов отчетов может быть сколько угодно много. В зависимости от задач, стоящих перед разработчиком, мы сможем выбирать тот или иной файл отчета и вызвать метод

```
procedure Open;
```

который загрузит из файла *.rav в память все необходимые для создания отчета данные. Работу с проектом отчета традиционно завершает процедура

```
procedure Close;
```

Предусмотрен ряд методов, позволяющих управлять группой отчетов, среди них SelectReport(), GetReportList(), ReportDescToMemo(), ReportDesc(), ReportName() и ReportFullName().

Компонент TRvProject нуждается в обязательной поддержке еще одного коллеги со странички **Rave** палитры компонентов Delphi — TRvSystem. Ссылка на системный элемент хранится в свойстве

```
property Engine: TRpComponent;
```

В руках главного компонента должен находиться и основной метод

```
procedure Execute;
```

Процедура дает команду на подготовку отчета к печати.

Системный компонент *TRvSystem*

Третий по счету обязательный компонент — TRvSystem отвечает за три направления работы: предварительный просмотр отчета, его вывод на печать и экспорт в файл. Выполнять одну из этих задач компонент начинает сразу после того, как его старший коллега TRvProject вызовет метод Execute(). Направление деятельности определяется состоянием свойства

```
property DefaultDest: TReportDest;
```

По умолчанию свойство установлено в состояние rdPreview — вывод окна предварительного просмотра, вы можете установить значение rdFile (экспорт в файл) или rdPrinter (постановка задания на печать).

Компонент обладает пятью свойствами, определяющими все аспекты взаимодействия будущего отчета и системы (табл. 33.1).

Таблица 33.1. *Настройка поведения компонента* `TRvSystem`

Опции	Описание
property `SystemFiler: TSystemFiler;`	Определяет порядок печати в файл
property `SystemOptions: TSystemOptions;`	Системная конфигурация компонента
property `SystemPreview: TSystemPreview;`	Особенности окна предварительного просмотра
property `SystemPrinter: TSystemPrinter;`	Характеристики системного принтера
property `SystemSetups: TSystemSetups;`	Допустимые опции печати

Основные события компонента `OnPreviewShow()`, `OnPreviewSetup()`, `OverridePreview()` и `OverrideStatus()` позволяют реагировать на вывод окна предварительного просмотра.

Компоненты экспорта отчета в файл

Неоспоримое преимущество системы построения отчетов Rave Report — возможность экспорта данных из отчета в широко распространенные форматы файлов. Более того, для организации экспорта в тот или иной формат достаточно просто разместить соответствующий компонент на форме. Там их сразу обнаружит компонент `TRvSystem` и осуществит все необходимые подготовительные операции.

Преобразование форматов осуществляют следующие компоненты:

- `TRvRenderText` — в текстовый формат;
- `TRvRenderRTF` — в расширенный текстовый формат (Rich Text Format, RTF);
- `TRvRenderHTML` — в страницы HTML;
- `TRvRenderPDF` — в формат Acrobat Reader.

ЗАМЕЧАНИЕ

При необходимости с сайта разработчика системы **http://www.nevrona.com** можно скачать компоненты `TRvRenderMetafile`, `TRvRenderJPEG` и `TRvRenderBitmap`, позволяющие экспортировать данные в графический формат.

Пример работы с редактором Rave Reports

Попробуем наши силы при создании отчета для демонстрационной базы данных "Книжный магазин".

Расположите в модуле данных либо на главной форме проекта следующий перечень компонентов:

- Предпочтительный для вас компонент — набор данных, например `TADOQuery`. Запрос позволит нам получить из БД все необходимые данные. Переименуйте компонент в `Query_Report`. Воспользовавшись свойством `Connection`, подключите запрос к компоненту-соединителю `ADOConnection1`.
- Для подключения к набору данных нам потребуется компонент `RvDataSetConnection1:TRvDataSetConnection1`. Воспользуйтесь свойством `DataSet` и подключите компонент к запросу `Query_Report`.
- Компонент `RvSystem1:TRvSystem`.
- Компонент `RvProject1:TRvProject`. С помощью свойства `Engine` подключите компонент к `RvSystem1`.

Подготовку отчета начинаем с формирования текста инструкции компонента `TADOQuery`. Отредактируйте текст запроса в свойстве SQL. Наш пример предназначен для получения информации обо всех книгах, имеющихся на складе книжного магазина, поэтому сбору подлежат практически все данные из таблиц БД. Например, на диалекте Access такой запрос будет выглядеть так, как в листинге 33.1.

Листинг 33.1. Текст запроса к БД "Книжный магазин"

```
SELECT SUPPLIERS.SUPPLIER, CONTRACTS.CONTRACT, CONTRACTS.CONTRDATE,
       ORDERS.ORDERNUM, ORDERS.ORDERDATE, DELIVERIES.AMOUNT,
       PUBLISHERS.PUBLISHER, BOOKS.TITLE, WRITERS.WRITER, GENRES.GENRE

FROM WRITERS

INNER JOIN ((SUPPLIERS INNER JOIN (PUBLISHERS INNER JOIN ((CONTRACTS
INNER JOIN ORDERS ON CONTRACTS.CONTRACT_KEY = ORDERS.CONTRACT_KEY)
INNER JOIN (GENRES INNER JOIN ((BOOKS INNER JOIN DELIVERIES ON BOOKS.BOOK_KEY
= DELIVERIES.BOOK_KEY) INNER JOIN [GENRES_BOOKS] ON BOOKS.BOOK_KEY =
[GENRES_BOOKS].BOOK_KEY)
ON GENRES.GENRE_KEY = [GENRES_BOOKS].GENRE_KEY) ON ORDERS.ORDER_KEY =
DELIVERIES.ORDER_KEY) ON PUBLISHERS.PUBLISHER_KEY = DELIVERIES.PUBLISHER_KEY)
ON SUPPLIERS.SUPPLIER_KEY = CONTRACTS.SUPPLIER_KEY) INNER JOIN [WRITERS_BOOKS]
ON BOOKS.BOOK_KEY = [WRITERS_BOOKS].BOOK_KEY) ON WRITERS.WRITER_KEY =
[WRITERS_BOOKS].WRITER_KEY

ORDER BY CONTRACTS.CONTRACT, CONTRACTS.CONTRDATE;
```

Впрочем, предложенный мною запрос вовсе не догма, поэтому в своем примере вы можете написать любую другую, устраивающую вас инструкцию `SELECT`.

В контекстном меню компонента `RvProject1` находим пункт **Rave Visual Designer** и вызываем дизайнер отчетов Rave Reports (рис. 33.2).

Дизайнер отчетов обладает интуитивно-понятным интерфейсом. Центральная часть редактора отведена под макет страницы будущего отчета. Слева от страницы расположилась панель свойств, она отображает свойства активного в текущий момент элемента управления. Изменяя значения свойств, мы можем настраивать внешний вид и определять особенности поведения компонентов. Справа от макета

страницы находится панель с иерархией элементов проекта. Позднее, когда мы сконструируем свой первый отчет, эта панель значительно упростит поиск задействованных в проекте компонентов.

Но пока иерархия отражает только узлы верхнего уровня:

1. **Report Library** (библиотека отчетов) содержит все входящие в состав проекта отчеты.
2. **Global Page Catalog** (глобальный каталог) представляет собой хранилище шаблонов, на основе которых можно создавать новые отчеты.
3. **Data View Dictionary** (словарь представления данных) после того, как мы присоединимся к набору данных БД, отобразит входящие в отчет поля данных.

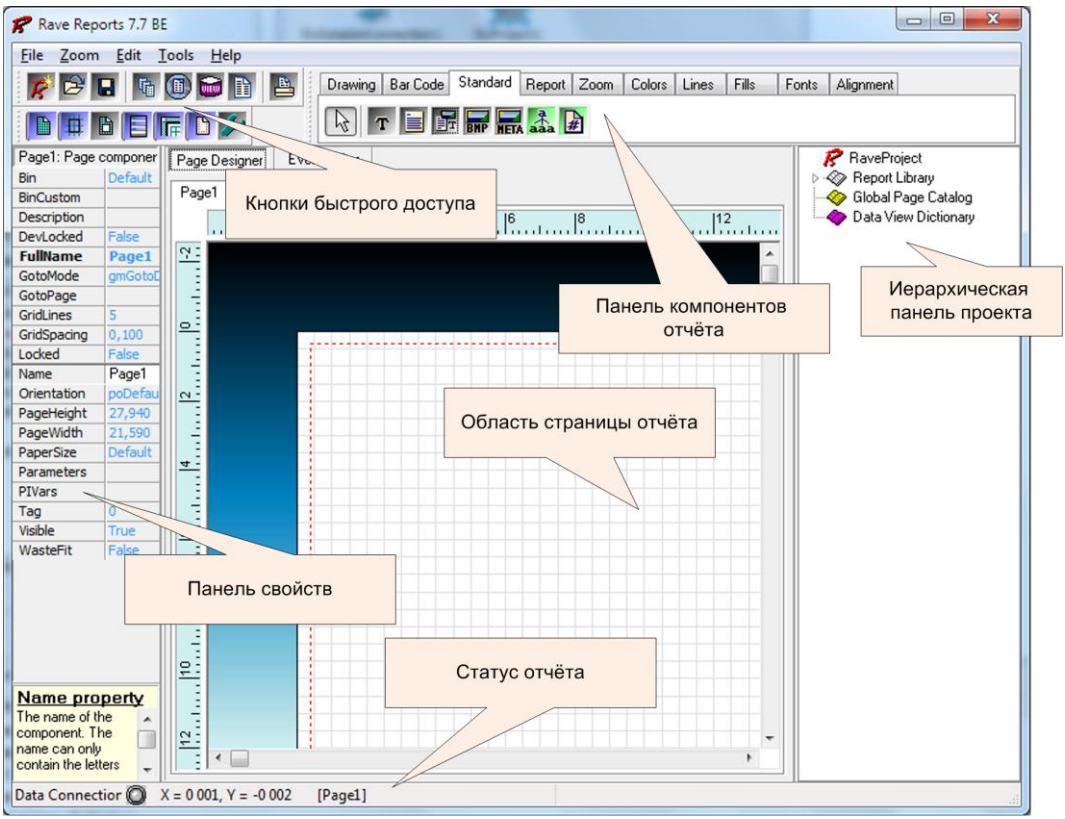


Рис. 33.2. Интерфейс дизайнера отчетов Rave Reports 7.7

Доступ ко всем компонентам, необходимым для построения отчета, а также к дополнительному функционалу по их настройке предоставляет панель компонентов отчета. Управление дизайнером осуществляется с помощью главного меню программы и кнопок быстрого доступа.

После знакомства с дизайнером Rave Reports сделаем очередной шаг. Нам предстоит определить способ соединения отчета с базой данных. Находим элемент

меню **File | New Data Object** (Файл | Новый объект данных) и в появившемся на экране диалоговом окне **Data Connections** (Соединения с данными) выбираем строку **Direct Data View** (Прямое представление данных). Эта операция укажет редактору отчета, что в качестве источника данных выбран компонент, потомок класса `TRvDataSetConnection`. Щелчок по кнопке **Next** выведет окно выбора доступных компонентов `TRvDataSetConnection`, в нашем примере он у нас единственный, поэтому смело нажимаем кнопку **Finish**. В результате у узла **Data View Dictionary** (на иерархической панели, расположенной в правой части редактора отчетов) появится дочерний узел с перечнем всех передаваемых в отчет полей данных (рис. 33.3).

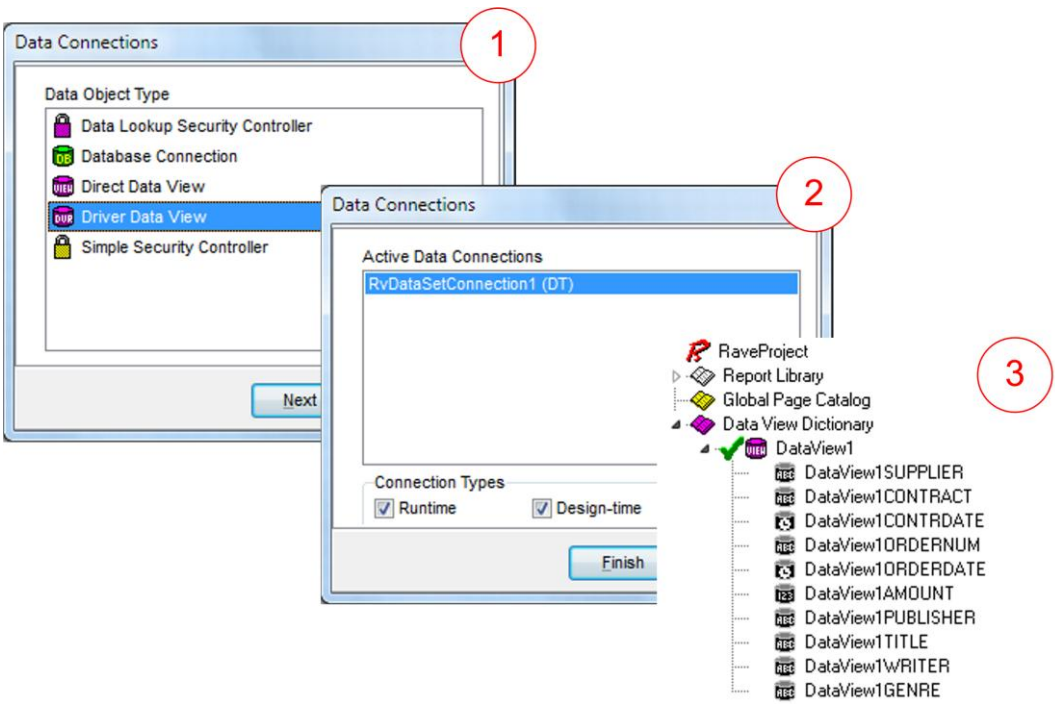


Рис. 33.3. Этапы подключения проекта отчета к источнику данных

Все необходимые подготовительные операции завершены. Для создания своего первого отчета Rave Reports воспользуемся услугами мастера отчетов. Выбираем пункт меню **Tools | Report Wizards | Simple Table** (Инструменты | Мастера отчетов | Простая таблица). В появившемся окне выбираем доступный просмотр данных, он у нас пока только один — DataView1 (рис. 33.4). В очередном окне отмечаем поля, данные из которых следует включить в отчет. Если востребованы все поля, то проще всего щелкнуть по кнопке **All** (Все). Следующее (третье по счету) окно предложит вам определить порядок следования колонок в отчете. В четвертом окне мастер попросит нас назначить заголовок отчета и отступы на странице. Пятое и последнее окно мастера предназначены для настройки характеристик шриф-

тов. Выбрав гарнитуру, стиль и предпочтительные размеры шрифта, нажимаем кнопку **Generate** (Генерировать). В ответ на это система сгенерирует новый отчет.

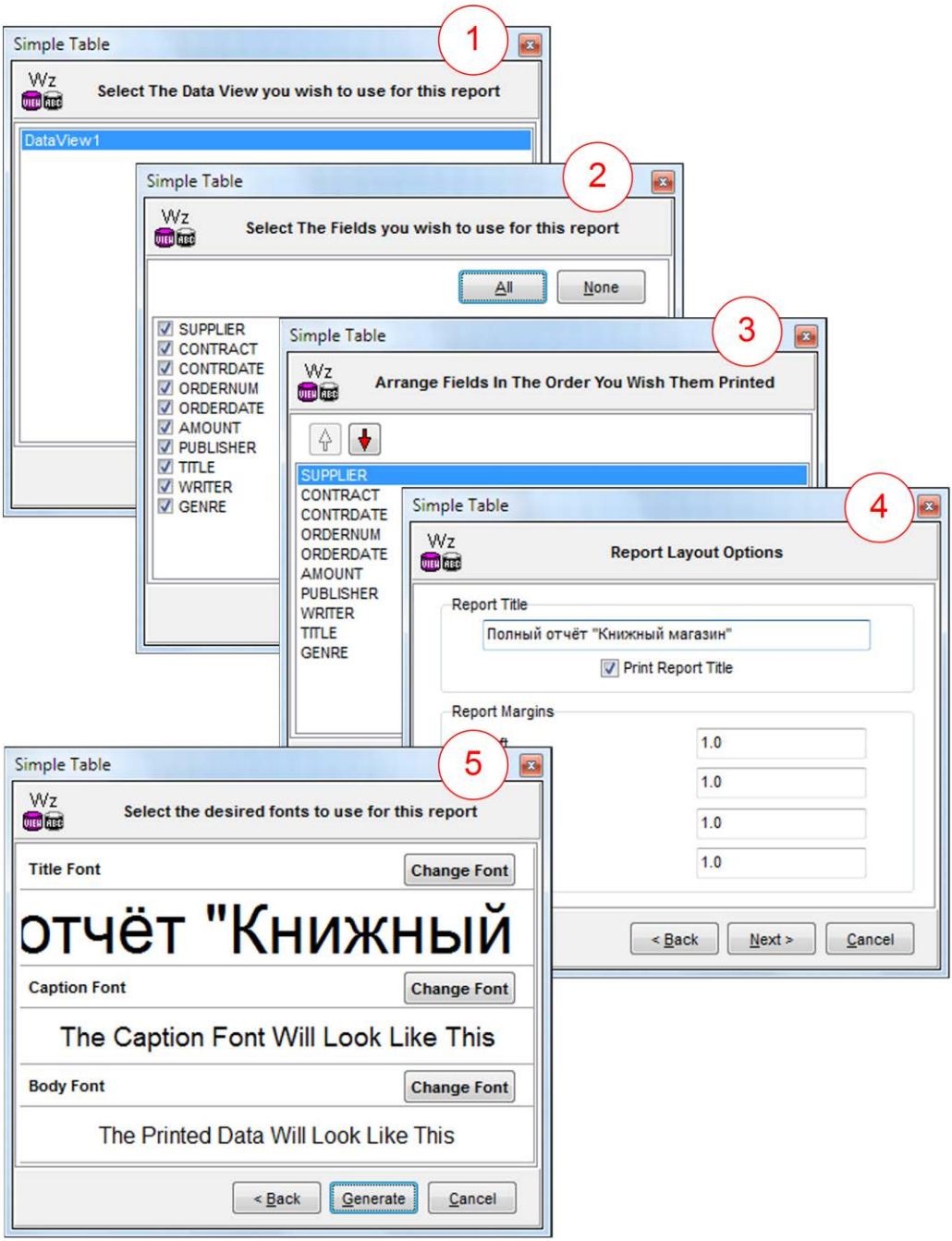


Рис. 33.4. Этапы работы мастера Simple Table

Чтобы взглянуть на плоды работы мастера, активизируйте компонент `Query_Report` и выберите пункт меню **File | Execute Report** (Файл | Выполнить отчет). Чтобы пока еще "сырой" отчет не отправился прямиком на принтер, в появившемся окне **Output options** (Выходные опции) проконтролируйте установку элемента **Preview** (Просмотр). Только после этого нажимаем кнопку **OK**. В результате на экране компьютера отобразится окно предварительного просмотра, благодаря которому вы сможете оценить соответствие внешнего вида будущего отчета вашим пожеланиям. Заранее не расстраивайтесь, мастер далеко не всесилен, и после его работы программисту обычно приходится вносить много правок.

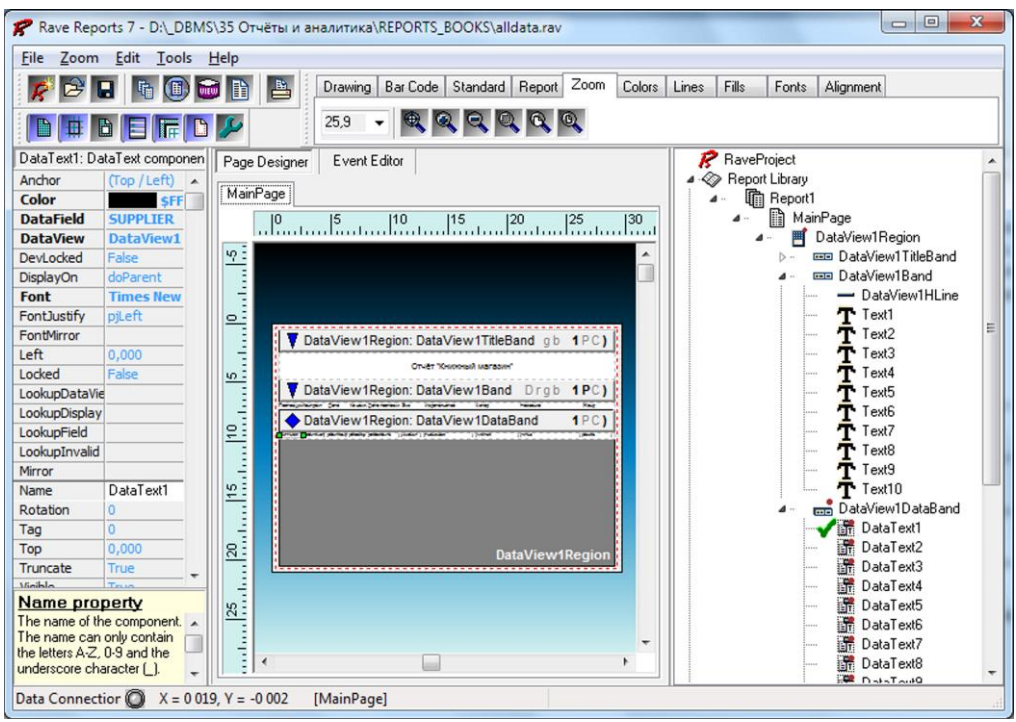


Рис. 33.5. Редактор Rave Report с готовым отчетом

В первую очередь может потребоваться внесение изменений в параметры листа бумаги. Например, в нашем случае, когда отчет содержит много колонок, предпочтительнее выбрать альбомную ориентацию листа. Для того чтобы добраться до этих важнейших характеристик бумажного листа, раскройте узел **Report Library**, предоставляющий доступ к библиотеке отчетов. На данный момент в библиотеке содержится единственный отчет с названием по умолчанию **Report1**. Развернув этот узел, вы получите доступ к характеристикам страницы отчета **Main Page** и областям отчета (рис. 33.5):

- **DataView1Region** — область всего отчета.
- **DataView1TitleBand** — полоса заголовка всего отчета.

- **DataView1Band** — полоса заголовков таблицы отчета. Раскрыв узел DataView1Band, вы получите доступ к текстовым компонентам, отвечающим за прорисовку шапки таблицы.
- **DataView1DataBand** — область основных данных. Узел содержит компоненты DataText, обеспечивающих отображение данных отчета.

Обратите внимание, что на выбор каждого из перечисленных узлов реагирует панель свойств отчета. Например, при выборе узла **Main Page** (Главная страница) на панели свойств отобразятся параметры страницы.

Все остальное в ваших руках. Настраивая на свой вкус свойства различных компонентов, разработчик придает отчету требуемый формат.

Завершив разработку отчета, обязательно сохраним наш труд, для этого выбираем пункт меню **File | Save** и записываем файл отчета alldata.rav в папку с проектом.

Вызов отчета из приложения

Для вызова готового отчета из приложения Delphi необходимо позаботиться о двух вещах: убедиться в готовности к работе компонента-набора данных и передать имя файла с отчетом компоненту `RvProject1`.

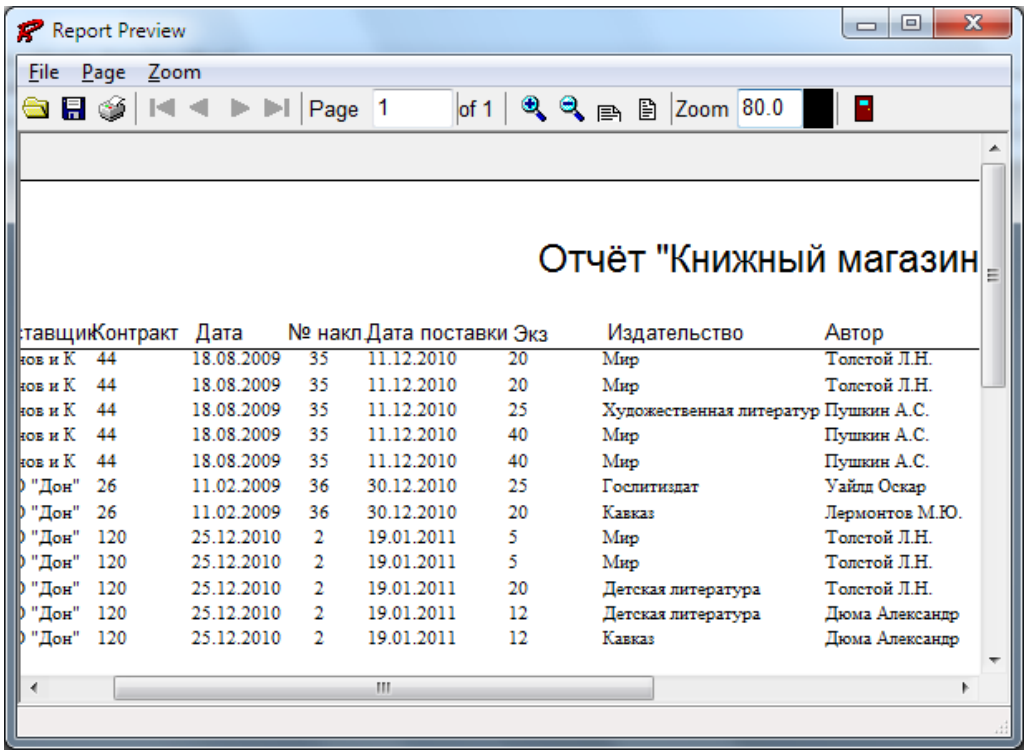


Рис. 33.6. Предварительный просмотр отчета перед печатью

Исходный код очень прост (листинг 33.2).

Листинг 33.2. Обращение к отчету Rave Reports из приложения Delphi

```
procedure TDataModule1.ExecuteRaveReport(FileName: string);
begin
if Query_Report.Active=false then Query_Report.open;
   RvProject1.ProjectFile:=FileName;
   RvProject1.Execute;
end;
```

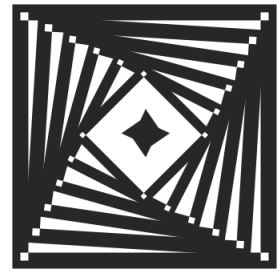
Результат нашей работы иллюстрирует рис. 33.6. Это окно предварительного просмотра, в котором отображен готовый к отправке на печать отчет демонстрационной БД книжного магазина.

Резюме

Дизайнер отчетов Rave Reports компании Nevrona Design поставляется вместе с Delphi достаточно давно. Отсчет ведется с вышедшей в свет в 2002 году седьмой версии среды проектирования. За это время у Rave Reports появилось много поклонников, которые среди достоинств Rave Reports в первую очередь выделяют доброжелательность и простоту системы проектирования отчетов. Нельзя не отметить профессиональную работу программистов компании Nevrona Design, создавших ряд незаурядных компонентов Rave. С одной стороны, компоненты идеальны для начинающего разработчика, т. к. практически не требуют дополнительной настройки, с другой стороны, эти компоненты обладают огромным потенциалом, который по заслугам оценивают многие профессионалы.

Несмотря на все свои неоспоримые преимущества, система Rave Reports не занимает доминирующее положение среди средств разработки отчетов. В следующих главах мы познакомимся с альтернативными подходами проектирования отчетов для приложений баз данных. И в первую очередь мы научимся ставить на службу проектов баз данных приложения из состава пакета программ Microsoft Office.

Глава 34



Разработка динамических библиотек для проектов БД

Рано или поздно любой программист приходит к однозначному выводу — во многих созданных им программных продуктах неоднократно встречаются абсолютно одинаковые (или очень близкие по духу) программные блоки. Подтверждение тому — пример из предыдущей главы: ни один проект приложений баз данных не обходится без генерации печатных отчетов. При создании очередного приложения мы вынуждены вновь и вновь писать тот же самый программный модуль. Не исключаю, что благодаря многократному повторению одних и тех же действий можно до такой степени отточить свое мастерство (в особенности скорость набора на клавиатуре), что вам просто не будет равных в этой области… А в действительности разработка проекта из творческого процесса превращается в рутинное занятие, в котором большая часть времени посвящается переписыванию строк, ранее изобретенных процедур и функций в новые модули. Такое утомительное времяпрепровождение гарантированно превратит даже самого безобидного флегматика в существо, по своей раздражительности успешно конкурирующее с носорогом средних размеров. Да и сами листинги начинают "пухнуть" не по дням, а по часам и, в конце концов, теряют читабельность. Вот такие печальные дела… К счастью, нет безвыходных ситуаций. Одним из решений может стать разделение проекта на исполняемый модуль и группу динамически подключаемых библиотек.

Общая характеристика DLL

Динамически подключаемая библиотека (Dynamic Link Library, DLL) представляет собой отдельный программный модуль, содержащий код, данные или ресурсы (или все вместе) и допускающий совместное их использование несколькими приложениями. Применение динамически подключаемых библиотек вносит неоспоримые улучшения как в процесс программирования, так и в эксплуатацию программного продукта.

- Исключается необходимость компоновки всего кода приложения в один исполняемый файл.

- Значительно снижаются размеры файлов проекта и потребляемые ими ресурсы.
- Появляется возможность применять ранее написанные библиотеки в новых проектах, отказываясь от многократного повтора одного и того же кода в разных приложениях.
- Допускается создание разделяемых между приложениями ресурсов, например, битовых образов, курсоров, анимированных картинок.
- Упрощается локализация ошибок в проекте.
- Выпуск новой версии программного продукта может свестись к обновлению одной-двух библиотек, а не проекта в целом.
- Предоставляется возможность одновременной работы над одним проектом группе программистов. Разделение труда, когда один работает над исполняемым модулем, второй с первой библиотекой и т. д., значительно упрощает и, что очень важно, ускоряет рождение программного продукта. А если библиотеки вдруг не стыкуются, то очень просто во всем обвинить руководителя проекта, который опять что-то не продумал (шутка).
- Сторонние разработчики ПО могут применять созданные нами DLL (и наоборот).
- Обеспечивается совместное использование библиотечного модуля несколькими процессами.

С помощью DLL проект из закостенелого кома превращается в группу разумно взаимодействующих модулей, становясь более открытым и надежным.

Именно благодаря названным преимуществам динамические библиотеки составляют основу архитектуры построения современных версий Windows. При изучении Windows вы, наверное, уже сталкивались с такими файлами, как KERNEL32.DLL, COMMDLG32.DLL, GDI32.DLL. Эти и многие другие динамические библиотеки содержат функции интерфейса программирования (API), ресурсы Windows, широко используемые программистами во время создания приложений. Например, модуль KERNEL32 отвечает за управление памятью и инициализацию процессов, COMMDLG32 содержит все стандартные диалоговые окна, GDI32 обслуживает графический интерфейс.

Для того чтобы понять принцип работы динамических библиотек, нужно знать ответ на вопрос: что происходит внутри операционной системы при запуске приложения? В первую очередь операционная система создает объект главного процесса (process kernel object). Процесс сразу запускает первичный поток (primary thread). В 32-разрядных версиях Windows стартующий процесс в свое распоряжение получит отдельное адресное пространство размером 2^{32} (4 294 967 296) байт (4 Гбайт) виртуальной памяти.

Для поддержания адресации распределяется участок памяти не только из оперативной памяти, но и из системного страничного файла (файла подкачки). Чтобы избежать загрузки всего исполняемого файла в оперативную память или в страничный файл, после создания объекта главного процесса создается новый объект, именуемый объектом отображения файла. Его задача — отобразить выполняемый файл в адресное пространство процесса.

После успешного отображения исполняемого файла в виртуальную память операционная система разыскивает внутри этого файла все ссылки на динамически подключаемые библиотеки и эти библиотеки отображаются в тот же процесс (где-то внутри участка адресуемой памяти главного процесса). Владельцем всех данных, объектов и переменных динамической библиотеки становится поток, вызвавший функцию DLL. Для каждого файла библиотеки создается свой собственный объект отображения файла. Если, в свою очередь, внутри динамической библиотеки существуют ссылки на другие файлы библиотек, то последние также отображаются в адресное пространство.

Главный процесс способен загружать одну и ту же динамическую библиотеку многократно, запуская дополнительные потоки. Но при этом отображение DLL в адресное пространство происходит только при первой загрузке. Последующие загрузки осуществляют приращение счетчика обращений к библиотеке. Если DLL загружалась N раз, то образ библиотеки в адресном пространстве процесса будет храниться до тех пор, пока не произойдет N выгрузок. С каждой выгрузкой динамической библиотеки счетчик обращений уменьшает свое значение на единицу. Нулевое значение счетчика — признак того, что библиотека более не нужна.

При завершении работы образ каждого DLL-файла удаляется из памяти вместе с соответствующим объектом отображения. Процесс завершится до удаления образа EXE-файла и объекта главного процесса. С завершением главного процесса автоматически освобождается используемая память.

Создание шаблона динамической библиотеки в Delphi

Для создания динамической библиотеки в среде Delphi предусмотрен отдельный мастер. Для его вызова в меню **File** выберите команду **New** и в появившемся на экране диалоговом окне **New Items** на странице **New** найдите пиктограмму DLL (рис. 34.1).

Нажав кнопку **OK**, вы получите заготовку проекта новой динамической библиотеки. Создайте проект DLL и сохраните его в отдельном каталоге под именем `mydll.dpr`. А теперь просмотрим листинг будущей библиотеки (листинг 34.1).

Листинг 34.1. Исходный код библиотеки mydll

```
library mydll;

{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  Project-View Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL--even those that
```

```
  are nested in records and classes. ShareMem is the interface unit to
  the BORLNDMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using BORLNDMM.DLL, pass string information
  using PChar or ShortString parameters. }

uses  SysUtils, Classes;

{$R *.RES}

begin

end.
```

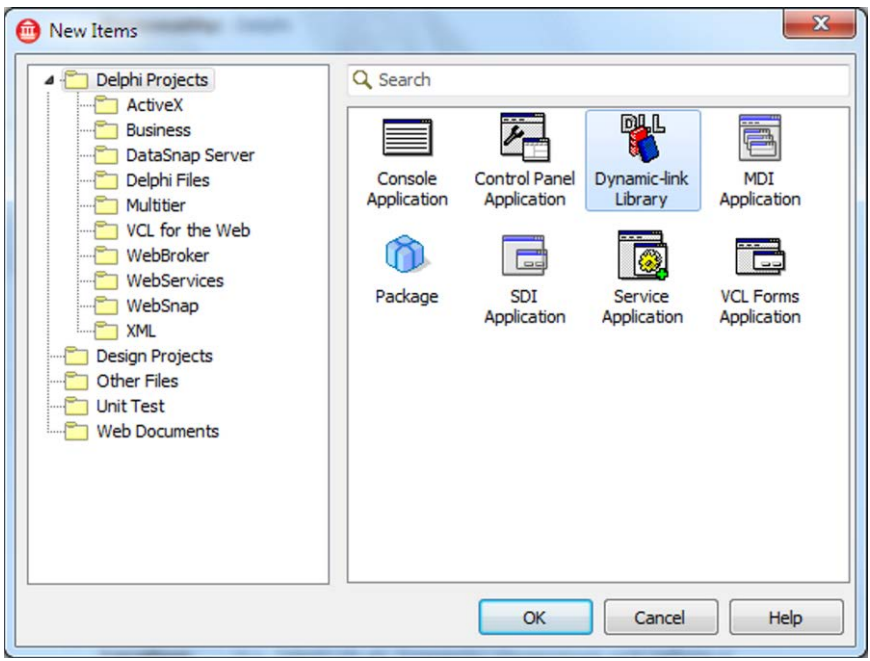


Рис. 34.1. Создание проекта DLL

В отличие от уже знакомых нам листингов, начинающихся со слова `program` или `unit`, в заглавной строке вы встретите ключевое слово `library`. Это и есть визитная карточка исходного кода динамически подключаемой библиотеки. Первое, что бросается в глаза, — устрашающий по объему текст комментария. В его строках разработчики Delphi предупреждают программиста о том, что если хотя бы одна из экспортируемых функций DLL требует данные строкового типа (в качестве параметра или в виде возвращаемого значения), то самым первым в операторе `uses` должен стоять модуль `ShareMem`. Такая необходимость обусловлена особенностью

размещения в памяти строкового типа данных `String`, `AnsiString` и `UnicodeString`. Дело в том, что перечисленные типы данных в дополнение к памяти, содержащей символьные данные, еще используют 12-байтовую структуру, хранящую техническую информацию о размере памяти, предназначенной для строки, счетчик обращения к строке и ее физическую длину. Модуль `ShareMem` позволяет корректно передать между библиотекой и исполняемым файлом как служебную, так и полезную информацию, в результате исполняемый и библиотечный модули начинают говорить "на одном языке".

ВНИМАНИЕ!

Единственный недостаток использования дополнительного модуля SHAREMEM заключается в том, что при установке своего программного продукта на машину заказчика в состав проекта необходимо включить еще одну библиотеку — BORLNDMM.DLL (найти ее вы сможете в каталоге \Bin). Успокаивает то, что ее размер не превышает 30 Кбайт. При желании от библиотеки BORLNDMM.DLL можно отказаться, но в этом случае строковую информацию следует передавать при помощи `PChar` или `ShortString`.

Экспортирование функций DLL

Чтобы приложение или другая библиотека могли обращаться к функциям внешней динамической библиотеки, эти функции должны быть помечены, как экспортируемые. Поэтому синтаксис объявления экспортируемых библиотекой методов имеет ряд небольших отличий от привычного описания обычных функций Delphi. Четко выделяются три особенности:

- необходимо выбрать соглашение о вызовах для передачи параметров от исполняемого модуля к динамической библиотеке;
- экспортируемая функция объявляется вместе с директивой `export`;
- имена экспортируемых библиотекой процедур и функций указываются в секции `exports`.

Прежде чем написать первую экспортируемую функцию, программист принимает решение о способе передачи данных между модулями проекта. В Delphi различают пять соглашений о вызовах (протоколов), отличающихся друг от друга особенностями:

- использования стека (иногда и регистров процессора) для передачи параметров функций;
- направлением заполнения стека (как есть, слева направо или справа налево);
- очистки стека.

Такое разнообразие протоколов объясняется достаточно просто — стремлением разработчиков Delphi к построению максимально гибкого программного продукта, совместимого с другими языками программирования.

Директива `REGISTER` считается наиболее эффективным протоколом передачи данных и устанавливается по умолчанию. Здесь по возможности для первых трех

передаваемых параметров используются расширенные регистры процессора (EAX, EDX и ECX). Остальные параметры (если они есть) передаются как в директиве PASCAL. К сожалению, из-за уникальности протокола он пригоден только для компиляторов Delphi или C Builder.

Соглашение о вызовах PASCAL по скорости слегка уступает REGISTER. К недостаткам протокола можно отнести отсутствие поддержки переменного числа параметров.

Директива CDECL предназначена для организации вызовов в духе языка Си. Протокол допускает передачу переменного числа параметров, но, к сожалению, в среде Delphi это не всегда имеет смысл.

Директивы STDCALL и SAFECALL служат для обращения к API Windows. Протокол SAFECALL предназначен для объявления методов с дуальным интерфейсом. Например, это соглашение о вызовах используется для обращения к объектам автоматизации.

ЗАМЕЧАНИЕ

Для того чтобы модуль был совместим с применяемыми в системе функциями прикладного интерфейса программирования Windows API, применяйте стандартное соглашение STDCALL.

Пример простой DLL

Закрепим наши знания на практике. Откроем ранее начатый проект MyDLL и опишем экспортируемую функцию SumFunc(), складывающую два числа и возвращающую результат операции. Чтобы не перегружать заголовочный модуль mydll избыточным кодом, добавьте к проекту DLL новый модуль. Для этого воспользуйтесь меню **File | New …** и на вкладке **New** диалога **New Items** выберите пиктограмму **Unit**.

Сохраните только что созданный модуль под именем MathFunc.pas. Обратите внимание, что в строке Uses библиотеки mydll появилась ссылка на MathFunc.pas.

Объявите экспортируемую функцию в разделе interface модуля MathFunc, не забыв указать стандартное соглашение о вызове STDCALL и директиву EXPORT. В разделе реализации опишите саму функцию (листинг 34.2).

Листинг 34.2. Описание экспортируемой функции в библиотеке

```
unit MathFunc;

interface

function SumFunc(X,Y:integer):integer; stdcall; EXPORT;
implementation
```

```
  function SumFunc(X,Y:integer):integer;
  begin
  Result:=X+Y;
  end;
end.
```

В заключение осталось опубликовать `SumFunc()` в разделе `EXPORTS` головного модуля библиотеки `mydll`. Для этого сразу после списка используемых модулей `uses` добавьте новую строку `EXPORTS SumFunc` (листинг 34.3).

Листинг 34.3. Доработка головного модуля библиотеки mydll

```
library mydll;
uses SysUtils, Classes;

EXPORTS SumFunc; //библиотека экспортирует функцию SumFunc

{$R *.RES}
begin

end.
```

Одновременно нажав клавиши <Ctrl>+<F9>, откомпилируйте библиотеку.

Взаимодействие динамической библиотеки с проектом

Различают два варианта загрузки динамически подключаемой библиотеки — *явная* (динамическая) и *неявная* (статическая) загрузка. Неявная загрузка предполагает жесткое подключение библиотеки к исполняемому модулю. В такой ситуации сразу после своего старта приложение приступает к загрузке всех необходимых ему дополнительных библиотек. В среде современной операционной системы статическая компоновка не всегда эффективна, т. к. приводит к неэкономному использованию одного из основных ресурсов — памяти. Реализация явной загрузки несколько сложнее, но вместе с тем обеспечивает бóльшую гибкость и надежность. Вам не придется подключать к программе никакие интерфейсные модули. Вместо этого приложение станет вызывать импортируемые функции при помощи функций Windows API.

Размещение файла DLL

Операционная система Windows самостоятельно обнаружит необходимую библиотеку, но при условии, что файл DLL размещен в одном из перечисленных далее каталогов:

- в каталоге, из которого запускается основной исполняемый EXE-файл;
- в текущем каталоге;
- в системном каталоге Windows;
- в каталоге Windows;
- в каталогах, включенных в переменную окружения `PATH`.

Явная загрузка DLL

При явной загрузке динамически подключаемой библиотеки программист должен сделать четыре шага:

1. Загрузить DLL.
2. Получить адрес нужной функции.
3. Вызвать функцию.
4. Выгрузить библиотеку из памяти.

Три из четырех действий реализуются при помощи функций Windows API. Отображение динамической библиотеки в адресное пространство процесса обычно осуществляет функция

```
function LoadLibrary(LibFileName: string):THandle;
```

Параметр `LibFileName` содержит имя подключаемой библиотеки. Примите во внимание, что в параметр не следует передавать путь к файлу DLL. В случае успеха функция возвращают дескриптор загрузившейся библиотеки.

Второй этап при явной загрузке динамически подключаемой библиотеки — получение адреса конкретной функции экспортируемой DLL с помощью функции

```
function GetProcAddress(hModule: THandle; ProcName: string): Pointer;
```

Параметр `hModule` соответствует дескриптору библиотеки, полученному из функции `LoadLibrary()`. Имя функции описывается в параметре `ProcName`. Возвращаемое значение — не что иное, как указатель на вызываемую функцию.

В заключение необходимо выгрузить DLL из памяти системы, вызвав функцию

```
function FreeLibrary(hModule: THandle ): LongBool;
```

Параметр `hModule` нам уже знаком, это дескриптор ранее загруженной библиотеки. Функция `FreeLibrary()` уменьшает на единицу значение счетчика количества пользователей загруженного модуля в адресном пространстве процесса. Если счетчик примет нулевое значение, то процесс выгружает библиотеку из памяти.

Как это выглядит на практике? Достаточно просто. Например, нам нужно воспользоваться услугами функции суммирования двух чисел из библиотеки mydll.dll, тогда следует выполнить код листинга 34.4.

Листинг 34.4. Явная загрузка DLL и вызов функции

```
Function DLL_SUM(X,Y:Integer):Integer;
var h:longint;
    SumFunc: Function(X,Y:Integer):Integer; STDCALL;
Begin
Result:=0;
h:=LoadLibrary('mydll.dll'); //загрузка DLL
  TRY
    if h>0 then //если загрузка DLL успешна
    begin
      @SumFunc:=GetProcAddress(h,'SumFunc'); //поиск функции в DLL
      IF Assigned(@SumFunc)=true    //если функция найдена
         then Result:=SumFunc(X,Y); //вызов функции
    end;
  FINALLY
  freelibrary(h); //выгружаем DLL
  END;
end;
```

ВНИМАНИЕ!

Перед первым запуском приложения скопируйте ранее созданную библиотеку mydll.dll в ту же папку, в которой находится исполняемый модуль проекта.

Неявная загрузка DLL

Организация неявной загрузки динамически подключаемой библиотеки несколько проще и включает два этапа:

1. Регистрация в секции `interface` прототипов импортируемых функций и процедур с указанием соглашения о вызовах.
2. Объявление в разделе констант секции `implementation` имени подключаемой библиотеки и импортируемых из нее функций и процедур.

В листинге 34.5 приведен фрагмент кода, демонстрирующий порядок неявной загрузки динамической библиотеки. Вызов экспортируемой функции осуществляется по щелчку по кнопке `Button1`.

Листинг 34.5. Пример неявной загрузки DLL

```
…
public
    { Public declarations }
end;
```

```
Function SumFunc(X,Y:integer):integer; stdcall;//опишем прототип

var frmMain: TfrmMain;

implementation

{$R *.dfm}
const mydll='mydll.dll'; //константа с именем библиотеки

Function SumFunc; external mydll name 'SumFunc'; //подключение DLL

procedure TfrmMain.Button1Click(Sender: TObject);
var X, Y:integer;
begin //вызов экспортируемой функции
 X:=StrToIntDef(EdValue1.Text,0);
 Y:=StrToIntDef(EdValue2.Text,0);
 Label1.Caption:=IntToStr(SumFunc(X, Y));
end;
```

Непосредственно вызов экспортируемой функции не вызывает никаких трудностей и ничем не отличается от вызова обычной функции, объявленной внутри модуля.

Пример DLL универсального генератора отчетов

Что может стать поводом для выноса части процедур и функций в отдельную библиотеку? Основная причина — интуитивное понимание программистом того, что задачи, решаемые модулем, могут повториться в перспективных проектах. Второй причиной может стать предположение, что процедуры и функции хотя и необходимы для построения полнофункционального проекта, но станут вызываться нерегулярно (как операция печати отчетов). В таком случае нецелесообразно перегружать основной модуль излишним кодом и нерационально занимать дополнительные ресурсы компьютера. Достаточно загрузить динамическую библиотеку на период выполнения требуемых операций, а затем освободить захваченные ресурсы. Кроме того, идея построения модульного проекта будет весьма кстати в случае, если вы предполагаете усовершенствование программного продукта в дальнейшем. При такой перспективе гораздо удобнее работать с отдельными модулями, чем с единым, громоздким продуктом. Это далеко не полный, но достаточный на первых порах перечень условий, при которых модульное построение проекта выглядит в более выигрышном свете.

На роль библиотеки как нельзя лучше подходит (*см. главу "Генератор отчетов своими руками" на прилагаемом DVD*) универсальный генератор отчетов. Согласитесь, что очень удобно иметь в своем распоряжении универсальную библиотеку, способную создавать простейшие табличные отчеты — это очень ускорит разработку программного продукта. Ядром программного продукта может выступать любое приложение баз данных. В случае, когда следует подготовить отчет, приложение запрашивает помощь у библиотеки (рис. 34.2).

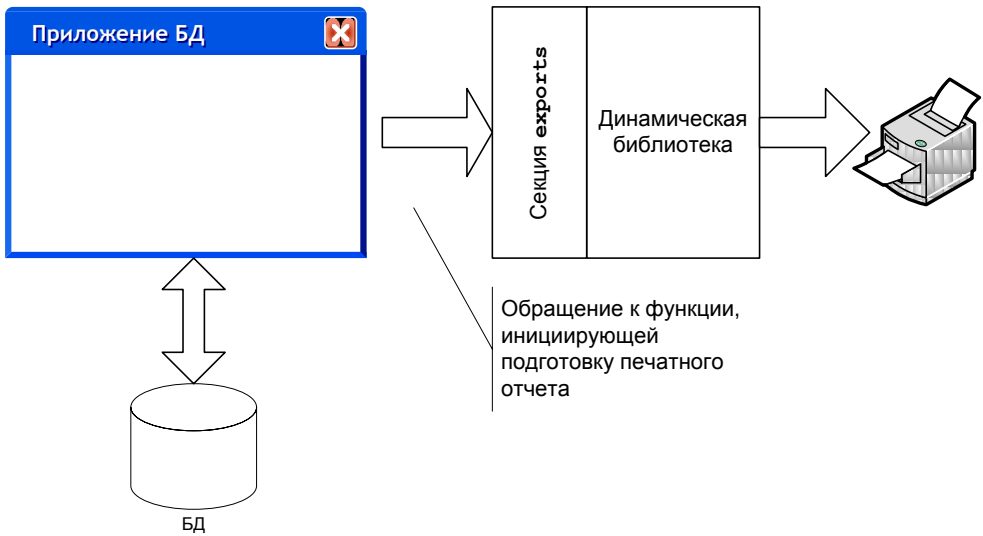


Рис. 34.2. Взаимодействие приложения и динамической библиотеки

Создайте шаблон новой динамической библиотеки и сохраните файл проекта в пустой каталог под именем universalreport.dproj. Если мы сейчас дадим команду на компиляцию проекта, Delphi создаст библиотеку universalreport.dll и поместит ее в тот же самый каталог, в котором расположены исходные файлы библиотеки. Это не вполне удобно с точки зрения отладки DLL. Во время отладки динамическую библиотеку лучше размещать в той же папке, где находится исполняемый файл, который будет пользоваться услугами библиотечного модуля. Чтобы указать среде программирования, что откомпилированный файл следует направлять в другой каталог, воспользуйтесь пунктом меню **Project | Options** и настройте параметр **Output directory** (рис. 34.3).

Следующим нашим действием будет подключение к проекту библиотеки уже готовых (если вы повторили пример из *главы "Генератор отчетов своими руками" на прилагаемом DVD*) файлов отчета — формы `frmReport.dfm` и соответствующего ей модуля report.pas. Просто скопируйте указанные файлы в папку с исходными файлами библиотеки и воспользуйтесь пунктом меню **Project | Add to project…**

Добавьте к проекту пустой модуль, выбрав в главном меню пункт **File | New | Unit Delphi**. При сохранении переименуйте его в access.pas. Этот модуль станет

отвечать за обеспечение доступа к форме отчета из внешнего приложения. Модуль доступа экспортирует одну-единственную процедуру `GetReport` (листинг 34.6).

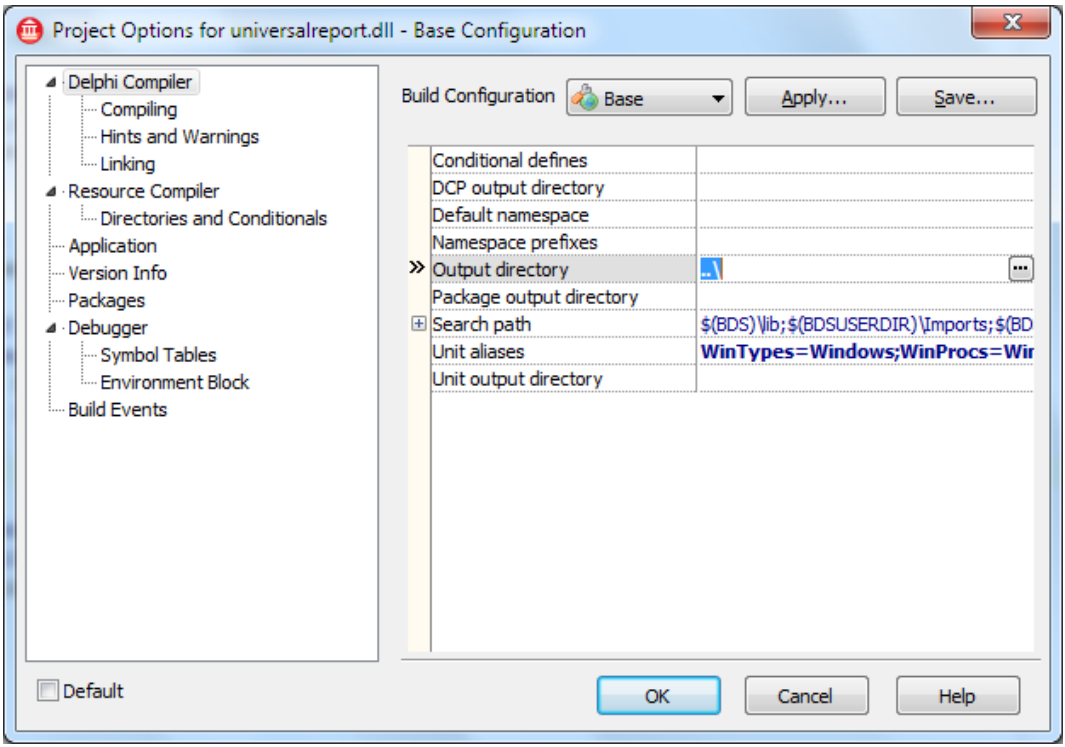


Рис. 34.3. Настройка компилятора Delphi при работе с DLL

Листинг 34.6. Экспорт процедуры

```
unit access;

interface
  uses forms,windows,DB,report;

  Procedure GetReport(AHandle:THandle; DS:TDataSet); STDCALL; Export;

implementation

Procedure GetReport(AHandle:THandle;DS:TDataSet);STDCALL; Export;
begin
Application.Handle:=AHandle;
```

```
 with TfrmReport.Create(Application) do
 begin
  ReportDataSet:=DS;
  ShowModal;
  Release;
 end;
Application.Handle:=0;
end;

end.
```

Особенность универсального генератора отчетов (*см. главу "Генератор отчетов своими руками" на прилагаемом DVD*) в том, что взаимодействие между формой отчета и главной формой проекта осуществлялось с помощью одной-единственной ссылки на набор данных — абстрактный класс `TDataSet`. Для этого в секции публичных объявлений формы отчета мы ввели специальную переменную

```
public
    ReportDataSet:TDataSet;
```

Мы сохранили аналогичный способ взаимодействия и сейчас. Процедура `GetReport()` снабжена параметром `DS:TDataSet`, благодаря которому обращающееся к библиотеке приложение передаст в DLL уже сформированный набор данных.

Теперь нам необходимо объявить секцию экспорта библиотеки. Для этого вызовем головной программный модуль (пункт меню **Project | View Source**) и добавим в него одну строку — `exports` (листинг 34.7).

Листинг 34.7. Объявление секции экспорта библиотеки

```
library universalreport;

uses SysUtils, Classes,
  report in 'report.pas' {frmReport},
  access in 'access.pas';

EXPORTS GetReport;//экспортируемая процедура

{$R *.res}

begin
end.
```

Одновременно нажав клавиши <Ctrl>+< F9>, откомпилируйте библиотеку. Теперь нам следует подключить библиотеку к приложению.

Замечание

Для запуска DLL на выполнение следует указать ведущее приложение. Воспользуйтесь пунктом меню **Run | Parameters…** и в появившемся окне заполните строку **Host application**.

Откройте приложение, из которого вы намерены вызывать библиотеку с универсальным генератором отчетов. Подготовьте процедуру обращения к динамической библиотеке (листинг 34.8).

Листинг 34.8. Обращение к библиотеке из приложения

```
Procedure TfrmMain.DLL_UniversalReport(DataSet:TDataSet);
var h:longint;
    GetReport : Procedure(AHandle:THandle; DS:TDataSet); STDCALL;
begin
h:=LoadLibrary('prntrep.dll');{загружаем библиотеку}
TRY
    if h>0 then {если библиотека успешно загружена}
    begin
      @GetReport:=GetProcAddress(h,'GetReport'); {доступ к процедуре}
      IF Assigned(@GetReport)=true then
         GetReport(Application.Handle,DataSet); {вызов процедуры}
    end;
FINALLY
freelibrary(h); {выгружаем библимотеку}
END;
end;
```

Теперь мы сможем сколько угодно раз пользоваться услугами отчета (листинг 34.9).

Листинг 34.9. Вызов отчета авторов

```
frmDM.ADOQuery1.SQL.Clear;
frmDM.ADOQuery1.SQL.Add('SELECT WRITER as "Автор" FROM WRITERS ORDER BY
WRITER');
DLL_UniversalReport(frmDM.ADOQuery1); //вызов библиотеки
```

Этот пример подготовит к печати список всех авторов книг. Обратите внимание на то, что в результирующем наборе колонка "WRITER" будет называться "Автор".

Следующий пример (листинг 34.10) подготовит запрос на диалекте базы данных Access и сгенерирует список контрактов с поставщиками печатной продукции.

Листинг 34.10. Вызов отчета контрактов поставщиков

```
WITH frmDM.ADOQuery1.SQL DO
BEGIN
 Clear;
 Add('SELECT SUPPLIER, CONTRACT, CONTRDATE FROM SUPPLIERS');
 Add('INNER JOIN');
 Add('CONTRACTS ON CONTRACTS.SUPPLIER_KEY=SUPPLIERS.SUPPLIER_KEY');
 Add('WHERE CONTRDATE BETWEEN #1/01/2010# AND #31/12/2010# ');
 Add('ORDER BY CONTRDATE,CONTRACT');
END;
DLL_UniversalReport(frmDM.ADOQuery1); //вызов библиотеки
```

Резюме

Динамически подключаемые библиотеки — это один из столпов, на котором держится современное программное обеспечение. Динамически подключаемая библиотека представляет собой отдельный программный модуль, содержащий код, данные и ресурсы и допускающий совместное их использование несколькими приложениями. Благодаря DLL существенно расширяется функциональность приложения, упрощается управление проектом, разделяются ресурсы между приложениями и достигается значительная экономия памяти.

Заключение

Современный рынок программного обеспечения уже давно разделен на сферы влияния крупных IT-корпораций, которые состоят в постоянной конкурентной борьбе друг с другом за каждый сегмент рынка. Операционные системы, средства проектирования и разработки программного обеспечения, текстовые и графические редакторы, системы автоматизированного проектирования, интернет-браузеры, пакеты системных утилит, мультимедиа, — вот далеко не полный перечень секторов рынка, за которые идет битва не на жизнь, а на смерть между именитыми компаниями.

Казалось, что в таких условиях программисту-одиночке (пусть даже очень талантливому) практически нереально разработать, реализовать, а затем и осуществлять поддержку своего программного продукта. К счастью, это не так. Как бы ни был широк охват различных сегментов рынка программных продуктов IT-корпорациями, они все равно не в состоянии удовлетворить пожелания конкретного пользователя. Множество коммерческих и государственных организаций и предприятий по-прежнему остро нуждаются в созданных специально для них программных продуктах и, в первую очередь — базах данных. В перечне ваших потенциальных партнеров магазины, склады, мастерские, кинотеатры, аптеки, библиотеки, больницы и поликлиники, учебные заведения, юридические агентства, туристические бюро и многие другие предприятия и организации. От вас, как от программиста, требуется только одно — создать профессиональный программный продукт.

Материал настоящей книги призван стать точкой опоры, которая позволит читателю начать карьеру разработчика баз данных и программиста на одном из самых совершенных языков программирования Delphi. Выбор Delphi в качестве среды для разработки приложений баз данных не случаен. Это по-настоящему быстрый, удобный, простой в освоении и эффективный инструмент разработки, позволяющий программисту решать весь спектр задач построения современного программного обеспечения. За эти и многие другие положительные качества Delphi отдают свое предпочтение тысячи профессиональных разработчиков программного обеспечения по всему миру.

Список литературы

1. Аткинсон Леон. MySQL. Библиотека профессионала. Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 624 с.
2. Браст Эндрю Дж., Форте Стивен. Разработка приложений на основе Microsoft SQL Server 2005. Мастер-класс. Пер. с англ. — М.: Издательство "Русская редакция", 2007. — 880 с.
3. Вигерс Карл. Разработка требований к программному обеспечению. Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2004. — 576 с.
4. Гарольд Э., Минс С. XML. Справочник. Пер. с англ. — СПб.: Символ-Плюс, 2002. — 576 с.
5. Грофф Дж., Вайнберг П., SQL: Полное руководство. Пер. с англ. — 2-е изд., перераб. и доп. — Киев: Издательская группа BHV, 2001. — 816 с.
6. Дейв Энсор, Йен Стивенсон. Oracle. Проектирование баз данных. Пер. с англ. — Киев: Издательская группа BHV, 1999. — 560 с.
7. Дейт, К. Дж. Введение в системы баз данных. Пер. с англ. — 8-е изд. — М.: Издательский дом "Вильямс", 2006. — 1328 с.
8. Елманова Н., Трепалин С., Тенцер А. Delphi и технология COM. — СПб.: Питер, 2004. — 704 с.
9. Кайт Том. Oracle для профессионалов. Пер. с англ. — СПб.: ООО "ДиаСофтЮП", 2003. — 672 с.
10. Клайн К. SQL. Справочник. Пер. с англ. — 2-е изд. — М.: КУДИЦ-ОБРАЗ, 2006. — 832 с.
11. Коннолли Томас, Бегг Каролин, Страчан Анна. Базы данных: проектирование, реализация и сопровождение. Теория и практика. Пер. с англ. — 2-е изд. — М.: Издательский дом "Вильямс", 2001. — 1120 с.
12. Молинаро Э. SQL. Сборник рецептов. Пер. с англ. — СПб.: Символ-Плюс, 2009. — 672 с.
13. Нильсен Пол. Microsoft SQL Server 2005. Библия пользователя. Пер. с англ. — М.: Издательский дом "Вильямс", 2008. — 1232 с.
14. Осипов Д. Delphi. Профессиональное программирование. — СПб.: Символ-Плюс, 2006. — 1056 с.
15. Осипов Д. Графика в проектах Delphi. — СПб.: Символ-Плюс, 2008. — 648 с.
16. Райордан Р. Основы реляционных баз данных. Пер. с англ. — М.: Издательско-торговый дом "Русская редакция", 2001. — 384 с.
17. Роб П., Коронел К. Системы баз данных: проектирование, реализация и управление. Пер с англ. — 5-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 1040 с.

18. Сворт Боб. Delphi 2010 DataSnap: новые возможности в управлении и доступе к данным. — www.drbob42.com, 2009.
19. Седжвик. Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ./Роберт Седжвик. — К.: Издательство «ДиаСофт», 2001. — 688 с.
20. Скляр А.Я. Введение в InterBase. — М.: Горячая линия — Телеком, 2002. — 517 с.
21. Тамре Л. Введение в тестирование программного обеспечения: Пер. с англ. — М.: Издательский дом "Вильямс", 2003. — 368 с.
22. Уорсли Дж., Дрейк Дж. PostgreSQL. Для профессионалов. Пер. с англ. — СПб.: Питер, 2003. — 496 с.
23. Хантер Дэвид, Рафтер Джефф, Фаусетт Джо, ван дер Влист Эрик и др. XML. Базовый курс. — 4-е изд.: Пер. с англ. — М.: ООО "И.Д. Вильямс", 2009. — 1344 с.
24. Ховард М., Лебланк Д. Защищенный код. Пер. с англ. — 2-е изд., испр. — М.: Издательско-торговый дом "Русская редакция", 2004. — 704 с.
25. Хомоненко А. Д., Цыганков В. М., Мальцев М. Г. Базы данных: Учебник для высших учебных заведений/ Под. ред. проф. А. Д. Хомоненко. — 4-е изд., перераб. и доп. — СПб.: КОРОНА принт, 2004 — 736 с.
26. Шаньгин В. Ф. Защита компьютерной информации. Эффективные методы и средства. — М.: ДМК Пресс, 2008. — 544 с.
27. Щеглов А. Ю. Защита компьютерной информации от несанкционированного доступа. — СПб.: Наука и техника, 2004. — 384 с.

Содержимое DVD

На DVD размещены дополнительные главы, а также материалы и пробные версии ПО компании Embarcadero, включая Delphi XE.

Дополнительные главы расположены в папке **Дополнительные главы**, которая включает следующие файлы: Отчеты в Microsoft Office.pdf, Генератор отчетов своими руками.pdf, Диаграммы и графики.pdf.

Примечание

Для прочтения дополнительных глав на компьютере должен быть установлен Acrobat Reader, который можно бесплатно скачать по адресу **http://www.adobe.com/products/reader/**.

Для доступа к материалам и дистрибутивам, предоставленным компанией Embarcadero, следует запустить файл **Embarcadero.exe**, откроется интерактивное меню, в котором можно выбрать интересующий вас раздел или программу.

Предметный указатель

A

ADO
  CreateUDLFile 416
  DataLinkDir 416
  GetDataLinkFiles 417
  GetProviderNames 417
  PromptDataLinkFile 417
  PromptDataSource 418
  строка соединения 392

C

CORBA 36

D

dbExpress 554
DBLogDlg
  LoginDialog 406
  LoginDialogEx 406
  RemoteLoginDialog 406
DLL 698
  GetProcAddress 705
  LoadLibrary 705
  неявная загрузка 706
  явная загрузка 705

E

EADOError
  Description 425
  Source 425
  SQLState 425
ER_модель
  атрибут 58
  подтип сущности 62
  рекурсивная связь 67
  связь 63
  сильная связь 66
  слабая связь 66
  тип сущности 58

I

IBFilterDialog
  DataSet 498
  DefaultMatchType 500
  Execute 498
  Fields 500
  Options 500

S

SQL
  Add 204
  Alter 204
  ALTER INDEX 207
  ALTER TABLE 204
  AVG 176
  BETWEEN 169
  CAST 161
  Check 201
  CLOSE 219
  Collate 200
  COMMIT 193
  COUNT 176
  CREATE DATABASE 198
  CREATE INDEX 206
  CREATE PROCEDURE 211
  CREATE ROLE 221
  CREATE SCHEMA 199
  CREATE TABLE 200
  CREATE TRIGGER 213
  CREATE TYPE 159

CREATE VIEW 209
DECLARE CURSOR 217
Default 200
DELETE 188
DROP DATABASE 200
DROP INDEX 207
DROP SCHEMA 200
EXISTS 171
FETCH 219
Foreign Key 201
FULL JOIN 172
Global 204
Grant 222
GRANT 222
GROUP BY 177
HAVING 178
IN 170
INNER JOIN 172
INSERT 185
IS NULL 170
JOIN..USING 172
LEFT JOIN 172
LIKE 169
MAX 176
MERGE 190
MIN 177
OPEN 218
ORDER BY 167
OUTER JOIN 172
Primary Key 201
RANK 179
References 201
REVOKE 225
RIGHT JOIN 172
ROLLBACK 193
SELECT 165
SET TRANSACTION 196
START TRANSACTION 191
SUM 176
Unique 201
UPDATE 187
USING 176
WHERE 167
битовая последовательность 155
встроенные функции 163
задачи языка 149
логические операторы 163
приближенные числовые типы данных 153
символьные строки 154
тип данных дата и время 156
типы данных 150
точные числовые типы данных 152

T

TADOCommand
CommandText 420
CommandTimeout 423
ConnectionString 420
Execute 421
ExecuteOptions 423
ParamCheck 420
Parameters 420
Prepared 423
TCommandType 420
TADOConnection
AfterConnect 396
AfterDisconnect 408
Attributes 415
BeforeConnect 396
BeforeDisconnect 408
BeginTrans 413
Cancel 395
CommandCount 411, 413
Commands 411
CommitTrans 413
Connected 394
ConnectionString 393
ConnectionTimeout 395
ConnectOptions 394
CursorLocation 415
DataSetCount 413
Errors 424
Execute 411
GetFieldNames 410
GetProcedureNames 410
GetTableNames 410
InTransaction 413
IsolationLevel 414
KeepConnection 394
LoginPrompt 394, 404
Mode 395
OnBeginTransComplete 413
OnCommitTransComplete 414
OnConnectComplete 396
OnDisconnect 408
OnExecuteComplete 412
OnInfoMessage 396
OnLogin 396
OnRollbackTransComplete 414

OnWillConnect 396
OnWillExecute 412
Open 394
OpenSchema 409
Provider 394
RollbackTrans 414
State 409
Version 408
TADODataSet
BlockReadSize 439
CacheSize 439
CancelBatch 443
CommandText 427
Connection 427
CursorLocation 428, 438
CursorType 441
DeleteRecords 443
EnableBCD 432
Filter 438
Filtered 438
FilterGroup 440
FilterOptions 438
First 430
GetIndexNames 436
IndexFieldCount 436
IndexFieldNames 436
IndexName 436
Last 430
LoadFromFile 444
LockType 437, 443
MarshalOptions 443
MaxRecords 439
MoveBy 430
Next 430
OnEndOfRecordset 435
OnFetchComplete 433
OnFetchProgress 433
OnFieldChangeComplete 434
OnMoveComplete 435
OnRecordChangeComplete 435
OnRecordsetChangeComplete 435
OnRecordsetCreate 428, 433
OnWillChangeField 433
OnWillChangeRecord 434
OnWillChangeRecordset 435
OnWillMove 435
Prior 430
RecNo 430
RecordCount 430
RecordSize 431
RecordStatus 431
SaveToFile 444
Seek 436
Sort 436
TADOQuery
DataSource 450
ExecSQL 449
ParamCheck 450
Parameters 449
Prepared 450
RowsAffected 450
SQL 449
TADOStoredProc
ExecProc 452
Open 452
Parameters 452
ProcedureName 451
TADOTable
GetIndexNames 445
IndexFieldNames 445
IndexName 445
MasterFields 446
MasterSource 446
ReadOnly 445
TableDirect 445
TableName 445
TAggregate
Active 378
AggregateName 377
Expression 377
GroupingLevel 377, 378
IndexName 378
TAggregateField
Active 379
Expression 378
TAutoIncField 326
TBCDField 327
Currency 328
Precision 328
TBinaryField 330
TBlobField
Assign 332
BlobType 332
Clear 333
LoadFromFile 332
LoadFromStream 332
Modified 332
SaveToFile 332
SaveToStream 332
SetFieldType 332
Transliterate 332

TBookmarkList
Clear 575
CurrentRowSelected 575
Delete 575
Find 575
IndexOf 575
Refresh 575
TBooleanField
DisplayValues 330
TBytesField 330
TCheckConstraint
CustomConstraint 376
ErrorMessage 376
ImportedConstraint 376
TClientDataSet
Add 377
AddIndex 372
AfterApplyUpdates 382
AfterExecute 387
AfterGetParams 387
AfterGetRecords 386
AfterRowRequest 386
Aggregates 377
AggregatesActive 377
ApplyRange 373
ApplyUpdates 382
BeforeApplyUpdates 382
BeforeExecute 387
BeforeGetParams 387
BeforeGetRecords 386
BeforeRowRequest 386
CancelRange 375
CancelUpdates 382
ChangeCount 382
CommandText 387
Constraints 376
CreateDataSet 348, 360
CreateTable 354
Data 367
DataSize 367
DeleteIndex 373
Delta 381
EditRangeEnd 375
EditRangeStart 375
EmptyDataSet 367
Execute 387
FetchBlobs 386
FetchOnDemand 386
FetchParams 387
FieldDefs 348, 360
FileName 362
GetNextPacket 386
IndexDefs 348
IndexFieldCount 372
IndexFieldNames 371
IndexFields 373
IndexName 371
KeyExclusive 376
LoadFromFile 361
LoadFromStream 361
LogChanges 381
OnDeleteError 385
OnEditError 385
OnPostError 385
OnReconcileError 383
PacketRecords 385
Params 387
ProviderName 380
RefreshRecord 386
RemoteServer 380
RevertRecord 383
SavePoint 383
SaveToFile 361
SaveToStream 361
SetRange 375
SetRangeEnd 373
SetRangeStart 373
StoreDefs 350
UndoLastChange 382
XMLData 367
TColumn
Alignment 579
Assign 580
AssignedValues 579
ButtonStyle 581
Color 579
DefaultAlignment 579
DefaultColor 580
DefaultFont 580
DefaultReadOnly 580
DefaultWidth 580
Depth 581
DisplayName 580
DropDownRows 581
Expandable 581
Expanded 581
Field 578
FieldName 578
Font 579
Grid 582
ParentColumn 582
PickList 580

PopupMenu 582
ReadOnly 578
RestoreDefaults 580
Showing 582
Title 579
Visible 579
Width 579
TCurrencyField 326
Precision 327
TCustomADODataSet
Recordset 426
RecordsetState 426
Supports 426
TCustomEdit
PasswordChar 586
ReadOnly 586
Text MaxLength 586
TCustomListBox
AddItem 600
Items 599
SetItemData 600
TDataSet
Active 278
ActiveBuffer 296
ActiveRecord 296
AfterOpen 279
Append 286
AppendRecord 286
AutoCalcFields 290
BlobFieldCount 288
Bof 281
BufferCount 296
Buffers 296
Cancel 287
CanModify 286
CheckBrowseMode 287
Close 279
ControlsDisabled 296
CurrentRecord 296
DataSource 296
DefaultFields 288
Delete 287
DisableControls 282
Edit 287
EnableControls 282
Eof 281
FieldByName 289
FieldByNumber 289
FieldCount 288
FieldDefList 289
FieldDefs 289
FieldList 289
Fields 289
FieldValues 289
Filter 291
Filtered 291
FilterOptions 292
FindField 289
First 280
Insert 286
InsertRecord 286
IsEmpty 281
IsLinkedTo 296
Last 280
Locate 292
Lookup 293
Modified 287
MoveBy 280
Next 280
ObjectView 290
OnCalcFields 290
OnFilterRecord 292
Open 279
Post 287
Prior 280
RecNo 281
RecordCount 281
Refresh 279
Resync 280
State 284
UpdateStatus 288
TDataSetProvider
AfterApplyUpdates 662
AfterExecute 663
AfterGetParams 663
AfterGetRecords 664
AfterUpdateRecord 663
BeforeApplyUpdates 662
BeforeExecute 663
BeforeGetParams 663
BeforeGetRecords 664
BeforeRowRequest 663
BeforeUpdateRecord 662
Constraints 659
DataSet 659
Exported 659
OnDataRequest 664
OnGetData 664
OnGetDataSetProperties 664
OnGetTableName 664
OnUpdateData 662
OnUpdateError 662

Options 660
ResolveToDataSet 659
UpdateMode 660
TDataSource
AutoEdit 572
DataSet 571
Edit 572
IsLinkedTo 571
OnDataChange 572
OnStateChange 572
OnUpdateData 572
State 571
TDateField
DisplayFormat 330
TDateTimeField
DisplayFormat 330
TDBCheckBox
Checked 591
State 591
ValueChecked 591
ValueUnchecked 591
TDBCtrlGrid
AllowInsert 593
ColCount 592
DoKey 593
EditMode 593
OnPaintPanel 593
Orientation 592
PanelCount 593
PanelHeight 593
PanelIndex 593
PanelWidth 593
RowCount 592
SelectedColor 593
ShowFocus 593
TDBEdit
DataField 573
DataSource 573
Field 573
ReadOnly 573
TDBGrid
Columns 576
Count 575
DataSource 573
DefaultDrawing 584
DrawingStyle 583
FieldCount 573
Fields 573
GradientEndColor 583
GradientStartColor 583
ObjectView 581
OnCellClick 582
OnColEnter 582
OnColExit 582
OnColumnMoved 583
OnDrawColumnCell 583
OnDrawDataCell 585
OnEditButtonClick 583
OnTitleClick 582
Options 574
SelectedField 573
SelectedIndex 573
SelectedRows 575
ShowPopupEditor 582
TDBGridColumns
Add 577
Columns 576
Grid 577
Items 576
LoadFromFile 577
LoadFromStream 577
RebuildColumns 578
RestoreDefaults 578
SaveToFile 577
SaveToStream 577
TDBImage
AutoDisplay 588
Picture 588
TDBListBox
Items 589
TDBLookupComboBox
CanModify 595
DataField 594
DataSource 594
DropDown 596
DropDownAlign 596
DropDownRows 596
DropDownWidth 596
KeyField 594
ListField 594
ListFieldIndex 595
ListFields 595
ListSource 594
NullValueKey 595
OnCloseUp 596
OnDropDown 596
ReadOnly 595
Text 596
TDBLookupListBox
CanModify 595
DataSource 594
KeyField 594

ListField 594
ListFieldIndex 595
ListFields 595
ListSource 594
NullValueKey 595
ReadOnly 595
RowCount 596
SelectedItem 596
TDBMemo
AutoDisplay 587
DataField 573
DataSource 573
Lines 587
LoadMemo 587
TDBNavigator
BeforeAction 597
ConfirmDelete 597
Flat 597
Hints 598
OnClick 597
VisibleButtons 597
TDBRadioGroup
ItemIndex 591
Items 590
Value 590
Values 590
TDBText
AutoSize 585
DataField 573
DataSource 573
WordWrap 585
TDSHTTPService
Active 640
AuthenticationManager 640
DSHostname 640
DSPort 640
Filters 639
HttpPort 640
RESTContext 640
Server 639
ServerSoftware 640
Trace 641
TDSHTTPServiceAuthenticationManager
HTTPAuthenticate 641
TDSServer
AutoStart 633
HideDSAdmin 633
LifeCycle 637
OnConnect 634
OnCreateInstance 638
OnDestroyInstance 638
OnDisconnect 634
OnError 636
OnGetClass 637
OnPrepare 635, 638
OnTrace 636
Start 633
Started 633
Stop 633
TDSServerClass
Server 637
TDSTCPServerTransport
BufferKBSize 640
Filters 639
MaxThreads 640
PoolSize 640
Port 639
Server 639
TField 289, 299
Alignment 314
AsAnsiString 307
AsBCD 307
AsBoolean 307
AsCurrency 307
AsDateTime 307
AsExtended 307
AsFloat 307
AsInteger 307
Assign 304
AssignValue 304
AsSQLTimeStamp 307
AsString 307
AsVariant 307
AsWideString 307
AutoGenerateValue 306
Calculated 301
CanModify 305
Clear 304
ConstraintErrorMessage 310
CurValue 305
CustomConstraint 310
DataSet 303
DataSize 308
DataType 301
DefaultExpression 309
DisplayLabel 314
DisplayName 314
DisplayText 313
DisplayWidth 314
EditMask 311
EditMaskPtr 311
FieldKind 301

FieldName 303, 314
FieldNo 303
FocusControl 314
GetData 306
HasConstraints 311
ImportedConstraint 311
Index 303
IsIndexField 312
IsNull 304
KeyFields 320
Lookup 301, 320
LookupCache 320
LookupDataSet 320
LookupKeyFields 320
LookupResultField 320
NewValue 305
OldValue 305
OnChange 315
OnGetText 315
OnSetText 315
OnValidate 314
Origin 303
ProviderFlags 306
ReadOnly 305
RefreshLookupList 320
Required 305
SetData 306
Size 308
Text 313
Validate 315
Value 304
Visible 313
TFieldDef
Attributes 351
ChildDefs 351
DataType 351
DisplayName 351
FieldClass 351
FieldNo 351
HasChildDefs 351
InternalCalcField 351
Name 351
ParentDef 352
Precision 351
Required 351
TFieldDefs
AddFieldDef 350
Clear 351
Delete 349
Find 349
Items 349
ParentDef 349
Update 349
TFloatField 326
Precision 327
TGraphicField 332
TGuidField
AsGuid 329
TIBBackupService
BackupFile 528
BlockingFactor 528
DatabaseName 528
Options 528
TIBConfigService
ActivateShadow 512
AlterJournal 516
BringDatabaseOnline 513
CreateJournal 516
CreateJournalArchive 516
Database 512
DatabaseName 512
DisableFlush 513
DropJournal 516
DropJournalArchive 516
FlushDatabase 513
GetJournalInformation 516
JournalInformation 514
ReclaimMemory 513
SetAsyncMode 513
SetDBSqlDialect 512
SetFlushInterval 513
SetGroupCommit 513
SetLingerInterval 513
SetPageBuffers 512
SetReadOnly 513
SetReclaimInterval 513
SetReserveSpace 513
SetSweepInterval 512
ShutdownDatabase 512
SweepDatabase 512
Transaction 512
TIBCustomService
Active 503
Attach 503
Detach 503
IsServiceRunning 504
LoginPrompt 503
OnAttach 503
OnLogin 503
Params 503
Protocol 502
ServerName 502

ServiceParamBySPB 504
ServiceStart 504
TraceFlags 504
TIBDatabase
AddEventNotifier 464
AddTransaction 464
AfterConnect 459
AfterDisconnect 461
AllowStreamedConnected 458
BeforeConnect 459
BeforeDisconnect 461
CheckActive 460
CheckDatabaseName 457
CheckInactive 460
CloseDataSets 461
Connected 457
CreateDatabase 456
DatabaseName 457
DBParamByDPB 457
DBSQLDialect 456
DefaultTransaction 464
DropDatabase 456
FindDefaultTransaction 464
FindTransaction 464
ForceClose 461
GetFieldNames 462
GetTableNames 462
IdleTimer 461
LoginPrompt 457
OnDialectDowngradeWarning 456
OnIdleTimer 461
OnLogin 460
Open 457
Params 457
RemoveEventNotifier 464
RemoveTransaction 464
RemoveTransactions 464
SQLDialect 456
TestConnected 459
TraceFlags 464
TransactionCount 464
Transactions 463
TIBDataBaseINI
CharacterSet 478
Database 477
DatabaseName 478
FileName 477
IniFileName 477
Password 478
ReadFromINI 477
SaveToINI 478
Section 478
SQLRole 478
UseAppPath 477
Username 478
WriteToDatabase 477
TIBDataSet
AfterDatabaseDisconnect 485
AfterTransactionEnd 485
ApplyUpdates 489
BatchInput 494
BatchOutput 494
BeforeDatabaseDisconnect 485
BeforeTransactionEnd 485
CachedUpdates 488
CancelUpdates 490
CanModify 488
Database 483
DatabaseFree 486
DeleteSQL 484
ForcedRefresh 488
GeneratorField 487
InsertSQL 484
IsSequenced 490
LiveMode 488
ModifySQL 484
OnUpdateError 486, 490
OnUpdateRecord 486, 489
Params 484
Prepare 484
QDelete 484
QInsert 484
QModify 484
QRefresh 484
QSelect 484
RecNo 490
RefreshSQL 484
RevertRecord 490
SelectSQL 484
Transaction 483
TransactionFree 485
Undelete 489
UniDirectional 485
UnPrepare 484
UpdateObject 492
UpdateRecordTypes 488
UpdatesPending 489
TIBEvents
Events 473
OnEventAlert 473
Registered 473
RegisterEvents 473

TIBExtract
ExtractObject 471
TIBInputDelimitedFile
ColDelimiter 495
FileName 495
RowDelimiter 495
TIBInputRawFile
FileName 495
TIBJournalInformation
CheckpointInterval 515
CheckpointLength 515
CreateJournalAttributes 515
CreateJournalLength 515
Directory 515
HasArchive 515
HasJournal 515
PageCache 515
PageLength 515
PageSize 515
TimestampName 515
TIBLogService
Eof 517
GetNextChunk 517
GetNextLine 517
TIBOutputDelimitedFile
ColDelimiter 495
FileName 495
RowDelimiter 495
TIBOutputRawFile
FileName 495
TIBQuery
BatchInput 494
BatchOutput 494
Database 483
ExecSQL 491
GenerateParamNames 492
Params 492
SQL 491
StmtHandle 492
TParams 343
Transaction 483
UpdateObject 491
TIBRestoreService
BackupFile 530
DatabaseName 530
Options 530
PageBuffers 530
PageSize 530
TIBSecurityService
ActiveUser 524
AddUser 525
DefaultRole 524, 527
DeleteUser 525
Description 524
DisplayUser 524
DisplayUsers 523
EnableEUA 527
FirstName 523
GroupID 524
GroupName 524
LastName 523
MiddleName 523
ModifyUser 525
Password 525
SuspendEUA 527
SystemUserName 524
UserDatabase 525
UserID 524
UserInfo 523
UserInfoCount 523
UserInfos 523
UserName 523
TIBServerProperties
AddAlias 508
AliasCount 508
AliasInfo 508
AliasInfos 507
ConfigParams 506
DatabaseInfo 506
DeleteAlias 508
Fetch 508
FetchAliasInfo 507
FetchConfigParams 506
FetchDatabaseInfo 505
FetchLicenseInfo 506
FetchLicenseMaskInfo 507
FetchVersionInfo 505
LicenseInfo 507
LicenseMaskInfo 507
Options 508
VersionInfo 505
TIBSQL
BatchInput 494
BatchOutput 494
Bof 482
Current 482
Database 480
Eof 482
ExecQuery 481
FieldByName 482
FieldIndex 482
Fields 482

GoToFirstRecordOnExecute 482
Next 482
OnSQLChanging 480
ParamByName 481
Params 481
Prepare 480
Prepared 481
SQL 480
SQLType 481
Transaction 480
TIBStatisticalService
DatabaseName 518
GetNextLine 520
Options 518
TableNames 518
TIBStoredProc
Database 483
ExecProc 492
Params 492
Prepared 492
StoredProcName 492
Transaction 483
TIBTable
Database 483
Exists 494
GotoCurrent 494
StoreDefs 350
TableTypes 494
Transaction 483
TIBTransaction
AddDatabase 466
AllowAutoStart 467
AutoStopAction 467
CheckAutoStop 469
CheckDatabasesInList 469
CheckInTransaction 469
CheckNotInTransaction 469
Commit 466
CommitRetaining 467
DatabaseCount 465
Databases 465
DefaultAction 468
DefaultDatabase 466
FindDatabase 466
Handle 470
IdleTimer 468
InTransaction 467
OnIdleTimer 468
Params 469
RemoveDatabase 466
RemoveDatabases 466
Rollback 466
RollbackRetaining 467
SQLObjectCount 466
SQLObjects 466
StartTransaction 466
TPB 469
TPBLength 470
TIBUpdateSQL
Apply 498
DataSet 497
DeleteSQL 497
ExecSQL 498
InsertSQL 497
ModifySQL 497
SetParams 498
SQL 497
TIBValidationService
DatabaseName 520
FetchLimboTransactionInfo 521
FixLimboTransactionErrors 522
GlobalAction 521
LimboTransactionInfoCount 522
LimboTransactionInfos 522
Options 520
TIndexDef
CaseInsFields 354
DescFields 354
Expression 354
FieldExpression 354
Fields 354
GroupingLevel 354
Options 354
Source 354
TIndexDefs
Add 352
AddIndexDef 352
Assign 352
Find 353
FindIndexForFields 353
GetIndexForFields 353
Items 352
Update 352
TIntegerField 326
MaxValue 326
MinValue 326
TLargeintField 326
TListItem
Caption 605
Data 606

Delete 606
Index 606
SubItems 605
TListItems
Add 606
Clear 606
Delete 606
Insert 606
TListView
Columns 605
Items 605
ViewStyle 604
TMemoField 332
TNavigateBtn 597
TNumericField
DisplayFormat 324
EditFormat 326
TObjectField
FieldCount 339
Fields 339
FieldValues 339
HasParent 340
ObjectType 340
ParentField 340
TParam
Clear 345
DataType 344
IsNull 345
Items 343
LoadFromFile 345
LoadFromStream 345
Name 344
NumericScale 344
ParamType 344
Precision 344
Size 344
Value 345
TParameters
ParamByName 450
ParamValues 449
TParams
AddParameter 344
Clear 344
Count 344
Delete 344
ParamByName 343
ParamValues 343
TQuery
TParams 343
TRvCustomConnection
DataSet 688
TRvProject
Close 689
Engine 689
Execute 689
Open 689
ProjectFile 689
TRvSystem
DefaultDest 689, 690
SystemFiler 690
SystemPreview 690
SystemPrinter 690
SystemSetups 690
TSimpleDataSet
DataSet 568
TSmallintField 326
TSQLConnection
ActiveStatements 549
AfterConnect 538
AfterDisconnect 538
AutoClone 549
BeforeConnect 538
BeforeDisconnect 538
CloneConnection 549
Close 537
CloseDataSets 547
Commit 548
Connected 537
ConnectionName 535
ConnectionState 538
DataSetCount 547
DataSets 547
DriverName 535
DsAuthPassword 551
DSAuthUser 551
Execute 549
ExecuteDirect 548
GetDriverFunc 535
GetFieldNames 550
GetIndexNames 550
GetPackageNames 550
GetProcedureNames 550
GetProcedureParams 550
GetTableNames 550
InTransaction 548
KeepConnection 538
LibraryName 536
LoadParamsFromIniFile 537
LoadParamsOnConnect 537
MaxStmtsPerConn 549

MultipleTransactionsSupported 547
OnLogin 538
Open 537
Params 536
Rollback 548
StartTransaction 548
TableScope 550
TransactionsSupported 547
VendorLib 536
TSQLDataSet
CommandText 556
CommandType 556
ExecSQL 561
GetMetadata 560
MaxBlobSize 563
ParamCheck 562
Params 562
ParseDeleteSql 563
ParseInsertSql 563
ParseSelectSql 563
ParseUpdateSql 563
SQLConnection 556, 561
TSQLMonitor
Active 552
AutoSave 552
CustomCategory 553
FileName 552
MaxTraceCount 551
Message 552
OnLogTrace 553
OnSQL 476
OnTrace 552
SQLConnection 551
TraceFlag 552
TraceFlags 476
TraceLevel 553
TraceList 551
TSQLQuery
ParamCheck 565
Params 565
SQL 565
TParams 343
TSqlServerMethod
Params 642
ServerMethodName 642
SQLConnection 642
TSQLServerMethod
ExecuteMethod 642
TSQLStoredProc
ExecProc 566
NextRecordSet 566
PackageName 566
ParamCheck 565
Params 565
PrepareStatement 566
SQLConnection 565
StoredProcName 565
TSQLTable
DeleteRecords 564
IndexFieldNames 564
IndexName 564
MasterFields 564
MasterSource 564
TableName 564
TSQLTimeStampField 331
TStringField
FixedChar 329
Transliterate 328
TStringGrid
Objects 607
TTable
StoreDefs 350
TTimeField
DisplayFormat 330
TTreeNode
AbsoluteIndex 612
Data 612
Index 612
Level 612
Text 612
TTreeView
AddChildObject 612
AddChildObjectFirst 613
AddObject 612
AddObjectFirst 612
InsertNode 613
InsertObject 613
TTreeNodes 612
TWordField 326

X

XML
<attribute> 256
<element> 246
DTD 236
XML Schemas 243
пространство имен 245
тип данных 247

А

Агрегатные поля 378
Архитектура ANSI/X3/SPARC 22
Атрибут 40

Б

Безопасность:
  авторизация 141
  аудит 144
  аутентификация 140
  доступность 135
  конфиденциальность 135
  резервное копирование 143
  целостность 135
  шифрование 141

Д, З

Домен 43
Защита БД 135

И

Иерархическая модель 18
Индексирование 92
  B-деревья 99
  битовые карты 103
  хеширование 93
Информационная система 119

К

Клиент-сервер 34
Ключ:
  вторичный 47
  первичный 47, 81

М

Многоуровневые БД 36
Модель:
  Entity-Relationship 58
  "сущность – связь" 58
Модуль:
  ADODB 391
  ADOInt 391
  DB 276, 573
  DBCtrls 573
  DBGrids 573
  DBLogDlg 406
  FMTBcd 328
  IBCustomDataSet 480
  IBDatabase 454
  IBDatabaseInfo 474
  IBEvents 473
  IBExtract 471
  IBFilterDialog 498
  IBHeader 457
  IBQuery 491
  IBServices 502
  IBStoredProc 492
  IBTable 494
  ShareMem 702
  SimpleDS 554
  SqlExpr 554

Н

Нормализация:
  1NF 77
  2NF 82
  3NF 85
  4NF 87
  5NF 88
  NF Бойса-Кодда 86

П

Планирование БД 123
Поле:
  агрегатное 290
  вычисляемое 290, 323
  динамическое 299
  набора данных 299
  статическое 299
Проектирование БД 127

Р

Реляционная алгебра 51
Реляционная модель 23, 38
Реляционная таблица 45

С

Связь 43
Сетевая модель 20
Системный каталог 13
Системы файлов 8
Соглашение о вызовах:
  Cdecl 703
  Pascal 703
  Register 703
Состояние набора данных 284
СУБД:
  компоненты 31
Сущность 40

Т

Тип данных 41
Транзакция 106
  метод блокировок 113
  метод временных меток 115
  метод двухфазной блокировки 116
  неактуальные чтения 109
  несогласованная обработка 110
  потерянные обновления 108
  уровни изоляции 117
  чтение строк-фантомов 111

У

Управление службой:
  CloseServiceHandle 671
  ControlService 672
  OpenSCManager 671
  OpenService 672
  QueryServiceStatus 673
  StartService 672

Ф

Файл-сервер 33

Ц

Целостность:
  данных 48
  доменов 48
  корпоративная 51
  ссылочная 50
  сущностей 49

Генератор отчетов своими руками

Существует несколько принципиально разных подходов к разработке отчетов. Первый и самый простой подход заключается во включении в проект специальных компонентов, упрощающих процесс проектирования отчетов. Уже несколько поколений Delphi пользуется услугами компонентов Rave Reports, разработанных компанией Nevrona Designs. Это очень удобные и простые в эксплуатации компоненты, в чем вы уже убедились, когда прочитали *главу 33*.

Второй способ заключается в использовании технологии автоматизации Windows. Практически на любом компьютере установлены приложения Microsoft Office, и почти каждый пользователь уже имеет навыки работы с текстовым процессором Word и редактором электронных таблиц Excel. Программисту остается лишь написать код, позволяющий передать данные из БД в соответствующие приложения. Об этом мы уже говорили в *главе "Отчеты в Microsoft Office"* (на прилагаемом DVD). С точки зрения программирования, второй способ более сложен и трудоемок, но в результате приносит существенно лучшие результаты. Во-первых, пользователь работает в привычной для него среде. Во-вторых, программные продукты Microsoft Office предоставляют огромный перечень сервисных возможностей по работе со своими документами. В-третьих, грамотно спроектированные отчеты сохранят свою работоспособность даже при выходе очередных версий офисного пакета.

Преемственность поколений — очень важный аспект разработки ПО. Старожилы Delphi помнят, что в первых версиях отчеты строились с помощью сторонней программы Report Smith. Затем в Borland (так тогда назывался владелец Delphi) от этой программы отказались, а попавшим в заложники программистам пришлось срочно (и бесплатно) переписать все свои старые приложения. Вслед за Report Smith в составе Delphi оказались компоненты Quick Report. Угадайте, где они сегодня? Правильно, получили отставку. Кто может дать гарантию, что через пару лет Embarcadero Technologies не поменяет линейку компонентов Rave Reports на компоненты другого производителя?

И наконец, третий способ подготовки отчетов заключается в создании отчета с опорой только на свои силы и на классические компоненты VCL. Такому отчету не грозят никакие катаклизмы и конкурентная борьба, возникающая между произво-

дителями программного обеспечения, ведь он "до последнего бита" принадлежит нам. В отчете, сделанном "с нуля", мы сможем, не боясь ограничений, реализовать все свои идеи. У подобных отчетов лишь один недостаток — чтобы их создать, придется потрудиться…

Для того чтобы научиться самому создавать отчеты, следует познакомиться с главным действующим лицом этой главы — печатающим устройством или, попросту говоря, принтером.

Технические характеристики принтера

Прежде чем потребовать от принтера, чтобы он нанес на лист бумаги первые строки нашего отчета, стоит разобраться, на какие "подвиги" наш принтер способен. Может быть, он не в состоянии печатать на листах формата А3, а нам нужен именно такой отчет. Или принтер не обладает достаточным разрешением? А может, принтер не поддерживает двустороннюю печать? Одним словом, у современного печатающего устройства очень много технических характеристик, которые следует учитывать при создании отчета.

В Windows API имеется универсальная функция, способная проинформировать нас о технических параметрах устройств вывода графической информации:

```
Function GetDeviceCaps(DC : HDC; Index : Integer) : Integer;
```

Наша новая знакомая способна рассказать о нескольких десятках ключевых характеристик устройства вывода (принтера, плоттера или экрана дисплея). Для этого в `GetDeviceCaps()` передается дескриптор исследуемого контекста графического устройства `DC`. Второй параметр — `Index` представляет собой один из почти трех десятков идентификаторов (некоторые из них вы найдете в табл. 1). В зависимости от значения идентификатора функция возвратит ту или иную характеристику устройства.

Таблица 1. *Возможные значения `Index` для исследования печатающего устройства*

Значение `Index`	**Возвращаемая характеристика**	
TECHNOLOGY	Технология драйвера. Может принимать одно из следующих значений:	
	DT_PLOTTER	Векторный плоттер
	DT_RASDISPLAY	Растровый дисплей
	DT_RASPRINTER	Растровый принтер
HORZSIZE	Физическая ширина области отображения в миллиметрах	
VERTSIZE	Физическая высота области отображения в миллиметрах	
HORZRES	Ширина области отображения в пикселах	
VERTRES	Высота области отображения в пикселах	

Таблица 1 *(окончание)*

Значение Index	Возвращаемая характеристика
LOGPIXELSX	Число пикселов в логическом дюйме по ширине области отображения
LOGPIXELSY	Число пикселов в логическом дюйме по высоте
BITSPIXEL	Число битов цвета для представления одного пиксела
ASPECTX	Относительная ширина пиксела устройства по горизонтали
ASPECTY	Относительная ширина пиксела устройства по вертикали
ASPECTXY	Относительная ширина пиксела устройства по диагонали
PHYSICALWIDTH	Ширина физической страницы в пикселах
PHYSICALHEIGHT	Высота физической страницы в пикселах
PHYSICALOFFSETX	Расстояние от левого края физической страницы до левой границы области печати в единицах устройства
PHYSICALOFFSETY	Расстояние от верхнего края физической страницы до верхней границы области печати в единицах устройства
SCALINGFACTORX	Коэффициент масштабирования для оси X принтера
SCALINGFACTORY	Коэффициент масштабирования для оси Y принтера

Описание принтера в Delphi, класс *TPrinter*

Среда Delphi предлагает один из наиболее простых и одновременно удобных способов работы с печатающим устройством — модуль `Printers`. В результате программист получает право обращаться к принтеру через функцию

```
function Printer: TPrinter;
```

Метод возвратит объект класса `TPrinter`, инкапсулирующий в своем составе практически весь интерфейс печати из Windows GDI.

Если вы создадите свою версию потомка класса `TPrinter`, то Delphi позволит заменить глобальный экземпляр принтера. Для этого следует обратиться к функции

```
function SetPrinter(NewPrinter: TPrinter): TPrinter;
```

входящей в состав модуля `Printers`.

Выбор принтера

В операционной системе можно установить неограниченное число принтеров: локальный принтер, сетевые печатающие устройства и различного рода эмуляторы

принтеров, например, весьма популярный Adobe Acrobat Distiller, "перегоняющий" любые документы в формат PDF или Microsoft Office Document Image Writer Driver, позволяющий формировать снимки документов в формате растрового изображения TIFF либо MDI (Microsoft Document Imaging). Весь перечень доступных устройств собирается в списке

```
property Printers: TStrings;
```

Индекс текущего принтера окажется в свойстве

```
property PrinterIndex: Integer;
```

Меняя порядковый номер, мы назначаем принтер, на который отправится задание. Для обращения к принтеру, установленному по умолчанию, присвойте свойству `PrinterIndex` значение –1. В листинге 1 приведен небольшой пример инициализации перечня доступных принтеров во время создания главной формы приложения.

Листинг 1. Получение списка зарегистрированных в системе принтеров

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
 ComboBox1.Items.Assign(Printer.Printers);
 ComboBox1.ItemIndex:=Printer.PrinterIndex;
end;
```

Управление страницей документа

Вспомним про еще один ингредиент, без которого печать невозможна, — лист бумаги. Средствами визуальной библиотеки компонентов можно легко получить данные о ключевых характеристиках бумаги. В первую очередь это высота и ширина листа:

```
property PageHeight: Integer;
property PageWidth: Integer;
```

Размеры передаются в логических единицах.

Свойство

```
property Orientation: TPrinterOrientation;
TPrinterOrientation = (poPortrait, poLandscape);
```

определяет ориентацию листа бумаги при печати. Предлагается всего два варианта ориентации — портретная или альбомная.

Информация о некоторых возможностях принтера по управлению заданием содержится в свойстве

```
property Capabilities: TPrinterCapabilities;
```

Описание `TPrinterCapabilities` приведено в табл. 2.

Таблица 2. *Возможные значения* `TPrinterCapability`

Значение	Описание
`pcCopies`	Пользователю разрешено определять число распечатываемых копий документа
`pcOrientation`	Принтер позволяет изменять ориентацию бумаги
`pcCollation`	При печати нескольких экземпляров одного документа указывает, что экземпляры разбираются по копиям

Формирование и отправка задания на печать

Вся идея работы с объектом принтера заключается в обращении к поверхности бумаги как к холсту, на котором можно рисовать. Именно для этого принтер инкапсулирует хорошо знакомый и горячо любимый программистами Delphi фирменный объект VCL — холст:

```
property Canvas: TCanvas;
```

Далее остается проявить умение и сноровку и отправить на печать данные. В простейшем случае достаточно написать менее 10 строк исходного кода (листинг 2).

Листинг 2. Отправка на печать текстовой строки

```
implementation
 uses Printers;

procedure TForm1.Button1Click(Sender: TObject);
begin
 with Printer do
 begin
    BeginDoc;     //старт документа
    Canvas.TextOut(10,10,'Тест печати'); //печать строки
    EndDoc;       //завершение документа
 end;
end;
```

Печать инициируется посредством обращения к процедуре

```
procedure BeginDoc;
```

Это команда *спулеру печати* (print spooler) Windows о начале нового задания[1]. Если все прошло успешно, то не забудьте попрощаться, известив спулер о завершении задания:

```
procedure EndDoc;
```

[1] Спулер — программа-планировщик, которая принимает отправленные на печать документы, сохраняет их и передает на доступный принтер.

Индикатором того, что принтер приступил к выполнению задания, станет переход в true свойства

```
property Printing: Boolean; //только для чтения
```

При необходимости разбить документ на несколько страниц, вспоминаем о существовании процедуры

```
procedure NewPage;
```

Обращение к этому методу вынуждает принтер завершить работу с текущим листом бумаги и "стащить" из лотка следующий. Номер распечатываемой в данный момент страницы хранится в свойстве

```
property PageNumber: Integer;
```

Для того чтобы сформированное нами задание на печать более профессионально смотрелось в окне менеджера печати, перед обращением к процедуре BeginDoc() определите заголовок заданию. Имя задания передается в свойство

```
property Title: string;
```

Отмена задания

В процессе печати документа спулер периодически проверяет состояние свойства

```
property Aborted: Boolean; //только для чтения
```

И если названное свойство возвратит значение true, то это станет командой на отмену текущего задания. Свойство доступно только для чтения, поэтому при необходимости отменить уже отправленное на принтер задание обратитесь к методу

```
procedure Abort;
```

Процедура самостоятельно позаботится о переводе Aborted в состояние true. В результате нераспечатанное задание сбрасывается, и принтер останавливается либо переходит к обслуживанию следующей задачи.

Печать многострочного текста

В девяти случаях из десяти принтер применяется для распечатки текста. При выводе многострочного неформатированного текста целесообразно воспользоваться услугами холста, а точнее — его метода TextOut(). Напомню, что первые два параметра метода определяют место вывода строки, это, соответственно, горизонтальный и вертикальный отступы от верхнего левого угла страницы. Если с горизонтальным отступом все понятно, при печати неформатированного, выровненного по левому краю текста он каждый раз принимает одно и то же значение, то с вертикальным отступом немного сложнее. Программист обязан предусмотреть порядок назначения вертикального отступа печати для каждой очередной строки. Поэтому в нашем примере перед отправкой задания мы с помощью метода TextHeight() выясняем высоту строки и сохраняем это значение в локальной пе-

ременной `LineHeight`. Позднее оно будет использоваться как множитель при пересчете места печати очередной строки из свойства `Lines` компонента `Memo1`:

```
Y:=Y+LineHeight*2;
```

Выяснив величину отступа, в качестве второго параметра передаем ее в функцию `TextOut()`, остальной код иллюстрирует листинг 3.

Листинг 3. Печать многострочного текста

```
var LineNum :Word; //номер строки
    LineHeight {высота строки}, Y, X : integer;
begin
Printer.Canvas.Font:=Memo1.Font;            //синхронизация шрифтов
LineHeight:=Printer.Canvas.TextHeight('A'); //выясняем высоту строки
Printer.Title:='Название задания';          //назначаем имя задания

X:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETX); {отступ от левого края}
Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY); {отступ от верхнего края}
Printer.BeginDoc; //старт задания
for LineNum:=0 To Memo1.Lines.Count-1 do {перебор всех строк Memo-поля}
begin
    If Y>Printer.PageHeight-20 then
    begin
    Printer.NewPage; {если страница закончилась, то начинаем очередную}
    Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY);
    end
    else Y:=Y+LineHeight*2;{вертикальный отступ для очередной строки}
  {печать строки}
   Printer.Canvas.TextOut(X,Y,Memo1.Lines.Strings[LineNum]);
end;
Printer.EndDoc; //завершаем задание
end;
```

При печати текстовых данных шрифтами разной гарнитуры следует проверять, имеются ли в распоряжении печатающего устройства требуемые шрифты. Их перечень содержится в свойстве

```
property Fonts: TStrings;
```

Свойство возвратит список строк с именами семейств шрифтов.

Пример универсального отчета

Теперь предлагаю опробовать наши силы на реальном задании. Разработаем программу, способную стать базовой для многих будущих отчетов. Для того чтобы

пример стал как можно более универсальным, поставим перед собой амбициозные задачи:

- Наш отчет должен воспринимать любой набор данных. Другими словами, обязательным входным параметром программы-отчета будет потомок класса `TDataSet`. Это очень удобно. Представьте себе, что пользователь работает с какой-то таблицей. Для формирования отчета достаточно будет передать в него ссылку на присоединенный к БД компонент-набор данных (например, запрос `T…Query`) и ничего больше.
- Отчет формируется в наиболее востребованном виде — табличном.
- Перед печатью пользователь должен получить примерный внешний вид отчета, т. е. нам следует создать окно предварительного просмотра.
- Пользователь должен обладать возможностью выбирать печатающее устройство и режим печати.

Для начала задач хватит. Приступим к реализации задуманного. Для этого вам понадобится любой действующий проект базы данных. Добавьте к проекту компонент-запрос, допустим `TADOQuery`, подключите его к базе данных и заполните свойство SQL подходящим для вас текстом запроса. Задача компонента-запроса — предоставить в наш отчет набор данных, который нам предстоит распечатать.

Добавьте к проекту новую форму, переименуйте ее в `frmReport` и сохраните в папке с остальными формами модуль report.pas, соответствующий форме. В строку `uses` формы-отчета добавьте ссылку на модуль печати:

```
uses printers;
```

В секции публичных объявлений формы отчета укажем переменную-набор данных

```
public
  ReportDataSet:TDataSet;
```

которая обеспечит связь между нашим отчетом и запросом.

Для вызова формы-отчета разместите на главной форме новую кнопку (пункт меню) и напишите строки, приведенные в листинге 4.

Листинг 4. Вызов формы отчета

```
frmReport:=TfrmReport.Create(frmMain);
frmReport.ReportDataSet:= ADOQuery1;
frmReport.Show;
```

Мы создаем новый экземпляр отчета, подключаем к нему заранее подготовленный запрос и выводим форму на экран. На этом подготовительные операции завершены.

Окно предварительного просмотра

Возможность просмотра потенциальных результатов печати на экране компьютера — бесспорный плюс любого программного продукта. Предоставив подобную

сервисную возможность, программист избавит пользователя от необходимости перевода дорогостоящей бумаги и картриджа принтера. Да что там бумага и чернила… Главное — сохраненные нервы и хорошее настроение пользователя.

Создание своего собственного окна предварительного просмотра если и не "высший пилотаж", то, по крайней мере, бесспорное свидетельство способности программиста самостоятельно "отрываться от земли". Ведь нужно умудриться обмануть компьютер, убедив его, что до смешного малая поверхность экранной формы на самом деле огромный лист бумаги.

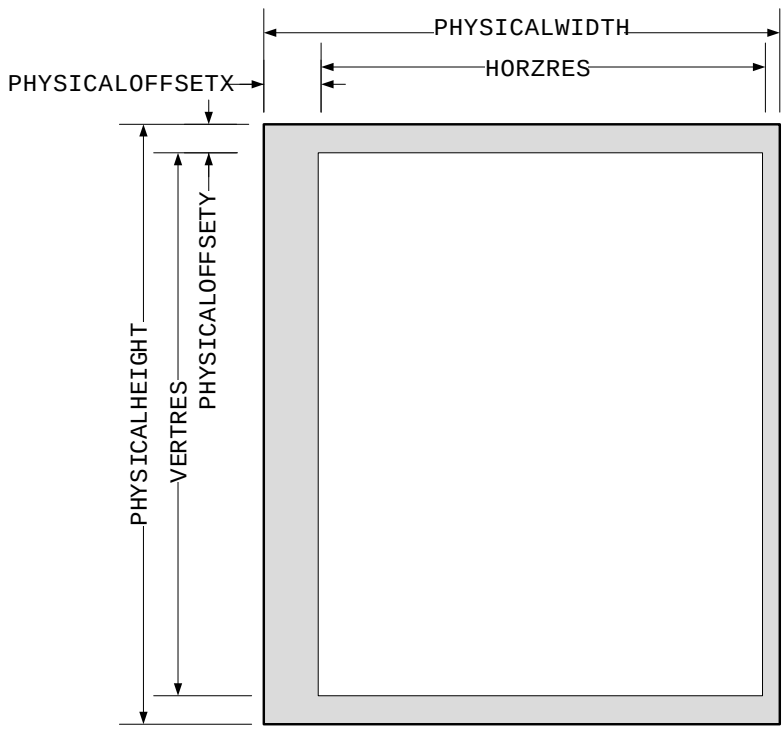


Рис. 1. Размеры листа бумаги, вычисляемые функцией `GetDeviceCaps()`

Практически ни один принтер не способен печатать на всей поверхности листа. Наличие "мертвых" зон по краям страницы объясняется особенностями конструкции печатающего устройства и в первую очередь зависит от его механической части. В технические подробности кинематики принтеров мы вдаваться не станем, но сделаем вывод, что перед формированием задания на печать целесообразно выяснить геометрические размеры страницы и величины недоступных для печати областей листа бумаги, чтобы часть распечатываемой информации не угодила в запретные зоны. Для решения этой задачи в качестве помощника привлечем функцию `GetDeviceCaps()`. Метод не только укажет ширину (`PHISICALWIDTH`) и высоту (`PHISICALHEIGHT`) страницы, но и уточнит физические возможности печатающего устройства по выводу данных. В первую очередь это отступы от левого

(PHISICALOFFSETX) и верхнего (PHISICALOFFSETY) краев листа. Размеры правой и нижней "мертвых" зон легко вычисляются, если мы знаем физическую ширину (HORZRES) и высоту (VERTRES) доступной для печати области (рис. 1).

Для хранения всей служебной информации в секции частных объявлений формы-отчета объявим некоторый перечень переменных (листинг 5).

Листинг 5. Переменные универсального отчета

```
private
 //==== ХАРАКТЕРИСТИКИ ПЕЧАТАЮЩЕГО УСТРОЙСТВА ============
    {разрешение по горизонтали и вертикали в лог. единицах}
    prLOGPIXELSX, prLOGPIXELSY,
    {размеры страницы в лог. единицах}
    prPHYSICALWIDTH, prPHYSICALHEIGHT,
    {поля отступа слева и сверху}
    prPHYSICALOFFSETX, prPHYSICALOFFSETY,
    {размеры рабочей области страницы}
    prHORZRES, prVERTRES:integer;
//==== МАССИВ СО ЗНАЧЕНИЯМИ ОТСТУПОВ ТЕКСТОВЫХ КОЛОНОК =====
    Columns:array of integer;
```

Такой внушительный перечень переменных нужен для обеспечения хранения данных о физических возможностях принтера по работе с листом бумаги. Для удобства группа переменных для взаимодействия с печатающим устройством снабжена префиксом pr.

Динамический массив Columns подготовлен для хранения отступов между левым краем рабочей области листа бумаги и колонками текста, выводимого в отчете. О нем мы поговорим немного позднее.

Для заполнения большинства из приведенных переменных создаем отдельную процедуру (листинг 6).

Листинг 6. Получение характеристик принтера

```
procedure TfrmReport.GetPrinterSettings;
begin
{узнаем разрешение принтера — число точек на дюйм}
  prLOGPIXELSX:=GetDeviceCaps(Printer.Handle, LOGPIXELSX);
  prLOGPIXELSY:=GetDeviceCaps(Printer.Handle, LOGPIXELSY);
{размеры выбранного листа бумаги в логических единицах принтера}
  prPHYSICALWIDTH:=GetDeviceCaps(Printer.Handle, PHYSICALWIDTH);
  prPHYSICALHEIGHT:=GetDeviceCaps(Printer.Handle, PHYSICALHEIGHT);
{отступы слева и сверху}
  prPHYSICALOFFSETX:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETX);
  prPHYSICALOFFSETY:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETY);
```

```
{ширина и высота рабочей области}
  prHORZRES:=GetDeviceCaps(Printer.Handle, HORZRES);
  prVERTRES:=GetDeviceCaps(Printer.Handle, VERTRES);
end;
```

Теперь займемся внешним видом формы-отчета (рис. 2). Нам потребуется помощь следующих компонентов:

- Панель группировки `GroupBox1`. Разместите панель в правой части формы, выбрав режим выравнивания `Align=alRight`.
- Комбинированный список `ComboBox1` для хранения перечня доступных принтеров. Расположите компонент на панели группировки `GroupBox1`.
- Диалог настройки принтера `PrinterSetupDialog1`.
- Кнопка `btnPrinterSetup` предназначена для вызова диалога настройки принтера.
- В нижней части панели группировки расположите еще две кнопки. Кнопка `btnPrint` возьмет ответственность за отправку задания на печать, `btnCancel` — закроет форму.
- В левой части формы расположим область для графического вывода `PaintBox1:TPainBox`. На его поверхности мы нарисуем окно предварительного просмотра. Установите свойство компонента `Align` в состояние `alClient`.

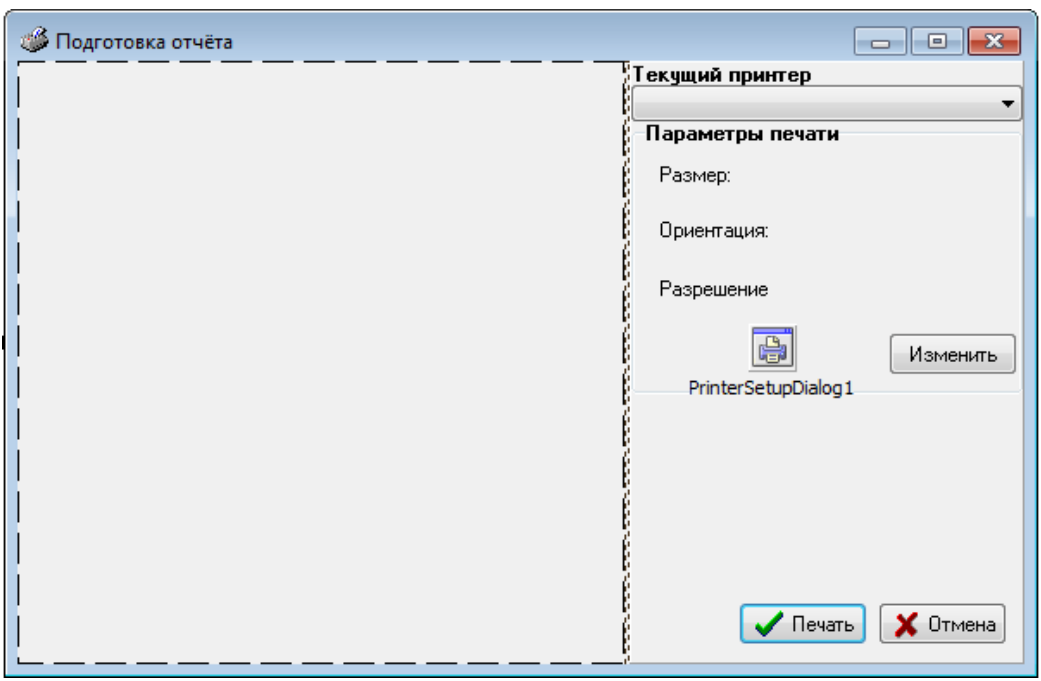


Рис. 2. Внешний вид формы-отчета

При выводе формы на экран генерируется событие OnShow(). Это самый удачный момент для сбора сведений об установленных в системе печатающих устройствах и их параметрах (листинг 7).

Листинг 7. Вывод формы на экран

```
procedure TfrmReport.FormShow(Sender: TObject);
begin
 ComboBox1.Items:=Printer.Printers;          {перечень принтеров}
 ComboBox1.ItemIndex:=Printer.PrinterIndex; {текущий принтер}
 GetPrinterSettings;           {узнаем характеристики текущего принтера}
 if ReportDataSet.Active=False
    then ReportDataSet.Open;      {активируем набор данных}
    SetLength(Columns,ReportDataSet.FieldCount);{распределяем память для мас-
сива отступов колонок}
end;
```

Перечень всех доступных принтеров передается в свойство Items комбинированного списка, затем мы "вычисляем" текущий принтер и получаем все необходимые для нашей дальнейшей работы характеристики печатающего устройства. Массив Columns предназначен для хранения отступов текстовых колонок от левого края рабочей области. Так как нам заранее не известно, сколько именно колонок содержится в отчете, то память под массив выделяется динамически. Мы узнаем число колонок, обратившись к свойству FieldCount набора данных ReportDataSet.

Если пользователь изменит текущий принтер, следует вновь собрать все его характеристики, поэтому в момент закрытия комбинированного списка вновь вызовем процедуру GetPrinterSettings() так, как в листинге 8.

Листинг 8. Изменение текущего принтера

```
procedure TfrmReport.ComboBox1CloseUp(Sender: TObject);
begin
 Printer.PrinterIndex:=ComboBox1.ItemIndex;//выбор принтера
 GetPrinterSettings;//получение сведений о принтере
 PaintBox1.Repaint; //перерисовка окна предварительного просмотра
end;
```

Кроме того, инициируем перерисовку области вывода PaintBox1. Эту операцию мы вызываем, немного забегая вперед, ведь пока в обработчике события OnPaint() компонента PaintBox1 не написано ни одной строки. Но это один из самых сложных этапов программирования, и нам необходимо к нему как следует подготовиться. Впрочем, мы уже близко подобрались к цели, нам осталось описать событие щелчка по кнопке btnPrinterSetup (листинг 9).

Листинг 9. Вызов диалога настройки принтера

```
procedure TfrmReport.btnPrinterSetupClick(Sender: TObject);
begin
  PrinterSetupDialog1.Execute(); //старт диалога
  ComboBox1.ItemIndex:=Printer.PrinterIndex;
  ComboBox1CloseUp(Sender); //вызываем событие OnCloseUp списка
end;
```

Щелчок по кнопке пробуждает к жизни диалог изменения настроек принтера `PrinterSetupDialog1`. Обращение к методу `Execute()` компонента выводит на экран стандартное окно **Настройка печати**. Благодаря диалоговому окну мы получаем возможность не только изменить текущий принтер, но и выбрать размеры листа бумаги, его ориентацию и качество печати (в первую очередь определяемое разрешением принтера).

Теперь наступает самый интересный и, вместе с тем, наиболее сложный этап работы. Нам предстоит написать исходный код для окна предварительного просмотра подлежащего печати текста. На роль этого окна мы уже выдвинули компонент `PaintBox1`. Такой чести компонент удостоился благодаря наличию у него своего собственного холста для рисования — свойства `Canvas`. На нашу долю остается немного "поколдовать" с кодом прорисовки в событии `OnPaint()` (листинг 10).

Листинг 10. Вывод окна предварительного просмотра

```
procedure TfrmReport.PaintBox1Paint(Sender: TObject);
var PaperRect,WorkRect:TRect;
    x,y:integer;
    s:string;
begin
with PaintBox1.Canvas do
begin
  Brush.Color:=clCream; //цвет кисти
  FillRect(PaintBox1.ClientRect);  //заливка области вывода кистью
  SetMapMode(Handle, MM_ISOTROPIC);//изотропный режим отображения
  {определяем протяженность логического окна, равную сторонам листа плюс 100
единиц}
  SetWindowExtEx(Handle,prPHYSICALWIDTH+100,prPHYSICALHEIGHT+100,nil);
  SetViewportExtEx(Handle,PaintBox1.ClientWidth,
                          PaintBox1.ClientHeight, nil);
  {перенос начала координат в верхний левый угол листа бумаги}
    SetWindowOrgEx(Handle, -50, -50, nil);
```

```
{прямоугольная область листа бумаги}
  PaperRect :=RECT(0, 0, prPHYSICALWIDTH,   prPHYSICALHEIGHT);
{прямоугольная область листа бумаги доступная для печати}
  WorkRect.Top   :=PaperRect.Top +prPHYSICALOFFSETY;
  WorkRect.Left  :=PaperRect.Left+prPHYSICALOFFSETX;
  WorkRect.Bottom:=WorkRect.Top  +prVERTRES;
  WorkRect.Right :=WorkRect.Left +prHORZRES;

  Brush.Color:=clWhite; //выбираем белую кисть
  FillRect(PaperRect); //закрашиваем белой кистью лист бумаги

  Font.Name:='Arial'; {шрифт для печати названий колонок}
  Font.Size:=14;
  Font.PixelsPerInch:=prLOGPIXELSX;

  Columns[0]:=WorkRect.Left+1; {отступ для печати текста слева}
  y:=WorkRect.Top;             {отступ сверху}

  {заголовки колонок}
  for x:=0 to ReportDataSet.FieldCount-1 do
  begin
    s:=ReportDataSet.Fields.Fields[x].DisplayLabel;
    TextOut(Columns[x],Y,s); //выводим название колонки
    {сдвиг вправо зависит от названия заголовка, точнее от его
    ширины в логических единицах}
    Columns[x+1]:=Columns[x]+TextWidth(S);
  end;

  {вывод основных данных}
  Font.Name:='Times New Roman';
  Font.Size:=12;
  Y:=Y+TextHeight(S)+5; {смещение вниз}

  ReportDataSet.First; {переход к первой строке данных}
  while ReportDataSet.Eof=false do
  begin
     for x:=0 to ReportDataSet.FieldCount-1 do
     begin
       s:=ReportDataSet.Fields.Fields[x].AsString;
       TextOut(Columns[x],Y,s);
     end;
```

```
      Y:=Y+TextHeight(S)+5;

      if Y+TextHeight(S)>=WorkRect.Bottom
            then break; //количество строк превысило размер страницы

      ReportDataSet.Next;
    end;
end;
end;
```

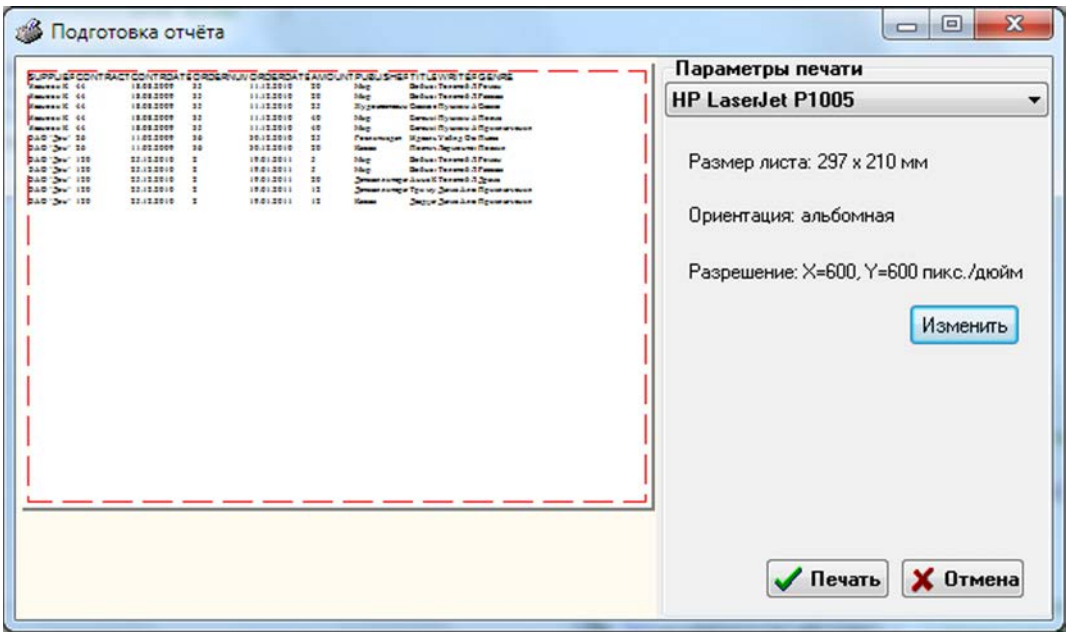


Рис. 3. Внешний вид окна предварительного просмотра

При решении подобных задач мы не сможем обойтись без поддержки функций Windows API, отвечающих за настройку режима отображения [15]. Сердце процедуры — код, переводящий холст в пользовательский (изотропный) режим отображения. Для этого вызывается функция `SetMapMode()` с ключом `MM_ISOTROPIC`. С данного момента мы приобретаем эксклюзивное право назначать свою собственную единицу измерения. Несколько позднее, благодаря изотропному режиму отображения и функциям GDI `SetWindowExtEx()` и `SetViewportExtEx()` программно изменяем значения протяженности холста. Мы без всякого зазрения совести обманули окно предварительного просмотра, с этого момента приложение начинает полагать, что крошечная поверхность компонента `PaintBox` вмещает в себя целый лист бумаги огромного размера. Для еще большего удобства с помощью `SetWindowOrgEx()` мы переносим точку отсчета координат в левый верхний угол листа бумаги. Все остальное — дело техники. Определяем координаты прямо-

угольной области, имитирующей лист бумаги (структура PaperRect), и рабочей области (структура WorkRect).

Для того чтобы не загромождать пример избыточным кодом, величина отступа определяется простейшим способом — с помощью функции TextWidth() выясняется, сколько логических единиц требуется для вывода на экран названия колонки. Более точный результат даст анализ свойства Size всех полей TField.

Разобравшись с отступами в цикле while..do, перебираем строки набора до тех пор, пока не достигнем конца набора или страницы. Величина вертикального сдвига каждой очередной строки рассчитывается с помощью функции TextHeight().

На рис. 3 показан внешний вид разработанного нами окна предварительного просмотра. Дело за малым — завершить проект, для этого осталось написать процедуру печати отчета на реальном принтере.

Вывод отчета на печать

Благодаря тому, что работа со свойством Canvas компонента PaintBox1 почти ничем не отличается от применения аналогичного свойства принтера TPrinter, нам остается просто перенести код из процедуры предварительного просмотра в процедуру печати (листинг 11).

Листинг 11. Код печати

```
procedure TfrmReport.btnPrintClick(Sender: TObject);
var x,y:integer;
    s:string;
begin
  Printer.Title:='Универсальный отчет';
  Printer.BeginDoc; {начало печати}
  Printer.Canvas.Font.Name:='Arial';
  Printer.Canvas.Font.Size:=14;

  y:=prPHYSICALOFFSETY; {вывод заголовков колонок}
  if ReportDataSet.Active=False Then  ReportDataSet.Open;
    for x:=0 to ReportDataSet.FieldCount-1 do
    begin
      s:=ReportDataSet.Fields.Fields[x].DisplayLabel;
      s:=AnsiUpperCase(s); //к верхнему регистру
      Printer.Canvas.TextOut(frmReport.Columns[x],y,s);
    end;
  y:=y+Printer.Canvas.TextHeight(S)+5;
  Printer.Canvas.Font.Name:='Times New Roman';
  Printer.Canvas.Font.Size:=12;
```

```
  ReportDataSet.First;
   while ReportDataSet.Eof=false do {цикл вывода основных данных}
   begin
       for x:=0 to ReportDataSet.FieldCount-1 do
       begin
         s:=ReportDataSet.Fields.Fields[x].AsString;
         Printer.Canvas.TextOut(Columns[x],y,s);
       end;

      y:=y+Printer.Canvas.TextHeight(S)+5;{смещение вниз}

      if Y+Printer.Canvas.TextHeight(S)>=prPHYSICALOFFSETY+prVERTRES then
         begin {количество строк превысило размер страницы}
          Printer.NewPage;      {старт очередной страницы}
          y:=prPHYSICALOFFSETY; {начинаем вывод текста сверху страницы}
         end;
      ReportDataSet.Next; //очередная строка
   end;

  Printer.EndDoc; //завершаем печать
  ReportDataSet.Close; //закрываем набор данных
end;
```

По сравнению с кодом предварительного просмотра процедура печати несколько упростилась. Причина в том, что здесь нам не нужно идти на "обман" области вывода `PaintBox1`, внушая компоненту, что он обладает тем же разрешением, что и принтер.

Пример завершен. Как видите, обладая определенными знаниями в области графической подсистемы Windows, можно создавать свои собственные отчеты. Код листинг а 35.11 — всего лишь универсальный шаблон, который достоин усовершенствования. Можно

- нарисовать сетку таблицы;
- улучшить качество масштабирования колонок, вплоть до предоставления пользователю возможности настраивать отступы мышью;
- ввести дополнительные служебные поля (номер страницы, дата печати);
- научить программу получать из полей таблиц BLOB-данные, например, изображения.

Самое главное, что для этого нам достаточно лишь помощи Delphi, и нет нужды задействовать программное обеспечение сторонних производителей.

Резюме

Между нами говоря, многие пользователи полагают, что компьютер — это лишь дорогая печатающая машинка. Поэтому взаимодействие с принтером в проектах Delphi — весьма злободневная тема, а организация печати — очень непростая задача. Основная сложность заключается в том, что программист не сможет обойтись без знания процедур и функций графической подсистемы Windows. Впрочем, в простейших случаях неоценимый вклад в проект может внести класс `TPrinter`, инкапсулирующий возможности принтера.

Диаграммы и графики

В составе визуальной библиотеки среды программирования Delphi почти нет компонентов, рожденных вне стен Embarcadero Technologies. Профессиональный уровень программистов этой компании вполне позволяет обходиться своими силами. Наглядное тому подтверждение — более сотни всевозможных элементов управления на палитре компонентов Delphi.

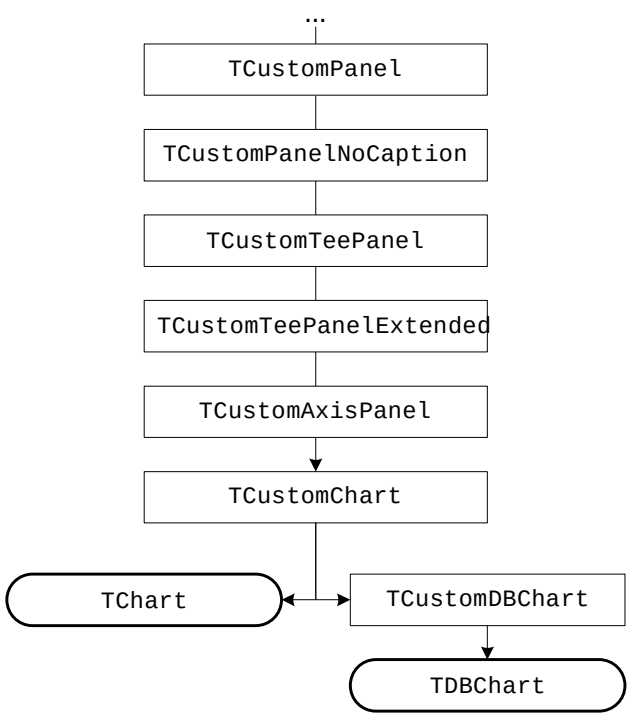


Рис. 1. Иерархия наследования диаграмм

Исключений из этого правила немного. Одно из них — компоненты-диаграммы, которые появились в Delphi в конце 1990-х годов. На такой беспрецедентный

шаг Borland (именно так раньше называлась компания-владелец Delphi) вдохновили блестящие возможности элементов управления TChart и TDBChart (страница **TeeChart Std**) по построению сложнейших диаграмм. Первоначальная идея разработать компоненты, способные строить диаграммы и графики, принадлежит сотруднику компании Steema Software программисту Дэвиду Бернеда (David Berneda).

При работе с диаграммами к проекту автоматически подключаются несколько модулей: TeEngine, ExtCtrls, TeeProcs и Chart (компонент TChart), TDBChart (компонент TDBChart). К сожалению, исходный код модулей для нас недоступен, но это законное право его владельца и нам не на что сердиться. Из немногочисленной информации о диаграммах доподлинно известно, что опорным классом VCL, на фундаменте которого построены диаграммы, служит обычная панель TCustomPanel. Но между панелью и диаграммами программисты Steema Software добавили полдесятка своих классов (рис. 1).

Компонент *TChart*

С точки зрения объектно-ориентированного программирования диаграмма представляет собой весьма сложную конструкцию. В своем составе TChart (TDBChart) инкапсулирует впечатляющий перечень взаимодействующих между собой, даже можно сказать переплетенных друг с другом объектов. Управление большинством из них организовано как из инспектора объектов, так и из специализированного редактора, вызываемого из контекстного меню компонента. К ключевым интегрированным объектам диаграммы относятся экземпляры классов (рис. 2):

- TChartSeriesList — контейнер для хранения отдельных графиков (в самом общем случае отдельный график реализуется на базе класса TChartSeries);
- TChartAxis, TChartDepthAxis — координатные оси диаграммы;
- TChartLegend — легенда (подписи графиков) диаграммы;
- TChartTitle — заголовок диаграммы.

Над оформлением диаграммы кропотливо трудится ряд специализированных графических классов:

- TCanvas3D — трехмерный холст (канва) диаграммы;
- TChartGradient — класс, описывающий способ заливки диаграммы;
- TChartPen — класс, определяющий характеристики пера;
- TChartBrush — вид кисти;
- TChartWall — класс, описывающий способ прорисовки стен;
- TView3DOptions — опции объемной графики.

Большинство из перечисленных графических классов представляют собой логическое развитие хорошо нам знакомых TCanvas, TPen, TBrush.

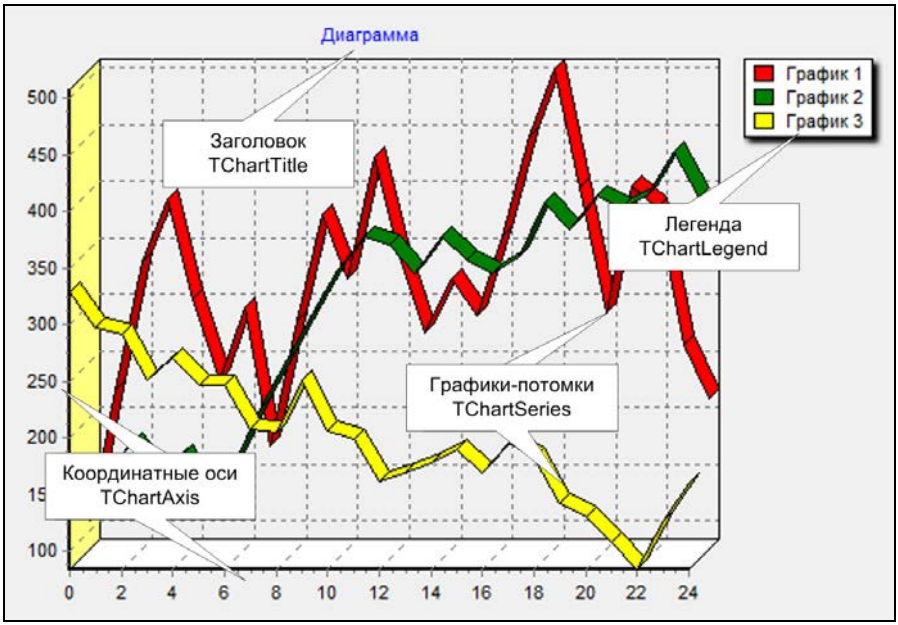


Рис. 2. Основные объекты, входящие в состав диаграммы `TChart`

Хранение графиков в диаграмме

Смысл существования компонента `TChart` состоит в умении хранить и выводить на экран разнообразные графики (потомки класса `TChartSeries`). Для "складирования" графиков в `TChart` предусмотрен специализированный контейнер. Если же контейнер пуст, то диаграмма представляет собой унылое зрелище и безответственно занимает драгоценное пространство на рабочей форме.

Без связи с диаграммой график оказывается бесполезным инструментом. Хотя он и вполне самостоятельный объект, но из-за того, что `TChartSeries` не умеет отображать свои данные, в отрыве от диаграммы практического интереса он не представляет. Обреченные жить в симбиозе, график и диаграмма взаимодействуют благодаря свойству

```
property Series[Index : Longint] : TChartSeries;
```

Это и есть "врата" в склад графиков компонента `TChart`. Для работы с отдельным экземпляром графика требуется указать его индекс. Обязанности главного счетовода склада возложены на метод

```
function SeriesCount : Longint ; //только для чтения
```

Только он знает о количестве графиков в контейнере. Обращаясь к свойству `Series` и владея числом графиков, диаграмма способна управлять каждым из графиков, например, окрасить их в небесные тона:

```
for i:=0 to Chart1.SeriesCount -1 do
    with  Chart1.Series[i] do SeriesColor := clAqua;
```

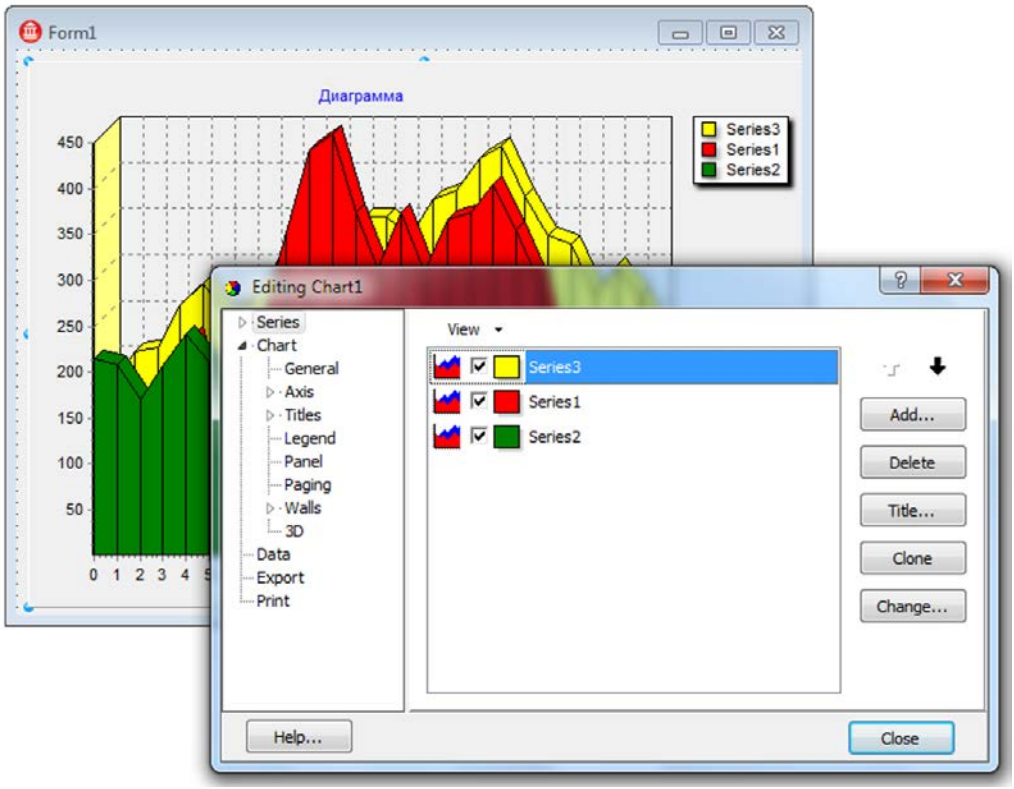


Рис. 3. Редактор компонента `Tchart`

Во время разработки приложения для вставки, изменения или удаления графика из диаграммы служит специальный редактор (рис. 3). Для его вызова в режиме проектирования достаточно дважды щелкнуть по компоненту `TChart`.

Содержимое контейнера допускается пополнять или освобождать во время выполнения программы. Для этого предусмотрен ряд методов. С операцией добавления нового графика в коллекцию лучше всех справится процедура

```
procedure AddSeries(ASeries : TChartSeries);
```

В качестве параметра в процедуру передается экземпляр графика, например, созданный конструктором `Create()`, как в листинге 1.

Листинг 1. Динамическое создание графика

```
var ASeries : TChartSeries;
begin
 ASeries:=TChartSeries.Create(Chart1);
 Chart1.AddSeries(ASeries);
 //…
end;
```

По сути, метод AddSeries() просто говорит графику, что теперь он переходит в единоличное владение к диаграмме, вызвавшей этот метод.

Обратную задачу по "экспроприации" у контейнера отдельного графика обеспечивает процедура

```
procedure RemoveSeries(ASeries : TChartSeries);
```

Подчеркну, что здесь не идет и речи об уничтожении экземпляра графика. Изъятый экземпляр просто покидает родительскую диаграмму (его свойство ParentChart переходит в nil). Позднее этот график без проблем можно вернуть на свое место или подключить к другой диаграмме.

Для полной очистки контейнера вызывается процедура

```
procedure RemoveAllSeries;
```

Базовый класс графиков — *TChartSeries*

Процесс создания нового графика всегда начинается с ответа на вопрос: "Какой тип графика программист рассчитывает увидеть на диаграмме?" Если поиск ответа начался еще во время проектирования приложения, то у нас есть превосходная подсказка. На основной странице встроенного редактора компонента TChart нажимаем кнопку **Add** и в новом окне подбираем пиктограмму графика с устраивающей нас внешностью (рис. 4), все остальное — проблемы диаграммы... Несколько сложнее обстоят дела, если формирование экземпляра графика возложено на код приложения. В таких обстоятельствах программисту предварительно следует разобраться со всем спектром графиков (графики описаны в модуле TeEngine): TLineSeries, TAreaSeries, TPointSeries, TBarSeries, THorizBarSeries, TPieSeries, TChartShape, TFastLineSeries, TArrowSeries, TGanttSeries, TBubbleSeries. Согласитесь, что список нельзя назвать скромным — разработчик компонента Дэвид Бернеда потрудился на славу… К счастью, Дэвид превосходно владеет канонами объектно-ориентированного программирования. Поэтому все перечисленные типы графиков предусмотрительно построены на основе единого базового класса — графика TChartSeries (рис. 5). Различия между родительским и дочерними классами сведены к незначительным деталям, а это существенно упрощает процесс эксплуатации диаграммы программистом. Оставив детали на самостоятельное изучение, мы подробно поговорим о главном — фундаментальном классе TChartSeries.

Независимо от класса, любой график инкапсулирует в своем составе ряд обязательных объектов. Вот основные из них:

- TChartValueList — специализированный список, предназначенный для ввода, редактирования и удаления массива значений (точек графика);
- TSeriesPointer — специализированный класс для создания в графике простейших геометрических фигур;
- TSeriesMarks — метки-пояснения над точками графика.

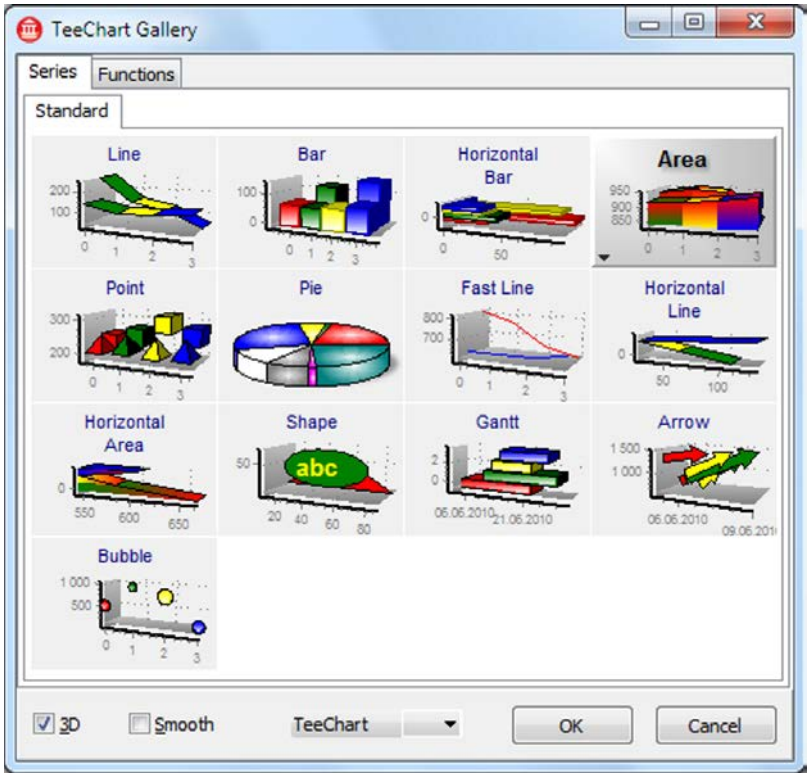


Рис. 4. Выбор вида графика

Для создания экземпляра графика программным способом вызывается его конструктор:

```
constructor Create(Chart : TCustomChart);
```

Единственный параметр конструктора — ссылка на диаграмму, в состав которой войдет только что рожденный график. Кстати, мы уже упоминали о том, что график способен как кошка гулять сам по себе, не принадлежа ни одной из диаграмм проекта, но тогда проку от него будет немного, т. к. без владельца он окажется невидимым. Поэтому в большинстве случаев для графика целесообразно назначить хозяина, а если этого не было сделано в конструкторе, то для переподчинения графика диаграмме требуется обратиться к его свойству

```
property ParentChart : TCustomChart;
```

Тем более, что никогда не поздно вывести график из-под опеки диаграммы — просто передайте в свойство `ParentChart` неопределенный указатель `nil` (листинг 2).

Листинг 2. Смена владельца графика

```
ASeries1.ParentChart:= Chart2;{график ASeries1 передан в подчинение диаграмме Chart2}
ASeries2.ParentChart:= nil; {график ASeries2 утратил владельца}
```

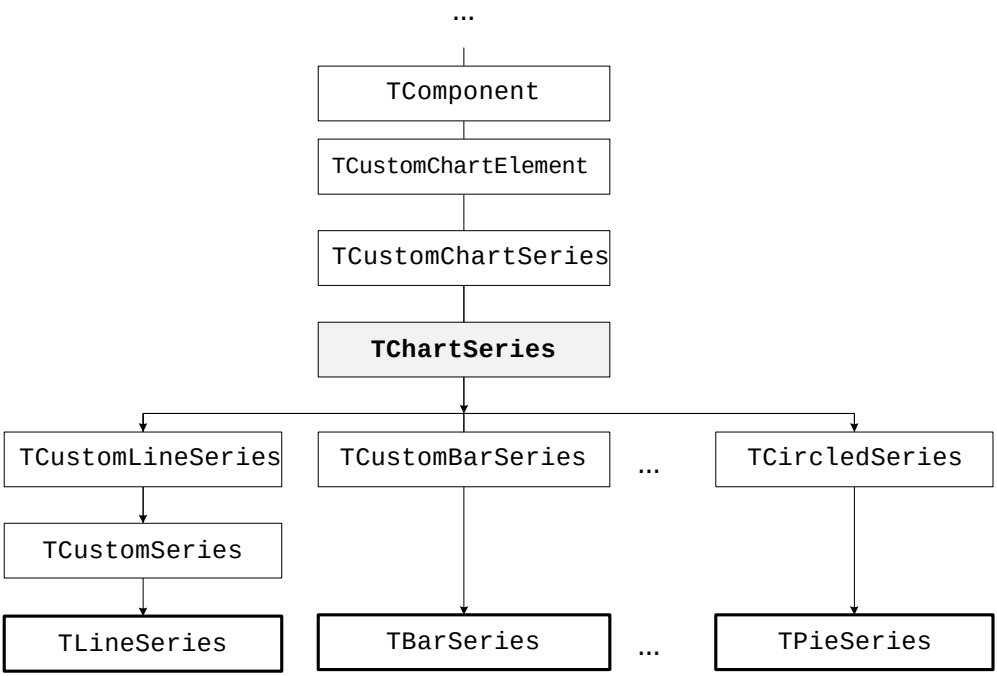


Рис. 5. Место класса `TChartSeries` в иерархии графиков

Точки графика, класс *TChartValueList*

График `TChartSeries` — это сердце диаграммы. Визуально его можно представить в виде обычной линии, заштрихованной области, пирамиды, цилиндра или чего-нибудь еще в том же духе. Рисунок графика строится по его точкам. Для хранения точек в рамках класса создано специальное хранилище — список точек:

```
property ValueList[Index : Longint] : TChartValueList;
```

В справочной системе названное свойство класса позиционируется, как применяемое только при разработке потомков `TChartSeries`, поэтому при программировании обычных приложений оно нам не потребуется. Вместо него у потомков `TChartSeries` предусмотрены два свойства-надстройки:

```
property XValues : TChartValueList; //значения по оси X
property YValues : TChartValueList; //значения по оси Y
```

Они обеспечивают раздельный доступ к коллекции значений по осям X и Y графика соответственно. Например, чтобы добраться до интересующей нас точки для редактирования ее содержимого, воспользуйтесь свойством

```
property Value[Index : Longint] : Double;
```

Пример приведен в листинге 3.

Листинг 3. Перерисовка графика

```
LineSeries1.YValues.Value[5] := 25 ;
LineSeries1.Repaint;
```

Еще более удобный способ, позволяющий адресовать точку графика без явного обращения к хранилищу `TChartValueList`, реализуют три свойства:

```
property XValue[Index : Longint] : Double; //значение по оси X
property YValue[Index : Longint] : Double; //значение по оси Y
property XLabel[Index : Longint] : String; //пояснительная метка
```

Здесь `Index` — порядковый номер точки.

Общее число элементов в хранилище возвратит функция

```
function Count : Longint;
```

Упомяну о существовании ряда свойств и методов графика, несущих исключительно информационную нагрузку (табл. 1).

Таблица 1. *Информационные свойства графика*

Свойство	Описание
`property Total : Double;`	Сумма всех значений графика
`property TotalABS : Double;`	Сумма абсолютных значений
`property MaxValue : Double;`	Максимальное значение в наборе
`property MinValue : Double;`	Минимальное значение в наборе
`function First : Double;`	Значение первой точки графика
`function Last : Double;`	Значение последней точки графика

Для облегчения поиска точки по ее значению создана функция

```
function Locate (Const Value : Double) : Longint;
```

В случае успеха метод возвратит индекс найденной точки в наборе, если такой точки не существует, то функция вернет –1.

Узнав, каким образом организовано хранение данных, пора познакомиться со способами заполнения графика значениями.

Методы вставки новой точки

В классе `TChartSeries` реализовано несколько методов вставки в график новой точки.

Методы

```
function Add(Const AValue :Double; Const ALabel : String;
             AColor : TColor):Longint;
function AddY(Const AYValue : Double; Const AXLabel: String;
              AColor: TColor): Longint;
```

добавляют новую точку в конец набора. Здесь AValue — значение точки по координате Y, ALabel — необязательный пояснительный текст метки и AColor — цвет точки. Функции возвратят индекс точки в наборе. Листинг 4 демонстрирует способ добавления к графику трех новых разноцветных точек.

Листинг 4. Построение графика

```
with Series1 do
begin
    Add(40, 'Точка 1', clRed );
    Add(60, 'Точка 2', clBlue );
    Add(30, 'Точка 3', clGreen );
end;
```

Если необходимо внести новую точку, но описываемую не одной, как в случае Add() и AddY(), а двумя координатами X и Y, то используйте метод

```
function AddXY(Const AXValue, AYValue: Double; Const AXLabel: String;
               AColor: TColor) : Longint;
```

В этом методе появился дополнительный параметр AXValue — значение по оси X.

Пример:

```
AddXY(10,20,'',clTeeColor);
```

Для вставки в график разрыва (пустой точки) применяйте метод

```
function AddNull(const ALabel : String) : Longint;
```

Помимо вставки значений с помощью методов Add..., график в состоянии воспринимать данные из другого графика, набора данных (TADOTable, TSQLQuery, TIBStoredProc и т. д.), элементов ActiveX, COM-объектов. Источник данных указывают в свойстве

```
property DataSource : TComponent;
```

Пример приведен в листинге 5.

Листинг 5. Подключение графика к набору данных

```
Series1.DataSource:=Table1;                    {компонент-таблица}
Series1.YValues.ValueSource:='Table1Field1';{поле с данными}
Series1.XLabelsSource:='Table1LabelField';{поле с подписями}
```

Еще один способ получения данных из другого графика обеспечивает метод

```
procedure AssignValues(Source : TChartSeries);
```

И, наконец, метод

```
procedure FillSampleValues(NumValues : Longint);
```

заполняет график псевдослучайными значениями. В параметр передается число генерируемых точек.

Операция вставки нового значения в график сопровождается вызовом пары событий:

```
property OnBeforeAdd : TSeriesOnBeforeAdd;
TSeriesOnBeforeAdd = Function(Sender: TChartSeries): Boolean of object;
```

и

```
property OnAfterAdd : TSeriesOnAfterAdd;
TSeriesOnAfterAdd = Procedure(Sender : TChartSeries;
                              ValueIndex : Longint) of object;
```

Первое событие возникает до момента вставки, второе — сразу после него.

Каждой точке графика соответствует метка, способная хранить поясняющий текст. Для того чтобы обратиться к метке, воспользуйтесь свойством

```
property XLabel[Index:Longint] : String;
```

Единственный параметр `Index` указывает порядковый номер метки в списке.

В качестве источника меток может выступать не только программный код, но и набор данных, предоставляемый таблицей или запросом. Для этого укажите источник данных в свойстве

```
property XLabelsSource : String;
```

Если в качестве источника данных для компонента выступает запрос или таблица, то потребуется свойство

```
property ValueSource : String;
```

которое уже встречалось в листинге 5. В `ValueSource` определяется имя поля, из которого график получает набор значений. Предусмотрен альтернативный способ применения данного свойства, это случай получения значений из другого графика так, как продемонстрировано в листинге 6.

Листинг 6. Подключение графика к набору данных

```
LineSeries1.DataSource:=BubbleSerie2;
LineSeries1.YValues.ValueSource:='Y';
```

Замечание

После всех операций, связанных со вставкой или удалением точки, желательно перерисовать график, вызвав метод `RefreshSeries()`. Процедура обновляет все графики, принадлежащие диаграмме.

Удаление точек, очистка графика

Для удаления точки с индексом `ValueIndex` из набора вызовите процедуру

```
procedure Delete(ValueIndex : Longint);
```

Для полной очистки содержимого графика используйте метод

```
procedure Clear;
```

В момент удаления всех значений вызывается событие

```
property OnClearValues : TSeriesOnClear;
TSeriesOnClear = Procedure(Sender: TChartSeries) of object;
```

Оформление графика

Сразу оговорюсь, что внешний вид графика в первую очередь определяется его классом и устанавливается еще на этапе его создания (см. рис. 4). А в этом подразделе мы говорим только о дополнительных возможностях по настройке внешнего вида отдельного графика.

У эстетов наибольшей популярностью пользуется свойство, определяющее цвет графика:

```
property SeriesColor : TColor;
```

Нужно отметить, что диаграмма `TChart` умеет самостоятельно назначать новый цвет для каждого, вновь создаваемого в ней графика. Но запас идей у диаграммы невелик: всего 16 цветовых решений, так что, начиная с 17-го графика, цвета начнут повторяться.

Если немного потрудиться, то вполне реально определять индивидуальную окраску для каждой точки одного и того же графика (вспоминайте метод `Add()`, последний параметр которого определяет цвет точки). Но при этом необходимо помнить, что эксклюзивный подбор цвета для точки возможен только при установке в `false` свойства

```
property ColorEachPoint : Boolean;
```

Если же это свойство хранит значение `true`, то все точки графика окрасятся в одинаковые тона.

В качестве источника цвета способны выступать данные из поля таблицы или запроса к базе данных (формат поля должен соответствовать типу данных `TColor`). Имя источника передается в свойство

```
property ColorSource : String;
```

Весьма полезный метод (листинг 7) позволит изменить цвет отдельных точек графика, принадлежащих определенному диапазону:

```
procedure ColorRange(AValueList: TChartValueList;
                     Const FromValue, ToValue: Double; AColor: TColor);
```

В первом параметре `AValueList` указывают список анализируемых значений, в параметрах `FromValue`, `ToValue` устанавливают диапазон значений, последний параметр — `AColor` определяет цвет.

Листинг 7. Изменение цвета диапазона точек

```
LineSeries1.ColorRange(LineSeries1.XValues,
                 EncodeDate(2011,1,1), EncodeDate(2014,12,31), clBlue);
LineSeries1.ColorRange( LineSeries1.YValues, 100,
                        LineSeries1.MaxYValue, clYellow);
```

В заключение упомянем о возможности перевода графика в монохромный режим:

```
property Monochrome : Boolean;
```

Это свойство пригодится при просмотре задания, отправляемого на черно-белый принтер.

Взаимодействие с мышью

График — это не только цветная картинка, призванная развлекать вашего шефа при подведении итогов работы компании за квартал. При должной сноровке график превращается в объект, способный общаться с пользователем. Для этого в нем инкапсулированы функции, позволяющие идентифицировать точку графика, над которым находится указатель мыши.

В подтверждение сказанному предложу небольшой пример. При полной инсталляции Delphi на компьютер устанавливаются таблицы демонстрационной базы данных dbdemos.mdb. Сейчас нас интересует таблица `Country`, в ней содержатся статистические данные о государствах Северной и Южной Америк, их столицах и населении. Давайте попробуем построить диаграмму с графиком, отражающим связь между страной и ее населением.

Для воплощения задуманного потребуется начать новый проект, разместить на его главной форме диаграмму `Chart1`, компонент доступа к данным `ADOTable1` и запрос `ADOQuery1`. Воспользуйтесь свойством `ConnectionString` компонентов `ADOTable1` и `ADOQuery1`, чтобы подключить их к БД dbdemos.mdb.

Переходим в редактор кода и объявляем переменную `BarSeries`, в которой будет храниться экземпляр будущего графика:

```
private
    BarSeries : TBarSeries; {подключите модуль Series}
```

Листинг 8 содержит исходный код, вызываемый при отображении формы на экране.

Листинг 8. Вывод формы на экран

```
procedure TForm1.FormShow(Sender: TObject);
var Color:TColor;
    p:double;
begin
Chart1.Title.Caption:='Данные о населении';
Chart1.AllowZoom:=False; //отключаем увеличение
BarSeries:=TBarSeries.Create(Chart1); //создаем график
BarSeries.ParentChart:=Chart1; //передаем график во владение диаграмме
with ADOTable1 do
begin
TableName:='country';
Open; //открываем таблицу
```

```
While Eof<>True do {цикл перебора всех записей в таблице}
  begin
   p:=FieldByName('Population').AsFloat;

   {от количества жителей зависит цвет}
   if p<10000000 then Color:=clYellow
                 else
   if p<50000000 then Color:=clGreen
                 else Color:=clRed;
  {вставляем новое значение в график}
   BarSeries.AddY(p, FieldByName('Name').AsString,Color);
   Next; //переход к следующей записи
  end;
  Close; //завершаем работу с таблицей
  end;
end;
```

Прокомментирую ключевые эпизоды события `OnShow()`. В момент появления на экране формы создается экземпляр графика, а ссылка на него заносится в переменную `BarSeries`. Затем мы подключаем компонент `ADOTable1` к таблице и перебираем все строки результирующего набора внутри цикла `while..do`. Во время перебора для каждой записи создаем соответствующую точку на графике. Точка графика запоминает данные о числе жителей (поле `Population`) и названии страны (поле `Name`). Специально акцентирую внимание на том, что название страны мы внесли в подпись точки по горизонтальной оси (свойство `XLabel`) — эти данные позднее пригодятся. Запустите приложение на выполнение — перед вами появится график с обещанными данными.

Прежде чем мы перейдем ко второй части примера, расскажу, что для преобразования координат указателя мыши в порядковый индекс точки графика обеспечит функция

```
function GetCursorValueIndex : LongInt;
```

Если метод возвращает –1, значит, позиция курсора не совпадает с точкой графика.

Теперь рассмотрим взаимодействие графика и мыши. Для этого нам потребуется помощь компонента `ADOQuery1` и событие `OnMouseDown()` диаграммы `Chart1` (листинг 9).

Листинг 9. Обработка события `OnMouseDown` диаграммы

```
procedure TForm1.Chart1MouseDown(Sender: TObject; Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
var Index:integer;
    Capital : String;
```

```
begin
Index:=BarSeries.GetCursorValueIndex; {узнаем, над какой точкой
                                       графика расположен указатель мыши}
if Index>-1 then
   with ADOQuery1 do
    begin
      SQL.Clear; //готовим запрос
      SQL.Add('Select Capital from country where Name="'
               +BarSeries.XLabel[Index]+'"');
    Open; //отправляем запрос
    Capital:=FieldByName('Capital').AsString; //сохраняем результат
    Close; //закрываем набор данных
    ShowMessage('Столица страны -'+Capital); //выводим результаты запроса
   end;
end;
```

Первая строка процедуры служит для выяснения индекса точки, над которой была нажата кнопка мыши. Если метод `GetCursorValueIndex()` вернул неотрицательное значение, то формируем запрос к таблице, задача которого заключается в поиске столицы страны с именем `Name`. Текст названия мы получим из свойства `XLabel` точки графика с индексом — из `Index`. Результаты запроса помещаются в строковую переменную `Capital` и выводятся в отдельном сообщении. На рис. 6 представлен экранный снимок приложения после обработки щелчка по точке "Mexico".

Замечание

Взаимодействие с базами данных — профессиональная обязанность диаграммы `TDBChart`. По сравнению со своим ближайшим родственником `TChart` этот компонент обладает рядом особенностей, к которым, в частности, относится свойство `DataSource`, обеспечивающее прямую связь с источником данных.

Познакомимся еще с некоторыми методами, связывающими график и манипулятор мышь. Индекс элемента графика вполне можно выяснить по его экранным координатам:

```
function Clicked(x, y: Integer) : Longint;
```

Функция требует, чтобы мы передали координаты (`X`, `Y`), и возвращает индекс элемента, описываемого этими точками.

Для того чтобы сразу прочитать значения (`X`, `Y`) выбранной точки, понадобится процедура

```
procedure GetCursorValues(var  x,y : Double );
```

Преобразование экранной координаты X (или Y) в хранящееся в точке графика значение осуществляет метод

```
function XScreenToValue(ScreenPos : Longint) : Double; //координата X
function YScreenToValue(ScreenPos : Longint) : Double; //координата Y
```

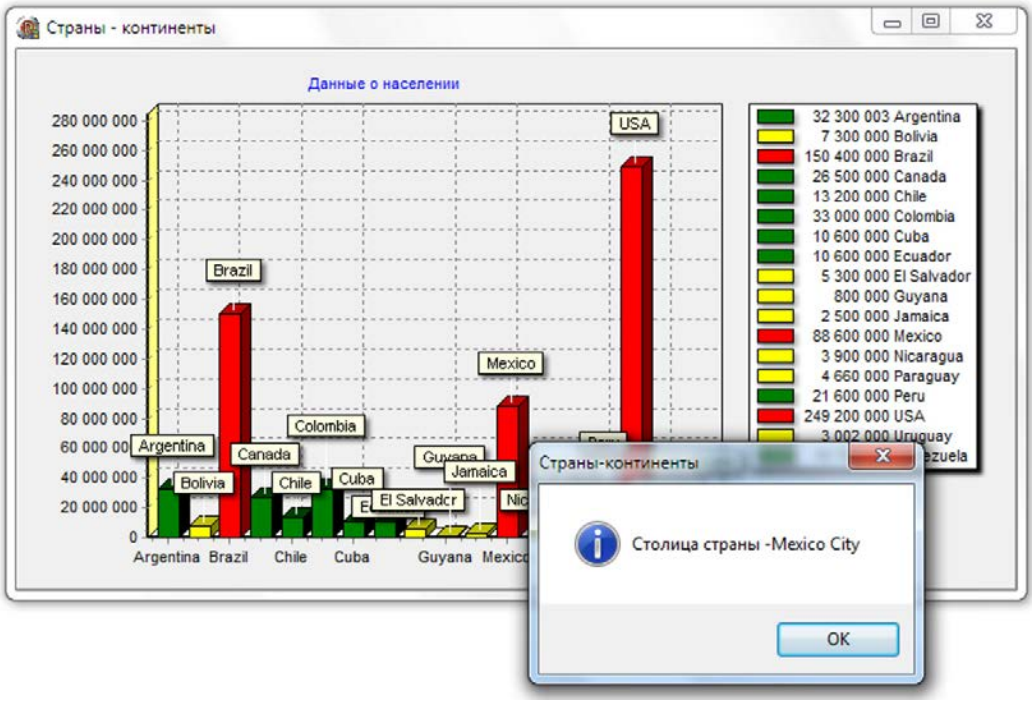


Рис. 6. Экранный снимок приложения "Страны-континенты"

В завершение приведем расширенное описание события `OnClick()` графика:

```
TSeriesClick = procedure(Sender : TChartSeries; ValueIndex: LongInt;
                         Button: TMouseButton; Shift: TShiftState;
                         X, Y: Integer);
```

Событие способно сразу возвратить индекс элемента — параметр `ValueIndex`.

Метки графика — *TSeriesMarks*

Помимо поясняющих подписей по оси X диаграммы каждая точка графика дополнительно снабжается своей собственной текстовой меткой. Визуально метка состоит из цветного прямоугольника с текстовой строкой на нем, например, на рис. 6 в метках отображались названия стран. Метки особенно удобны в ситуации, когда подписи на горизонтальной оси чрезмерно велики и не помещаются на экране.

Общую настройку всех меток графика обеспечивает свойство

```
property Marks : TSeriesMarks;
```

Самое коварное, что может сделать `TSeriesMarks`, — отключить вывод меток на экран. Для этого нужно установить в `false` свойство

```
property Visible : Boolean;
```

Второе по важности свойство назначает стиль меток. Стиль определяет, какие именно данные будут отображены в метке:

```
property Style : TSeriesMarksStyle;
TSeriesMarksStyle = (smsValue, smsPercent, smsLabel, smsLabelPercent,
                     smsLabelValue, smsLegend, smsPercentTotal,
                     smsLabelPercentTotal, smsXValue);
```

В зависимости от выбранного стиля в тексте метки окажется только числовое значение точки, значение в процентах или описание. По умолчанию назначается стиль `smsLabel` — текст метки будет идентичен подписи данной точки по оси X диаграммы. А если в нашем примере с населением Северной и Южной Америк установить стиль меток в `smsLabelPercent`, то, помимо текста с названием страны, в метку попадут дополнительные данные — процентное соотношение населения данной страны ко всем остальным государствам (листинг 10).

Листинг 10. Изменение стиля меток

```
With BarSeries.Marks do
begin
  Visible:=True;
  Style:=smsLabelPercent;
end;
```

Соединительная линия, связывающая точку графика и соответствующую ей метку, рисуется пером, описываемым в свойстве

```
property Arrow : TChartPen;
```

Длина соединительной линии устанавливается в свойстве

```
property ArrowLength : Integer;
```

Единица измерения — пикселы.

Особенности прорисовки границ прямоугольной области метки задаются в свойстве

```
property Frame : TChartPen;
```

Свойство

```
property BackColor : TColor;
```

определяет цвет фона метки.

Вместо заливки метки цветом, при желании допускается сделать их прозрачными, передав в свойство

```
property Transparent : Boolean;
```

значение `true`.

Шрифт метки определяется в свойстве

```
property Font : TFont;
```

Контроль за тем, чтобы метки не выходили за пределы рабочей области, осуществляется в свойстве

```
property Clip : Boolean;
```

Если свойство установлено в true, то метки отсекаются.

Единственный метод, объявленный у класса TSeriesMarks, преобразует экранные координаты в индекс метки:

```
property Clicked(X, Y : Integer):Integer ;
```

Все перечисленные свойства доступны в инспекторе объектов и на вкладке **Marks** встроенного редактора диаграммы.

Легенда диаграммы — *TChartLegend*

Пояснительная надпись о графиках, содержащихся в диаграмме, называется легендой. Доступ к описывающему легенду свойству обеспечивается из свойства

```
property Legend : TChartLegend;
```

За включение (отключение) показа легенды ответственно свойство

```
property Visible : Boolean;
```

Свойство

```
property LegendStyle : TLegendStyle;
TLegendStyle = (lsAuto,lsSeries,lsValues,lsLastValues);
```

назначает стиль легенды.

Свойство

```
property TextStyle : TLegendTextStyle;
TLegendTextStyle = (ltsPlain, ltsLeftValue, ltsRightValue,
                    ltsLeftPercent, ltsRightPercent, ltsXValue);
```

определяет стиль текста.

Местоположение области легенды определяет свойство

```
property Alignment : TLegendAlignment;
TLegendAlignment = (laLeft,laRight,laTop,laBottom);
```

Координаты прямоугольника с легендой описываются в свойстве

```
property RectLegend : TRect; //только для чтения
```

доступном только для чтения.

Координатные оси диаграммы — *TChartAxis*

Все координатные оси диаграммы создаются на основе описанного в модуле TeEngine класса TChartAxis. Диаграмма умеет поддерживать до пяти осей одновременно, для этого в TChart объявлены соответствующие свойства:

```
property BottomAxis : TChartAxis; //нижняя горизонтальная ось
property TopAxis : TChartAxis;   //верхняя горизонтальная ось
property LeftAxis : TChartAxis;  //левая вертикальная ось
property RightAxis : TChartAxis;  //правая вертикальная ось
property DepthAxis : TChartAxis;  //ось – Z для 3-мерных графиков
```

Каждая ось идентифицирует свое место в диаграмме посредством двух свойств:

```
property Horizontal : Boolean; //только для чтения
property OtherSide : Boolean; // только для чтения
```

Соответствие между значениями, возвращаемыми данными свойствами и местом оси в диаграмме, приведено в табл. 2.

Таблица 2. *Идентификация оси диаграммы*

Координатная ось	**Свойства оси**	
	`Horizontal`	`OtherSide`
`LeftAxis`	`False`	`False`
`TopAxis`	`True`	`True`
`RightAxis`	`False`	`True`
`BottomAxis`	`True`	`False`

Свойство

```
property AxisVisible : Boolean;
```

определяет видимость координатных осей.

По умолчанию значения точек начала и окончания отсчета по осям задаются автоматически. Для этого ось анализирует хранящиеся в точках графика данные и выявляет самое большое и самое малое содержащееся в них значение. Если в этой услуге мы не нуждаемся, то для отказа от автоматической настройки точек отсчета оси нужно перевести в состояние `false` свойство

```
property Automatic : Boolean; //по умолчанию true
```

Далее программист самостоятельно назначает начало отсчета:

```
property Minimum : Double;
```

и окончание отсчета:

```
property Maximum : Double;
```

В этом случае код, назначающий размерность для левой вертикальной оси от нуля до сотни, будет выглядеть примерно так, как в листинге 11.

Листинг 11. Настройка размерности оси диаграммы

```
with Chart1.LeftAxis do
begin
 Automatic:= False ;
 Minimum  := 0;
 Maximum  := 100;
end;
```

Аналогичную задачу решает метод

```
procedure SetMinMax(Const AMin, AMax : Double);
```

Процедура сама переведет свойство `Automatic` в состояние `false` и передаст в свойства `Minimum` и `Maximum` значения аргументов `AMin` и `AMax`.

Замечание

Самое большое и самое малое значения точек графика, связанного с осью координат, проще всего выяснить, обратившись к методу `CalcMinMax()`.

Иногда обязанности по определению точек отсчета целесообразно распределить между осью диаграммы и программистом. Допустим, мы планируем, чтобы отсчет всегда начинался с нулевой отметки, а максимальное его значение доверяем выбрать диаграмме. Для этого у оси предусмотрена пара свойств

```
property AutomaticMaximum : Boolean;
property AutomaticMinimum : Boolean;
```

определяющих, какая из точек настраивается автоматически (листинг 12).

Листинг 12. Автоматический выбор верхнего предела оси

```
With Chart1.LeftAxis do
begin
  Automatic := False;
  Minimum := 0;
  AutomaticMaximum := True;
end;
```

При необходимости координатную ось допустимо привести к логарифмическому виду, установив в `true` свойство

```
property Logarithmic : Boolean;
```

Для того чтобы задать общую подпись для координатной оси, воспользуйтесь свойством

```
property Title : TChartAxisTitle;
```

Пример:

```
Chart1.BottomAxis.Title.Caption:='м/с';
```

Для того чтобы рядом с координатной осью выводились метки, содержащие текущее значение, установите в `true` свойство

```
property Labels : Boolean;
```

Свойства

```
property LabelsFont : TFont; // шрифт метки
property LabelsAngle : Integer;//угол поворота метки, по умолчанию – 0
property LabelStyle: TAxisLabelStyle;
```

определяют внешний вид меток.

Описание `TAxisLabelStyle` приведено в табл. 3.

Таблица 3. *Класс* `TAxisLabelStyle`

Значение	Описание
`talAuto`	Автоматический выбор стиля
`talNone`	Меток нет

Таблица 3 *(окончание)*

Значение	Описание
talValue	Метки оси основываются на текущих значениях соответствующего графика и находятся в пределах его минимального и максимального значений
talMark	Каждый отдельный график описывается свойством Marks. Такие метки схожи с обычными всплывающими подсказками
talText	Стиль метки такой же, как свойства графика — XLabels

Расстояние от границы диаграммы до меток устанавливается в свойстве

```
property LabelsSize : Integer;
```

Расстояние измеряется в пикселах, по умолчанию установлено в ноль.

Формат выводимых значений устанавливается в маске ввода:

```
property AxisValuesFormat : String;
```

По умолчанию шаблон маски ввода выглядит следующим образом: `'#,##0.###'`.

Если координатная ось предназначена для отображения времени, то форматирующая маска вывода даты/времени описывается в свойстве

```
property DateTimeFormat : String;
```

Пример форматирующей строки:

```
Chart1.BottomAxis.DateTimeFormat:='dd/mm/yyyy'; // 15.11.2004
```

Текст, выводимый в метках оси, можно назначить индивидуально для каждой отдельной отметки. Для этого предназначено событие диаграммы TChart:

```
property OnGetAxisLabel : TAxisOnGetLabel;
TAxisOnGetLabel = procedure(Sender : TChartAxis; Series : TChartSeries;
                  ValueIndex: LongInt; Var LabelText: String) of object;
```

Событие вызывается перед выводом метки на экран. Здесь Series — график, ValueIndex — индекс точки, LabelText — текст метки.

Минимальный шаг между метками на оси описывает свойство

```
property Increment : Double;
```

Чтобы сделать ось видимой, установите в true ее свойство

```
property Visible : Boolean;
```

Свойство

```
property Axis : TChartPen;
```

отвечает за прорисовку координатной оси, в нем описываются характеристики пера, которым прочерчивается ось.

Свойство

```
property Grid : TChartPen;
```

определяет особенности вывода на экран линий сетки.

Масштабирование

Доброжелательность диаграммы не имеет границ. Она всегда прилагает максимальные усилия, чтобы обслуживаемый ею график со всеми удобствами и, главное, целиком поместился в ее рабочей области. Но, к сожалению, такие труды компонента `TChart` достойны уважения только тогда, когда график намерен отображать не более двух-трех десятков точек. Если же массив точек превысит этот предел, то их "утрамбовка" в границах сравнительно небольшого окна окажется бессмысленной затеей — получится нагромождение отсчетов, в котором не разобраться. Для решения проблемы можно разбить рабочую область диаграммы на несколько страниц с ограниченным числом точек на одной странице.

Компонент `TChart` умеет отображать не только график в целом, но и его отдельный участок. Для этого достаточно настроить уже знакомые нам свойства `Minimum` и `Maximum` обеих осей диаграммы. Допустим, отсчеты по осям X и Y начинались в точке (0,0) и заканчивались в (100, 100). Если мы хотим увидеть только центральную часть рабочей области диаграммы, то нам поможет листинг 13.

Листинг 13. Вывод части графика

```
With Chart1 do
begin
 BottomAxis.Automatic := False;
 BottomAxis.Minimum   := 20;
 BottomAxis.Maximum   := 80;

 LeftAxis.Automatic := False ;
 LeftAxis.Minimum   := 20;
 LeftAxis.Maximum   := 80;
end;
```

Замечание

Практически того же результата можно достичь без единой строки кода. Чтобы выбрать увеличиваемую прямоугольную область, пользователю достаточно нажать левую кнопку мыши и, удерживая ее, выделить весь прямоугольник, начав с левого верхнего угла. Далее диаграмма автоматически настроится на показ увеличиваемой области. Для уменьшения масштаба пользователь выполняет обратные действия: нажимает левую кнопку мыши в правом нижнем углу прямоугольника и, удерживая кнопку, выделяет прямоугольник в обратном направлении.

Для того чтобы разрешить диаграмме изменять свой масштаб при помощи описанных манипуляций с мышью, свойству

```
property AllowZoom : Boolean;
```

должно быть присвоено значение `true`.

При масштабировании можно явно указать координаты прямоугольной области:

```
procedure ZoomRect(Const Rect : TRect);
```

Пример:

```
Chart1.ZoomRect( Rect( 5,5,Chart1.Width-5,Chart1.Height-5 ) );
```

Простейший способ масштабирования — указание процента увеличения в свойстве

```
procedure ZoomPercent( Const  PercentZoom : Double );
```

Пример:

```
Chart1.ZoomPercent(200);  { Увеличение на 200% }
```

Если передаваемое значение больше 100%, то масштаб увеличивается, если меньше — уменьшается. Будьте внимательны, за 100% принимается текущее состояние диаграммы.

В момент изменения масштаба генерируется событие

```
property OnZoom : TNotifyEvent;
```

Возврат к нормальному режиму увеличения достигается при помощи метода

```
procedure UndoZoom;
```

Событие

```
property OnUndoZoom : TNotifyEvent;
```

устанавливает масштаб по умолчанию.

Процесс изменения масштаба может сопровождаться анимацией, что несколько замедляет ход увеличения (уменьшения) масштаба, но делает диаграмму более привлекательной. Для включения анимации установите в `true` свойство

```
property AnimatedZoom : Boolean;
```

Включение анимации заставит диаграмму переходить от одного масштаба к другому не моментально, а постепенно, отображая промежуточные состояния. Количество переходных состояний указывается в свойстве

```
property AnimatedZoomSteps : Integer;
```

Многостраничные диаграммы

Нередко программист сталкивается с ситуацией, когда все точки графика физически невозможно отобразить на рабочей поверхности диаграммы. В таком случае допускается разбиение диаграммы на несколько страниц.

Самое главное, что нужно сделать для разделения диаграммы на страницы, — указать максимальное число точек, размещаемых на одной странице:

```
property MaxPointsPerPage : LongInt;
```

Допустим, что график включает в свой состав 95 точек и свойство `MaxPointsPerPage` равно 10, тогда последняя страница будет содержать всего 5 точек.

Свойство

```
property ScaleLastPage : Boolean;
```

позволяет уточнить, как разместятся точки графика на последней странице. Если оно установлено в `false`, то точки графика равномерно распределятся по горизонтальной оси, иначе точки прижмутся к левой границе диаграммы.

Об общем количестве страниц, собранных в диаграмме, проинформирует свойство

```
function NumPages : Longint;
```

Индивидуальный номер страницы служит для ее идентификации или для перехода к новой странице:

```
property Page : LongInt;
```

Два метода позволяют переходить к следующей или возвращаться к предыдущей странице:

```
procedure NextPage;
procedure PreviousPage;
```

О переходе к новой странице нам просигнализирует событие

```
property OnPageChange : TNotifyEvent;
```

Вариант использования события приведен в листинге 14.

Листинг 14. Информирование о переходе к новой странице

```
procedure TForm1.Chart1PageChange(Sender: TObject);
begin
Label1.Caption:=Format('Страница %d из %d',
              [ (Sender as TChart).Page, (Sender as TChart).NumPages]);
end;
```

Экспорт диаграмм

В классе `TChart` предусмотрен ряд методов, предназначенных для экспорта текущего изображения диаграммы в графический файл:

```
procedure SaveToBitmapFile( Const FileName : String );{формат *.bmp}
procedure SaveToMetafile(Const FileName : String); {формат *.wmf}
procedure SaveToMetafileEnh(Const FileName : String); { формат *.emf}
```

Единственный параметр предназначен для указания имени файла, в который будет сохранено изображение.

Во время выполнения программы текущие данные вполне можно сохранить во временный файл с именем `AName`:

```
procedure SaveChartToFile(AChart : TCustomChart; const AName : String);
```

Для загрузки данных из временного файла понадобится метод

```
procedure LoadChartFromFile(var AChart : TCustomChart;
                            const AName:String);
```

Метод

```
procedure LoadChartFromURL(var AChart : TCustomChart; Const URL : String);
```

загружает файл из Интернета по адресу, указанному в параметре `URL`.

Отмечу, что в классе `TChart` подготовлены специальные методы для работы с буфером обмена:

```
procedure CopyToClipboardBitmap;
procedure CopyToClipboardMetafile(Enhanced:Boolean);
```

Последний метод копирует рабочую область диаграммы в буфер обмена Windows в виде метафайла. Формат метафайла определяется параметром `Enhanced`: `false` — формат WMF, `true` — EMF.

Печать диаграммы

Ориентация страницы печати осуществляется при помощи метода

```
procedure PrintOrientation ( AOrientation : TPrinterOrientation );
```

Можно назначить портретную или альбомную ориентацию.

Другие способы:

```
procedure PrintPortrait ; //портретная ориентация странице
procedure PrintLandscape; //альбомная ориентация странице
```

Отступы полей страницы настраивают при помощи свойства

```
property PrintMargins : TRect;
```

По умолчанию отступ всех полей равен 15.

Для отправки на печать всей клиентской части диаграммы вызовите метод

```
procedure Print;
```

В момент печати свойство

```
property Printing : Boolean; //только для чтения
```

устанавливается в `true`.

Достаточно интересные методы позволят распечатать диаграмму в строго определенном месте на странице:

```
procedure PrintRect(const R : TRect) ;
procedure PrintPartial(const R : TRect) ;
```

Место печати указывают в прямоугольной структуре `R:TRect`. Максимальный размер участка для печати — (`0,0,Printer.PageWidth-1, Printer.PageHeight-1`). Единственное отличие метода `PrintRect()` от `PrintPartial()` в том, что последний не вызовет методы принтера `BeginDoc` (новый документ) и `EndDoc` (конец документа).

При желании можно заставить диаграмму отобразить саму себя на любом объекте, обладающем свойством `Canvas`, с помощью метода

```
procedure Draw(UserCanvas : TCanvas ; Const UserRect : TRect);
```

Во втором параметре нужно установить границы прямоугольника, в пределах которого будет осуществляться графический вывод. Листинг 15 демонстрирует порядок вывода диаграммы на печать с помощью метода `Draw()`.

Листинг 15. Отправка диаграммы на печать

```
Var UserRect : TRect;
Begin
 UserRect.Left  := 0;
 UserRect.Right := Printer.PageWidth div 2;
```

```
 UserRect.Top   := 0;
 UserRect.Bottom:= Printer.PageHeight div 3;

 Printer.BeginDoc;
 Chart1.Draw(Printer.Canvas, UserRect);
 Printer.EndDoc;
End;
```

Для печати в монохромном виде целесообразно установить в true свойство

```
property MonochromePrinting : Boolean;
```

Упорядочивание графиков внутри диаграммы

Так как диаграмма может содержать несколько графиков, то зачастую пользователь хочет получить возможность менять их местами — выводить на передний план или прятать назад. Для этого предназначены два метода:

```
procedure SeriesDown ( ASeries : TChartSeries );//назад
procedure SeriesUp ( ASeries : TChartSeries ); //вперед
```

Каждый график (потомок класса TChartSeries) обладает специальным свойством, хранящим его порядковый индекс:

```
property ZOrder : LongInt;
```

Индекс самого верхнего графика равен нулю. Свойство доступно только для чтения.

Максимально возможное порядковое значение графика можно прочитать в свойстве диаграммы

```
property MaxZOrder : Longint;
```

Обеспечение объемного вида диаграммы

Для представления диаграммы в объемном виде установите в true свойство

```
property View3d : Boolean;
```

Степень объемности назначает свойство

```
property Chart3dPercent : Integer;
```

Оно может принимать любое значение из диапазона от 1 до 100. По умолчанию установлено значение 15%.

Свойство

```
property View3dWalls : Boolean;
```

включает объемные стены.

Два справочных свойства (только для чтения) указывают размер (в пикселах), зарезервированный для вывода изображений в трехмерном виде:

```
property Height3D : Longint;
property Width3D : Longint;
```

Еще два свойства (только для чтения) характеризуют высоту и ширину отдельного графика в диаграмме при их объемном отображении:

```
property SeriesHeight3D : Longint;
property SeriesWidth3D : Longint;
```

Как правило, последние четыре свойства программист использует при рисовании на канве диаграммы.

Особенности компонента *TDBChart*

Чтобы упростить формирование аналитических отчетов и форм в проектах БД, создатели диаграммы TChart доработали компонент, введя дополнительный промежуточный класс — TCustomDBChart (см. рис. 1). В результате на свет появилась диаграмма TDBChart, обладающая всей функциональностью рассмотренного ранее компонента TChart и способная взаимодействовать с набором данных (потомком TDataSet) и источником данных (TDataSource). Для работы с данными диаграмма TDBChart обращается за поддержкой к опорному модулю баз данных DB.pas.

Чтобы увидеть внешнюю сторону усовершенствований компонента TDBChart, добавьте к диаграмме график и переключитесь на страничку выбора источника данных **Data Source**. К имевшимся ранее (у компонента TChart) источникам **Manual**, **Random**, **Series**, **Function** и **Cross Tab** добавились еще три: **Single Record**, **Summary** и **Dataset**.

Для того чтобы графики диаграмм автоматически загружали данные, сразу после открытия набора данных проконтролируйте, чтобы свойство

```
property AutoRefresh: Boolean;//по умолчанию true
```

находилось в состоянии true. Если режим автоматического обновления графиков отключен, то для построения графика следует вызвать метод

```
procedure RefreshDataSet(ADataSet: TDataSet; ASeries: TChartSeries);
```

Здесь ADataSet — ссылка на исходный набор данных; ASeries — график, чьи данные следует обновить. Если следует обновить все графики диаграммы, то вызываем процедуру

```
procedure RefreshData;
```

Если исходные данные изменяются очень часто, то имеет смысл установить периодичность обновления (в секундах):

```
property RefreshInterval: Integer;//по умолчанию 0
```

В демонстрационной базе данных dbdemos.mdb есть таблица служащих EMPLOYEE, хранящая данные о денежных средствах, выплаченных работникам компании. Попробуем подключиться к таблице EMPLOYEE с помощью компонента TDBChar, для этого нам понадобятся услуги компонента TADOQuery.

С помощью свойства ConnectionString подключите компонент-запрос к файлу dbdemos.mdb. А затем воспользуйтесь строками кода из листинга 16.

Листинг 16. Построение графика на основе набора данных

```
procedure TForm1.FormShow(Sender: TObject);
begin
 with ADOQuery1 do
 begin
  SQL.Clear;
  SQL.Add('SELECT (LastName+" "+ FirstName) as Name, HireDate,Salary');
  SQL.Add('FROM employee Order By LastName,FirstName');
  Open;
 end;

 with TPointSeries.Create(DBChart1) do
 begin
  ParentChart:=DBChart1;
  DataSource:=ADOQuery1;
  XValues.ValueSource:='HireDate'; {дата выплаты}
  XValues.DateTime:=True;
  YValues.ValueSource:='Salary'; {размер оплаты}
  XLabelsSource:= 'Name'; {метки}
 end;
end;
```

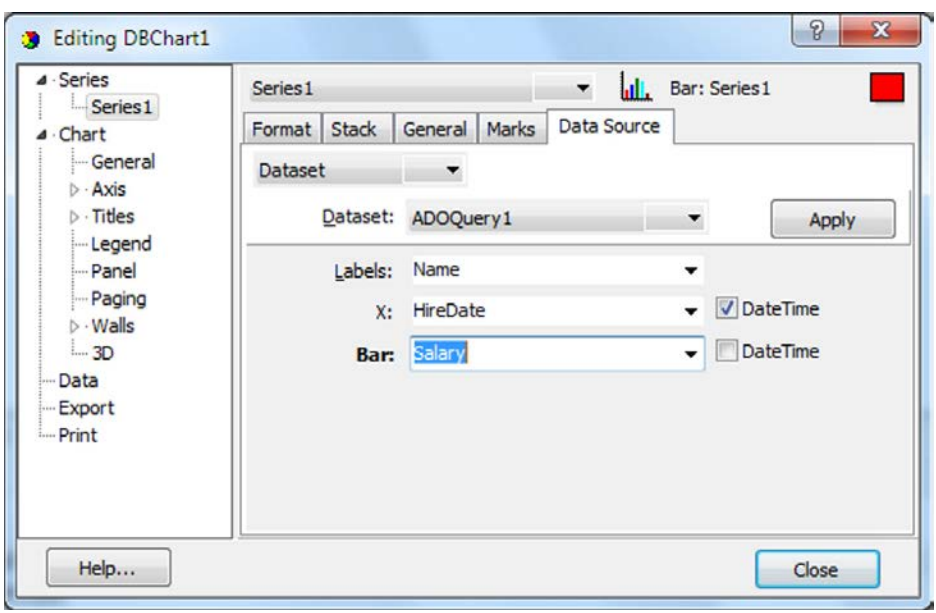


Рис. 7. Подключение к источнику данных с помощью встроенного редактора диаграммы

В рассмотренном примере мы динамически создаем график с заданными свойствами, однако эту операцию можно проделать и вручную, следует лишь заполнить соответствующие поля на вкладке **Data Source** графика (рис. 7).

Графики диаграммы в качестве набора данных

Долгое время компоненты-диаграммы обладали одним существенным недостатком: во время работы приложения пользователю было крайне сложно внести в отображаемые в них графики свои правки. Чтобы реализовать этот важный элемент пользовательского интерфейса, программистам пришлось проявить выдумку и… сочинить много нового кода. Однако с появлением компонентов `TChartDataSet` и `TSeriesDataSet` ситуация принципиально изменилась, теперь для предоставления пользователю возможности редактировать данные графиков программисту достаточно пару раз щелкнуть мышью.

Взгляните на иерархию классов `TChartDataSet` и `TSeriesDataSet` (рис. 8), наши новые знакомые являются наследником одного из наиболее важных классов в иерархии VCL — универсального набора данных `TDataSet`. Что это дает? Ответ прост. Новые классы позволяют работать с содержащимися в графиках данными так, будто это потомки `TDataSet`, другими словами, — обычные таблицы.

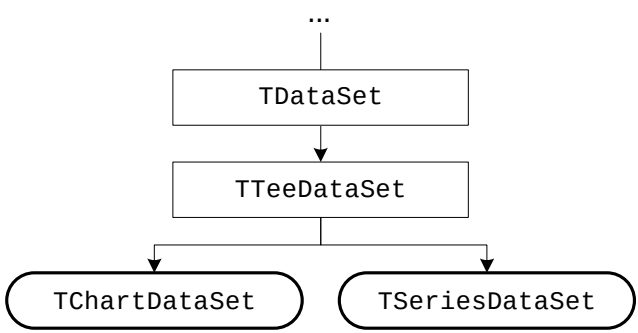


Рис. 8. Иерархия источников данных для работы с графиками

Основное свойство компонента `TSeriesDataSet`:

```
property Series: TChartSeries;
```

Оно позволяет источнику данных подключиться к любому графику диаграммы.

Основное свойство компонента `TChartDataSet`:

```
property Chart: TCustomChart;
```

Благодаря этому свойству мы присоединяемся ко всем графикам диаграммы.

После подключения к графикам, следует установить в `true` свойство `Active` и предоставить пользователю удобные средства по редактированию данных. Вос-

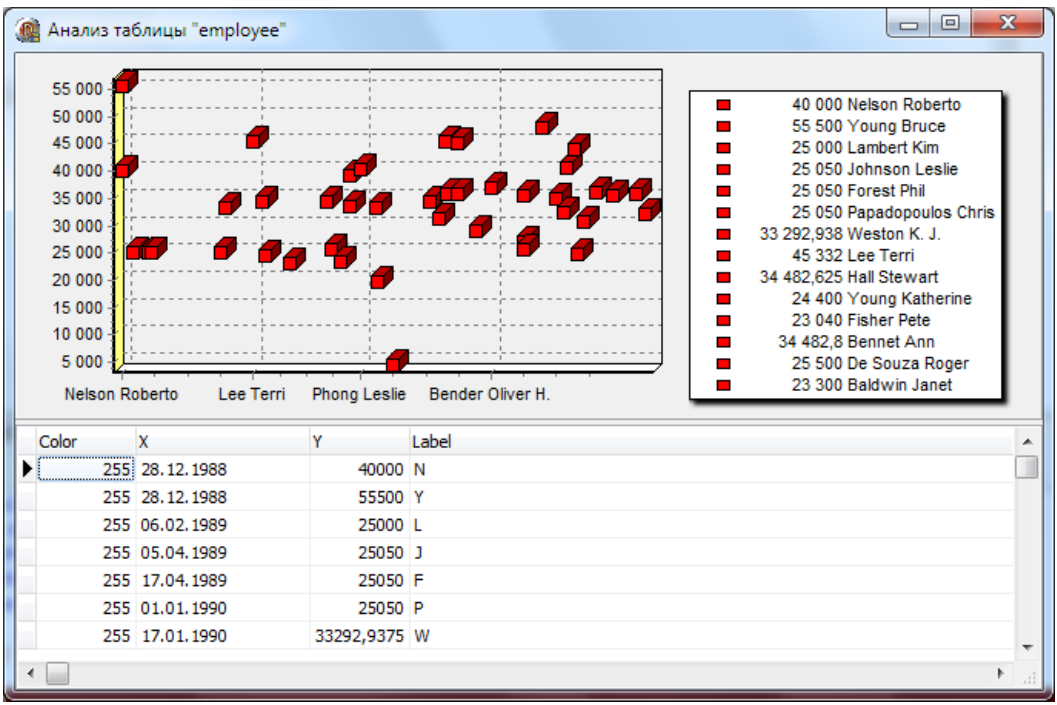
пользуйтесь предыдущим примером с демонстрационной таблицей служащих `EMPLOYEE`. Добавьте на форму три компонента:

- набор данных `ChartDataSet1:TChartDataSet` — с помощью свойства `Chart` присоедините компонент к диаграмме `DBChart1`;
- источник данных `DataSource1: TDataSource` — воспользовавшись свойством `DataSet`, подключите источник к набору данных `ChartDataSet1`;
- сетка `DBGrid1:TDBGrid` — обратившись к свойству `DataSource`, подключите сетку к источнику данных.

Добавьте в конец листинга 16 (событие `OnShow()` главной и единственной формы примера) строку

```
ChartDataSet1.Open;
```

На этом программирование закончилось, результаты наших трудов вы обнаружите на рис. 9.



Color	X	Y	Label
255	28.12.1988	40000	N
255	28.12.1988	55500	Y
255	06.02.1989	25000	L
255	05.04.1989	25050	J
255	17.04.1989	25050	F
255	01.01.1990	25050	P
255	17.01.1990	33292,9375	W

Рис. 9. Диаграмма с подключенной сеткой `TDBGrid`

Заметьте, что представленные в сетке данные никоим образом не дублируют содержимое таблицы, к которой подключен график. Вместо этого в полях сетки содержится техническое описание каждой из точек графика: цвет (**Color**), значение `X`, значение `Y` и метка (**Label**). И, что очень важно, все перечисленные колонки доступны для редактирования.

Резюме

Приложения БД предназначены не только для организации доступа к данным предприятия. Современный программный продукт должен предоставлять пользователю удобный и наглядный способ анализа этих данных.

Компоненты-диаграммы окажут неоценимую помощь при построении различного рода аналитических приложений, представив данные в визуальной форме в виде разнообразных графиков. Программист обладает широкими возможностями по определению способа вывода данных на экран, настройке объемного вида, масштабирования и анимации графиков.

Отчеты в Microsoft Office

Одним из существенных достижений программистов Microsoft по праву считается создание **многокомпонентной модели объектов** (Component Object Model, COM). COM — это спецификация, она описывает способ взаимодействия программ любого типа, программа-сервер (сервер COM) предоставляет в наше распоряжение собственные службы, программа-клиент (клиент COM) пользуется услугами доступных служб. Взаимодействующие COM-объекты могут функционировать как в рамках одного и того же процесса, так и в разных процессах, в том числе протекающих на разных машинах под управлением различных операционных систем.

У идей COM много наследников, один из них — технология *автоматизации* (automation). Автоматизация (как впрочем, и COM) нацелена на предоставление сервиса от одних приложений к другим. Все нововведения автоматизации неразрывно связаны с внедрением в технологию COM двух элементов:

- интерфейса `IDispatch`, умеющего вызывать методы сервера автоматизации по их имени;
- механизма *позднего связывания* (late binding). Благодаря позднему связыванию соединение между методом интерфейса объекта COM и контроллером автоматизации устанавливается не во время проектирования, а в момент выполнения приложения.

В итоге автоматизация позволила обращаться к методам сервера автоматизации по их имени, а не только при посредничестве указателей, как требует классика работы, COM-объектами, и процесс программирования существенно упростился.

Автоматизацию поддерживает подавляющее большинство приложений из обоймы программного обеспечения Microsoft. В первую очередь это все приложения из состава Microsoft Office и Internet Explorer. Не обошли стороной автоматизацию и такие гранды индустрии программного обеспечения, как Corel, Adobe и многие другие производители.

Если вы еще никогда не сталкивались с разработкой приложений на основе технологии автоматизации, то на первых порах приемы программирования вам покажутся весьма необычными. Допустим, что приложению требуется получить информацию о составе таблиц базы данных Microsoft Access. Пока не спешите искать компоненты-наборы данных со странички **dbGo** палитры Delphi. Для такого зада-

ния имеется и более элегантное решение. Процедура, приведенная в листинге 1, предоставит нам список List всех таблиц, принадлежащих БД из файла БД с именем DBName.

Листинг 1. Демонстрация автоматизации в Access

```
uses comobj;
…
procedure TForm1.GetAccessDBTables(DBName: string; List: TStrings);
var A:OleVariant;
    i:integer;
begin
  try
    A:=CreateOleObject('Access.Application');{создаем сервер}
    A.OpenCurrentDatabase(DBName); //вызов метода сервера
    for i:=0 to A.CurrentData.AllTables.Count - 1 do
          List.Add(A.CurrentData.AllTables[i].Name);
 finally
   A:=Unassigned; //отключение от сервера автоматизации
 end;
end;
```

В самом начале листинга создается сервер автоматизации — приложение Microsoft Access. Для этого в функцию

```
function CreateOleObject(const ClassName: string): IDispatch;
```

направляется обычный текстовый идентификатор сервера. В ответ функция создает запрашиваемый нами объект и возвращает в программу интерфейс сервера автоматизации. В нашем примере результат работы функции передается в переменную A. Дальше начинается самое необычное. Вполне понятно, что у переменной, пусть даже универсальной OleVariant, не может быть ни свойств, ни методов. Однако, буквально в следующей строке кода, мы обращаемся к функции OpenCurrentDatabase(), так, будто этот метод открытия базы данных Access всегда принадлежал переменной A. И, что самое удивительное, не встречаем никаких возражений со стороны Delphi! Почему так происходит?

Поняв, что переменная A предназначена для взаимодействия с сервером автоматизации, компилятор Delphi не превращает вызов методов в обычный бинарный код, а оставляет в коде исполняемого файла текстовое описание, содержащее названия функций и их аргументов. В этом несложно убедиться, открыв откомпилированный EXE-файл, содержащий листинг 1, в любом текстовом редакторе. Вы без труда найдете названия методов автоматизации (рис. 1). Когда приложение запускается на выполнение, сохраненные строки просто передаются интерфейсу IDispatch, именно он (а не компилятор Delphi) проведет синтаксический разбор команд и запустит их на выполнение.

Замечание

Способ работы с COM-объектом, когда связывание имен свойств и методов объекта с их кодом осуществляется не на этапе компиляции, а при выполнении программы, называют *поздним связыванием* (late binding). В отличие от раннего связывания, здесь отпадает необходимость в применении библиотеки типов.

```
00074D20: FF FF FF FF 12 00 00 00|41 00 63 00 63 00 65 00 | яяяя....A.c.c.e.
00074D30: 73 00 73 00 2E 00 41 00|70 00 70 00 6C 00 69 00 | s.s...A.p.p.l.i.
00074D40: 63 00 61 00 74 00 69 00|6F 00 6E 00 00 00 00 00 | c.a.t.i.o.n.....
00074D50: 01 01 00 08 4F 70 65 6E|43 75 72 72 65 6E 74 44 | ....OpenCurrentD
00074D60: 61 74 61 62 61 73 65 00|01 00 00 43 6F 75 6E 74 | atabase....Count
00074D70: 00 00 00 00 01 00 00 41|6C 6C 54 61 62 6C 65 73 | .......AllTables
00074D80: 00 00 00 00 01 00 00 43|75 72 72 65 6E 74 44 61 | .......CurrentDa
00074D90: 74 61 00 00 01 00 00 4E|61 6D 65 00 02 01 00 03 | ta.....Name.....
00074DA0: 41 6C 6C 54 61 62 6C 65|73 00 00 00 55 8B EC 6A | AllTables...U<mj
```

Рис. 1. Фрагмент бинарного кода контроллера автоматизации с поздним связыванием

Замечание

Библиотека типов (Type library) представляет собой описание интерфейсов, методов, свойств и структур, имеющихся в распоряжении объектов COM. Библиотека компилируется в двоичный файл и может использоваться клиентским приложением, как некий эквивалент заголовочных файлов. Именно благодаря библиотеке типов реализуется механизм раннего связывания, позволяющий будущему клиенту COM узнать все об интерфейсах сервера еще на этапе компиляции.

Некая "отрешенность" компилятора Delphi от процесса проверки корректности кода программы имеет простое объяснение — Delphi ничего не знает о возможностях сервера автоматизации и, как говорится, верит программисту на слово. Можете проверить это утверждение и добавить в листинг 1 строку с любой нелепостью, скажем

```
A.HelloWorld;
```

и Delphi без всякого зазрения совести включит ее в исполняемый код. Все будет хорошо ровно до тех пор, пока мы не запустим проект на выполнение и не "поприветствуем" сервер автоматизации командой `HelloWorld`. Интерфейс `IDispatch` не имеет чувства юмора, поэтому, не найдя у сервера запрашиваемого метода, "с удовольствием" сгенерирует исключительную ситуацию `EOleError` (рис. 2). Следовательно, работая по законам технологии автоматизации, мы должны быть абсолютно уверены в каждом символе своего кода.

Перед изложением материала, посвященного работе с программами из состава пакета Microsoft Office, а точнее об использовании технологии автоматизации, реализованной в приложениях офиса, сделаю одно важное замечание относительно излагаемого далее материала. Синтаксис описываемых свойств и методов программ Microsoft Office является чем-то вроде моего перевода с языка Microsoft Visual Basic на язык Delphi и никоим образом не претендует на академичность.

Рис. 2. Сообщение о возникновении исключительной ситуации у сервера автоматизации

И еще одно примечание, касающееся специфики передачи значений параметров в процедуры и функции Basic. В методах Basic различают *обязательные* (required) и *необязательные* (optional) параметры. С обязательными параметрами все просто, они (как и в языке Delphi) должны всегда присутствовать при вызове метода. А вот работа с необязательными аргументами для программиста Delphi покажется необычной.

Предположим, что существует некая гипотетическая функция `InsertFile()`, предназначенная для вставки файла в документ Office:

```
Function InsertFile(FileName : string;
                    [Range : Variant;
                     ConfirmConversions, Link : Boolean;
                     Attachment : Variant]): Variant;
```

В этом методе первый параметр обязательный, а остальные четыре (они прячутся внутри квадратных скобок) — факультативные. Если программист к ним не обратился, то им присваиваются значения по умолчанию. В простейшей ситуации, вызов данного метода ничем не отличается от привычного для нас вызова метода в среде Delphi:

```
InsertFile('C:\File.txt');//передается только обязательный параметр
```

Но если мы хотим передать какие-то данные в необязательные параметры, то сталкиваемся с непривычным для программиста Delphi синтаксисом:

```
InsertFile(FileName:='C:\File.txt',ConfirmConversions:=False,Link:=True);
```

Внимание!

При обращении к факультативному параметру метода сначала указывается имя параметра и затем присваивается соответствующее значение, например `DemoFunction(Param1:=True)`.

Константы Microsoft Office

Во всех языках программирования широко используются именованные константы. Опытный программист Delphi, столкнувшись с константой `clRed`, сразу скажет, что она предназначена для описания красного цвета, константа `mrOk` уве-

домит, что пользователь нажал кнопку **OK** на окне диалога, константа MAX_PATH определяет максимальное число символов в пути к файлу. Констант очень много, в Delphi они хранятся в различных модулях, таких как Windows, Types, SysUtils и т. п. По необходимости разработчик подключает нужный модуль и получает возможность работать с объявленными в нем константами.

Объекты Microsoft Office также нуждаются в помощи многочисленных именованных констант, но беда в том, что они объявлены в программных модулях Office и не могут быть напрямую подключены к проектам Delphi. Выход из сложившегося положения — разработка собственных модулей на языке Delphi и описание в них констант из пакета Microsoft Office. Для этого достаточно создать отдельный файл с расширением .pas.

В листинге 2 приведен фрагмент файла wordtypes.pas, предназначенный для работы с Microsoft Word.

Листинг 2. Пример подключения констант MS Word к проекту Delphi

```
unit wordtypes;

interface

const
{WdSaveOptions - опции закрытия приложения}
  wdDoNotSaveChanges=0;
  wdSaveChanges=-1;
  wdPromptToSaveChanges=-2;

{WdOrientation - ориентация страницы документа}
 wdOrientPortrait = 0;
 wdOrientLandscape = 1;

{WdPaperSize - размеры страницы документа}
 wdPaper10x14 = 0;
 wdPaper11x17 = 1;
 wdPaperLetter = 2;
 wdPaperLetterSmall = 3;
//
end.
```

Единожды созданный файл с константами мы сможем использовать в дальнейшем во всех наших проектах, взаимодействующих с Microsoft Office.

Как узнать имена и значения констант? В первую очередь, импортировав библиотеку типов интересующего вас сервера автоматизации. Для этого воспользуйтесь пунктом меню **Component | Import component** (рис. 3).

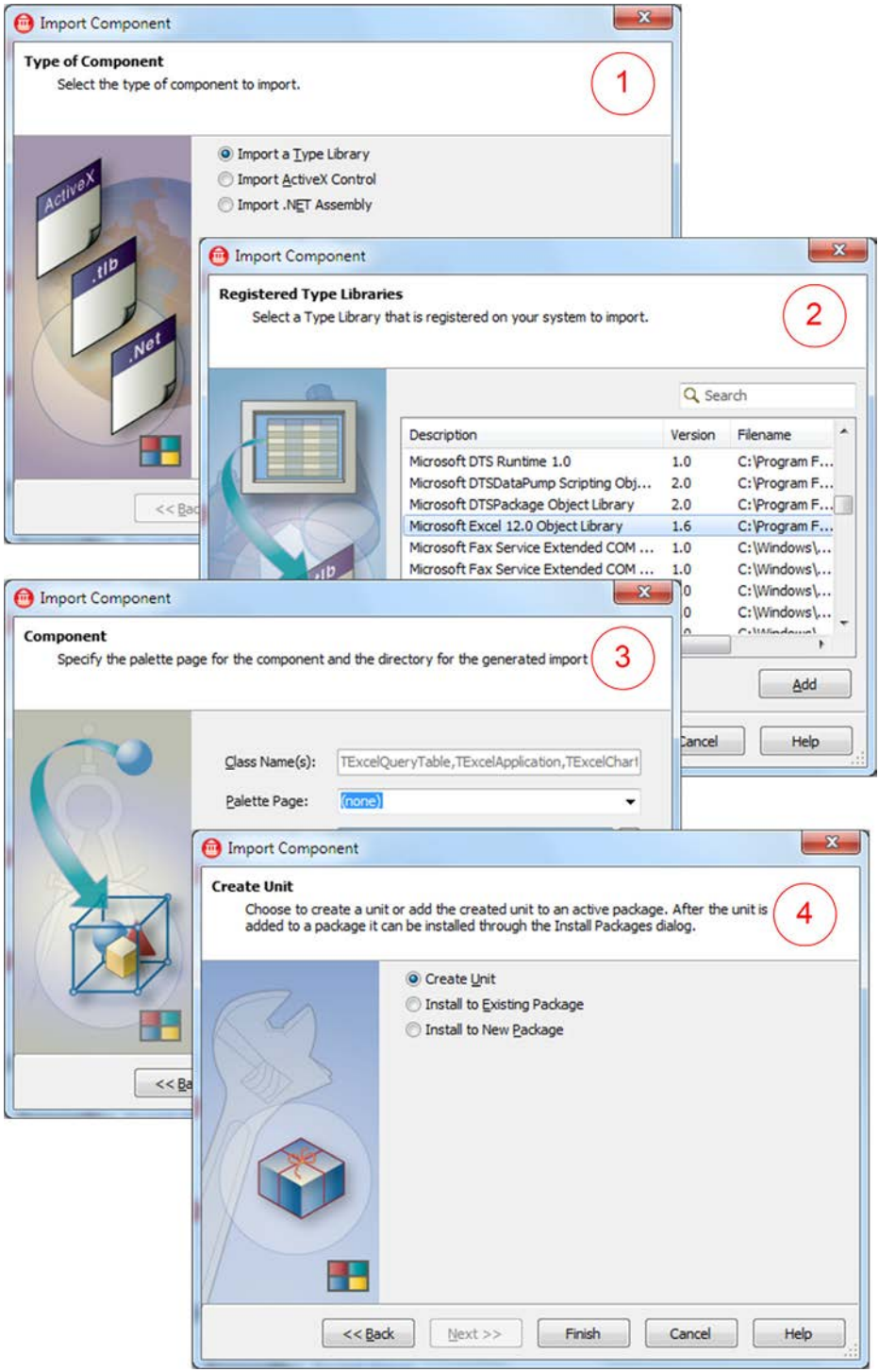


Рис. 3. Мастер импорта библиотеки типов

Нажатие на кнопку **Next** выведет на экран перечень зарегистрированных в системе библиотек типов. Разработчику предстоит выбрать интересующую его библиотеку. В следующем окне изменим только параметр **Unit Dir Name**, определяющий путь к папке, в которую импортируется файл с библиотекой. Укажите здесь путь к каталогу вашего проекта автоматизации и вновь нажмите кнопку **Next**. В заключительном окне предлагается выбрать форму импорта, в нашем случае предлагаю ограничиться пунктом **Create Unit**. Это действие укажет помощнику, что разработчик рассчитывает получить в свое распоряжение отдельный файл библиотеки типов с описанием интерфейсов сервера автоматизации в формате PAS.

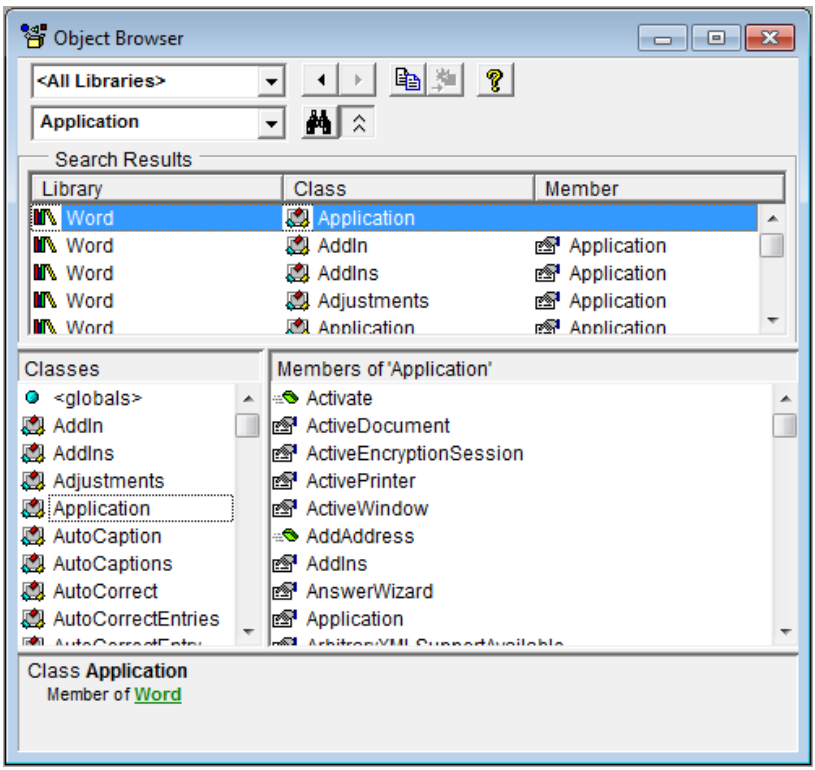


Рис. 4. Окно **Object Browser** редактора Microsoft Visual Basic с активным объектом `Word.Application`

Если вас интересуют только избранные имена констант, то предложу еще один удобный способ "шпионажа". Для его апробации откройте любую программу из пакета офиса и одновременно нажмите клавиши <Alt>+<F11>. В появившемся на экране редакторе языка Visual Basic меню выберите пункт **View | Object Browser** (Представление | Просмотр объектов) или просто нажмите клавишу <F2>. В результате вы увидите окно **Object Browser**, в котором можно исследовать всю объектную модель приложения, в том числе имена и значения интересующих нас констант (рис. 4).

Обращение к серверу автоматизации

Для того чтобы проект Delphi приобрел возможность взаимодействия с программами Microsoft Office (и любыми другими приложениями, поддерживающими технологию Automation), первое, что нужно сделать, упомянуть модуль `ComObj` в строке `uses` проекта. Данный модуль понадобится нам ради обращения к функции `CreateOleObject()`, реализующей экземпляр объекта автоматизации. Единственный параметр функции — название класса создаваемого объекта. Например, идентификатором приложения Excel выступает строка `Excel.Application`, Visio — `Visio.Application`, Word — `Word.Application` и. т. д.

В листинге 3 приведен пример инициализации приложения Word.

Листинг 3. Запуск сервера автоматизации MS Word

```
var Word_App : OleVariant;
begin
  Word_App:= CreateOleObject('Word.Application');
try
//операции с объектом автоматизации
Finally
  Word_App:= Unassigned;//освобождаем переменную
End;
```

Строка `Word_App:=Unassigned` освобождает переменную типа `OleVariant`. Впрочем, это необязательная операция, т. к. за уничтожение экземпляра объекта отвечает сам объект. На мой взгляд, это весьма удобно. Объект автоматизации проверяет, сколько раз он был вызван, и если число ссылок равно нулю, "самоотверженно" ликвидируется.

Иногда, вместо инициализации нового экземпляра объекта автоматизации, оказывается проще подключиться к уже существующему объекту (запущенному ранее пользователем или другими "потусторонними" силами). Для решения задач подобного рода обращайтесь к методу `GetActiveOleObject` (листинг 4).

Листинг 4. Подключение к исполняющемуся серверу автоматизации MS Excel

```
var Excel_App : OleVariant;
begin
 try
   try
   Excel_App:=GetActiveOleObject('Excel.Application');
   {работа с сервером MS Excel}
   Finally
     Excel_App:= Unassigned;//освобождает переменную
   End;
```

```
  except
    MessageBoxEx(Application.Handle,
              PChar('Нет активного экземпляра EXCEL'),
              PChar('Ошибка'),
              MB_ICONERROR,
              Lang_Russian);
 end;
end;
```

Будьте внимательны, не исключено, что в памяти компьютера не окажется экземпляра вызываемого объекта. В нашем примере при таком неприятном стечении обстоятельств просто появляется сообщение об отсутствии требуемого объекта.

Для того чтобы код приобрел максимальную гибкость, предлагаю разработать универсальную функцию, способную вернуть интерфейс любого из зарегистрированных в системе серверов автоматизации. В первый параметр передаем текстовый идентификатор приложения из пакета Microsoft Office, второй параметр функции призван возвратить ссылку на экземпляр сервера (листинг 5).

Листинг 5. Универсальная функция доступа к объекту автоматизации

```
function GetAutomationServer(ObjectID: String;
                             var App:OleVariant):Boolean;
begin
Result:=True;
 try
   App:=GetActiveOleObject(ObjectID);
  except
    try
    App:=CreateOleObject(ObjectID);
     except
      Result:=false;
      MessageBoxEx(Application.Handle,
      PChar('В системе отсутствует сервер автоматизации '+ ObjectID+'!'),
      PChar('Ошибка'),
      MB_ICONERROR,
      Lang_Russian);
     end;
     end;
 end;
```

Если сервер уже запущен на выполнение, то функция постарается подключиться к активному экземпляру. Если активный экземпляр отсутствует, пытаемся создать свой собственный. В случае успеха функция возвращает ссылку на экземпляр сервера и значение `true`, иначе — функция вернет `false`.

Понятие коллекции

Зачастую входящие в объектную модель объекты объединяются в коллекции объектов. Основная задача коллекций — хранение и предоставление доступа к группе однотипных объектов. Например, рабочая книга Excel содержит коллекцию листов, документ Word — коллекции абзацев, таблиц, списков. Для простоты запоминания все коллекции из состава Microsoft Office вооружены общими свойствами и методами. В первую очередь коллекция всегда владеет информацией, какому приложению она принадлежит:

```
Property Application : OleVariant;
```

Общее число элементов в коллекции мы выясним из свойства

```
Property Count : Integer;
```

Для пополнения коллекции новым объектом вызывается метод

```
Function Add([Параметр(ы) функции]) : OleVariant;
```

Для обращения к конкретному элементу коллекции потребуется функция

```
Function Item(Index : Integer): OleVariant
```

Здесь Index — порядковый номер элемента в массиве.

ВНИМАНИЕ!

Будьте внимательны, отсчет элементов в коллекции элементов Microsoft Office начинается не с нуля, как мы привыкли в программах Delphi, а с единицы.

Текстовый процессор Microsoft Word

Приложение Microsoft Word обычные пользователи часто называют текстовым редактором. Какое неслыханное оскорбление! Уже давным-давно Word вырос из штанишек банального редактора. Word умеет создавать широчайший спектр разноплановых документов, работает в Интернете, отправляет почту, взаимодействует со всеми приложениями из пакета Microsoft Office, обладает мощным, и в то же время простым интегрированным языком программирования, и самое главное — Word способен выступать в роли сервера автоматизации.

Я больше чем уверен, что вы уже имеете опыт работы с редактором Word и, по крайней мере, набирали в нем текст. Сегодня мы познакомимся с другой стороной текстового процессора — с возможностью его применения в качестве универсального генератора отчетов в наших проектах баз данных. А для этого нам необходимо разобраться с иерархией объектов и коллекцией Microsoft Word (рис. 5).

На рис. 5 приведен лишь фрагмент внушительной объектной модели Microsoft Word. На вершине айсберга расположен самый главный объект — приложение (Application). Названный объект имеется у всех основных программ из пакета Microsoft Office. Именно благодаря Application мы получаем доступ ко всем сервисным возможностям сервера автоматизации. С точки зрения потребителя услуг текстового процессора нас с вами больше всего интересует документ Word, он ин-

капсулируется в объекте `Document`. Для доступа к отдельному документу следует познакомиться с коллекцией документов `Documents`. Документ Word представляет собой совокупность множества коллекций и объектов. Например, абзацы текста доступны благодаря коллекции абзацев `Paragraphs`, таблицы — коллекции `Tables`, параметры страницы документа настраиваются с помощью объекта `PageSetup`. Воздействуя на свойства и методы объектов текстового процессора Word, мы создадим новый или редактируем существующий документ.

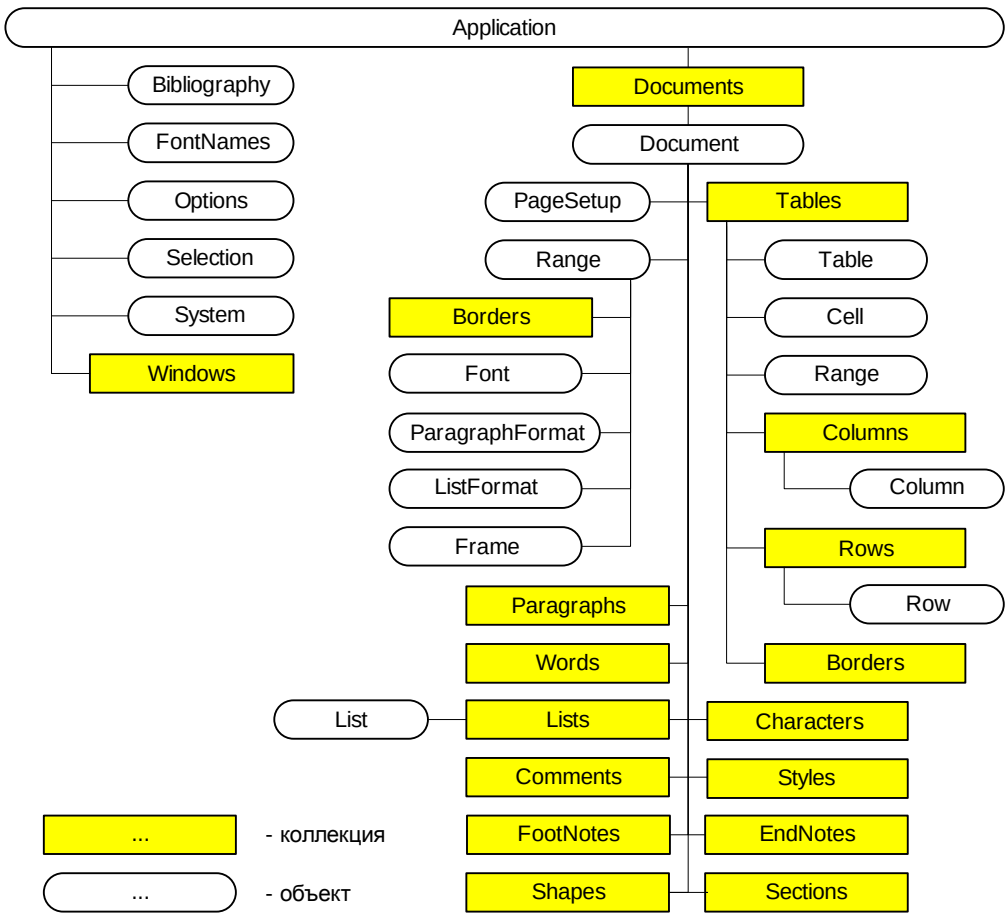


Рис. 5. Основные элементы объектной модели Microsoft Word

Приложение Word — *Application*

Для получения доступа к текстовому процессору Word понадобится объект `Application` (строковый идентификатор `Word.Application`). Простейшее, и в то же время самое главное, что мы можем сделать с экземпляром приложения

Microsoft Word, — отобразить его на экране. Для показа или скрытия приложения используйте свойство

```
Property Visible : Boolean;
```

Основным свойством приложения можно считать коллекцию всех документов

```
Property Documents : OleVariant;
```

Простейший способ завершения работы с приложением — вызов метода

```
Procedure Quit([SaveChanges:WdSaveOptions]; [OriginalFormat: WdOriginalFor-
mat]);
```

Все присутствующие в процедуре параметры опциональные, поэтому в простейшем случае метод может вызываться без аргументов. Первый параметр `SaveChanges` определяет реакцию приложения на закрытие в ситуации, когда какой-то из документов не сохранен. Параметр может принимать три значения:

```
const wdDoNotSaveChanges=0;      //не сохранять изменения
      wdSaveChanges=-1;          //сохранить изменения
      wdPromptToSaveChanges=-2; //запросить о сохранении изменений
```

Второй параметр уточняет формат сохраняемого документа:

```
const wdWordDocument=0;            //формат документа Word
      wdOriginalDocumentFormat=1; //не изменять текущий формат
      wdPromptUser=2;              //запросить пользователя
```

Коллекция документов *Documents* и документ *Document*

Основное назначение текстового процессора Microsoft Word заключается в предоставлении нам возможности создавать и редактировать текстовые документы. Текстовый документ описывается объектом `Document`. В процессоре Word может одновременно сосуществовать несколько документов. Ссылки на них находятся в списке документов — коллекции `Documents`, а доступ к коллекции мы получим из приложения `Application`.

Сначала поговорим о коллекции документов. Для открытия файла с документом, совместимым с Word, следует обратиться к методу коллекции

```
Function Open(S:string):OleVariant;
```

Процедура

```
Procedure Save;
```

сохранит все документы коллекции. Если в коллекции есть новый документ, до сего момента еще не сохраненный на диске, то для него будет вызван стандартный диалог **Сохранить как...** и пользователю будет предложено указать имя файла.

Для того чтобы закрыть все документы, вызываем процедуру

```
Procedure Close(SaveChanges:WdSaveOptions;
                OriginalFormat: WdOriginalFormat);
```

Напомню, что у всех коллекций из состава Microsoft Office имеются одноименные методы. Метод `Add()`создает новый документ и одновременно добавляет его в

состав коллекции Documents, метод Item()обеспечивает доступ к отдельному документу, свойство Count подскажет число документов в коллекции.

Пример управления коллекцией документов приведен в листинге 6.

Листинг 6. Пример управления коллекцией документов

```
var Word_App,Documents,Document : OleVariant;
    i:integer;
begin
if GetAutomationServer('Word.Application',Word_App)=false then exit;
  TRY
   Documents:=Word_App.Documents;{доступ к коллекции документов}
   if FileExists('c:\test.docx') then {открываем файл}
      Document:=Documents.Open('c:\test.docx');
    Document:=Documents.ADD;{добавляем в коллекцию новый документ}
      {в последнем документе выводим перечень открытых документов}
    for i:=1 to Documents.Count do
             Document.Range.InsertAfter(Documents.Item(i).Name+#13);

    Word_App.Visible:=True; //отображаем приложение на экране
   FINALLY
     Word_App:= Unassigned;
   END;
end;
```

Закончим разговор о коллекции и перенесем внимание на отдельный документ Document. Мы уже знаем, что для обращения к документу требуется указать его индекс в коллекции документов приложения. Например, для того чтобы получить доступ к последнему документу, нужно написать следующую строку кода

```
Document:=Documents.Item(Documents.Count);
```

У документа есть имя, оно хранится в свойстве

```
Property Name : string;
```

По умолчанию имена документа и файла документа совпадают. Кроме того, мы в состоянии выяснить полное имя (включающее путь к файлу на диске):

```
Property FullName : string; //только для чтения
```

В коллекции документов одновременно может быть активным лишь один из них. Для того чтобы активизировать документ, вам пригодится метод

```
Procedure Activate;
```

Для закрытия документа потребуется помощь метода

```
Procedure Close([SaveChanges:WdSaveOptions];
                [OriginalFormat: WdOriginalFormat]);
```

Обратите внимание, что параметры процедуры нам уже хорошо знакомы, т. к. встречались в одноименном методе коллекции документов `Documents` и в методе `Quit()` приложения `Application`.

Приведенной информации уже вполне достаточно для организации поиска в коллекции нужного нам документа (листинг 7).

Листинг 7. Поиск документа в коллекции

```
for i:=1 to Documents.Count do
 if Documents.Item(i).Name='test.docx' then {ищем документ 'test.docx' }
   begin
     {если документ обнаружен}
     Document:=Documents.Item(i);
     Document.Close(SaveChanges:=wdSaveChanges); //закрываем документ
     Break;
   end;
```

Каждый документ знает, какому приложению он принадлежит, для этого предназначено свойство

```
Property Application : OleVariant;
```

Документ Word обладает рядом свойств, которые предоставляют доступ к основным коллекциям и объектам (табл. 1).

Таблица 1. *Основные свойства документа* `Document`

Свойство	Описание
`Property Paragraphs : OleVariant;`	Коллекция абзацев документа
`Property Tables : OleVariant;`	Все таблицы документа
`Property Lists : OleVariant;`	Списки документа
`Property Footnotes: OleVariant;`	Сноски
`Property Endnotes: OleVariant;`	Примечания
`Property Range : OleVariant;`	Область документа.
`Property PageSetup : OleVariant;`	Установки страницы (размеры, ориентация, поля)

Для систематизации большого числа документов очень полезным окажется свойство

```
Property BuiltinDocumentProperties : OleVariant;
```

которое позволяет читать и редактировать служебные пометки документа Microsoft Word.

Для этого нужно обратиться к свойству Value коллекции, указав, какие служебные пометки документа нас заинтересовали. Например:

```
Document.BuiltinDocumentProperties[wdPropertyTitle].Value:='Документ1';
Document.BuiltinDocumentProperties[wdPropertyAuthor].Value:='Иванов.И.';
```

В листинге 8 приведен перечень констант, позволяющих обратиться к информационным свойствам документа.

Листинг 8. Перечень констант информационных свойств документа

```
{wdPropertyWords — информационные свойства документа}
const wdPropertyTitle=1;            //название документа
      wdPropertySubject=2;          //принадлежность
      wdPropertyAuthor=3;           //автор
      wdPropertyKeywords=4;         //ключевые слова
      wdPropertyComments=5;         //комментарии
      wdPropertyTemplate=6;         //шаблон
       wdPropertyLastAuthor=7;     //автор последней редакции
      wdPropertyRevision=8;         //номер версии
      wdPropertyAppName=9;          //имя приложения
      wdPropertyTimeLastPrinted=10; //дата последней печати
      wdPropertyTimeCreated=11;    //дата создания
      wdPropertyTimeLastSaved=12;  //дата последнего сохранения
      wdPropertyVBATotalEdit=13;   //суммарное время редактирования
      wdPropertyPages=14;          //число страниц
      wdPropertyWords=15;          //число слов
      wdPropertyCharacters=16;     //число символов
      wdPropertySecurity=17;       //права доступа
      wdPropertyCategory=18;       //категория
      wdPropertyFormat=19;         //формат
      wdPropertyManager=20;        //менеджер
      wdPropertyCompany=21;        //компания
      wdPropertyBytes=22;          //число в байтах
      wdPropertyLines=23;          //число строк
      wdPropertyParas=24;          //число абзацев
      wdPropertySlides=25;         //число слайдов
      wdPropertyNotes=26;          //количество комментариев
      wdPropertyHiddenSlides=27;   //скрытых слайдов
      wdPropertyMMClips=28;        //мультимедиа
      wdPropertyHyperlinkBase=29;  //гиперссылки
      wdPropertyCharsWSpaces=30;   //число символов без пробелов
```

Параметры страницы, объект *PageSetup*

Параметрами страницы ведает объект PageSetup. Чаще всего программист сталкивается с задачей изменения размера страницы. Для этого предназначено свойство

```
Property PaperSize : WdPaperSize; // тип данных wdPaperSize
```

Свойство может принимать любое значение из диапазона от 0 до 41, наиболее часто используемые величины приведены в листинге 9.

Листинг 9. Перечень констант размера страницы

```
type wdPaper10x14 = 0; {WdPaperSize - размеры страницы}
     //…
     wdPaperA3 = 6;          // A3 (297 x 420)мм
     wdPaperA4 = 7;          // A4 (210 x 297)мм
     wdPaperA4Small = 8;     // A4 уменьшенный
     wdPaperA5 = 9;          // A5 (148 x 210)мм
     //…
     wdPaperCustom = 41;     // Пользовательский формат
```

Если предопределенные (стандартные) размеры листа нам не по душе, то обращаемся к паре свойств:

```
Property PageHeight : cardinal;
Property PageWidth : cardinal;
```

Размер задается в соответствии с текущей системой измерений (миллиметры, дюймы). После внесения любых изменений в свойства PageHeight и PageWidth процессор Word переводит свойство PaperSize в состояние 41 (константа wdPaperCustom) — размеры пользователя.

Ориентацию страницы задает свойство

```
Property Orientation : integer;
```

Вариантов немного, имеется всего два допустимых значения: wdOrientPortrait=0 — портретная ориентация и wdOrientLandscape=1 — альбомная ориентация.

Пример:

```
Document.PageSetup.PaperSize:= wdPaperA4;
Document.PageSetup.Orientation:= wdOrientLandscape;
```

Поля страницы (расстояние от края листа до начала текста) задаются свойствами

```
Property LeftMargin : cardinal;    //левое поле
Property RightMargin : cardinal;   //правое поле
Property TopMargin : cardinal;     //верхнее поле
Property BottomMargin : cardinal;  //нижнее поле
```

В качестве единицы измерения применяется условная точка, в одном дюйме (2,54 см) — 72 точки.

При формировании документа с зеркальными полями установите в true свойство

```
Property MirrorMargins : Boolean;
```

Расстояния от нижнего края листа до нижнего колонтитула и от верхнего среза листа до верхнего колонтитула назначает пара свойств

```
Property FooterDistance : single; //расстояние до нижнего колонтитула
Property HeaderDistance : single; //расстояние до верхнего колонтитула
```

Объект настройки страницы документа способен также определить некоторые особенности колонтитулов страницы:

```
Property DifferentFirstPageHeaderFooter : Boolean; {различать колонтитул первой страницы}
Property OddAndEvenPagesHeaderFooter : Boolean;{различать колонтитулы четной и нечетной страницы}
```

Объект PageSetup обеспечивает доступ к текстовым колонкам документа — коллекции TextColumns. Теперь мы можем изменить число колонок в документе (по умолчанию создается документ, форматирующий текст в одну колонку). Для этого вызовем метод коллекции TextColumns:

```
Procedure SetCount(NumColumns:cardinal);
```

Для форматирования документа в две колонки, разделенные линией, можно написать следующие строки кода:

```
Document.PageSetup.TextColumns.SetCount(2);       //две колонки
Document.PageSetup.TextColumns.LineBetween:=True; //разделенные линией
```

Область документа *Range*

Объект Range предоставляет доступ к определенной области документа, в которую мы имеем право вставлять текст и редактировать его. Для того чтобы добраться до объекта Range, обратитесь к одноименному свойству документа (см. рис. 1). Кроме того, свойством Range снабжен еще ряд объектов документа (например, абзац Paragraph и таблица Table).

В область документа могут входить один или несколько абзацев документа, несколько слов, отдельное слово, отдельный символ. Для определения диапазона необходимо указать его начало и окончание (обратите внимание на то, что параметры Start и End имеют тип Variant):

```
Function Range(Start, End : Variant) : Range;
```

Пример работы с областью приведен в листинге 10.

Листинг 10. Работа с областью Range

```
var Document, Range : OleVariant;
    …
    Document:=Documents.Open('c:\test.docx');//открываем документ
    Range:=Document.Range(20,50);            //выбираем область
    Range.Font.Size:=30;                     //изменяем размер шрифта
```

В примере открывается документ, выделяется область с 20-го по 50-й символ и изменяется размер шрифта для этой области.

Для выделения целых абзацев применяется немного другая конструкция:

```
Range:= Document.Paragraphs.Item(1).Range;
```

Здесь мы обратились к самому первому абзацу документа.

Для того чтобы включить в область несколько абзацев, предлагаю повторить следующую строку кода:

```
Range:=Document.Range(Document.Paragraphs.Item(1).Range.Start,
                      Document.Paragraphs.Item(3).Range.End);
```

Пример позволит объединить в диапазон три абзаца документа — с первого по третий.

Для того чтобы узнать размеры области, описываемой объектом `Range`, воспользуйтесь свойствами области `Start` и `End`:

```
Property Start : Integer;
Property End : Integer;
```

Это индексы текстового символа начала и окончания области.

Метод

```
Procedure Select;
```

позволяет выделить данные, входящие в область.

Текст, содержащийся в области, доступен из свойства

```
Property Text : string;
```

Для того чтобы удалить все, находящееся в области, вызовите метод `Delete()`:

```
Procedure Delete;
```

Для вставки текста в область можете обратиться к методам

```
Procedure InsertBefore(Text : string);
Procedure InsertAfter(Text : string);
```

Первый метод вставляет текст перед областью, второй — после области (листинг 11).

Листинг 11. Вставка текста в область Range

```
Document:=Documents.Add;
   Range:=Document.Range;
   Range.InsertBefore('Hello, ');
   Range.InsertAfter('World!');
```

Ряд методов области предназначен для вставки нового абзаца в указанное место.

Метод

```
Procedure InsertParagraph;
```

заменяет указанную область пустым абзацем.

Процедуры

```
Procedure InsertParagraphBefore;
Procedure InsertParagraphAfter;
```

отвечают за вставку нового абзаца соответственно до, и после области `Range`.

Для вставки це*лого текстового файла воспользуйтесь методом

```
Function InsertFile(FileName : string;
                    [ConfirmConversions, Link : Boolean]) : OleVariant;
```

Здесь FileName — имя вставляемого файла — обязательный аргумент, остальные аргументы необязательны. Параметр ConfirmConversions, установленный в false, отказывается от подтверждения о преобразовании файла при открытии, Link — определяет, внедрять документ (false) в файл или просто установить с ним связь (true). Минимальный синтаксис требует передачи одного параметра — имени файла:

```
Document.Range.InsertFile(Filename:='c:\Test.txt',
                          ConfirmConversions:=false);
```

Если потребуется разорвать страницу, колонку, раздел, используйте метод

```
Procedure InsertBreak(Type : integer)
```

Здесь Type — константа wdBreakType. Для вставки новой страницы — wdPageBreak (значение 7), начало нового раздела со следующей страницы — wdSectionBreakNextPage (2), разрыв колонки — wdColumnBreak(8).

Манипулируя свойствами диапазона, мы получаем возможность обращаться к коллекциям объектов или отдельным объектам, находящимся внутри данного диапазона. Их базовый перечень приведен в табл. 2.

Таблица 2. *Основные объекты и коллекции, доступные из* Range

Свойство	Описание
`Property Tables : OleVariant;`	Коллекция таблиц Tables внутри диапазона
`Property Columns : OleVariant;`	Коллекция колонок Columns
`Property Rows : OleVariant;`	Коллекция рядов Rows
`Property Cells : OleVariant;`	Коллекция ячеек Cells
`Property Bookmarks : OleVariant;`	Коллекция закладок Bookmarks
`Property ListFormat : OleVariant;`	Объект ListFormat, обеспечивающий форматирование списков
`Property Comments : OleVariant;`	Коллекция комментариев Comments
`Property Endnotes : OleVariant;`	Коллекция примечаний Endnotes
`Property Footnotes : OleVariant;`	Коллекция сносок Footnotes
`Property Paragraphs : OleVariant;`	Коллекция абзацев Paragraphs
`Property ShapeRange: OleVariant;`	Доступ к объектам Shape области
`Property Words : OleVariant;`	Коллекция слов Words

Объект выделения *Selection*

Объект Selection инкапсулирует выделенную часть документа, доступ к выделению обеспечивается напрямую из приложения Application. Выделение может быть проведено как пользователем, так и программно. Наиболее простой программный способ создания выделения основан на использовании метода Select() объекта Range (листинг 12).

Листинг 12. Получение доступа к выделенной области

```
var Word_App, Document,Range,Selection : OleVariant;
    …
    Range:=Document.Range(Document.Paragraphs.Item(2).Range.Start,
                          Document.Paragraphs.Item(3).Range.End);
    Range.Select;
    Selection:=Word_App.Selection;
```

В листинге 12 мы выделили второй и третий абзацы документа.

Доступ к тексту выделения Selection обеспечивает свойство

```
Property Text:String;
```

Для замены выделенного текста новой строкой, применим код из листинга 13.

Листинг 13. Пример замены выделенного текста

```
Selection:=Word_App.Selection;
Word_App.Options.ReplaceSelection:=True;
Selection.Text:='Замена выделенного текста';
```

С помощью объекта Selection можно применять к выделению заранее предопределенные стили форматирования. Для этого предназначено свойство

```
Property Style: WdBuiltinStyle;
```

Предопределенных стилей форматирования очень много, поэтому мы ограничимся представлением лишь некоторых из них (листинг 14).

Листинг 14. Некоторые из предопределенных стилей форматирования

```
{WdBuiltinStyle - стили форматирования}
const wdStyleNormal=-1;   //нормальный
      wdStyleHeading1=-2; //заголовок 1-го уровня
      wdStyleHeading2=-3; //заголовок 2-го уровня
      …
      wdStyleList=-48;     //список
      …
      wdStyleDate=-77;     //дата
```

Порядок применения форматирования заголовка 1-го уровня к выделению приведен в листинге 15.

Листинг 15. Форматирование выбранной области

```
Selection:=Word_App.Selection;
Selection.Style:=wdStyleHeading1;
```

Ряд процедур предназначен для снятия форматирования выделенной области документа:

```
Procedure ClearFormatting;              //снятие всего форматирования
Procedure ClearCharacterAllFormatting; //снятие форматирования шрифта
Procedure ClearParagraphAllFormatting; //снятие форматирования абзаца
```

Выделение можно скопировать, вырезать или заменить данными из буфера обмена:

```
Procedure Copy;
Procedure Cut;
Procedure Paste;
```

Коллекция абзацев *Paragraphs* и абзац документа *Paragraph*

Коллекция `Paragraphs` организует доступ к перечню всех абзацев документа. Абзац `Paragraph` значительно упрощает форматирование текста. За выравнивание текста в абзаце отвечает свойство

```
Property Alignment : wdParagraphAlignment;
```

которое может принимать значения в диапазоне от 0 до 8. Названия констант, обозначающих наиболее востребованные способы выравнивания, приведены в листинге 16.

Листинг 16. Константы выравнивания

```
const wdAlignParagraphLeft=0;    //Влево
      wdAlignParagraphCenter=1;  //По центру
      wdAlignParagraphRight=2;   //Вправо
      wdAlignParagraphJustify=3; //По ширине
```

Междустрочный интервал в абзаце назначается множителем. Значения междустрочного интервала устанавливаются в свойстве

```
Property LineSpacingRule : wdLineSpacing;
```

Допустимый диапазон значений от 0 до 5:

```
const wdLineSpaceSingle=0; //Одинарный
wdLineSpace1pt5=1;     //Полуторный
wdLineSpaceDouble=2;   //Двойной
wdLineSpaceAtLeast=3; //Минимум
wdLineSpaceExactly=4; //Точно
wdLineSpaceMultiple=5;//Множитель
```

Если вас не удовлетворяют предопределенные значения междустрочного интервала, то воспользуйтесь свойством

```
Property LineSpacing: Single;
```

и укажите приемлемый размер. При этом свойство `LineSpacingRule` автоматически примет значение 5 (`wdLineSpaceMultiple` — множитель).

Свойства

```
Property LeftIndent : single;
Property RightIndent : single;
```

определяют отступы текста слева и справа.

Отступ красной строки задает свойство

```
Property FirstLineIndent : single;
```

Для того чтобы заключить абзац в рамку, обратитесь к объекту `Border` и включите его:

```
Property Border : OleVariant;
```

Пример форматирования абзаца приведен в листинге 17.

Листинг 17. Форматируем абзац

```
Paragraph:=Document.Paragraphs.Item(1);     //выбор абзаца
Paragraph.Alignment:= wdAlignParagraphRight;// выравнивание вправо
Paragraph.LineSpacingRule:= wdLineSpace1pt5;//межстрочный интервал
Paragraph. LeftIndent:=5; //отступ слева
```

Коллекция списков *Lists* и список *List*

Форматирование текста в виде списка, как правило, применяется для выделения однотипных перечислений (скажем список покупок, за которыми вас пытается отправить супруга как раз во время трансляции чемпионата Европы по футболу). В Microsoft Word различают три основных типа списков:

1. Маркированный.
2. Нумерованный.
3. Многоуровневый.

Для того чтобы программным образом преобразовать несколько абзацев в список, потребуется сделать следующие шаги:

- Объединить нужные абзацы в диапазон.
- Обратиться к свойству диапазона `ListFormat`.
- Применить к объекту `ListFormat` метод `ApplyBulletDefault()`.

Рассмотрите код, приведенный в листинге 18.

Листинг 18. Демонстрация работы со списками

```
var Word_App,Documents,Document,Paragraph,Range,ListFormat : OleVariant;
begin
if GetAutomationServer('Word.Application',Word_App)=false then exit;
 try
  Document:=Word_App.Documents.ADD;
  Paragraph:=Document.Paragraphs.Add;
  Paragraph.Range.InsertBefore('Объединить в диапазон нужные абзацы');
  Paragraph:=Document.Paragraphs.Add;
  Paragraph.Range.InsertBefore('
                          Обратиться к свойству диапазона ListFormat');
  Paragraph:=Document.Paragraphs.Add;
  Paragraph.Range.InsertBefore('Применить к объекту ListFormat метод
                               ApplyBulletDefault');
  {Включаем в область три абзаца}
   Range:=Document.Range(Document.Paragraphs.Item(1).Range.Start,
                         Document.Paragraphs.Item(3).Range.End);
  {С помощью свойства диапазона ListFormat преобразуем абзацы к списку}
   ListFormat:= Range.ListFormat;
   ListFormat.ApplyBulletDefault;
    Word_App.Visible:=True;
   Finally
     Word_App:= Unassigned;
   End;
End;
```

Создав документ, мы вставили в него три абзаца (по одной строке в каждом). Затем эти абзацы объединили в область `Range`. Далее при помощи свойства `ListFormat`, определили способ форматирования абзацев в список.

Форматирование списка

Объект `ListFormat` обеспечивает форматирование списка. В нем заложено три метода форматирования по умолчанию. Одним из них мы уже воспользовались в предыдущем примере:

```
Procedure ApplyBulletDefault([DefaultListBehavior : Variant]);
```

Эта процедура создает маркированный список. Ее допускается использовать без параметра `DefaultListBehavior`. Собственно назначение параметра — указать Microsoft Word применять при формировании списка определенное форматирование:

```
const wdWord8ListBehavior=0;  //Старый стиль Word 97
      wdWord9ListBehavior=1;  // Web-ориентированное форматирование
      wdWord10ListBehavior=2; //Современное форматирование
```

Второй метод формирует нумерованный список:

```
Procedure ApplyNumberDefault(DefaultListBehavior : Variant);
```

И, наконец, третий — многоуровневый список:

```
Procedure ApplyOutlineNumberDefault(DefaultListBehavior : Variant);
```

Коллекция списков

Если в документе Microsoft Word существует хотя бы один список, то он окажется в коллекции списков — объекте `Lists`. Отдельный элемент названной коллекции обслуживается объектом `List`, это просто список.

В коллекции `Lists` реализован метод, возвращающий конкретный объект `List` списка по его индексу

```
Function Item (Index : Integer) : OleVariant;
```

Как всегда общее число элементов в коллекции выясняется из свойства `Count`

```
Property Count : Integer;
```

Пример приведен в листинге 19.

Листинг 19. Работа с коллекцией списков

```
Document:=Word.Documents.Open('c:\test.docx');
  if Document.Lists.Count>0 then List:= Document.Lists.Item(1);
```

После того как мы научились получать доступ к отдельному списку `List`, более подробно познакомимся с его возможностями.

Чтобы узнать, сколько пунктов содержится в списке, используйте метод

```
Function CountNumberedIteMicrosoft : Integer;
```

Как и большинство объектов Word, список может возвратить свою область — `Range`:

```
Property Range : OleVariant;
```

Каждый отдельный пункт списка (абзац) хранится в коллекции абзацев — свойстве

```
Property ListParagraphs : OleVariant:
```

В свою очередь в качестве отдельного элемента коллекции `ListParagraphs` выступает объект `Paragraph`.

В листинге 20 приведен фрагмент кода, демонстрирующий способ вставки в список нового элемента.

Листинг 20. Вставка нового элемента в список

```
List:= Document.Lists.Item(1);
if List.CountNumberedIteMicrosoft>=2 then
   begin
    Range:=List.ListParagraphs.Item(2).Range;
```

```
    Range.InsertParagraphAfter;
    Range:=List.ListParagraphs.Item(3).Range;
    Range.InsertBefore('Третий элемент списка');
  end;
```

Мы обращаемся к самому первому списку в документе, и если в нем два (или более) элемента, вставляем третий элемент и заполняем его текстом.

Коллекция разделов *Sections* и раздел документа *Section*

Создаваемый по умолчанию документ Word включает всего один раздел, описываемый объектом `Section`. Доступ к разделу производится с помощью коллекции разделов документа `Sections`:.

```
Section:= Document.Sections.Item(Doc.Sections.Count);
```

Коллекция разделов также может пригодиться для добавления нового раздела в документ. Для этой цели используют ее метод `Add()`:

```
Procedure Add(Range : OleVariant; [const Start : wdSectionStart]);
```

Параметр `Range` указывает область документа, в которую будет вставляться раздел. Параметр `Start` — константа типа `wdSectionStart`, определяющая особенности вставки нового раздела: "со следующей страницы", "на текущей странице", "с четной страницы" и "с нечетной страницы" (листинг 21).

Листинг 21. Константы разделов документа wdSectionStart

```
{wdSectionStart - разделы документа}
const  wdSectionContinuous=0; //Раздел на текущей странице
       wdSectionNewColumn=1;  //Новая колонка
       wdSectionNewPage=2;    //Раздел с новой страницы
       wdSectionEvenPage=3;   //Раздел с четной страницы
       wdSectionOddPage=4;    //Раздел с нечетной страницы
```

Параметр не обязательный, и если вы его пропустите, то просто добавится разрыв страницы.

Замечание

Еще один способ вставки нового раздела предоставляет диапазон `Range`, для этого у него имеется метод `InsertBreak()`. Несколько позже мы его рассмотрим подробнее.

А теперь поговорим об объекте раздела — `Section`. Основная задача раздела — обеспечение индивидуального форматирования части документа. Раздел хранит информацию об ориентации страницы, полях, колонтитулах, последовательности номеров страниц. Например, обратившись к свойству `PageSetup` раздела, мы полу-

чим право поменять ориентацию страницы и организовать многоколоночное форматирование одного из разделов документа (листинг 22).

Листинг 22. Работа с разделом документа

```
var Word_App,Document,Section,Range,Paragraph,Header : OleVariant;
    i:integer;
begin
 if GetAutomationServer('Word.Application',Word_App)=false then exit;
   try
    Document:=Word_App.Documents.ADD;

    for i:=1 to 9 do {создаем разделы документа}
         begin
          Paragraph:=Document.Paragraphs.Add;
          Range:=Paragraph.Range;
          Document.Sections.Add(Range,Start:=wdSectionNewPage);
         end;

    for i:=1 to 10 do {перебираем разделы документа}
       begin
         Section:=Document.Sections.item(i);
          if Odd(i) then
              Section.PageSetup.Orientation:=wdOrientPortrait
          else
              Section.PageSetup.Orientation:=wdOrientLandscape;

          Range:=Section.Range;
          Range.InsertBefore('Раздел '+IntToStr(i));
       end;

      Header:=Section.Headers.Item(1); //вставили колонтитул
      Header.PageNumbers.Add;          //внесли в колонтитул № страницы

    Word_App.Visible:=True;
   Finally
     Word_App:= Unassigned;
   End;
end;
```

Благодаря тому, что раздел ссылается на область (объект `Range`), мы получаем широкие полномочия по редактированию содержимого секции. Кроме того, раздел

документа предоставляет доступ к верхнему и нижнему колонтитулам раздела. Так свойства

```
Property Headers : OleVariant;
Property Footers : OleVariant;
```

возвращают ссылку на коллекцию колонтитулов HeadersFooters.

Коллекция таблиц *Tables* и таблица *Table*

Все существующие в документе таблицы хранятся в коллекции таблиц Tables этого документа. Для добавления к списку новой таблицы вызывайте метод

```
Function Add(Range : OleVariant;
             NumRows, NumColumns : cardinal) : OleVariant;
```

Метод применяется к списку таблиц и в своей минимальной нотации требует передачи трех параметров. Аргумент Range указывает область, в которую будет вставлена таблица, параметры NumRows и NumColumns отвечают соответственно за число рядов и колонок в будущей таблице. Функция возвратит объект-таблицу Table (листинг 23).

Листинг 23. Вставка в документ новой таблицы

```
var Word_App,Document,Range,Table : OleVariant;
…
   Document:=Word_App.Documents.add;
   Range:=Document.Paragraphs.Item(Document.Paragraphs.Count).Range;
   Table:=Document.Tables.Add(Range,3,4);
```

Приведенный пример демонстрирует способ создания в документе Document таблицы размерностью 3×4 ячейки. Таблица добавляется в самый конец документа.

С каждой вновь добавленной таблицей значение свойства

```
Property Counts: integer; //количество таблиц в коллекции Tables
```

коллекции таблиц Tables получает приращение.

Для того чтобы обратиться к какой-либо таблице из коллекции, воспользуйтесь методом

```
Function Item(Index : integer) : OleVariant;
```

Пример:

```
Table:=Doc.Tables.Item(5);// 5-я таблица
```

У коллекции таблиц не так много методов и свойств. Но у самой таблицы Table их хоть отбавляй. За оформление таблицы в первую очередь отвечает свойство

```
Property AutoFormatType : Integer; //только для чтения
```

Чтобы посмотреть, что это такое, запустите Word, создайте таблицу с произвольным числом строк и столбцов, а затем на панели быстрого доступа выберите

вкладку **Конструктор | Стили таблиц**. В новом окне (рис. 6) мы получим впечатляющий список стилей таблиц (простая таблица, классическая таблица, сетка таблицы и т. д.).

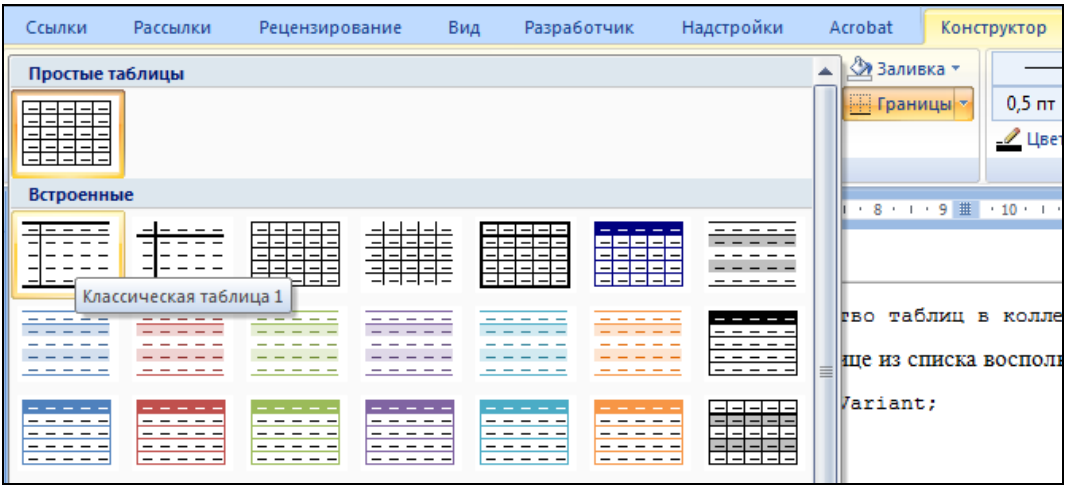


Рис. 6. Фрагмент окна с настройкой стиля таблиц Microsoft Word

Для того чтобы изменить стиль таблицы, обратитесь к методу

```
Procedure AutoFormat(FormatIndex : Integer);
```

Для демонстрации работы процедуры `AutoFormat()`предложен листинг 24, формирующий несколько таблиц различных стилей.

Листинг 24. Просмотр стилей таблиц

```
var Word_App,Document,Range,Paragraph,Table : OleVariant;
    i:integer;
begin
if GetAutomationServer('Word.Application',Word_App)=false then exit;
   try
   Document:=Word_App.Documents.add;
   for i:=0 to 42 do
     begin
      Paragraph:=Document.Paragraphs.Add;
      Paragraph.Range.InsertBefore(IntToStr(i));
      Paragraph:=Document.Paragraphs.Add;
      Document.Tables.Add(Paragraph.Range,4,4);
      Document.Tables.Item(i+1).AutoFormat(i);
     end;
```

```
  Finally
    Word_App:= Unassigned;
  End;
end;
```

Значение, передаваемое в параметр FormatIndex, должно находиться в диапазоне значений от 0 до 42. Выборочные значения констант формата таблицы приведены в листинге 25.

Листинг 25. Значения констант формата таблицы

```
{WdTableFormat - форматы таблиц}
const wdTableFormatNone=0;        //Обычная таблица
      wdTableFormatSimple1=1;     //Простая таблица 1
      wdTableFormatSimple2=2;     //Простая таблица 2
      wdTableFormatSimple3=3;     //Простая таблица 3
      wdTableFormatClassic1=4;    //Классическая таблица 1
      wdTableFormatColorful1=8;   //Цветная таблица 1
```

Таблица состоит из рядов и колонок. Список этих объектов мы обнаружим в соответствующих коллекциях:

```
Property Rows : OleVariant;    //коллекция рядов Rows
Property Columns : OleVariant; //коллекция колонок Columns
```

Названные коллекции в свою очередь обладают методами, обеспечивающими доступ к своим элементам — рядам и колонкам:

```
Function Item(RowIndex : Integer) : OleVariant;   // ряд таблицы
Function Item(ColumnIndex : Integer) : OleVariant;// колонка таблицы
```

Общее число колонок или рядов в коллекции как всегда находим в свойстве Count коллекции.

Для добавления нового ряда или новой колонки соответственно воспользуйтесь методами

```
Function Add(BeforeRow : OleVariant) : OleVariant; //объект Row
Function Add(BeforeColumn : OleVariant) : OleVariant; //объект Column
```

Параметр Before... указывает, перед каким элементом мы планируем осуществить вставку. Методы возвратят ссылку на вставленный объект — столбец или ряд.

Для удаления ряда или колонки вызываем стандартный метод коллекции Delete():

```
Table.Rows.IteMicrosoft(1).Delete;
```

Для изменения программным образом ширины колонки понадобится свойство

```
Property Width : Integer;
```

Высоту ряда задает свойство

```
Property Height : Integer;
```

Весьма полезный метод, осуществляющий автоматическую настройку размеров столбцов и рядов:

```
Procedure AutoFitBehavior(Behavior: WdAutoFitBehavior);
{WdAutoFitBehavior автоформат таблиц}
const wdAutoFitFixed=0;   //фиксированный размер
      wdAutoFitContent=1; //по содержимому
      wdAutoFitWindow=2;  //по ширине окна
```

В качестве параметра лучше всего передавать константу `wdAutoFitWindow` (автоподбор по ширине окна). Перед вызовом этого метода необходимо установить в `true` свойство, разрешающее автоматическую настройку таблицы:

```
Property AllowAutoFit: boolean;
```

Для того чтобы при переходе таблицы на новую страницу она вновь начиналась с заголовка, установите в `true` свойство

```
Property HeadingFormat : Boolean;
```

Если необходимо сдвинуть таблицу вправо от границы листа, то воспользуйтесь свойством

```
Property LeftIndent : integer;
```

Пример работы с таблицей приведен в листинге 26.

Листинг 26. Пример работы с таблицей

```
Table:=Document.Tables.Add(Document.Range(0,0),4,4);
 Table.Columns.Add(Table.Columns.Item(2));
 Table.Rows.Add(Table.Rows.Item(2));
 Table.Rows.Item(1).Height:=20;
 Table.Columns.Item(1).Width:=20;
 Table.AllowAutoFit:=True;
 Table.AutoFitBehavior(wdAutoFitContent);
 Table.Rows.Item(1).HeadingFormat:=True;
 Table.Rows.LeftIndent:=260;
```

ВНИМАНИЕ!

Программисты Delphi привыкли, что в компонентах-сетках библиотеки VCL при выборе ячейки первым параметром идет колонка, а вторым — ряд (как в свойстве сетки Cells сетки `TStringGrid`). Ячейки в приложениях Microsoft идентифицируются в обратном порядке — сначала ряд, а затем колонка.

Значения в колонке можно отсортировать, воспользовавшись методом

```
Procedure Sort;
```

Основой любой таблицы служит ячейка (объект `Cell`), она хранится в коллекции ячеек (`Cells`) и идентифицируется двумя значениями — номерами ряда и колонки:

```
Property Cell(aRow, aCol : Integer) : OleVariant;
```

Для того чтобы внести в ячейку текстовые данные, необходимо вспомнить изученный ранее объект-область Range. В листинге 27 для заполнения таблицы данными вызывается метод InsertAfter().

Листинг 27. Вставка текста в ячейку таблицы

```
var …, Document, Table : OLEVariant;
       aRow : Integer;
//…
Table:= Document.Tables.Add(Document.Range(0,0),9,2);
For aRow:=Table.Rows.Count Downto 1 do
          Table.Cell(aRow,1).Range.InsertAfter(aRow);
Table.Columns.Item(1).Sort;
```

В рассмотренном примере создается таблица, ее первая колонка заполняется цифрами в обратном порядке (от 9 до 1), а затем вызывается метод сортировки, упорядочивающий цифры по возрастанию.

Для удаления ячейки следует обратиться к процедуре

```
Procedure Delete(ShiftCells : wdDeleteCells);
{wdDeleteCells - удаление ячеек таблицы}
const wdDeleteCellsShiftLeft=0;    //Со сдвигом влево
      wdDeleteCellsShiftUp=1;      //Со сдвигом вверх
      wdDeleteCellsEntireRow=2;    //Удалить весь ряд
      wdDeleteCellsEntireColumn=3; //Удалить всю колонку
```

Параметр ShiftCells определяет порядок удаления ячейки, например

```
Table.Cell(4,1).Delete(wdDeleteCellsEntireColumn); {удаляем первую колонку таблицы}
```

Ячейка таблицы Microsoft Word обладает способностью делиться на части или, наоборот, объединяться со своими соседками. Для того чтобы разделить ячейку, вызовите процедуру

```
Procedure Split(NumRows, NumColumns : Integer);
```

Параметры NumRows и NumColumns определяют, на сколько рядов и колонок мы планируем разбить выбранную ячейку:

```
Table.Cell(1,1).Split(2,2);
```

Процедура

```
Procedure Merge(MergeTo : OleVariant);
```

выполняет обратную операцию — слияние ячеек.

В качестве параметра передается ссылка на ячейку, с которой мы хотим объединиться:

```
Table.Cell(1,1).Merge(Table.Cell(1,2));
```

Часто возникает необходимость повернуть текст в ячейке. Следующие две строки кода демонстрируют способ поворота текста на 90° против часовой стрелки.

```
Range:=Table.Cell(1,1).Range;
Range.Orientation:=2;// wdTextOrientationUpward
```

Для решения этой задачи мы воспользовались свойством `Orientation()` области `Range`.

Внедрение в документ OLE-объектов

OLE-объект (рисунок, форма, элемент ActiveX) хранится в документе в специализированном объекте `Shape`. В свою очередь `Shape` содержится в коллекции `Shapes`.

Для пополнения OLE-объектами коллекция `Shapes` вооружена богатым перечнем методов. Для вставки в документ файла изображения понадобится функция

```
Function AddPicture(FileName : String;
      [LinkToFile, Left, Top, Width, Height : Integer]) : OleVariant;
```

Метод вставляет в документ рисунок из файла `FileName`. Минимальный синтаксис требует передать только имя файла:

```
Document.Shapes.AddPicture('C:\Image.bmp');
```

Параметр `LinkToFile` определяет, каким образом изображение станет храниться в документе: в качестве внедренного в состав документа (`false`) или же с файлом картинки будет установлена связь (`true`). Параметры `Left`, `Top` назначают местоположение левого верхнего угла картинки. Параметры `Width`, `Height` отвечают за ширину и высоту картинки.

Пример:

```
Doc.Shapes.AddPicture(FileName:='C:\Image.bmp', LinkToFile:=False,
Width:=100);
```

Еще более унифицированный метод внедрит в документ любой зарегистрированный в системе OLE-объект:

```
Function AddOLEObject([const ClassType : string;]
                      FileName : String;
                     [LinkToFile,
                      DisplayAsIcon : Boolean;
                      IconFileName : String;
                      IconIndex : Integer;
                      IconLabel : String;
                      Left, Top, Width, Height : Integer]) : OleVariant;
```

Здесь `ClassType` — идентификатор внедряемого OLE-объекта. Список зарегистрированных в системе объектов вы увидите, воспользовавшись вкладкой **Вставка** панели быстрого доступа.

Замечание

Напомню, что идентификаторы зарегистрированных в системе OLE-объектов регистрируются в реестре Windows в секцию HKEY_CLASSES_ROOT\CLSID\{…}\ ProgID.

Если вы внедряете в документ объект из файла, то достаточно передать имя файла через параметр FileName:

```
Document.Shapes.AddOleObject(FileName:='C:\Книга1.xls');
```

Параметр LinkToFile определит способ хранения OLE-объекта. Если вы хотите, чтобы OLE-объект внедрялся в документ, то передаете значение false. В противном случае документ установит связь с файлом объекта OLE. Внедренный объект можно представить в виде иконки, для этого присвойте параметру DisplayAsIcon значение True. Если для отображения иконки OLE-объекта вы планируете использовать свою собственную пиктограмму, то передайте путь к ней в параметре IconFileName. Если в объект OLE внедрены несколько иконок, то в параметре IconIndex можно передать порядковый номер иконки. Текст иконки описывается в аргументе IconLabel. Местоположение и размеры внедренного объекта определяются аргументами Left, Top, Width, Height.

Простейшие графические фигуры (доступные на панели **Фигуры**) также представляют собой OLE-объекты. Для того чтобы внедрить в документ линию, воспользуйтесь методом

```
Procedure AddLine(BeginX, BeginY, EndX, EndY : Integer);
```

Пример:

```
Document.Shapes.AddLine(50,50,100,100);
```

Для вставки в документ фигуры предусмотрена процедура

```
Procedure AddShape(Const Type : Integer;
                   Left, Top, Width, Height : Integer);
```

Здесь ключевой аргумент Type — это константа фигуры. Начиная с Microsoft Word 2000, аргумент Type может принимать любое значение из диапазона чисел 1…130:

```
Document.Shapes.AddShape(16, 10, 10, 110, 110);
```

Элемент коллекции Shapes — объект Shape обладает широким спектром возможностей по настройке внешнего вида объекта. Местоположение и размеры объекта описываются свойствами:

```
Property Left : Integer;
Property Top : Integer;
Property Width : Integer;
Property Height : Integer;
```

Для поворота объекта вокруг горизонтальной (вертикальной) оси понадобится процедура

```
Procedure Flip(Const FlipCmd : Integer);
```

Для поворота по горизонтали передавайте 0, по вертикали — 1.

Метод

```
Procedure Delete;
```

удаляет объект.

Оформление документа

Разработчики Office реализовали широкий перечень объектов, нашедший свое применение во всех приложениях пакета. К числу наиболее востребованных объектов офиса относится шрифт:

```
Property Font : OleVariant;
```

Спектр настройки характеристик шрифта весьма широк: начиная от имени гарнитуры, размера, начертания и заканчивая включением контура и тени (табл. 3).

Таблица 3. *Свойства объекта* `Font`

Название	Описание
`Property Name : String;`	Имя шрифта
`Property Size : Cardinal;`	Размер шрифта в пунктах
`Property Kerning : Cardinal;`	Значение минимального размера шрифта, после которого автоматически включается кернинг
`Property Spacing : Integer;`	Степень уплотнения шрифта в пунктах. Если значение больше нуля, то шрифт разрежен, меньше нуля — уплотнен, 0 — нормальное состояние
`Property Position : Integer;`	Смещение строки текста относительно условной линии в пунктах. Отрицательные значения — смещение вниз, положительные — вверх
`Property Scaling : Integer;`	Горизонтальное масштабирование шрифта. От 1 до 99 — сжат по горизонтали, 100 — нормальное состояние, от 101 до 600 — растянут по горизонтали
`Property Bold : Boolean;`	`True` — жирный, `False` — обычный
`Property Italic : Boolean;`	`True` — курсив, `False` — обычный
`Property UnderLine : Boolean;`	`True` — нижнее подчеркивание, `False` — без подчеркивания
`Property UnderLineColor : ColorRef;`	Цвет линии подчеркивания. Имеет смысл, если свойство `UnderLine` установлено в `true`
`Property StrikeThrough : Boolean;`	`True` — зачеркивание, `false` — без зачеркивания
`Property DoubleStrikeThrough : Boolean;`	`True` — двойное зачеркивание, `false` — без зачеркивания
`Property Outline : Boolean;`	`True` — только внешний контур, `false` — обычный
`Property Shadow : Boolean;`	`True` — с тенью, `false` — без тени
`Property Color : ColorRef;`	Цвет, для использования подходят константы, объявленные в Delphi

Граница области

Границу области описывает объект `Border`. Для обращения к требуемой границе коллекции границ `Borders` вызываем метод

```
Function IteMicrosoft(Index : Integer) : OleVariant;
```

В качестве параметра передается константа `wdBorderType`, определяющая, к какой из границ (рис. 7) мы хотим получить доступ.

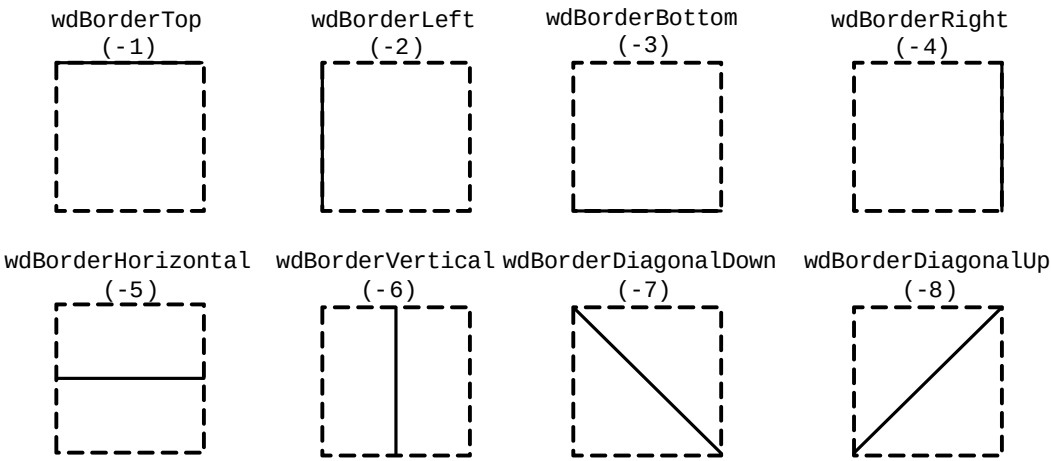


Рис. 7. Константы границ области

Внимание!

Границы `wdBorderDiagonalDown`, `wdBorderDiagonalUp`, `wdBorderVertical` и `wdBorderHorizontal` применяются только к объекту `Cell` (ячейка таблицы `Table`).

Границу можно включить или отключить, за это отвечает свойство

```
Property Enable : Boolean;
```

Свойство

```
Property OutsideColor : ColorRef;
```

определяет цвет границы.

Для включения тени границы необходимо установить в `true` свойство

```
Property Shadow : Boolean;
```

Свойство

```
Property LineStyle: wdLineStyleNone;
```

определяет стиль линии.

Именованные константы (листинг 28) задают стиль линий.

Листинг 28. Именованные константы, задающие стиль линии

```
const wdLineStyleNone=0; {нет линии}
      wdLi-
neStyleSingle=1;{одинарная}
      wdLineStyleDot=2;  {точки}
wdLineStyleDashSmallGap=3;{тире-
пробел}
wdLineStyleDashLargeGap=4;{тире
большой пробел}
wdLineStyleDashDot=5;{тире-точка}
wdLineStyleDashDotDot=6;{тире-
точка-точка}
wdLineStyleDouble=7;{двойная}
wdLineStyleTriple=8;{тройная}
wdLineStyleThinThickSmall-
Gap=9;{тонкая —утолщенная}
```

В заключение подраздела приведем небольшой пример работы с границами (листинг 29).

Листинг 29. Граница абзаца документа

```
{включаем границу первого абзаца документа}
Document.Paragraphs.Item(1).Borders.Enable:=true;
{делаем невидимой верхнюю линию}
Document.Paragraphs.Item(1).Borders.Item(wdBorderTop).LineStyle:=
   wdLineStyleNone;
```

Пример универсального генератора отчетов

Наступила пора применить свои знания о сервере автоматизации Microsoft Word на практике. Предлагаю создать полезную утилиту — программу, позволяющую экспортировать в документ Word данные из таблиц формата Microsoft Access. Для этого нам понадобится четыре компонента VCL:

- Диалог открытия файла `OpenDialog1`.
- Две кнопки `TButton`. Переименуйте первую кнопку в `btnFileSelect`, этот компонент предназначен для обращения к диалогу выбора файла базы данных Access. Вторую кнопку назовем `btnWordReport` — она дает команду на генерацию отчета.
- Список `ListBox1`, он специализируется на сборе имен таблиц из БД Access. Переименуйте элемент управления в `lboxTableNames`.
- Любой компонент-запрос, например `ADOQuery1`.

Разместите все компоненты на главной форме нового проекта и приступайте к делу.

Работа начинается с подключения к проекту модуля `ComObj`:

```
uses ComObj;
```

В этом модуле объявлена "бесценная" функция `CreateOleObject()`, инициирующая объект автоматизации. Затем настроим диалог открытия файлов. Для этого в момент создания главной формы проекта определим фильтр, пропускающий файлы только с расширением mdb и accdb (БД Access старого и нового форматов). Кроме того, внесем некоторые изменения в опции диалога (листинг 30).

Листинг 30. Создание главной формы проекта

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  OpenDialog1.Filter:='Таблицы Microsoft Access|*.mdb;*.accdb';
  OpenDialog1.Options:=OpenDialog1.Options+
                              [ofReadOnly]-[ofAllowMultiSelect];
end;
```

Переходим к событию `OnClick()`кнопки `btnFileSelect`. Листинг 31 иллюстрирует порядок обращения к целевой базе данных.

Листинг 31. Диалог выбора БД Microsoft Access

```
procedure TForm1.btnFileSelectClick(Sender: TObject);
var FileName : string;
begin
 if OpenDialog1.Execute then //запуск диалога открытия файла
     BEGIN
     lboxTableNames.Items.Clear; //очищаем список таблиц
     FileName:=OpenDialog1.FileName; //имя файла БД
     {собираем имена всех доступных таблиц}
     GetAccessDBTables(OpenDialog1.FileName, lboxTableNames.Items);

    if lboxTableNames.Count>0 then
      begin
        lboxTableNames.ItemIndex:=0;
        btnWordReport.Enabled:=True;
      end else btnWordReport.Enabled:=True;

   {формируем соединительную строку}
    ADOQuery1.ConnectionString:='Provider=Microsoft.ACE.OLEDB.12.0;
                Data Source='+ FileName +';Persist Security Info=False';
    END;
end;
```

Обратите внимание на процедуру `GetAccessDBTables()`, она предназначена для заполнения списка `lboxTableNames:TListBox` именами таблиц базы данных. Код процедуры мы написали в начале главы (листинг 1).

Переходим к ядру нашей программы — процедуре генератора отчетов. Для этого в секции частных объявлений проекта опишем заголовок метода

```
Procedure CreateWordReport(TableName: string);
```

Реализация метода приведена в листинге 32.

Листинг 32. Метод построения отчета

```
procedure TForm1.GetWordReport(TableName: string);
var Word_App, Document, Range, Paragraph, Table, Selection : OleVariant;
    aRow, aCol : integer;
begin
{вызываем сервер автоматизации Word}
if GetAutomationServer('Word.Application',Word_App)=false then exit;

With ADOQuery1 do {Готовим запрос к таблице}
   begin
     If Active then Close;
     SQL.Clear;
     SQL.Add('Select * from '+TableName);
     Open;
   end;
TRY
  Document:=Word_App.Documents.Add;          {новый документ}
  Document.PageSetup.PaperSize:=wdPaperA4;   {формат A4}
  Document.PageSetup.Orientation:=wdOrientLandscape; {альбомная ориентация}

  Range:=Document.Range;
  Range.InsertBefore('Таблица '+TableName); {первая строка на странице}
  Range.Select;
  Selection:=Word_App.Selection;    {выделенный объект}
  Selection.Style:=wdStyleHeading1; {присваиваем выделению стиль заголовка}
  Paragraph:=Document.Paragraphs.Add;{добавим пару пустых абзацев}
  Paragraph:=Document.Paragraphs.Add;

{Вставляем в документ пустую таблицу, количество строк и столбцов определяется
числом записей RecordCount и полей FieldCount в файле базы данных}
  Table:=Document.Tables.Add(Paragraph.Range,
                             ADOQuery1.RecordCount+1,
                             ADOQuery1.FieldCount);
  {формируем заголовок таблицы из названий полей}
  for aCol:=1 to ADOQuery1.FieldCount do
      Table.Cell(1,aCol).Range.InsertAfter(
                             ADOQuery1.Fields[aCol-1].DisplayName);
```

```
  aRow:=2; //переводим счетчик строк на 2-ю строку
  WHILE ADOQuery1.Eof<>true DO
  BEGIN {перебираем строки файла и заполняем таблицу}
    for aCol:=1 to ADOQuery1.FieldCount do {перебор колонок}
        {не пропускаем в отчет BLOB-поля}
        if ADOQuery1.Fields[aCol-1].IsBlob=False then
           Table.Cell(aRow,aCol).Range.InsertAfter(
                                    ADOQuery1.Fields[aCol-1].AsString)
         else Table.Cell(aRow,aCol).Range.InsertAfter('BLOB');

    Inc(aRow);      //приращение счетчика строк
    ADOQuery1.Next; //переход к следующей записи
   END;
  Table.AllowAutoFit:=True; {форматируем таблицу}
  Table.AutoFormat(wdTableFormatColorful1); {8 цветная таблица}

  {заполняем основные свойства документа}
  Document.BuiltinDocumentProperties[wdPropertyTitle].Value:=
                                              'Таблица '+TableName;
  Document.BuiltinDocumentProperties[wdPropertyAuthor].Value:=
                                                  'Фамилия И.О.';

  Word_App.Visible:=true; {вывод на экран}
FINALLY
  Word_App:= Unassigned;
  ADOQuery1.Close;
END;
end;
```

Для завершения работы над приложением осталось совсем немного. Сделаем последний шаг — научим программу передавать в процедуру `GetWordReport()` имя файла таблицы. Для этого нам необходим обработчик щелчка по кнопке `btnWordReport` (листинг 33).

Листинг 33. Выбор таблицы для построения отчета

```
procedure TForm1.btnWordReportClick(Sender: TObject);
var TableName:string;
begin
Screen.Cursor:=crHourgLass;
  TableName:=lboxTableNames.Items.Strings[lboxTableNames.ItemIndex];
  GetWordReport(TableName);
Screen.Cursor:=crDefault;
end;
```

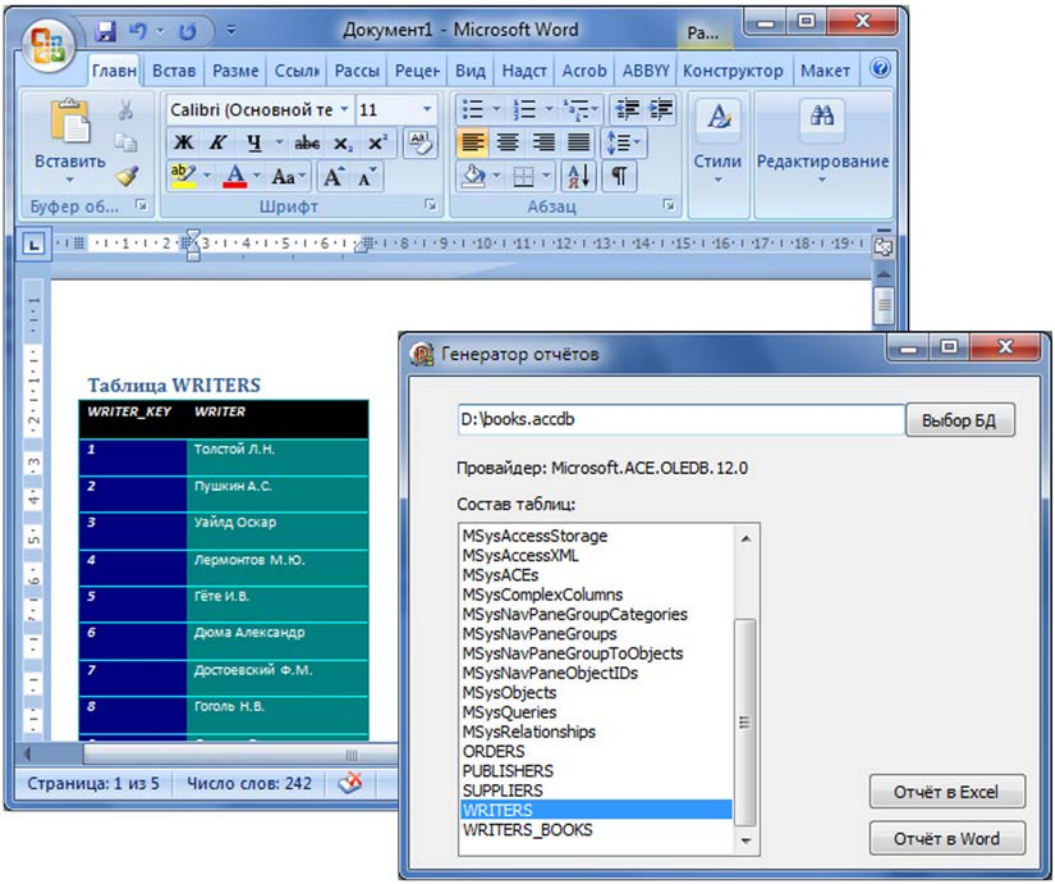


Рис. 8. Демонстрация построения отчета средствами Microsoft Word

Вот и все! Пользователь выбирает интересующий его файл БД Microsoft Access. Весь перечень таблиц из БД собирается в список `lboxTableNames`. Пользователь выделяет интересующую его строку с именем таблицы и затем вызывается процедура генерации отчета. На мой взгляд, у нас получилась весьма элегантная и, главное, универсальная утилита (рис. 8).

Электронные таблицы Microsoft Excel

Вряд ли стоит отдельно представлять один из наиболее популярных продуктов из пакета Microsoft Office. За простоту и удобство электронные таблицы Excel заслужили признательность практически у всех, кто хотя бы раз в жизни попробовал подсчитать в них свои налоги и чужие доходы.

Ключевые объекты Microsoft Excel (рис. 9):

- приложение `Application`;

- рабочая книга `WorkBook`;
- лист в книге `Sheet`;
- лист электронной таблицы `Worksheet`;
- лист диаграммы `Chart`;
- область `Range`.

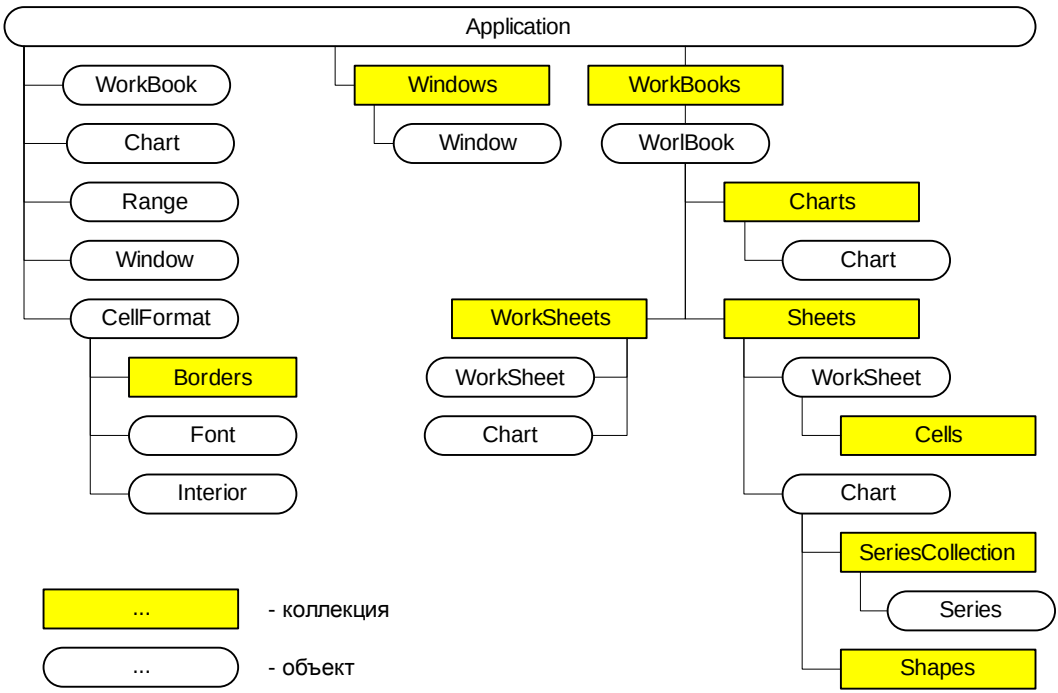


Рис. 9. Основные элементы объектной модели Microsoft Excel

Приложение Excel — *Application*

Фундаментальным объектом, обеспечивающим работу с программой электронных таблиц Microsoft Excel, является приложение — объект `Application`. В рамках `Application` опубликованы методы и свойства, реализующие доступ к ключевым объектам приложения: коллекциям рабочих книг `WorkBooks`, текущей активной книге, активному листу, активной ячейке. Объекты и коллекции модели Microsoft Excel очень сильно взаимоувязаны (см. рис. 9), доступ к одному и тому же объекту можно получить несколькими альтернативными способами.

Приложение электронных таблиц из программы Delphi вызывается точно так же, как и любой другой сервер автоматизации: с помощью методов `CreateOleObject()` и `GetActiveOleObject()`.

Замечание

Напомню, что в начале главы мы разработали универсальную процедуру `GetAutomationServer()`, позволяющую подключиться к любому серверу автоматизации.

Для того чтобы избавиться от излишних предупреждающих сообщений, установите в `false` свойство

```
Property DisplayAlerts : Boolean;
```

Для показа приложения передайте значение `true` в свойство

```
Property Visible : Boolean;
```

Приложение знает, какие из принадлежащих ему объектов в настоящий момент активны:

```
Property ActiveWorkbook : OleVariant;//активная рабочая книга
Property ActiveSheet : OleVariant; //активный лист рабочей книги
Property ActiveChart : OleVariant; //активная диаграмма
Property ActiveCell : OleVariant;  //диапазон активных ячеек
```

Интерфейс приложения `Application` хранит ссылки на ключевые коллекции объектов Excel. Чаще всего мы будем обращаться к свойствам

```
Property Workbooks : OleVariant;//все существующие рабочие книги
Property Sheets : OleVariant;   //коллекция листов активной книги
Property Rows : OleVariant;     //диапазон рядов активного листа
Property Columns : OleVariant;  //диапазон колонок активного листа
```

Свойство

```
Property Cells : OleVariant; // только для чтения
```

возвращает область, в данном случае содержащую все ячейки активного листа.

Коллекция *WorkBooks* и книга *Workbook*

Каждая книга Excel может включать в свой состав листы (`Sheet`) и диаграммы (`Chart`), при сохранении книги в виде файла на диске создается файл с расширением xlsx (для Excel 2007–2010) или xls (для Excel 97–2003). Все открытые книги хранятся в коллекции `WorkBooks`.

Для добавления новой книги в коллекцию `WorkBooks` воспользуйтесь методом

```
Function Add : OleVariant;
```

Функция возвратит экземпляр новой книги `WorkBook`, содержащей по умолчанию три листа (листинг 34).

Листинг 34. Создание книги Excel

```
var Excel_APP,WorkBook:OleVariant;
begin
 if GetAutomationServer('Excel.Application',Excel_APP)=false then exit;
 TRY
```

```
   WorkBook:=Excel_APP.WorkBooks.Add;
   //другие операции
   Excel_APP.Visible:=True;
 FINALLY
   Excel_APP:=Unassigned;
 END;
end;
```

Новую книгу можно создать на основе предопределенного шаблона:

```
WorkBook:=Excel_APP.WorkBooks.Add('c:\Program Files\Microsoft Of-
fice\Templates\1049\Авансовый_отчет.xltx');
```

Для того чтобы сделать книгу активной, нужно вызвать метод

```
Procedure Activate;
```

Простейший способ открытия файла предоставляет функция

```
Function Open(FileName : string) : OleVariant;
```

которая возвращает объект `WorkBook`.

Элементарный способ закрытия книги — вызов метода

```
Procedure Close;
```

Если с момента последнего сохранения файла в книге происходили изменения, то перед пользователем появится стандартный диалог сохранения файла. Закрыть файл можнло и без показа диалога сохранения. Для этого следует вызвать расширенную версию процедуры:

```
Procedure Close(SaveChanges : Boolean; FileName : String);
```

Если параметр `SaveChanges` установить в `false`, то книга закроется, не предприняв попытки сохранения. Если же в этот параметр передать значение `true`, то изменения направятся в файл `FileName`:

```
WorkBook.Close(SaveChanges:=true, FileName:='c:\test.xlsx');
```

Для сохранения данных в файл по умолчанию используйте метод

```
Procedure Save;
```

Но если эта процедура применяется для недавно созданной книги (пока не сохраненной на диск), метод вызовет стандартный диалог сохранения файла. Чтобы не встречаться с этим диалогом, лучше вызвать процедуру

```
Procedure SaveAs(FileName : string);
```

ВНИМАНИЕ!

Методы `Open()` и `SaveAs()` приведены с минимальным набором параметров. На самом деле они включают аргументы, обеспечивающие защиту документа паролем, настройку формата файла, режим "только для чтения" и многое другое.

Следует также упомянуть свойство, хранящее путь и имя файла книги

```
Property Path : string;
```

и его эквивалент

```
Property FullName : string;
```

Если свойство пусто, значит, книга ни разу не сохранялась.

Имя файла вы сможете узнать с помощью свойства

```
Property Name : string;
```

Для того чтобы уяснить, имеются ли в книге несохраненные изменения, обратитесь к свойству

```
Property Saved : Boolean;
```

Если книга не нуждается в сохранении, то свойство отрапортует значением `true`.

Доступ к ключевым коллекциям книги предоставляют свойства:

```
Property Worksheets : OleVariant; //коллекция листов книги
Property Charts : OleVariant;     //коллекция диаграмм книги
```

Первое свойство ссылается на все имеющиеся в книге листы, второе — на диаграммы книги.

Листы Excel

В таблицах Microsoft Excel различают два типа листов — обычный лист электронной таблицы `Worksheet` и лист диаграммы — `Chart`. Для хранения каждого типа листа существует специализированная коллекция: `Worksheets` и `Charts`. Помимо названных коллекций для обслуживания листов в Excel реализована универсальная коллекция `Sheets`, способная содержать оба типа листов (`Worksheet` и `Chart`).

Универсальная коллекция листов *Sheets*

Информация о числе доступных листов находится в свойстве `Count`. Для обращения к отдельному листу в коллекции служит свойство `Item`:

```
Property Item(Index : Integer) : OleVariant;
```

Единственная особенность — возвращаемый результат. Свойство возвращает страницу `WorkSheet` или `Chart`, в зависимости от того, какой индекс `Index` передан в свойство.

Свойство

```
Property Visible : Boolean;
```

отображает (скрывает) все листы коллекции.

Свойства

```
Property VPageBreaks : Boolean; //вертикальные границы
Property HPageBreaks : Boolean; //горизонтальные границы
```

включают/отключают вывод границ страниц.

Для добавления листа в коллекцию предусмотрен стандартный для коллекции метод `Add()`.

Вместе с тем, функция `Add()` обладает расширенным синтаксисом:

```
Function Add([Before | After : OleVariant;] [Count : Cardinal;]
             const Type : XlSheetType) : OleVariant;
{XlSheetType – тип листа}
```

```
type xlChart=-4109;              //лист диаграммы Chart
     xlDialogSheet=-4116;        //лист диалога DialogChart
     xlWorksheet=-4167;          //обычный лист Worksheet
```

Программист вправе определить, в какое место коллекции вставляется лист или листы. Для этого используют параметр `Before` (перед каким листом) или `After` (после какого листа) вставляется новый. Заостряю внимание читателя на том, что одновременно эти аргументы не действуют. В параметре `Count` указывается, какое число страниц нужно добавить. Константа `Type` определяет тип создаваемого листа, описываемый параметром `XlSheetType`. Если параметр `Type` не указывать, то создается обычный лист.

Следующий фрагмент кода (листинг 35) демонстрирует порядок добавления в коллекцию нескольких экземпляров листов.

Листинг 35. Добавление листов в коллекцию

```
WorkBook:=Excel_APP.WorkBooks.Add;
WorkSheet:=WorkBook.Sheets.Add;//добавление обычного листа к коллекции
WorkBook.Sheets.Add(After:=WorkSheet,2);//вставим еще 2 листа
Chart:=WorkBook.Sheets.Add(Type:=xlChart); //добавим лист диаграммы
```

Для удаления всех листов из коллекции используйте процедуру

```
Procedure Delete;
```

Замечание

Ряд опасных операций `Excel` сопровождает предупреждающими сообщениями. Напомню, что для отказа от сообщений понадобится установить свойство `DisplayAlerts` в `false`.

Метод

```
Procedure Move(Before или After : OleVariant);
```

перемещает листы `Sheet` внутри коллекции.

Параметры определяют новое местоположение листа: `Before` — перед каким листом, `After` — после какого листа размещается лист. На практике достаточно только одного параметра:

```
WorkBook.Sheets.Item[3].Move(Before:=WorkBook.Sheets.Item[1]);
```

Строка кода перемещает третий лист книги на первую позицию.

Замечание

Специализированные коллекции `Worksheets` и `Charts` обладают аналогичными с универсальной коллекцией `Sheets` свойствами и методами.

Общие свойства листов *Worksheet* и *Chart*

Каждому из вновь создаваемых листов приложение присваивает имя, например: в русскоязычной версии Microsoft Excel имя листа складывается из слова

"Лист" или "Диаграмма" и его порядкового номера в книге. Название листа записывается в свойство

```
Property Name : String;
```

В листинге 36 приведен фрагмент кода, обеспечивающий поиск листа по его имени.

Листинг 36. Поиск листа "Лист5"

```
for i:=1 to WorkBook.Sheets.Count-1 do
    if WorkBook.Sheets.Item[i].Name='Лист5' then
    begin
      WorkSheet:=WorkBook.Sheets.Item[i];
      WorkSheet.Activate;
      Break;
    end;
```

Отдельный лист можно сделать видимым (или спрятать), обратившись к свойству

```
Property Visible : Boolean;
```

Для удаления листа используйте процедуру

```
Procedure Delete;
```

Excel предоставляет возможность сохранить лист в отдельном файле. Для этого воспользуйтесь процедурой `SaveAs()`, имеющей аналогичный синтаксис с одноименной процедурой рабочей книги.

Параметры страницы — объект *PageSetup*

Параметры страницы определяются с помощью объекта `PageSetup`:

```
Property PageSetup : OleVariant;
```

Размер листа назначается свойством

```
Property PaperSize : XlPaperSize;
```

Диапазон допустимых значений от 1 до 41, каждое из них соответствует предопределенным размерам бумажного листа. Кроме того, имеется константа `xlPaperUser`, позволяющая назначать пользовательские размеры листа (листинг 37).

Листинг 37. Размеры страниц листа Excel

```
{XlPaperSize - размеры страниц}
const xlPaperLetter = 1;
      xlPaperLetterSmall = 2;
      xlPaperTabloid = 3;
      xlPaperLedger = 4;
      xlPaperLegal = 5;
      xlPaperStatement = 6;
      xlPaperExecutive = 7;
      xlPaperA3 = 8;
      xlPaperA4 = 9;
      xlPaperA4Small = 10;
      …
      xlPaperUser = 256;
```

Портретную (xlPortrait=1) или альбомную ориентацию листа (xlLandscape=2) установит свойство

```
Property Orientation : Integer;
```

Поля листа настраиваются свойствами:

```
Property LeftMargin : Double;  //левое поле
Property RightMargin : Double; //правое поле
Property TopMargin : Double;   //верхнее поле
Property BottomMargin : Double;//нижнее поле
```

Пример:

```
Worksheet.PageSetup.RightMargin :=Excel.CentimetersToPoints(2);
```

ЗАМЕЧАНИЕ

Для преобразования сантиметров в условные точки используйте метод приложения (объект Application) CentimetersToPoints().

Особенности листа электронной таблицы *Worksheet*

Лист Worksheet приложения Excel 2007 представляет собой двумерный массив-сетку из 1 048 576 рядов и 16 384 колонок. Ряды листа нумеруются по возрастанию, сверху вниз, начиная с единицы. Колонки именуются символами латинского алфавита A, B, …, AA, AB,… Элемент данного массива называется ячейкой. Отдельная ячейка массива однозначно адресуется номером ряда и именем колонки. Например, самая верхняя левая ячейка страницы обладает адресом "A1", ячейка правее ее — "B1", ниже — "A2". Подобная система адресации в Microsoft Excel называется стилем ссылок A1. Альтернативный способ адресации именуется стилем ссылок R1C1. В таком режиме адресации ячейка описывается номером ряда и номером колонки. На мой взгляд, для программирования более удобен стиль R1C1, однако, для того чтобы программист мог пользоваться обоими стилями, при изложении материала я буду рассматривать смешанную адресацию.

Коллекция, включающая все ячейки листа, доступна в свойстве

```
Property Cells : OleVariant;
```

Для выделения одной ячейки из области Cells воспользуйтесь свойством

```
Property Item[Row, Col : Integer] : OleVariant;
```

Здесь в квадратных скобках передаются ряд (Row) и колонка (Col), однозначно идентифицирующие ячейку:

```
Range := WorkSheet.Cells.Item[10,1]; {ячейка из 10-го ряда 1-й колонки — стиль R1C1}
```

Однако допускается и сокращенный синтаксис — без указания Item.

Пример:

```
Range := WorkSheet.Cells[5,2]; {ячейка из 5-го ряда 2-й колонки — стиль R1C1}
Range:=Sheet.Range[Sheet.Cells[1,1],Sheet.Cells[3,3]];{область ячеек — стиль R1C1}
```

Помимо доступа к элементарному объекту сетки ячейки, допустимо обращение к коллекциям рядов и колонок в целом. Для этого предназначены два свойства:

```
Property Rows : OleVariant;
Property Columns : OleVariant;
```

ЗАМЕЧАНИЕ

В Excel не реализованы отдельные объекты "ряд", "колонка" или "ячейка". Вместо этого, для доступа к отдельной ячейке (или их совокупности) разработан специализированный объект Range.

Страница Worksheet в состоянии содержать внедренные диаграммы. Для того чтобы обратиться к коллекции диаграмм на листе, потребуется метод

```
Function ChartObjects : OleVariant;
```

За инициирование пересчета формул на странице электронной таблицы Worksheet отвечает метод

```
Procedure Calculate;
```

Коллекция диаграмм *Charts* и лист диаграммы *Chart*

На хранении диаграмм Chart специализируется коллекция Charts. В роли ключевого метода коллекции выступает функция вставки листа диаграммы в книгу Excel:

```
Function Add([Before | After:OleVariant];[Count:Cardinal]) : OleVariant;
```

В простейшем случае функцию можно вызывать без параметров, однако если вы собираетесь добавить в рабочую книгу несколько листов-диаграмм, причем в определенное место, то параметр Count понадобится для указания количества экземпляров. Место вставки назначает аргумент After (перед каким листом) или Before (после какого листа).

Сам по себе, пустой лист диаграммы Chart малоинтересен. Разнообразие и функциональность в него вносят графики — объекты Series. Однако сразу после создания лист диаграммы Chart не содержит ни одного графика. Чтобы исправить такое положение вещей, нам понадобится свойство диаграммы:

```
Property SeriesCollection : OleVariant;
```

Это не что иное, как коллекция графиков, содержащихся на листе. Как и все рассмотренные ранее коллекции она обладает стандартным набором свойств и методов, ключевым из которых является функция добавления графика:

```
Function Add(Source : Olevariant,
             [const Rowcol : xlRowcol;
              SeriesLabels, CategoryLabels : Boolean]) : OleVariant;
```

Из четырех аргументов обязателен только параметр Source, определяющий область (Range), содержащую числовые значения, на основе которых будет строиться график. Другими словами, это столбец или ряд заполненных данными ячеек из листа электронной таблицы. Второй параметр может принимать одно из двух значений: xlColumns (значение 2) или xlRows (значение 1). Задача этого параметра —

указать графику, как в диапазоне Source располагаются значения (вертикально или горизонтально). Если параметр не передается, то по умолчанию график строится из вертикальных значений, если такое положение дел вас не устраивает, то установите его в единицу.

Пример создания диаграммы приведен в листинге 38.

Листинг 38. Подготовка данных и создание диаграммы

```
procedure TForm1.btnSeriesDemoClick(Sender: TObject);
var  Excel_App,WorkBook,Chart,WorkSheet,SeriesCollection,
     Range : OleVariant;
     row : integer;
begin
 if GetAutomationServer('Excel.Application',Excel_APP)=false then exit;
try
 Excel_App.SheetsInNewWorkbook :=1; //в книге 1 страница
 WorkBook:=Excel_App.Workbooks.Add; //создаем книгу
 WorkSheet:=WorkBook.WorkSheets[1]; //лист электронной таблицы
 Chart:=WorkBook.Charts.Add(Before:=WorkSheet); //лист диаграммы

 WorkSheet.Cells.Item[1,2]:='Данные 1';
 WorkSheet.Cells.Item[1,3]:='Данные 2';

for row:=2 to 11 do {заполняем случайными данными}
  begin
   //колонка A с текстом
   WorkSheet.Cells.Item[row,1]:='Значение '+IntToStr(row-1);
   //колонка B со случайными значениями
   WorkSheet.Cells.Item[row,2]:=1+Random(10);
   //колонка C со случайными значениями
   WorkSheet.Cells.Item[row,3]:=1+Random(10);
  end;
 //Работа с диаграммой
 Range:=WorkSheet.Range['B2:B11']; //область с данными 1
 SeriesCollection:=Chart.SeriesCollection;//коллекция диаграмм
 SeriesCollection.Add(Source:=Range); //создаем график 1
 Range:=WorkSheet.Range['C2:C11']; //область с данными 2
 SeriesCollection.Add(Source:=Range); //создаем график 2

 Excel_APP.Visible:=True;
finally
  Excel_APP:=Unassigned;
end;
end;
```

В книге Excel создается два листа: лист электронной таблицы WorkSheet и лист диаграммы Chart. В цикле for..do лист электронной таблицы заполняется двумя колонками случайных чисел (по 10 в каждом). На основе данных из этих колонок формируются две гистограммы.

Вновь вернемся к методу Add(), добавляющему график, и обсудим его третий параметр — SeriesLabels. При установке параметра в состояние true график начинает считать, что самая первая ячейка в области Source содержит не числовое значение, а текст — название графика. Обратите внимание на ячейку B1, отображающую текст "Данные 1" (рис. 10). Для того чтобы содержимое данной ячейки стало названием графика, следует внести некоторые доработки в код (листинг 39).

Листинг 39. Доработка листинга 38

```
//Работа с диаграммой
Range:= WorkSheet.Range['B1:B11']; //область с данными 1
SeriesCollection:=Chart.SeriesCollection;//коллекция диаграмм
SeriesCollection.Add(Source:=Range, SeriesLabels:=True);
```

Четвертый (необязательный) параметр CategoryLabels при установке в true информирует график, что в первой колонке (или ряду) области Source находятся не данные, а подписи к оси категорий. Для подключения подписей к оси категорий (горизонтальной оси) диаграммы мы вновь незначительно модифицируем наш пример (листинг 40).

Листинг 40. Подключение подписей к оси категорий

```
 //Работа с диаграммой
 Range:=WorkSheet.Range['A1:B11'];
 SeriesCollection:=Chart.SeriesCollection;
 SeriesCollection.Add(Source:=Range,
                      SeriesLabels:=True,
                      CategoryLabels:=True);
 Range:=WorkSheet.Range['C1:C11'];
 SeriesCollection.Add(Source:=Range, SeriesLabels:=True);
 Excel_APP.Visible:=True;
finally
...
```

Ключевое изменение коснулось описания первого графика. В обновленном листинге 40 область Range охватывает не только данные (диапазон B2:B11), но и столбец с подписями (диапазон A1:A11). Кроме того, в метод Add() направляется дополнительный параметр CategoryLables:=true. Благодаря этому нижняя ось приобрела подписи (см. рис. 10).

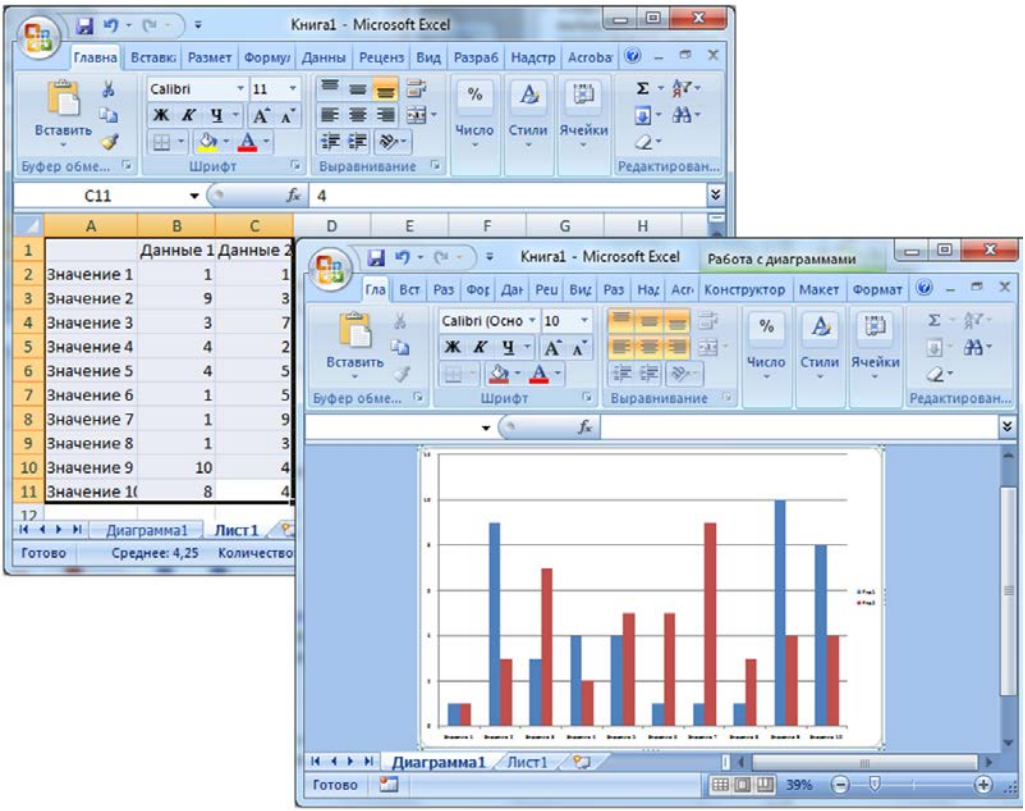


Рис. 10. Создание диаграммы из данных электронной таблицы

Одним из важнейших свойств диаграммы `Chart` считается свойство, определяющее ее тип:

```
Property ChartType : XlChartType;
```

По умолчанию, вновь создаваемая диаграмма имеет тип гистограммы. Но если это вас не устраивает, то нужно более плотно познакомиться с константами `xlChartType` (листинг 41).

Листинг 41. Избранные константы типа диаграммы `xlChartType`

```
{XlChartType - тип диаграмм}
const xlLine=4;    {Линия}
xlLineMarkersStacked=66;{Маркерная}
xl3DLine=-4101;    {3d линия}
xlRadar=-4151;     {Лепестковая}
xlRadarFilled=82;{Лепестковая с залив-
кой}
xlArea=1;    {Область}
xl3DArea=-4098; {3d область}
xlPie=5;           {Круговая}
xlPieExploded=69;{Разрезанная круго-
вая}
xl3DPie=-4102; {3d круговая}
xlBubble=15;     {Пузырьковая}
xlBubble3DEffect=87;{3d пузырьковая}
xlColumnClustered=51;{Гистограмма}
xl3DColumn=-4100;{3d гистограмма}
```

Благодаря перечисленным константам мы сможем получить любой из имеющихся в распоряжении Excel тип диаграммы (рис. 11).

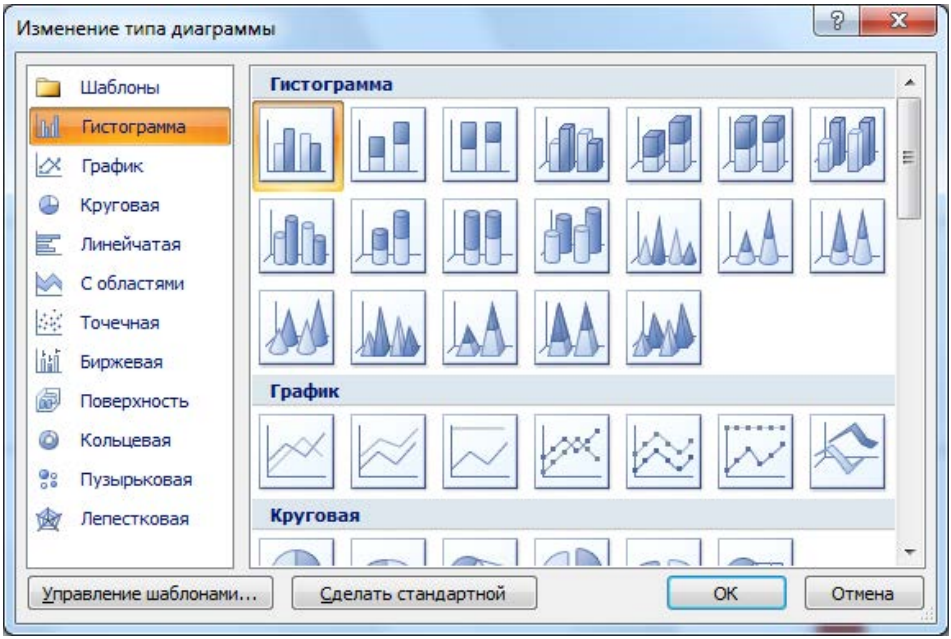


Рис. 11. Стандартный диалог Excel по изменению типа диаграммы

Диаграмма `Chart` включает в свой состав ряд объектов, отвечающих за оформление внешнего вида диаграммы. Для того чтобы добраться до заголовка диаграммы, обратитесь к объекту `ChartTitle` (листинг 42).

Листинг 42. Настройка заголовка диаграммы

```
Chart.HasTitle:=True;                          {включим заголовок}
Chart.ChartTitle.Text:='Название диаграммы'; {назначим название}
Chart.ChartTitle.Font.Color:=clRed;            {цвет шрифта}
Chart.ChartTitle.Shadow:=True;                 {включили тень}
```

Для отображения таблицы с данными диаграммы используйте свойство

```
Property HasDataTable : Boolean
```

Для отказа от вывода легенды установите в `false` свойство

```
Property HasLegend : Boolean;
```

Для того чтобы каждая точка графика приобрела возможность проинформировать пользователя о своих основных характеристиках, диаграмма `Chart` обладает методом

```
Procedure ApplyDataLabels(Type: XlDataLabelsType);
{XlDataLabelsType - тип данных метки}
```

```
const  xlDataLabelsShowNone=-4142;        //Не показывать информацию
       xlDataLabelsShowValue=2;           //Показывать значение
       xlDataLabelsShowPercent=3;         //Показывать долю в процентах
       xlDataLabelsShowLabel=4;           //Название точки
       xlDataLabelsShowLabelAndPercent=5; //Название и доля в процентах
       xlDataLabelsShowBubbleSizes=6;     //Отображать размер
```

Если вы передадите любое значение в диапазоне от 2 до 6, то рядом с точкой будет выведена метка с информацией об этой точке:

```
Chart.ApplyDataLabels(xlDataLabelsShowLabelAndPercent);
```

Область ячеек *Range*

Область `Range` предназначена для организации программного доступа к ячейкам сетки электронной таблицы. Способы обращения к области `Range` вы обнаружите в перечне свойств или методов практически всех ключевых объектов и коллекций Microsoft Excel (`Application`, `Sheets`, `Worksheets`, `Worksheet`). Более того, их может оказаться несколько. Например, у объекта `Worksheet` помимо одноименного свойства `Range` есть весьма сходное свойство `Cells`, способное (в зависимости от способа обращения) возвратить как все ячейки страницы, так и только некоторые из них.

В листинге 43 предложен ряд практических способов доступа к ячейке или объединению ячеек на странице с использованием стилей адресации A1 и R1C1.

Листинг 43. Варианты адресации ячеек

```
Range:=WorkSheet.Cells;          {все ячейки листа}
Range:=WorkSheet.Cells.Item[1,1];{ячейка из 1-го ряда и 1-го столбца}
Range:=WorkSheet.Cells[2,3];{ячейка из 2-го ряда и 3-го столбца}
Range:=WorkSheet.Range['F1'];   {ячейка F1}
Range:=WorkSheet.Range['A3:C6'];{диапазон ячеек от A3 до C6}
Range:=Sheet.Range[Sheet.Cells[1,1],Sheet.Cells[2,2]]; {диапазон от ячейки 1:1
до 2:2}
Range:=WorkSheet.Columns[1];{все ячейки 1-й колонки}
Range:=WorkSheet.Rows[2];   {все ячейки 2-го ряда}
```

Как видите, для обращения к конкретной ячейке листа, нужно воспользоваться свойством листа `Cells`. Если необходимо работать с несколькими ячейками одновременно, то требуется применять свойства, возвращающие область (`Range`, `Rows`, `Columns`).

Замечание

Напомню, что в Excel при обращении к ячейке таблицы с использованием стиля R1C1 первым аргументом выступает номер ряда, вторым — номер колонки.

Для вставки значения любого типа в область воспользуйтесь свойством

```
Property Value : Variant;
```

Несколько примеров вставки значения приведено в листинге 44.

Листинг 44. Варианты заполнения ячейки

```
WorkSheet.Cells[1,1].Value:='Текст'; //текст
WorkSheet.Cells[1,2].Value:=10;      //целое число
WorkSheet.Cells[1,3].Value:=11.1;    //вещественное число
WorkSheet.Cells[1,4].Value:=Date;    //дата
WorkSheet.Cells[1,5].Value:=Time;    //время
WorkSheet.Cells[1,2]:=10;       //сокращенный синтаксис присвоения
```

Формат представления чисел в ячейках электронной таблицы определяется свойством

```
Property NumberFormat : String;
```

В свойстве описывается формат вывода содержимого диапазона. Например, если мы планируем, чтобы число выводилось с точностью до четвертого знака, то примените маску следующего вида:

```
WorkSheet.Cells[1,1].NumberFormat:='0.0000';
```

Эта операция является аналогом вызова окна **Формат ячеек** (рис. 12) в редакторе электронных таблиц Microsoft Excel.

Внимание!

Будьте внимательны, осуществляя операции с неограниченной областью Range (например, если в область входят все ячейки листа). При попытке присвоить свойству Value такой области какое-то значение, Microsoft Excel не остановится, пока не заполнит все ячейки листа.

Основное достоинство Excel заключается в возможности проводить моментальные вычисления в соответствии с формулами, описываемыми пользователем в той или иной ячейке электронной таблицы. Для определения формулы используйте свойство

```
Property Formula : String;
```

Не останавливаясь на особенностях построения формул в Excel (эти знания вы сможете приобрести из справочной системы Microsoft Office), рассмотрим небольшой пример (листинг 45), демонстрирующий способ работы с формулами.

Листинг 45. Применение формул

```
var ..., Worksheet : Variant;
    Row : cardinal;
…
   for Row:=1 to 10 do Worksheet.Cells[Row,2]:=Random(1000);
```

```
  WorkSheet.Range['A11']:='Сумма =';
  WorkSheet.Range['B11'].Formula:='=Sum(B1:B10)';
…
```

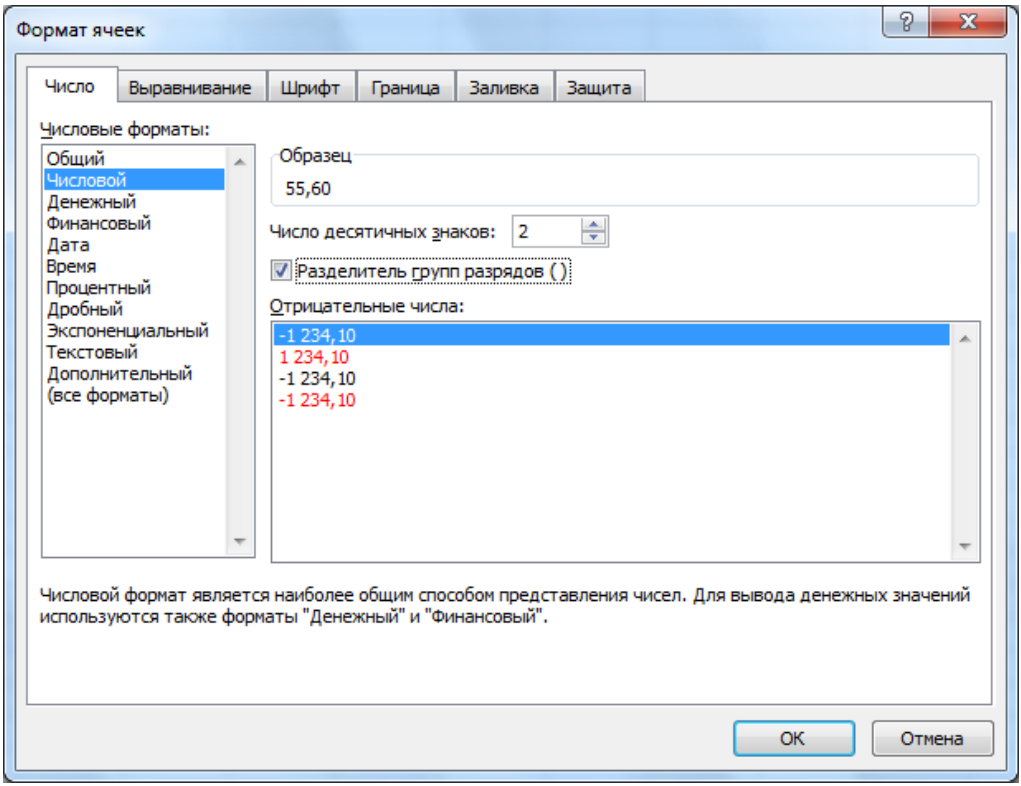


Рис. 12. Форматирование ячеек средствами Excel

В цикле `for … do` мы заполняем вторую колонку электронной таблицы случайными значениями. Затем в ячейку B11 записываем формулу, суммирующую все десять значений.

Если вам более удобен стиль адресации R1C1, то пригодится свойство

```
Property FormulaR1C1 : String;
```

Вот как должна измениться последняя строка листинга 45:

```
WorkSheet.Cells[11,2].FormulaR1C1:='=SUM(R1C2:R10C2)';
```

Слияние ячеек

Для слияния нескольких ячеек в одну нужно воспользоваться свойством

```
Property MergeCells : Boolean;
```

При установке свойства в `true` ячейки области объединяются в одну (листинг 46).

Листинг 46. Слияние ячеек

```
Range:=WorkSheet.Range['B2:D3'];
Range.MergeCells:=true;
```

Результат слияния отражен на рис. 13.

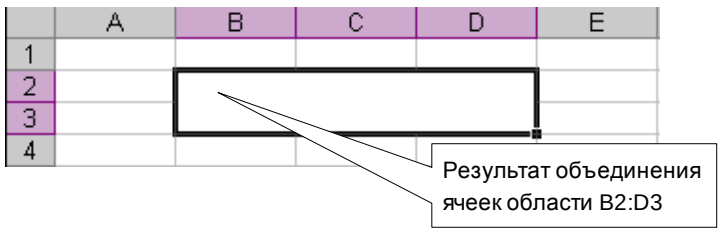


Рис. 13. Слияние ячеек

ВНИМАНИЕ!

Адрес объединенной ячейки соответствует адресу левой верхней ячейки, входящей в состав сливающихся ячеек. Например, адрес ячейки, приведенной на рис. 13, — "B2".

Оформление ячеек

Свойство

```
Property Font : OleVariant;
```

определяет шрифт текста в области `Range`.

Свойства объекта `Font` в Microsoft Excel идентичны свойствам шрифта Microsoft Word.

Объект `Range` предоставляет доступ к областям рядов и колонок электронной таблицы. Все колонки и все ряды из области `Range` мы найдем в свойствах

```
Property Columns : IRange;
Property Rows : IRange;
```

Для того чтобы обратиться к конкретной колонке или ряду, нужно указать ее индекс:

```
WorkSheet.Rows[1].Font.Size:=14;
```

Данная строка кода задает высоту шрифта в первой строке страницы, равную 14 пунктам.

При обращении к конкретной колонке вместо индекса допустимо указывать ее имя:

```
WorkSheet.Columns['AB'].Font.Name := 'Arial';
```

Для индивидуальной настройки ширины колонки или высоты ряда воспользуйтесь свойствами

```
Property ColumnWidth : Real;
Property RowHeight : Real;
```

Ширина и высота измеряются в условных точках.

Пример:

```
WorkSheet.Columns['A'].ColumnWidth := 5;
```

Два свойства

```
Property HorizontalAlignment : xlHAlign;//выравнивание по горизонтали
Property VerticalAlignment : xlVAlign;  //выравнивание по вертикали
```

задают выравнивание текста внутри области (листинг 47).

Пример иллюстрирует листинг 51.

Листинг 47. Константы выравнивания текста

```
{xlHAlign - выравнивание по горизонтали}
const    xlHAlignRight=-4152;
         xlHAlignLeft=-4131;
         xlHAlignJustify=-4130;
         xlHAlignDistributed=-4117;
         xlHAlignCenter=-4108;
         xlHAlignGeneral=1;
         xlHAlignFill=5;
    xlHAlignCenterAcrossSelection=7;
```

```
{xlVAlign - выравнивание по вертикали}
const xlVAlignTop=-4160;
      xlVAlignJustify=-4130;
      xlVAlignDistributed=-4117;
      xlVAlignCenter=-4108;
      xlVAlignBottom=-4107;
```

В листинге 48 приведен пример выравнивания текста внутри области.

Листинг 48. Определение выравнивания текста в 10-м ряду

```
Range:=WorkSheet.Rows[10];
Range.HorizontalAlignment:= xlHAlignLeft;
Range.VerticalAlignment:= xlVAlignCenter;
```

В Microsoft Excel за описание обрамления ячеек отвечает объект `Border`, входящий в состав коллекции `Borders`:

```
Property Borders : IBorders;
```

Добравшись до границ ячеек, мы приобретем достаточно широкие возможности по дизайну электронной таблицы, например, стиль линии задает свойство

```
Property LineStyle : xlLineStyle;
```

Перечень значений констант содержит листинг 49.

Листинг 49. Перечень значений констант при задании стиля линии

```
const xlLineStyleNone=-4142; {без линии}
      xlDouble=-4119; {двойная}
      xlDot=-4118;    {точка}
      xlDash=-4115;   {тире}
      xlContinuous=1;  {обычная}
      xlDashDot=4;{тире-точка}
      xlDashDotDot=5;{тире-точка-точка}
      xlSlantDashDot=13; {наклонное тире-точка}
```

Свойство

```
Property Color : ColorRef;
```

определяет цвет линии.

Свойство

```
Property Weight : Integer;
```

задает толщину линии.

Коллекция `Borders` включает в себя восемь элементарных объектов — `Border`. Они практически идентичны границам в Word (см. рис. 7), единственное отличие в том, что в Excel они описываются другими константами — `xlBordersIndex` (листинг 50).

Листинг 50. Перечень констант при задании границ линии

```
{XlBordersIndex – границы области}
const  xlDiagonal-
Down=5;{Диагональ}
  xlDiagonalUp=6; {Диагональ}
  xlEdgeLeft=7; {Левая граница}
  xlEdgeTop=8; {Верхняя граница}
  xlEdgeBottom=9; {Нижняя граница}
  xlEdgeRight=10; {Правая граница}
  xlInsideVertical=11; {Внутренняя вертикаль}
  xlInsideHorizontal=12;{Внутренняя горизонталь}
```

Для того чтобы обратиться к намеченной границе, необходимо передать ее константу коллекции `IBorders`. Например, код из листинга 51 проводит линию из точек по нижней границе области A1:D1.

Листинг 51. Работа с границей области

```
Border:=WorkSheet.Range['A1:D1'].Borders[xlEdgeBottom];
Border.LineStyle:= xlDouble;
Border.Color:=RGB(0,255,0);
```

Заливка области — *Interior*

За организацию заливки области отвечает специализированный объект `Interior`. Для того чтобы к нему обратиться, воспользуйтесь одноименным свойством области `Range`:

```
Property Interior : OleVariant;
```

С заливкой области связаны свойства, определяющие цвет фона ячейки:

```
Property Color : ColorRef;
```

Цвет заливки:

```
Property PatternColor : ColorRef;
```

И способ заливки:

```
Property Pattern : XlPattern; {xlPattern - заливка на основе узоров}
```

В листинге 52 приведен перечень значений констант при заливке.

Листинг 52. Значения констант при заливке

```
const xlPatternVertical=-4166;
      xlPatternUp=-4162;
      xlPatternNone=-4142;
      xlPatternHorizontal=-4128;
      xlPatternGray75=-4126;
      xlPatternGray50=-4125;
      xlPatternGray25=-4124;
      xlPatternDown=-4121;
      xlPatternAutomatic=- 4105;
      xlPatternChecker=-9;
      xlPatternSolid=1;
      xlPatternSemiGray75=10;
      xlPatternLightHorizontal=11;
      xlPatternLightVertical=12;
      xlPatternLightDown=13;
      xlPatternLightUp=14;
      xlPatternGrid=15;
      xlPatternCrissCross=16;
      xlPatternGray16=17;
      xlPatternGray8=18;
```

Свойство `Pattern` определяет особенности узора, применяемые при закрашивании области (листинг 53).

Листинг 53. Закрашивание ячейки 1:1

```
Range:=WorkSheet.Cells[1,1];
Range.Interior.Color:=clRed;
Range.Interior.PatternColor:=clGreen;
Range.Interior.Pattern:= xlPatternVertical;
```

Усовершенствование универсального генератора отчетов

Завершив теоретическую часть, посвященную работе с электронными таблицами, предлагаю достать с пыльной полки наш универсальный генератор отчетов и дополнить его одним полезным методом. Это процедура `CreateExcelReport()`, умеющая готовить отчеты в Excel (листинг 54).

Листинг 54. Подготовка отчета в Excel

```
procedure TForm1.CreateExcelReport(TableName:String);
var Excel_App, WorkBook, WorkSheet, Range : OleVariant;
    aRow, aCol : integer;
begin
if GetAutomationServer('Excel.Application',Excel_App)=false then exit;
       ADOQuery1.Close; {Готовим запрос}
       ADOQuery1.SQL.Clear;
       ADOQuery1.SQL.Add('Select * from '+TableName);
       ADOQuery1.Open;
    try
     Excel_App.DisplayAlerts:=false;     {отключаем вывод предупреждений}
     WorkBook:=Excel_App.WorkBooks.Add;   {добавляем новую книгу}
     WorkSheet:=WorkBook.Sheets.Item[1];  {выбираем первый лист}
     WorkSheet.Name:=TableName;           {переименуем лист}

    {из имен полей формируем заголовки колонок}
     for aCol:=1 to ADOQuery1.FieldCount do
                  WorkSheet.Cells.Item[1,aCol]:=
                              ADOQuery1.Fields[aCol-1].DisplayLabel;
      WorkSheet.Rows[1].Font.Size:=12;    //выделим заголовки шрифтом
      WorkSheet.Rows[1].Font.Color:=clRed;
      aRow:=2;

      while ADOQuery1.Eof=False do
      begin {перебираем записи таблицы и передаем их в сетку}
        for aCol:=1 to ADOQuery1.FieldCount do {перебор колонок}
         begin
          if ADOQuery1.Fields[aCol-1].IsBlob=False then
            WorkSheet.Cells.Item[aRow,aCol]:=
                           ADOQuery1.Fields[aCol-1].AsString
               else {не пропускаем BLOB поля}
               WorkSheet.Cells.Item[aRow,aCol]:='BLOB';
            end;

        Inc(aRow);      //приращение счетчика строк
        ADOQuery1.Next; //переход к очередной строке
      end;
{Включим показ границ ячеек}
  Range:=WorkSheet.Range[WorkSheet.Cells[1,1],
                         WorkSheet.Cells[aRow-1,ADOQuery1.FieldCount]];
  Range.Borders.Color:=clBlack;
```

```
  Excel_App.Visible:=true;
Finally
  Excel_App := Unassigned;
  ADOQuery1.Close;
end;
end;
```

Нам осталось добавить к проекту новую кнопку и подготовить код для вызова процедуры (листинг 55).

Листинг 55. Вызов процедуры `CreateExcelReport`

```
procedure TForm1.btnExcelReportClick(Sender: TObject);
var TableName:string;
begin
Screen.Cursor:=crHourgLass;
  TableName:=lboxTableNames.Items.Strings[lboxTableNames.ItemIndex];
  GetExcelReport(TableName);
Screen.Cursor:=crDefault;
end;
```

Универсальный генератор отчетов завершен.

Резюме

Умение работать с объектами автоматизации позволяет программисту использовать в своих проектах все лучшее из программных продуктов сторонних разработчиков. Технологию автоматизации поддерживают все приложения Microsoft Office, а также все ведущие производители программного обеспечения, в частности, такие известные корпорации, как Corel и Adobe. Названные фирмы доминируют на рынке программного обеспечения и установлены на большинстве персональных компьютеров, значит, мы не столкнемся с особыми трудностями при распространении своих проектов.