АНАТОЛИЙ ХОМОНЕНКО
ВЛАДИМИР ГОФМАН

РАБОТА С БАЗАМИ ДАННЫХ В DELPHI

3-Е ИЗДАНИЕ

ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

ТЕХНОЛОГИИ BDE, ADO, dbExpress И INTERBASE EXPRESS

РАБОТА С ЛОКАЛЬНЫМИ И УДАЛЕННЫМИ БАЗАМИ ДАННЫХ

ПРОГРАММИРОВАНИЕ НА SQL

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Анатолий Хомоненко
Владимир Гофман

РАБОТА С БАЗАМИ ДАННЫХ В DELPHI

3-Е ИЗДАНИЕ

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.06
ББК 32.973.26-18.2
Х76

Хомоненко А. Д., Гофман В. Э.

Х76 Работа с базами данных в Delphi. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2005. — 640 с.: ил.

ISBN 5-94157-361-8

Рассматривается использование средств Delphi 7 для разработки приложений баз данных. Даются понятия баз данных, характеризуются элементы и описываются этапы проектирования реляционных баз данных, изложена технология разработки информационных систем, освещаются приемы работы с данными, создание таблиц и приложений баз данных, подготовка отчетов. Рассматриваются навигационный и реляционный способы доступа к данным с помощью технологий BDE, ADO, dbExpress и Interbase Express, основы программирования на SQL. Показывается использование локальных и удаленных баз данных, включая создание многоуровневых информационных систем и публикацию баз данных в Интернете. Также рассматриваются возможности Delphi 2005 по работе с базами данных. Благодаря подробному изложению тем и большому числу примеров книга может служить практическим руководством по работе с базами данных.

Для программистов

УДК 681.3.06
ББК 32.973.26-18.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 01.02.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 51,6.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04
от 11.11.2004 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 5-94157-361-8

© Хомоненко А. Д., Гофман В. Э., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Предисловие **1**

Часть I. Основы работы с базами данных **5**

Глава 1. Основные понятия баз данных **7**

Банки данных 7
Модели данных 8
Базы данных и приложения 9
BDE 10
ADO 11
dbExpress 11
Варианты архитектуры для BDE 11

Глава 2. Реляционные базы данных и средства работы с ними **16**

Элементы реляционной базы данных 16
Таблицы баз данных 16
Ключи и индексы 19
Методы и способы доступа к данным 21
Связь между таблицами 23
Механизм транзакций 27
Бизнес-правила 28
Словарь данных 29
Таблицы форматов dBase и Paradox 29
Средства для работы с реляционными базами данных 34
Инструменты 34
Компоненты 35
Исключения баз данных 40

Глава 3. Проектирование баз данных **43**

Нормализация базы данных 43
Избыточность данных и аномалии 44
Приведение к нормальным формам 46
Первая нормальная форма 47
Вторая нормальная форма 48
Третья нормальная форма 49
Средства CASE 51

Глава 4. Технология создания информационной системы 55

Создание таблиц базы данных 55
Описание полей 58
Задание индексов 60
Задание ограничений на значения полей 62
Задание ссылочной целостности 65
Задание паролей 67
Задание языкового драйвера 69
Задание таблицы для выбора значений 70
Просмотр списка подчиненных таблиц 74
Изменение структуры таблицы 74
Создание приложения BDE 75
Использование модуля данных 77

Глава 5. Компоненты доступа к данным 81

Наборы данных 81
Состояния наборов данных 84
Режимы наборов данных 88
Доступ к полям 90
Особенности набора данных *Table* 92
Особенности набора данных *Query* 100
Объекты поля 105
Редактор полей 107
Операции с полями 115
Доступ к значению поля 116
Проверка типа и значения поля 119
Форматирование отображаемого значения поля 124
Источник данных 127

Часть II. Технологии доступа к данным 131

Глава 6. Визуальные компоненты для работы с данными 133

Отображение и редактирование значения логического поля 135
Отображение и выбор значения поля 136
Отображение и выбор значения поля в списке 138
Простой и комбинированный списки 138
Списки, сформированные по значениям поля набора данных 139
Представление записей в табличном виде 139
Характеристики сетки 140
Столбцы сетки 143
Использование модифицированной сетки 149

Использование навигационного интерфейса ... 152
Вывод графических изображений ... 154
Построение диаграмм ... 158

Глава 7. Навигационный доступ к данным с помощью BDE ... 164

Операции с таблицей БД ... 165
Создание, удаление и переименование ... 165
Установка уровня доступа ... 168
Сортировка набора данных ... 169
Навигация по набору данных ... 171
Перемещение по записям ... 172
Переход по закладкам ... 181
Фильтрация записей ... 184
Фильтрация по выражению ... 185
Фильтрация по диапазону ... 192
Навигация с псевдофильтрацией ... 196
Поиск записей ... 197
Поиск в наборах данных ... 197
Поиск по индексным полям ... 204
Модификация набора данных ... 206
Редактирование записей ... 208
Добавление записей ... 214
Удаление записей ... 216
Пример формы приложения ... 217
Работа со связанными таблицами ... 224
Пример приложения ... 225
Использование механизма транзакций ... 234

Глава 8. Реляционный доступ к данным с помощью BDE ... 236

Основные сведения о языке SQL ... 237
Функции языка ... 239
Определение данных ... 239
Создание и удаление таблицы ... 240
Изменение состава полей таблицы ... 242
Создание и удаление индекса ... 243
Отбор данных из таблиц ... 244
Описание инструкции *SELECT* ... 244
Управление полями ... 245
Простое условие отбора записей ... 249
Сложные критерии отбора записей ... 252
Группирование записей ... 253
Сортировка записей ... 254
Соединение таблиц ... 257

Модификация записей ... 259
Редактирование записей ... 259
Вставка записей ... 260
Удаление записей ... 262
Статический и динамический запросы ... 262

Глава 9. Технология dbExpress ... 266

Общая характеристика ... 266
Установление соединения с сервером ... 267
Компоненты доступа к данным ... 271
Универсальный доступ к данным ... 272
Просмотр таблиц ... 277
Выполнение SQL-запроса ... 277
Выполнение хранимых процедур ... 278
Компонент редактирования набора данных ... 279
Отладка соединения с сервером ... 282

Глава 10. Технология ADO ... 284

Общая характеристика ... 284
Установление соединения ... 286
Управление соединением и транзакциями ... 289
Компоненты доступа к данным ... 291
Доступ к таблицам ... 293
Выполнение запросов ... 294
Вызов хранимых процедур ... 294
Компонент *ADODataSet* ... 295
Команды ADO ... 295
Пример приложения ... 297

Глава 11. Создание и просмотр отчетов с помощью Rave Reports ... 301

Характеристика генератора отчетов ... 301
Визуальное конструирование отчетов ... 302
Интерфейс визуального конструктора ... 302
Состав проекта отчетов ... 303
Редактор событий ... 306
Компоненты, представленные на многостраничной панели инструментов ... 307
Компоненты отображения данных ... 309
Компоненты управления отчетом ... 312
Компонент-проект отчета ... 312
Компонент управления отчетом ... 313

Компоненты установления соединения ... 313
Схема управления отчетом и подсоединения данных ... 314
Примеры создания и просмотра отчетов ... 316
Предварительный просмотр отчета ... 317
Простой отчет приложения базы данных ... 319

Глава 12. Инструменты ... 322

Программа BDE Administrator ... 322
Работа с псевдонимами ... 323
Параметры драйвера ... 326
Системные установки ... 329
Использование конфигурационных файлов ... 330
Программа Database Desktop ... 331
Редактирование записей таблиц ... 332
Работа с псевдонимами ... 333
Работа с SQL-запросами ... 333
Визуальное конструирование запросов ... 335
Отбор записей из таблицы ... 336
Редактирование записей ... 338
Вставка и удаление записей ... 339
Связывание таблиц ... 341
Программа SQL Builder ... 342
Программа SQL Explorer ... 349
Программа Data Pump ... 351

Часть III. Удаленные базы данных ... 355

Глава 13. Введение в работу с удаленными базами данных ... 357

Основные понятия ... 357
Архитектура "клиент-сервер" ... 358
Сервер и удаленная БД ... 359
Средства работы с удаленными БД ... 359
Сервер InterBase ... 361
Бизнес-правила ... 362
Организация данных ... 363
Запуск сервера ... 364
Особенности приложения ... 365
Соединение с базой данных ... 365
Соединение с базой из программы IBConsole ... 366
Компонент *Database* ... 366
Компонент *Session* ... 371
Соединение с базой данных из приложения ... 374

Глава 14. Работа с удаленными базами данных 377

Создание базы данных 377
Управление структурой таблиц 380
Описание столбца 383
Ограничения столбца 383
Описание ключей 387
Определение ограничений ссылочной целостности 388
Использование индексов 390
Использование доменов 391
Использование просмотров 392
Хранимые процедуры 393
Создание и удаление хранимых процедур 394
Язык хранимых процедур 395
Виды хранимых процедур 401
Вызов хранимой процедуры выбора 401
Вызов хранимой процедуры действия 403
Использование триггеров 405
Создание и изменение триггера 406
Примеры использования триггера 407
Создание генераторов 409
Использование функций, определяемых пользователем 410
Реализация механизма транзакций 412
Использование механизма кэшированных изменений 414
Использование компонентов *Database*, *Table* и *Query* 415
Использование компонента *UpdateSQL* 419
Использование механизма событий сервера 422
Управление привилегиями 425
Манипулирование данными 427

Глава 15. Технология InterBase Express 430

Общая характеристика 430
Установление соединения с сервером 431
Управление транзакциями 432
Компоненты доступа к данным 434
Генераторы для автоинкрементных полей 435
Доступ к таблицам 436
Выполнение запросов 436
Получение и редактирование данных 437
Компонент *IBSQL* 441
Пример приложения 442

Глава 16. Инструменты для работы с удаленными базами данных 446

Программа IBConsole 446
Управление сервером 447

Подключение к серверу ... 447
Регистрация сервера ... 449
Просмотр протокола работы сервера ... 450
Операции с сертификатами ... 450
Управление пользователями ... 451
Управление БД ... 451
Регистрация базы данных ... 452
Подключение базы данных ... 452
Создание базы данных ... 453
Просмотр метаданных ... 453
Сбор мусора ... 454
Проверка состояния базы данных ... 454
Анализ статистики ... 455
Сохранение и восстановление базы данных ... 456
Интерактивное выполнение SQL-запросов ... 460
Программа SQL Monitor ... 463

Глава 17. Трехуровневые приложения ... 466

Принципы построения трехуровневых приложений ... 466
Сервер приложений ... 468
Приложение клиента ... 473

Часть IV. Базы данных и Интернет ... 483

Глава 18. Основные элементы интернет-технологий ... 485

Сценарии JavaScript, JScript и VBScript ... 486
Элементы управления ActiveX ... 487
Апплеты и сервлеты Java ... 488
Интерфейсы CGI и WinCGI ... 488
Интерфейсы ISAPI/NSAPI ... 489
Страницы ASP, PHP и IDC/HTX ... 489
Формирование Web-страниц ... 490
Интерфейсы OLE DB, ADO, ODBC ... 491
Статическая публикация БД ... 492
Динамическая публикация БД ... 492
Web-приложения ... 493
Протоколы передачи данных ... 494
Универсальный указатель ресурсов (URL) ... 495
Язык разметки гипертекста HTML ... 495
Расширяемый язык разметки XML ... 496
Программа XML Mapper ... 504

Глава 19. Web-приложения и интерфейсы 508

Характеристика Web-приложений 508
Web-приложения в сетях интранет 509
Web-приложения с модулями расширения серверной части 511
Web-приложения с модулями расширения клиентской части 513
Архитектура Web-приложений, публикующих БД 514
Двухуровневые Web-приложения 515
Трехуровневые Web-приложения 516
Интерфейсы Web-приложений 517
Общий интерфейс взаимодействия CGI 517
Переменные окружения 520
Стандартный вывод 522
Интерфейс программирования серверных приложений ISAPI 522

Глава 20. Публикация баз данных средствами Delphi 526

Компоненты, используемые при разработке Web-приложений 526
Статическая публикация 528
Компоненты генерации HTML-страниц 533
Компонент *PageProducer* 533
Компонент *DataSetPageProducer* 534
Компонент *DataSetTableProducer* 534
Компонент *QueryTableProducer* 537
Пример генератора HTML-страниц 537
Динамическая публикация 545
Создание модуля CGI 545
Создание ISAPI-модуля расширения сервера 554
Обработка пользовательского ввода в модуле ISAPI 560
Публикация графики 562
Использование технологии ADO 569

Глава 21. Работа с электронной почтой 572

Использование функции *ShellExecute* 572
Использование интерфейса MAPI 573

Глава 22. Характеристика Web-служб 579

Основные понятия 579
Документ WSDL 580
Вызываемый интерфейс 584
Страница *WebServices* Палитры компонентов 584
Схема взаимодействия клиента и сервера 588
Разработка клиента для Web-службы 589
Импортирование документа WSDL 589
Обращение к вызываемому интерфейсу 591
Пример создания клиента Web-службы 594

Часть V. Delphi 2005 597

Глава 23. Работа с БД в Delphi 2005 599

Общая характеристика Delphi 2005 599
Характеристика VCL для .NET 601
Разработка приложения VCL Forms 602
Характеристика способов 602
Приложения БД VCL Forms для .NET 603
Приложения БД VCL Forms для Win32 604
Технология ADO.NET 604
Общая характеристика 604
Схема доступа к данным 607
ADO.NET в Delphi 2005 607
Провайдеры данных для .NET 608
Приложение Windows Forms с ADO.NET и BDP.NET 611

Приложение.
Фрагменты иерархии классов VCL 616

Предметный указатель 619

Предисловие

Система визуального программирования Delphi обладает большой популярностью среди широкого круга пользователей: от неспециалистов до системных программистов, занимающихся разработкой сложных приложений и информационных систем.

Delphi позволяет быстро и удобно разрабатывать эффективные приложения, включая приложения для работы с базами данных. Система имеет развитые возможности по созданию пользовательского интерфейса, широкий набор функций, методов и свойств для решения прикладных расчетно-вычислительных задач. В системе имеются развитые средства отладки, облегчающие разработку приложений.

Традиционно Delphi относят к системам быстрой разработки приложений. Вместе с тем эта система обладает практически всеми возможностями современных СУБД, таких как Microsoft Access и Visual FoxPro. Она позволяет удобно создавать приложения с помощью инструментальных программных средств, визуально подготавливать запросы к базам данных, а также непосредственно писать SQL-запросы к базам данных.

Delphi позволяет создавать приложения для работы с локальными и удаленными базами данных, включая публикацию баз данных в Интернете. Применительно к работе с базами данных Delphi обеспечивает широкий набор инструментальных средств, поддерживает современные технологии, в том числе многоуровневую технологию "клиент-сервер".

Книга посвящена рассмотрению технологий разработки приложений для работы с базами данных и Интернетом в Delphi 7. В книге описываются основные компоненты, свойства, методы и события для работы с базами данных. Приводятся примеры реально работающих программ, которые читатель может использовать в своих разработках.

При написании книги мы и использовали версию Enterprise системы Delphi 7, как предоставляющую наибольшие возможности. Другие версии (Professional и Persona) могут отличаться, например, составом Хранилища объектов или значками объектов, в том числе форм.

По существу, книга представляет собой исправленное и существенно дополненное (по отношению ко 2-му изданию тех же авторов) произведение по работе с базами данных, которое доработано применительно к работе в среде Delphi 7 и снабжено материалами, характеризующими разработку приложений для баз данных в Delphi 2005.

Ее отличия от предыдущего издания в основном заключаются в освещении ряда новых средств и технологий, появившихся или получивших дальнейшее развитие в системе (технологий dbExpress и ADO работы с базами данных, технологии InterBase Express работы с сервером баз данных InterBase и др.), описании

техники работы с новым генератором отчетов Rave 5.0, знакомстве с основами построения и использования Web-служб, а также в уточнении состава визуальных компонентов системы и др.

Книга ориентирована на широкий круг читателей — от подготовленных пользователей до специалистов по программированию.

Ниже перечислены наиболее важные, на наш взгляд, новые средства и возможности системы Delphi 7 по работе с базами данных и Интернетом.

- Технологии работы с базами данных:
  - драйверы dbExpress обновлены для поддержки работы с Informix SE, Oracle 9i, DB2 7.2, InterBase 6.5 и MySQL 3.23.49; имеется новый драйвер для MS SQL 2000;
  - имеется несколько новых и измененных компонентов для работы с базами данных; вместо SQL Links для доступа к базам данных SQL Server рекомендуется использовать dbExpress;
  - можно задать специальный модуль данных SOAP в сервере приложений, имеющий многочисленные модули данных (свойство `SOAPServerIID` или добавление интерфейса модулей данных в конец URL); использовать компонент-соединение SOAP для вызова расширений интерфейса серверов приложения (свойство `SOAPServerIID` и метод `GetSOAPServer`).
- Технология CORBA: появилась новая версия комилятора IDL2PAS для разработки CORBA-приложений, рекомендуемая для использования вместо старых средств, интегрированных с поддержкой технологии COM.
- Web-технологии:
  - в Delphi входит IntraWeb из состава AtoZed Software; его можно использовать для разработки серверных Web-приложений с помощью стандартных средств формы, а также для разработки страниц приложений Web Broker и WebSnap;
  - Delphi поддерживает Apache 2 как целевой тип для Web Broker, WebSnap и SOAP (Simple Object Access Protocol); вместо интерфейса Win-CGI рекомендуется использование CGI, ISAPI/NSAPI или Apache в качестве целевого типа;
  - новый UDDI-обозреватель позволяет размещать и импортировать WSDL-документы, учитываемые в журналах регистрации UDDI;
  - новые классы и интерфейсы позволяют читать или вставлять заголовки в конверты SOAP, передающие сообщения между клиентами и серверами; клиентские и серверные приложения Web-служб получили возможность управлять прикрепленными сообщениями;
  - предоставляются большие возможности по управлению взаимодействием между вызываемым интерфейсом и документом WSDL на основе использования публикуемых событий;

- новый интерфейс IRIOAccess позволяет осуществлять доступ к удаляемому объекту интерфейса, который реализует вызываемый интерфейс.

Для полноты картины приводим также новые средства и возможности Delphi 2005 по работе с базами данных.

- в части провайдеров доступа к данным BDP.NET для обеспечения новых свойств и улучшения их поддержки:
  - добавлен диалог для хранимых процедур;
  - теперь BDP.NET включает пространство имен для поддержки Sybase как провайдера;
  - многие обновления сделаны для улучшения надежности и производительности провайдеров;
  - обеспечивается возможность программно перемещать данные от одного провайдера к другому;
  - добавлены классы, позволяющие связывать приложение БД с многими провайдерами;
  - для `BdpDataAdapter` добавлена поддержка схемы таблицы и полей.
- применительно к VCL для поддержки .NET по работе с базами данных:
  - BDE для .NET обновлен для обеспечения динамической загрузки DLL без указания пути, для улучшения производительности управления объектами типа BLOB и для включения классов, ранее отсутствовавших в .NET;
  - обновления для технологии dbExpress;
  - для DataSnap импортирован ряд компонентов .NET, связанных с установлением соединений.
- в интересах упрощения проектирования и разработки BDP.NET-приложений для БД усилены возможности Проводника кода (Data Explorer):
  - поддерживается перенос данных от одного провайдера к другому;
  - хранимые процедуры можно перетаскивать (Drag-and-drop) от одного провайдера в Проводнике кода на форму в Конструкторе;
  - обеспечивается множество сервисов метаданных, что позволяет просматривать и модифицировать схему базы данных.

Авторы[1]

[1] Материалы 9—11, 15, 22 и 23 глав подготовлены А. Хомоненко; 18—20 глав — Е. Мещеряковым и А. Хомоненко при участии В. Гофмана. Остальные главы подготовлены В. Гофманом и А. Хомоненко совместно. Исправление и дополнение материалов книги применительно к версии Delphi 7/2005 выполнил А. Хомоненко.

ЧАСТЬ I

Основы работы с базами данных

Глава 1

Основные понятия баз данных

Часто для успешного функционирования различным организациям требуется развитая информационная система, реализующая автоматизированный процесс сбора, манипулирования и обработки данных.

Банки данных

Современной формой информационных систем являются *банки данных*, имеющие в своем составе:

- вычислительную систему;
- систему управления базами данных (СУБД);
- одну или несколько баз данных (БД);
- набор прикладных программ (приложений БД).

База данных (БД) обеспечивает хранение информации, а также удобный и быстрый доступ к данным. Она представляет собой совокупность данных различного характера, организованных по определенным правилам. Информация в БД должна быть:

- непротиворечивой;
- неизбыточной;
- целостной.

Система управления базой данных (СУБД) — это совокупность языковых и программных средств, предназначенных для создания, ведения и использования БД. По характеру применения СУБД разделяют на персональные и многопользовательские.

Персональные СУБД обеспечивают возможность создания локальных БД, работающих на одном компьютере. К персональным СУБД относятся Paradox, dBase, FoxPro, Access и др.

Замечание

СУБД Access 97/2000/2002/2003 также обеспечивают возможность многопользовательского доступа к данным.

Многопользовательские СУБД позволяют создавать информационные системы, функционирующие в архитектуре "клиент-сервер". Наиболее известными многопользовательскими СУБД являются Oracle, Informix, SyBase, Microsoft SQL Server, InterBase.

В состав языковых средств современных СУБД входят:

- язык описания данных, предназначенный для описания логической структуры данных;
- язык манипулирования данными, обеспечивающий выполнение основных операций над данными — ввод, модификацию и выборку;
- язык структурированных запросов (SQL, Structured Query Language), обеспечивающий управление структурой БД и манипулирование данными, а также являющийся стандартным средством доступа к удаленным БД;
- язык запросов по образцу (QBE, Query By Example), обеспечивающий визуальное конструирование запросов к БД.

Прикладные программы, или *приложения*, служат для обработки данных, содержащихся в БД. Пользователь осуществляет управление БД и работу с ее данными именно с помощью приложений, которые также называют *приложениями БД*.

Иногда термин "база данных" трактуют в более широком смысле и обозначают им не только саму БД, но и приложения, обрабатывающие ее данные.

Замечание

Хотя система Delphi и не является СУБД в буквальном смысле этого слова, тем не менее, она обладает вполне развитыми возможностями СУБД. Предоставляемые Delphi средства обеспечивают создание и ведение локальных и клиент-серверных БД, а также разработку приложений для работы практически с любыми БД. Назвать систему Delphi обычной СУБД мешает, наверное, только то, что, с одной стороны, у нее нет "своего" формата таблиц (языка описания данных), поэтому она использует форматы таблиц других СУБД, например, dBase, Paradox или InterBase (это вряд ли является недостатком, поскольку названные форматы хорошо себя зарекомендовали); с другой стороны, в плане создания приложений различного назначения, в том числе приложений БД, возможности Delphi не уступают возможностям специализированных СУБД, а зачастую и превосходят их.

Модели данных

База данных содержит данные, используемые какой-либо прикладной информационной системой (например, системами "Сирена" или "Экспресс" продажи авиа- и железнодорожных билетов).

В зависимости от вида организации данных различают следующие основные модели представления данных в базе:

- иерархическую;
- сетевую;
- реляционную;
- объектно-ориентированную.

В *иерархической* модели данные представляются в виде древовидной (иерархической) структуры. Подобная организация данных удобна для работы с иерархически упорядоченной информацией, однако при оперировании данными со сложными логическими связями иерархическая модель оказывается слишком громоздкой.

В *сетевой* модели данные организуются в виде произвольного графа. Недостатком сетевой модели является жесткость структуры и высокая сложность ее реализации.

Кроме того, значительным недостатком иерархической и сетевой моделей является то, что структура данных задается на этапе проектирования БД и не может быть изменена при организации доступа к данным.

В *объектно-ориентированной* модели отдельные записи базы данных представляются в виде объектов. Между записями базы данных и функциями их обработки устанавливаются взаимосвязи с помощью механизмов, подобных соответствующим средствам объектно-ориентированных языков программирования. Объектно-ориентированные модели сочетают особенности сетевой и реляционной моделей и используются для создания крупных БД со сложными структурами данных.

Реляционная модель, предложенная в 70-х годах XX века сотрудником фирмы IBM Эдгаром Коддом, получила название от английского термина relation (отношение). Реляционная БД представляет собой совокупность таблиц, *связанных отношениями.* Достоинствами реляционной модели данных являются простота, гибкость структуры, удобство реализации на компьютере, наличие теоретического описания. Большинство современных БД для персональных компьютеров являются реляционными. При последующем изложении материала речь пойдет именно о реляционных БД.

Базы данных и приложения

В зависимости от взаимного расположения приложения и БД можно выделить:

- локальные БД;
- удаленные БД.

Для выполнения операций с локальными БД разрабатываются и используются так называемые *локальные приложения*, а для операций с удаленными БД — *клиент-серверные приложения.*

Расположение БД в значительной степени влияет на разработку приложения, обрабатывающего содержащиеся в этой базе данные.

Так, различают следующие виды приложений:

- приложения, использующие локальные базы данных, называют *одноуровневыми* (однозвенными) приложениями, поскольку приложение и базы данных образуют единую файловую систему;
- приложения, использующие удаленные базы данных, разделяют на двухуровневые (двухзвенные) и многоуровневые (многозвенные). *Двухуровневые* приложения содержат клиентскую и серверную части;
- *многоуровневые* (обычно трехуровневые) приложения кроме клиентской и серверной частей имеют дополнительные части. К примеру, в трехуровневых приложениях имеются клиентская часть, сервер приложений и сервер базы данных.

Одно- и двухуровневые приложения Delphi могут осуществлять доступ к локальным и удаленным БД с использованием следующих механизмов:

- BDE (Borland Database Engine — процессор баз данных фирмы Borland), предоставляющий развитый интерфейс API для взаимодействия с базами данных;
- ADO (ActiveX Data Objects — объекты данных ActiveX) осуществляет доступ к информации с помощью OLE DB (Object Linking and Embedding Data Base — связывание и внедрение объектов баз данных);
- dbExpress обеспечивает быстрый доступ к информации в базах данных с помощью набора драйверов;
- InterBase реализует непосредственный доступ к базам данных InterBase.

Выбор варианта технологии доступа к информации в базах данных, кроме прочих соображений, определяется с учетом удобства подготовки разработанного приложения к распространению, а также дополнительного расхода ресурсов памяти. К примеру, инсталляция для BDE требует примерно 15 Мбайт внешней памяти на диске и настройки псевдонимов используемых баз данных. Вариант InterBase вряд ли можно назвать конкурентоспособным, поскольку он ориентирован строго на работу с одноименным сервером баз данных.

Трехуровневые приложения Delphi 7 можно создавать с помощью механизма DataSnap. Используемые при создании трехуровневых (многоуровневых) приложений баз данных компоненты расположены на страницах **DataSnap** и **Data Access** Палитры компонентов.

BDE

BDE представляет собой совокупность динамических библиотек и драйверов, обеспечивающих доступ к данным. Процессор BDE должен устанавливаться на всех компьютерах, на которых выполняются Delphi-приложения, осуществляющие работу с БД. Приложение через BDE передает запрос к базе данных, а обратно получает требуемые данные. Механизм BDE до 7-й версии системы Del-

phi получил самое широкое распространение ввиду широкого спектра предоставляемых им возможностей. Идеологи фирмы Borland планируют отказаться от его поддержки, заменив его другими механизмами. Мы приводим множество примеров и описание технологии применения BDE для работы с базами данных в связи с тем, что накоплено большое количество приложений с использованием этого подхода.

ADO

Механизм ADO доступа к информации базы данных является стандартом фирмы Microsoft. Использование этой технологии подразумевает использование настраиваемых провайдеров данных. Технология ADO обеспечивает универсальный механизм доступа из приложений к информации источников данных. Эта технология основана на стандартных интерфейсах COM, являющихся системным механизмом Windows. Это позволяет удобно распространять приложения баз данных без вспомогательных библиотек.

dbExpress

Механизм доступа dbExpress подразумевает использование совокупности драйверов, компонентов, инкапсулирующих соединения, транзакций, запросов, наборов данных и интерфейсов, с помощью которых обеспечивается универсальный доступ к функциям этого механизма. Обеспечение взаимодействия с серверами баз данных по технологии dbExpress основано на использовании специализированных драйверов. Последние для получения данных применяют запросы SQL. На стороне клиента при этом нет кэширования данных, здесь применяются только однонаправленные курсоры и не обеспечивается возможность прямого редактирования наборов данных.

Варианты архитектуры для BDE

Здесь мы рассмотрим различные варианты архитектуры информационной системы на примере технологии BDE. Варианты архитектуры для других технологий доступа к данным рассмотрим позже при непосредственном их описании.

Локальные БД располагаются на том же компьютере, что и работающие с ними приложения. В этом случае говорят, что информационная система имеет локальную архитектуру (рис. 1.1). Работа с БД происходит, как правило, в *однопользовательском* режиме. При необходимости можно запустить на компьютере другое приложение, одновременно осуществляющее доступ к этим же данным. Для управления совместным доступом к БД необходимы специальные средства контроля и защиты. Эти средства могут понадобиться, например, в случае, когда приложение пытается изменить запись, которую редактирует другое приложение. Каждая разновидность БД осуществляет подобный контроль своими способами и обычно имеет встроенные средства разграничения доступа.

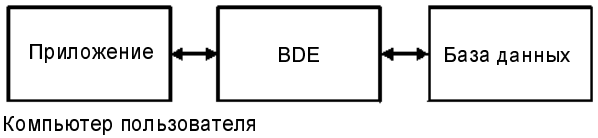


Рис. 1.1. Локальная архитектура

Для доступа к локальной БД процессор баз данных BDE использует стандартные драйверы, которые позволяют работать с форматами БД dBase, Paradox, FoxPro, а также с текстовыми файлами.

При использовании локальной БД в сети можно организовать многопользовательский доступ. В этом случае файлы БД и предназначенное для работы с ней приложение располагаются на сервере сети. Каждый пользователь запускает со своего компьютера это расположенное на сервере приложение, при этом у него запускается копия приложения. Такой сетевой вариант использования локальной БД соответствует архитектуре "файл-сервер". Приложение при использовании архитектуры "файл-сервер" также может быть записано на каждый компьютер сети, в этом случае приложению отдельного компьютера должно быть известно местонахождение общей БД (рис. 1.2).

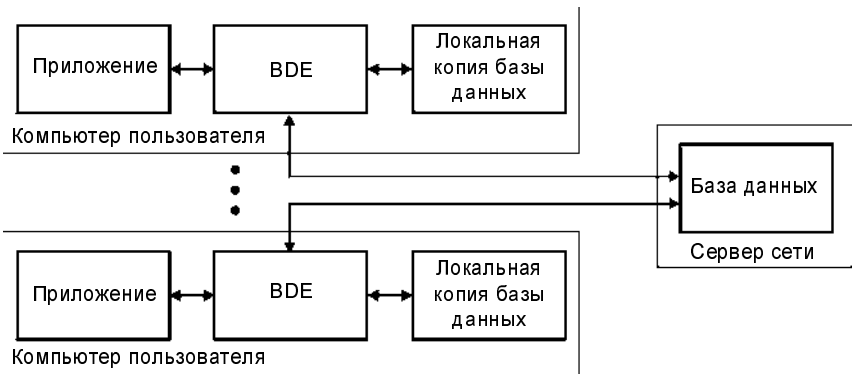


Рис. 1.2. Архитектура "файл-сервер"

При работе с данными на каждом пользовательском компьютере сети используется локальная копия БД. Эта копия периодически обновляется данными, содержащимися в БД на сервере.

Архитектура "файл-сервер" обычно применяется в сетях с небольшим количеством пользователей, для ее реализации подходят персональные СУБД, например, Paradox или dBase. Достоинствами этой архитектуры являются простота реализации, а также то, что приложение фактически разрабатывается в расчете на одного пользователя и не зависит от компьютера сети, на который оно устанавливается.

Однако архитектура "файл-сервер" имеет и существенные недостатки.

- Пользователь работает со своей локальной копией БД, данные в которой обновляются при каждом запросе к какой-либо из таблиц. При этом с сервера пересылается новая копия всей таблицы, данные из которой затребованы. Таким образом, если пользователю необходимо несколько записей таблицы, с сервера по сети пересылается вся таблица. В результате циркуляции в сети больших объемов избыточной информации *резко возрастает нагрузка на сеть*, что приводит к соответствующему снижению ее быстродействия и производительности информационной системы в целом.
- В связи с тем, что на каждом компьютере имеется своя копия БД, изменения, сделанные в ней одним пользователем, в течение некоторого времени являются неизвестными другим пользователям. Поэтому требуется постоянное обновление БД. Кроме того, возникает *необходимость синхронизации* работы отдельных пользователей, связанная с блокировкой в таблицах записей, которые в данный момент редактирует другой пользователь.
- Управление БД осуществляется с разных компьютеров, поэтому в значительной степени затруднена *организация управления доступом*, соблюдения *конфиденциальности* и поддержания *целостности* БД.

Удаленная БД размещается на компьютере-сервере сети, а приложение, осуществляющее работу с этой БД, находится на компьютере пользователя. В этом случае мы имеем дело с архитектурой "клиент-сервер" (рис. 1.3), когда информационная система делится на неоднородные части — сервер и клиент БД. В связи с тем, что компьютер-сервер отделен от клиента, его также называют *удаленным сервером.*

Клиент — это приложение пользователя. Для получения данных клиент формирует и отсылает запрос удаленному серверу, на котором помещена БД. Запрос формулируется на языке SQL, который является стандартным средством доступа к серверу при использовании реляционных моделей данных. После получения запроса удаленный сервер направляет его программе SQL Server (серверу баз данных) — специальной программе, управляющей удаленной БД и обеспечивающей выполнение запроса и выдачу его результатов клиенту.

Таким образом, в архитектуре "клиент-сервер" клиент посылает запрос на предоставление данных и получает только те данные, которые действительно были затребованы. Вся обработка запроса выполняется на удаленном сервере. Такая архитектура обладает следующими достоинствами.

- Снижение нагрузки на сеть, поскольку теперь в ней циркулирует только нужная информация.
- Повышение безопасности информации, связанное с тем, что обработка запросов всех клиентов выполняется единой программой, расположенной на сервере. Сервер устанавливает общие для всех пользователей правила использования БД, управляет режимами доступа клиентов к данным, запрещая, в частности, одновременное изменение одной записи различными пользователями.

- Уменьшение сложности клиентских приложений за счет отсутствия в них кода, связанного с контролем БД и разграничением доступа к ней.

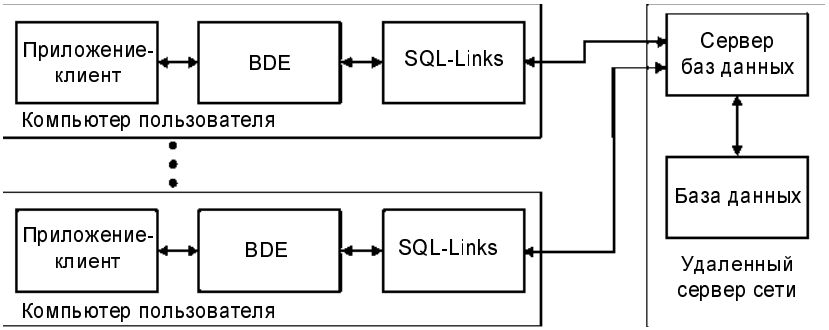


Рис. 1.3. Архитектура "клиент-сервер" ("толстый" клиент)

Для реализации архитектуры "клиент-сервер" обычно используются многопользовательские СУБД, например, Oracle или Microsoft SQL Server. Подобные СУБД также называют *промышленными*, т. к. они позволяют создать информационную систему организации или предприятия с большим числом пользователей. Промышленные СУБД являются сложными системами и требуют мощной вычислительной техники и соответствующего обслуживания. Обслуживание выполняет специалист (или группа специалистов), называемый *системным администратором БД* (администратором). Основные задачи системного администратора:

- защита БД;
- поддержание целостности БД;
- обучение и подготовка пользователей;
- загрузка данных из других БД;
- тестирование данных;
- резервное копирование и восстановление;
- внесение изменений в информационную систему.

Доступ Delphi-приложения к промышленным СУБД осуществляется через драйверы SQL-Links. Отметим, что при работе с "родной" для Delphi СУБД InterBase можно обойтись без драйверов SQL-Links.

Описанная архитектура является двухуровневой (уровень приложения-клиента и уровень сервера БД). Клиентское приложение также называют *сильным*, или "толстым", клиентом. Дальнейшее развитие данной архитектуры привело к появлению трехуровневого варианта архитектуры "клиент-сервер" (приложение-клиент, сервер приложений и сервер БД) (рис. 1.4).

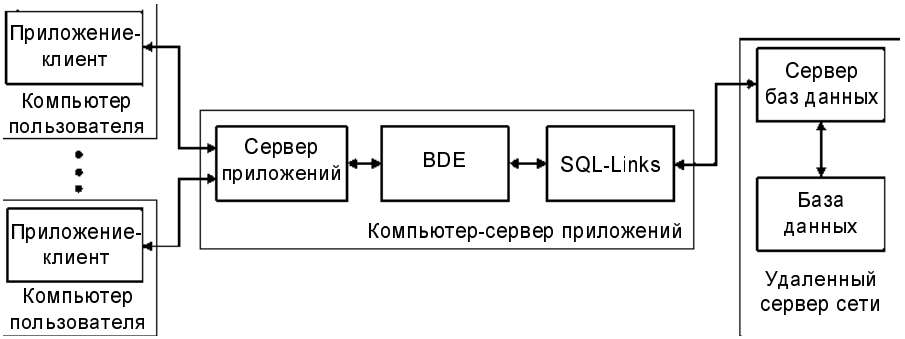


Рис. 1.4. Трехуровневая архитектура "клиент-сервер" ("тонкий" клиент)

В трехуровневой архитектуре часть средств и кода, предназначенных для организации доступа к данным и их обработке, из приложения-клиента выделяется в сервер приложений. Само клиентское приложение при этом называют *слабым*, или "тонким", клиентом. В сервере приложений удобно располагать средства и код, общие для всех клиентских приложений, например, средства доступа к БД.

Основные достоинства трехуровневой архитектуры "клиент-сервер" состоят в следующем:

- разгрузка сервера от выполнения части операций, перенесенных на сервер приложений;
- уменьшение размера клиентских приложений за счет разгрузки их от лишнего кода;
- единое поведение всех клиентов;
- упрощение настройки клиентов — при изменении общего кода сервера приложений автоматически изменяется поведение приложений-клиентов.

Напомним, что локальные приложения БД называют *одноуровневыми*, а клиент-серверные приложения БД — *многоуровневыми*.

Глава 2

Реляционные базы данных и средства работы с ними

Большинство современных баз данных для персональных компьютеров являются реляционными. Достоинства реляционной модели организации данных — простота, гибкость структуры, удобство реализации на компьютере, наличие теоретического описания.

Элементы реляционной базы данных

Реляционная база данных (БД) состоит из взаимосвязанных таблиц. Каждая таблица содержит информацию об объектах одного типа, а совокупность всех таблиц образует единую БД.

Таблицы баз данных

Таблицы, образующие БД, находятся в каталоге (папке) на жестком диске. Таблицы хранятся в файлах и похожи на отдельные документы или электронные таблицы (например, табличного процессора Microsoft Excel), их можно перемещать и копировать обычным способом, скажем, с помощью Проводника Windows. Однако в отличие от документов, таблицы БД поддерживают *многопользовательский* режим доступа, это означает, что их могут одновременно использовать несколько приложений.

Для одной таблицы создается несколько файлов, содержащих данные, индексы, ключи и т. п. Главным из них является файл с данными, имя этого файла совпадает с именем таблицы, которое задается при ее создании. В некотором смысле понятия таблицы и ее главного файла являются синонимами, при выборе таблицы выбирается именно ее главный файл: для таблицы dBase это файл с расширением dbf, а для таблицы Paradox — файл с расширением db. Имена остальных файлов таблицы назначаются автоматически — все файлы имеют одинаковые имена, совпадающие с именами таблиц, и разные расширения, указывающие на содержимое соответствующего файла. Расширения файлов приведены ниже в данной главе *в разд. "Таблицы форматов dBase и Paradox".*

Каждая таблица БД состоит из строк и столбцов и предназначена для хранения данных об однотипных объектах информационной системы. Строка таблицы называется *записью*, столбец таблицы — *полем* (рис. 2.1). Каждое поле должно иметь уникальное в пределах таблицы имя.

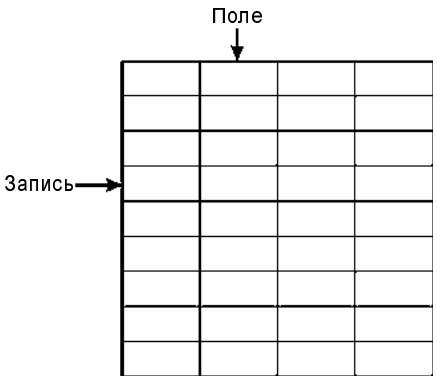


Рис. 2.1. Схема таблицы базы данных

Поле содержит данные одного из допустимых типов, например, строкового, целочисленного или типа "дата". При вводе значения в поле таблицы автоматически производится проверка соответствия типа значения и типа поля. В случае, когда эти типы не совпадают, а преобразование типа значения невозможно, генерируется исключение.

Особенности организации таблиц зависят от конкретной СУБД, используемой для создания и ведения БД. Например, в локальной таблице dBase и в таблице сервера InterBase нет поля автоинкрементного типа (с автоматически наращиваемым значением), а в таблице dBase нельзя определить ключ. Подобные особенности необходимо учитывать при выборе типа (формата) таблицы, т. к. они влияют не только на организацию БД, но и на построение приложения для работы с этой БД. Однако, несмотря на все различия таблиц, существуют общие правила создания и ведения БД, а также разработки приложений, которые и будут далее рассмотрены.

Замечание

В зависимости от типа таблиц и системы разработки приложений может использоваться различная терминология. Например, в InterBase поле таблицы называется столбцом.

Основу таблицы составляет описание ее полей, каждая таблица должна иметь хотя бы одно поле. Понятие структуры таблицы является более широким и включает:

- описание полей;
- ключ;

- индексы;
- ограничения на значения полей;
- ограничения ссылочной целостности между таблицами;
- пароли.

Иногда ограничения на значения полей, ограничения ссылочной целостности между таблицами, а также права доступа называют одним общим термином "ограничения".

Отметим, что отдельные элементы структуры зависят от формата таблиц, например, для таблиц dBase нельзя задать ограничения ссылочной целостности (т. к. у них нет ключей). Все элементы структуры задаются на физическом уровне (уровне таблицы) и действуют для всех программ, выполняющих операции с БД, включая средства разработки и ведения БД (например, программу Database Desktop). Многие из этих элементов (например, ограничения на значения полей или поля просмотра) можно также реализовать в приложении программно, однако в этом случае они действуют только в пределах своего приложения.

С таблицей в целом можно выполнять следующие операции:

- создание (определение структуры);
- изменение структуры (реструктуризация);
- переименование;
- удаление.

При *создании* таблицы задаются структура и имя таблицы. При сохранении на диске создаются все необходимые файлы, относящиеся к таблице. Их имена совпадают с именем таблицы.

При *изменении структуры* таблицы в ней могут измениться имена и характеристики полей, состав и наименования ключа и индексов, ограничения. Однако имена таблицы и ее файлов остаются прежними.

При *переименовании* таблица получает новое имя, в результате чего новое имя также получают все ее файлы. Для этого используются соответствующие программы (утилиты), предназначенные для работы с таблицами БД, например, Database Desktop или Data Pump.

Замечание

Таблицу нельзя переименовать, просто изменив названия всех ее файлов, например, с помощью Проводника Windows.

При *удалении* таблицы с диска удаляются все ее файлы. В отличие от переименования, удаление таблицы можно выполнить посредством любой программы (в том числе и с помощью Проводника Windows).

Ключи и индексы

Ключ представляет собой комбинацию полей, данные в которых однозначно определяют каждую запись в таблице. Простой ключ состоит из одного поля, а составной (сложный) — из нескольких полей. Поля, по которым построен ключ, называют *ключевыми*. В таблице может быть определен только один ключ. Ключ обеспечивает:

- однозначную идентификацию записей таблицы;
- ускорение выполнения запросов к БД;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

Ключ также называют *первичным ключом* или *первичным (главным) индексом*.

Информация о ключе может храниться в отдельном файле или совместно с данными таблицы. Например, в БД Paradox для этой цели используется отдельный файл (ключевой файл или файл главного индекса) с расширением px. В БД Access вся информация содержится в одном общем файле с расширением mdb. Значения ключа располагаются в определенном порядке. Для каждого значения ключа имеется уникальная ссылка, указывающая на расположение соответствующей записи в таблице (в главном ее файле). Поэтому при поиске записи выполняется не последовательный просмотр всей таблицы, а прямой доступ к записи на основании упорядоченных значений ключа.

Ценой, которую разработчик и пользователь платят за использование такой технологии, является увеличение размера БД вследствие необходимости хранения значений ключа, например, в отдельном файле. Размер этого файла зависит не только от числа записей таблицы (что достаточно очевидно), но и от полей, составляющих ключ. В ключевом файле, кроме ссылок на соответствующие записи таблицы, сохраняются и значения самих ключевых полей. Поэтому при вхождении в состав ключа длинных строковых полей размер ключевого файла может оказаться соизмеримым с размером файла с данными таблицы.

Таблицы различных форматов имеют свои особенности построения ключей. Вместе с тем существуют и общие правила.

- Ключ должен быть уникальным. У составного ключа значения отдельных полей (но не всех одновременно) могут повторяться.
- Ключ должен быть достаточным и не избыточным, т. е. не содержать поля, которые можно удалить без нарушения уникальности ключа.
- В состав ключа не могут входить поля некоторых типов, например, графическое поле или поле комментария.

Выбор ключевых полей не всегда является простой и очевидной задачей, особенно для таблиц с большим количеством полей. Нежелательно выбирать в качестве ключевых поля, содержащие фамилии людей в таблице сотрудников организации или названия товаров в таблице данных склада. В этом случае высока

вероятность существования двух и более однофамильцев, а также товаров с одинаковыми названиями, которые различаются, к примеру, цветом (значение другого поля). Для указанных таблиц можно использовать, например, поле кода сотрудника и поле артикула товара. При этом предполагается, что указанные значения являются уникальными.

Удобным вариантом создания ключа будет использование для него поля соответствующего типа, которое автоматически обеспечивает поддержку уникальности значений. Для таблиц Paradox таким является поле автоинкрементного типа, еще одним достоинством которого является небольшой размер (4 байта). В то же время в таблицах dBase и InterBase поле подобного типа отсутствует, и программист должен обеспечивать уникальность значений ключа самостоятельно, например, используя специальные генераторы.

Отметим, что при создании и ведении БД правильным подходом считается задание в каждой таблице ключа даже в случае, если на первый взгляд он не нужен. В примерах таблиц, которые приводятся при изложении материала, как правило, ключ создается, и для него вводится специальное автоинкрементное поле с именем `Code` или `Number`.

Индекс, как и ключ, строится по полям таблицы, однако он может допускать повторение значений составляющих его полей — в этом и состоит его основное отличие от ключа. Поля, по которым построен индекс, называют *индексными*. Простой индекс состоит из одного поля, а составной (сложный) — из нескольких полей.

Индексы при их создании именуются. Как и в случае с ключом, в зависимости от СУБД индексы могут храниться в отдельных файлах или совместно с данными. Создание индекса называют *индексированием таблицы*.

Использование индекса обеспечивает:

- увеличение скорости доступа к данным (поиска);
- сортировку записей;
- установление связи между отдельными таблицами БД;
- использование ограничений ссылочной целостности.

В двух последних случаях индекс применяется совместно с ключом второй таблицы.

Как и ключ, индекс представляет собой своеобразное оглавление таблицы, просмотр которого выполняется перед обращением к ее записям. Таким образом, использование индекса повышает *скорость доступа* к данным в таблице за счет того, что доступ выполняется не последовательным, а индексно-последовательным методом.

Сортировка представляет собой упорядочивание записей по полю или группе полей в порядке возрастания или убывания их значений. Можно сказать, что индекс служит для сортировки таблиц по индексным полям. В частности, в Delphi записи набора `Table` можно сортировать только по индексным полям.

Набор данных `Query` позволяет выполнить средствами SQL сортировку по любым полям, однако и в этом случае для индексированных полей упорядочивание записей выполняется быстрее.

Для одной таблицы можно создать несколько индексов. В каждый момент времени один из них можно сделать текущим, т. е. активным. Даже при существовании нескольких индексов таблица может не иметь текущего индекса (текущий индекс важен, например, при выполнении поиска и сортировки записей набора данных `Table`).

Ключевые поля обычно автоматически индексируются. В таблицах Paradox ключ также является главным (первичным) индексом, который не именуется. Для таблиц dBase ключ не создается, и его роль выполняет один из индексов.

Замечание

Создание ключа может привести к побочным эффектам. Так, если в таблице Paradox определить ключ, то записи автоматически упорядочиваются по его значениям, что в ряде случаев является нежелательным.

Таким образом, использование ключей и индексов позволяет:

- однозначно идентифицировать записи;
- избегать дублирования значений в ключевых полях;
- выполнять сортировку таблиц;
- ускорять операции поиска в таблицах;
- устанавливать связи между отдельными таблицами БД;
- использовать ограничения ссылочной целостности.

Одной из основных задач БД является обеспечение *быстрого доступа к данным* (поиска данных). Время доступа к данным в значительной степени зависит от используемых для поиска данных методов и способов.

Методы и способы доступа к данным

Выделяют следующие методы доступа к данным таблиц:

- последовательный;
- прямой;
- индексно-последовательный.

При *последовательном* методе выполняется последовательный просмотр всех записей таблицы и поиск нужных из них. Этот метод доступа является крайне неэффективным и приводит к значительным временным затратам на поиск, прямо пропорциональным размеру таблицы (числу ее записей). Поэтому его рекомендуется использовать только для относительно небольших таблиц.

При *прямом* доступе нужная запись выбирается в таблице на основании ключа или индекса. При этом просмотр других записей не выполняется. Напомним, что значения ключей и индексов располагаются в упорядоченном виде и содержат ссылку, указывающую на расположение соответствующей записи в таблице. При поиске записи выполняется не последовательный просмотр всей таблицы, а непосредственный доступ к записи на основании ссылки.

Индексно-последовательный метод доступа включает в себя элементы последовательного и прямого методов доступа и используется при поиске группы записей. Этот метод реализуется только при наличии индекса, построенного по полям, значения которых должны быть найдены. Суть его заключается в том, что находится индекс первой записи, удовлетворяющей заданным условиям, и соответствующая запись выбирается из таблицы на основании ссылки. Это является прямым доступом к данным. После обработки первой найденной записи осуществляется переход к следующему значению индекса, и в таблице выбирается запись, соответствующая значению этого индекса. Так последовательно перебираются индексы всех записей, удовлетворяющих заданным условиям, что является последовательным доступом.

Достоинством прямого и индексно-последовательного методов является максимальная возможная скорость доступа к данным, плата за которую — расход памяти на хранение информации о ключах и индексах.

Указанные методы доступа реализуются СУБД и не требуют специального программирования. Задачей разработчика является определение соответствующей структуры БД, в данном случае — определение ключей и индексов. Так, если для поля создан индекс, то при поиске записей по этому полю автоматически используется индексно-последовательный метод доступа, в противном случае — последовательный метод.

Замечание

При создании составного индекса важен порядок составляющих его полей. Например, если индекс создан по полям "фамилия", "номер отдела" и "дата рождения", а поиск ведется одновременно по полям "фамилия", "дата рождения" и "номер отдела", то такой индекс использован не будет. В результате доступ к таблице осуществляется последовательным методом. В подобной ситуации (для таблицы с большим числом записей) разработчик должен создать также индекс, построенный по полям "фамилия", "дата рождения" и "номер отдела".

При выполнении операций с таблицами используется один из следующих *способов доступа к данным*:

- навигационный;
- реляционный.

Навигационный способ доступа заключается в обработке каждой отдельной записи таблицы. Этот способ обычно используется в локальных БД или в удаленных БД небольшого размера. Если необходимо обработать несколько записей, то все они обрабатываются поочередно.

Реляционный способ доступа основан на обработке сразу *группы записей*, при этом если необходимо обработать одну запись, то обрабатывается группа, состоящая из одной записи. Так как реляционный способ доступа основывается на SQL-запросах, его также называют *SQL-ориентированным*. Этот способ доступа ориентирован на выполнение операций с удаленными БД и является предпочтительным при работе с ними, хотя его можно использовать и для локальных БД.

Способ доступа к данным выбирается программистом и зависит от средств доступа к БД, используемых при разработке приложения. Например, в приложениях, создаваемых в Delphi, реализацию навигационного способа доступа можно осуществить посредством компонентов `Table` или `Query`, а реляционного — с помощью компонента `Query`.

Таким образом, методы доступа к данным определяются структурой БД, а способы доступа — приложением.

Связь между таблицами

В частном случае БД может состоять из одной таблицы, содержащей, например, дни рождения сотрудников организации. Однако обычно реляционная БД состоит из набора взаимосвязанных таблиц. Организация связи (отношений) между таблицами называется *связыванием* или *соединением таблиц*.

Связи между таблицами можно устанавливать как при создании БД, так и при выполнении приложения, используя средства, предоставляемые СУБД. Связывать можно две или несколько таблиц. В реляционной БД, кроме связанных таблиц, могут быть и отдельные таблицы, не соединенные ни с одной другой таблицей. Это не меняет сути реляционной БД, которая содержит единую информацию об информационной системе, связанную не в буквальном смысле (связь между таблицами), а в функциональном смысле — вся информация относится к одной системе.

Для связывания таблиц используются *поля связи* (иногда применяется термин "совпадающие поля"). Поля связи обязательно должны быть индексированными. В подчиненной таблице для связи с главной таблицей задается индекс, который также называется *внешним ключом*. Состав полей этого индекса должен полностью или частично совпадать с составом полей индекса главной таблицы.

Особенности использования индексов зависят от формата связываемых таблиц. Так, для таблиц dBase индексы строятся по одному полю и нет деления на ключ (главный или первичный индекс) и индексы. Для организации связи в главной и подчиненной таблицах выбираются индексы, составленные по полям совпадающего типа, например, целочисленного.

Для таблиц Paradox в качестве полей связи главной таблицы должны использоваться поля ключа, а для подчиненной таблицы — поля индекса. Кроме того, в подчиненной таблице обязательно должен быть определен ключ. На рис. 2.2 показана схема связи между таблицами БД Paradox.

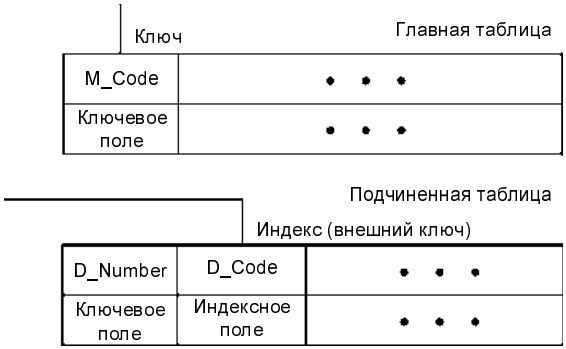


Рис. 2.2. Схема связи между таблицами базы данных Paradox

В главной таблице определен ключ, построенный по полю `M_Code` автоинкрементного типа. В подчиненной таблице определен ключ по полю `D_Number` также автоинкрементного типа и индекс, построенный по полю `D_Code` целочисленного типа. Связь между таблицами устанавливается по полям `D_Code` и `M_Code`. Индекс по полю `D_Code` является внешним ключом. В названия полей включены префиксы, указывающие на принадлежность поля соответствующей таблице. Так, названия полей главной таблицы начинаются с буквы `M` (Master), а названия полей подчиненной таблицы начинаются с буквы `D` (Detail). Подобное именование полей облегчает ориентацию в их названиях, особенно при большом количестве таблиц.

Замечание

Как отмечалось, поля связи должны быть индексированными, хотя, строго говоря, это требование не всегда является обязательным. При доступе к данным средствами языка SQL можно связать (соединить) между собой таблицы и по неиндексированным полям. Однако в этом случае скорость выполнения операций будет низкой.

Связь между таблицами определяет отношение подчиненности, при котором одна таблица является *главной* (родительской, или мастером — Master), а вторая — *подчиненной* (дочерней, или детальной — Detail). Саму связь (отношение) называют связь "главный-подчиненный", "родительский-дочерний" или "мастер-детальный". Существуют следующие виды связи:

- отношение "один-к-одному";
- отношение "один-ко-многим";
- отношение "много-к-одному";
- отношение "много-ко-многим".

Отношение *"один-к-одному"* означает, что одной записи в главной таблице соответствует одна запись в подчиненной таблице. При этом возможны два варианта:

- для каждой записи главной таблицы есть запись в подчиненной таблице;
- в подчиненной таблице может не быть записей, соответствующих записям главной таблицы.

Отношение "один-к-одному" обычно используется при разбиении таблицы с большим числом полей на несколько таблиц. В этом случае в первой таблице остаются поля с наиболее важной и часто используемой информацией, а остальные поля переносятся в другую (другие) таблицу.

Отношение *"один-ко-многим"* означает, что одной записи главной таблицы в подчиненной таблице может соответствовать несколько записей, в том числе ни одной. Этот вид отношения встречается наиболее часто. После установления связи между таблицами при перемещении на какую-либо запись в главной таблице в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы. Такой отбор записей подчиненной таблицы является своего рода фильтрацией.

Типичным примером является, например, организация учета выдачи книг в библиотеке, для которой удобно создать следующие две таблицы:

- таблицу карточек читателей, содержащую такую информацию о читателе, как фамилия, имя, отчество, дата рождения и домашний адрес;
- таблицу выдачи книг, в которую заносится информация о выдаче книги читателю и о возврате книги.

В этой ситуации главной является таблица карточек читателей, а подчиненной — таблица выдачи книг. Один читатель может иметь на руках несколько книг, поэтому одной записи в главной таблице может соответствовать несколько записей в подчиненной таблице. Если читатель сдал все книги или еще не брал ни одной книги, то для него в подчиненной таблице записей нет. После связывания обеих таблиц при выборе записи с данными читателя в таблице выдачи книг будут доступны только записи с данными о книгах, находящихся на руках этого читателя.

В приведенном примере предполагается, что после возврата книги соответствующая ей запись удаляется из таблицы выдачи книг. Вместо удаления записи можно заносить в соответствующее поле дату возврата книги.

Отношение *"много-к-одному"* отличается от отношения "один-ко-многим" только напрвлением. Если на отношение "один-ко-многим" посмотреть со стороны подчиненной таблицы, а не главной, то оно превращается в отношение "много-к-одному".

Отношение *"много-ко-многим"* имеет место, когда одной записи главной таблицы может соответствовать несколько записей подчиненной таблицы и одновременно одной записи подчиненной таблицы — несколько записей главной таблицы. Подобное отношение реализуется, например, при планировании занятий в институте и устанавливается между таблицами, в которых хранится информация о номерах аудиторий и номерах учебных групп. Так как учебная группа может заниматься в разных аудиториях, то одной записи об учебной группе (первая таблица) может соответствовать несколько записей о занимаемых этой группой аудиториях. В то же время в одной аудитории могут заниматься разные учебные группы (даже одновременно), поэтому одной записи об аудитории может соответствовать несколько записей об учебных группах (вторая таблица).

На практике отношение "много-ко-многим" используется достаточно редко. Причинами являются сложность организации связи между таблицами и взаимодействия между их записями. Кроме того, многие СУБД, в том числе Paradox, не поддерживают организацию ссылочной целостности для подобного отношения. Отметим также, что для отношения "много-ко-многим" понятия главной и подчиненной таблицы не имеют смысла.

Среди рассмотренных отношений наиболее общим является отношение "один-ко-многим". Другие виды отношений можно рассматривать как его варианты — отношение "один-к-одному" представляет собой частный случай этого отношения, а отношение "много-к-одному" является его "переворотом". Отношение "много-ко-многим" можно свести к отношению "один-ко-многим", соответствующим образом преобразовав и разделив таблицы. В дальнейшем предполагается, что таблицы связаны именно отношением "один-ко-многим".

Работа со связанными таблицами имеет следующие особенности.

- При изменении (редактировании) поля связи может нарушиться связь между записями двух таблиц. Поэтому при редактировании поля связи записи главной таблицы нужно соответственно изменять и значения поля связи всех подчиненных таблиц.
- При удалении записи главной таблицы нужно удалять и соответствующие ей записи в подчиненной таблице (каскадное удаление).
- При добавлении записи в подчиненную таблицу значение ее поля связи должно быть установлено равным значению поля связи главной таблицы.

Ограничения по установке, изменению полей связи и каскадному удалению записей могут быть наложены на таблицы при их создании. Эти ограничения, наряду с другими элементами, например описаниями полей и индексов, входят в структуру таблицы и действуют для всех приложений, которые выполняют операции с БД. Указанные ограничения можно задать при создании или реструктуризации таблицы, например, в среде программы Database Desktop, которая позволяет устанавливать связи между таблицами при их создании.

Ограничения, связанные с установкой, изменением значений полей связи и каскадным удалением записей, могут и не входить в структуру таблицы (таблиц), а реализовываться программным способом. В этом случае программист должен обеспечить:

- организацию связи между таблицами;
- установку значения поля связи подчиненной таблицы (это может также выполняться автоматически);
- контроль (запрет) редактирования полей связи;
- организацию (запрет) каскадного удаления записей.

Например, в случае удаления записи из главной таблицы программист должен проверить наличие соответствующих записей в подчиненной таблице. Если такие записи есть, то необходимо удалить и их или, наоборот, запретить удаление записей из обеих таблиц. И в том, и в другом случае пользователю должно быть выдано предупреждение.

Механизм транзакций

Информация БД в любой момент времени должна быть целостной и непротиворечивой. Одним из путей обеспечения этого является использование механизма транзакций.

Транзакция представляет собой выполнение последовательности операций. При этом возможны две ситуации.

- Успешно завершены все операции. В этом случае транзакция считается успешной, и все изменения в БД, которые были произведены в рамках транзакции отдельными операциями, подтверждаются. В результате БД переходит из одного целостного состояния в другое.
- Неудачно завершена хотя бы одна операция. При этом вся транзакция считается неуспешной, и результаты выполнения всех операций (даже успешно выполненных) отменяются. В результате происходит возврат БД в состояние, в котором она находилась до начала транзакции.

Таким образом, успешная транзакция переводит БД из одного целостного состояния в другое. Использование механизма транзакций необходимо:

- при выполнении последовательности взаимосвязанных операций с БД;
- при многопользовательском доступе к БД.

Транзакция может быть неявной или явной. *Неявная* транзакция стартует автоматически, а по завершении также автоматически подтверждается или отменяется. *Явной* транзакцией управляет программист с использованием компонента `Database` и/или средств SQL.

Часто в транзакцию объединяются операции над несколькими таблицами, когда производятся действия по внесению в разные таблицы взаимосвязанных изменений. Пусть осуществляется перенос записей из одной таблицы в другую. Если запись сначала удаляется из первой таблицы, а затем заносится во вторую таблицу, то при сбое, например из-за перерыва в энергопитании компьютера, возможна ситуация, когда запись уже удалена, но во вторую таблицу не попала. Если запись сначала заносится во вторую таблицу, а потом удаляется из первой таблицы, то при сбое возможна ситуация, когда запись будет находиться в двух таблицах. В обоих случаях имеет место нарушение целостности и непротиворечивости БД.

Для предотвращения подобной ситуации операции удаления записи из одной таблицы и занесения ее в другую таблицу объединяются в одну транзакцию. Выполнение этой транзакции гарантирует, что при любом ее результате целостность БД нарушена не будет.

Еще одним примером, демонстрирующим необходимость применения механизма транзакций, является складской учет товара. При поступлении товара на склад в таблицу движения товара заносится запись с данными о названии, количестве товара и дате его поступления. Затем в таблице склада соответственно количеству поступившего товара увеличивается наличное количество этого товара. При возникновении какой-либо ошибки, связанной с записью наличного

количества товара, новое значение может быть не занесено в соответствующую запись, в результате чего будет нарушена целостность БД, и она будет содержать некорректные значения. Такая ситуация возможна, например, в случае многопользовательского доступа к БД при редактировании этой записи другим приложением. Поэтому в случае невозможности внести изменения в наличное количество товара должно блокироваться и добавление новой записи в таблицу движения товара.

Для реализации механизма транзакций СУБД предоставляют соответствующие средства.

Бизнес-правила

Бизнес-правила представляют собой механизмы управления БД и предназначены для поддержания БД в целостном состоянии, а также для выполнения ряда других действий, например, накапливания статистики работы с БД.

Замечание

В данном контексте бизнес-правила являются просто правилами управления БД и не имеют отношения к бизнесу как предпринимательству.

В первую очередь бизнес-правила реализуют следующие ограничения БД:

- задание допустимого диапазона значений;
- задание значения по умолчанию;
- требование уникальности значения;
- запрет пустого значения;
- ограничения ссылочной целостности.

Бизнес-правила можно реализовывать как на физическом, так и на программном уровнях. В первом случае эти правила (например, ограничения ссылочной целостности для связанных таблиц) задаются при создании таблиц и входят в структуру БД. В дальнейшей работе нельзя нарушить или обойти ограничение, заданное на физическом уровне.

Вместо бизнес-правил, заданных на физическом уровне, или в дополнение к ним можно определить бизнес-правила на программном уровне. Действие этих правил распространяется только на приложение, в котором они реализованы. Для программирования в приложении бизнес-правил используются компоненты и предоставляемые ими средства. Достоинство такого подхода заключается в легкости изменения бизнес-правил и определении правил "своего" приложения. Недостатком является снижение безопасности БД, связанное с тем, что каждое приложение может устанавливать свои правила управления БД. В *главе 19*, посвященной навигационному способу доступа, приводится пример программирования бизнес-правил в приложении, связанный с каскадным удалением записей связанных таблиц.

При работе с удаленными БД в архитектуре "клиент-сервер" бизнес-правила можно реализовывать также на сервере.

Словарь данных

Словарь данных представляет собой совокупность определений БД и атрибутов. Словарь данных позволяет сформировать и сохранить характеристики, которые в дальнейшем можно использовать для описания БД.

Использование словаря данных позволяет:

- ускорить процесс создания БД;
- облегчить изменение характеристик БД;
- придать единообразный вид визуальным компонентам приложения.

Определение БД является специализированной БД, которая описывает структуру базы, но не содержит данных. Это описание структуры можно использовать для создания других БД.

Атрибуты представляют собой совокупности характеристик отдельных полей. Заданные через атрибуты характеристики поля при выполнении приложения устанавливаются в качестве значений соответствующих свойств визуальных компонентов, которые отображают значения поля, например, `DBGrid` или `DBText`. Приведем некоторые наиболее распространенные *характеристики полей* (они же *свойства* визуальных компонентов):

- `Alignment` — выравнивание;
- `DisplayLabel` — заголовок столбца (сетки `DBGrid`);
- `ReadOnly` — недоступность поля для редактирования (поле предназначено только для чтения);
- `Required` — требование обязательного ввода значения;
- `Visible` — видимость;
- `DisplayFormat` — формат отображаемого значения;
- `MinVàlue` — минимальное значение;
- `MaxValue` — максимальное значение.

Для работы со словарем данных удобно использовать программу SQL Explorer.

Таблицы форматов dBase и Paradox

Delphi не имеет своего формата таблиц, но поддерживает как собственные два типа локальных таблиц — dBase и Paradox. Каждая из этих таблиц имеет свои особенности.

Таблицы dBase являются одним из первых появившихся форматов таблиц для персональных компьютеров и поддерживаются многими системами, которые связаны с разработкой и обслуживанием приложений, работающих с БД.

Основные достоинства таблиц dBase:

- простота использования;
- совместимость с большим числом приложений.

В табл. 2.1 содержится список типов полей таблиц dBase IV. Для каждого типа приводится символ, используемый для его обозначения в программе Database Desktop (это программа создания и редактирования таблиц, SQL-запросов и запросов QBE), а также описание значений, которые может содержать поле рассматриваемого типа.

Таблица 2.1. *Типы полей таблиц dBase IV*

Тип	Обозначение	Описание значения
`Character`	`C`	Строка символов. Длина не более 255 символов
`Float`	`F`	Число с плавающей точкой. Диапазон -10^{308} ... 10^{308}. Точность 15 цифр мантиссы
`Number`	`N`	Число в двоично-десятичном формате BCD (Binary Coded Decimal)
`Date`	`D`	Дата
`Logical`	`L`	Логическое значение. Допустимы значение `True` (истина) и `False` (ложь). Разрешается использование прописных букв
`Memo`	`M`	Строка символов. Длина не ограничена. Символы хранятся в файле с расширением dbt
`OLE`	`O`	Данные в формате, который поддерживается технологией OLE. Данные хранятся в файле с расширением dbt
`Binary`	`B`	Последовательность байтов. Длина не ограничена. Байты содержат произвольное двоичное значение, хранятся в файле с расширением dbt

Замечание

При работе с таблицей в среде программы Database Desktop значения полей типа `Binary`, `Memo` и `OLE` не отображаются.

Таблицы dBase являются достаточно простыми и используют для своего хранения на дисках относительно мало физических файлов. По расширению файлов можно определить, какие данные они содержат.

- dbf — таблица с данными.
- dbt — данные больших двоичных объектов, или BLOB-данные (Binary Large OBject). К ним относятся двоичные, Memo- и OLE-поля. Memo-поле также называют полем комментариев.

- mdx — поддерживаемые индексы.
- ndx — индексы, непосредственно не поддерживаемые форматом dBase. При использовании таких индексов программист должен обрабатывать их самостоятельно.

Имя поля в таблице dBase должно состоять из букв и цифр и начинаться с буквы. Максимальная длина имени составляет 10 символов. В имена нельзя включать специальные символы и пробел.

К недостаткам таблиц dBase относится то, что они не поддерживают автоматическое использование парольной защиты и контроль целостности связей, поэтому программист должен кодировать эти действия самостоятельно.

Таблицы Paradox являются достаточно развитыми и удобными для создания БД. Можно отметить следующие их достоинства:

- большое количество типов полей для представления данных различных типов;
- поддержка целостности данных;
- организация проверки вводимых данных;
- поддержка парольной защиты таблиц.

Большой набор типов полей позволяет гибко выбирать тип для точного представления данных, хранимых в базе. Например, для представления числовой информации можно использовать один из пяти числовых типов. В табл. 2.2 содержится список типов полей для таблиц Paradox 7. Для каждого типа приводятся символ, используемый для обозначения этого типа в программе Database Desktop, и описание значений, которые может содержать поле рассматриваемого типа.

Таблица 2.2. *Типы полей таблиц в Paradox 7*

Тип	Обозначение	Описание значения
`Alpha`	`A`	Строка символов. Длина не более 255 символов
`Number`	`N`	Число с плавающей точкой. Диапазон -10^{307} ... 10^{308}. Точность 15 цифр мантиссы
`Money`	`$`	Денежная сумма. Отличается от типа `Number` тем, что в значении отображается денежный знак. Обозначение денежного знака зависит от установок Windows
`Short`	`S`	Целое число. Диапазон –32 768 ... 32 767
`LongInteger`	`I`	Целое число. Диапазон –2 147 483 648 ... 2 147 483 647
`BCD`	`#`	Число в двоично-десятичном формате
`Date`	`D`	Дата. Диапазон 01.01.9999 до н. э. ... 31.12.9999
`Time`	`T`	Время

Таблица 2.2 *(окончание)*

Тип	Обозначение	Описание значения
`Timestamp`	`@`	Дата и время
`Memo`	`M`	Строка символов. Длина не ограничена. Первые 240 символов хранятся в файле таблицы, остальные в файле с расширением mb
`Formatted Memo`	`F`	Строка символов. Отличается от типа `Memo` тем, что строка может содержать форматированный текст
`Graphic`	`G`	Графическое изображение. Форматы BMP, PCX, TIFF, GIF и EPS. При загрузке в поле изображение преобразуется к формату BMP. Для хранения изображения используется файл с расширением mb
`OLE`	`O`	Данные в формате, который поддерживается технологией OLE. Данные хранятся в файле с расширением mb
`Logical`	`L`	Логическое значение. Допустимы значения `True` (истина) и `False` (ложь). Разрешается использование прописных букв
`Autoincrement`	`+`	Автоинкрементное поле. При добавлении к таблице новой записи в поле автоматически заносится значение, на единицу большее, чем в последней добавленной записи. При удалении записи значение ее автоинкреметного поля больше не будет использовано. Значение автоинкреметного поля доступно для чтения и обычно используется в качестве ключевого поля
`Binary`	`B`	Последовательность байтов. Длина не ограничена. Байты содержат произвольное двоичное значение. Первые 240 байтов хранятся в файле таблицы, остальные в файле с расширением mb
`Bytes`	`Y`	Последовательность байтов. Длина не более 255 байтов

Замечание

При работе с таблицей в среде программы Database Desktop значения полей типа `Graphic`, `Binary`, `Memo` и `OLE` не отображаются.

Имя поля в таблице Paradox должно состоять из букв (допускается кириллица) и цифр и начинаться с буквы. Максимальная длина имени составляет 25 символов. В имени можно использовать такие символы, как пробел, `#`, `$` и некоторые другие. Не рекомендуется использовать символы `.`, `!` и `|`, т. к. они зарезервированы в Delphi для других целей.

При задании ключевых полей они должны быть первыми в структуре таблицы.

Замечание

Если требуется обеспечить перенос или совместимость данных из таблиц Paradox с таблицами других форматов, желательно выбирать имя поля длиной не более 10 символов и составлять его из латинских букв и цифр.

Поддержка концепции целостности данных обеспечивает правильность ссылок между таблицами. Например, если в БД имеются таблицы клиентов и заказов, то эти таблицы могут быть связаны следующим образом: каждая запись таблицы заказов ссылается через индексное поле на запись в таблице клиентов, соответствующую сделавшему заказ клиенту. Если в таблице клиентов любым способом удалить запись с информацией о каком-либо клиенте, то BDE автоматически удалит все записи, соответствующие этому клиенту, и из таблицы заказов. Подобное удаление взаимосвязанных записей называют *каскадным*.

Для полей можно определить специальный диапазон, в котором должны находиться вводимые в них значения. Кроме того, для каждого поля можно определить минимальное и максимальное допустимые значения. При попытке ввода в поле значения, выходящего за допустимые границы, возникает исключение, значение не вводится и содержимое поля не изменяется. Например, для поля `Salary` (оклад) в качестве минимального значения логично указать ноль, тем самым в это поле запрещается вводить отрицательные значения. Максимальное значение поля `Salary` зависит от организации, страны и от других факторов.

Наряду с диапазоном допустимых значений для каждого поля можно задать значение по умолчанию, которое автоматически заносится в поле при добавлении к таблице новой записи.

При работе с конфиденциальной информацией может потребоваться защита таблиц и их полей. Для каждой таблицы Paradox следует указать основной пароль, который используется при изменениях во всей таблице, связанных со сменой структуры или с редактированием данных в любом поле. Возможно также задание дополнительного пароля, позволяющего ограничить доступ к конкретному полю или группе полей таблицы, а также набора операций, применимых к таблице (например, разрешения только на чтение записей таблицы без возможности их модификации). После установки паролей они будут автоматически запрашиваться и контролироваться при любой попытке доступа к таблице.

Определенным недостатком таблиц Paradox является наличие относительно большого количества типов файлов, требуемых для хранения содержащихся в таблице данных. При копировании или перемещении какой-либо таблицы из одного каталога в другой необходимо обеспечить копирование или перемещение всех файлов, относящихся к этой таблице. Файлы таблиц Paradox имеют следующие расширения:

- db — таблица с данными;
- mb — BLOB-данные;
- px — главный индекс (ключ);
- xg* и yg* — вторичные индексы;

- val — параметры для проверки данных и целостности ссылок;
- tv и fam — форматы вывода таблицы в программе Database Desktop.

Замечание

Указанные файлы создаются только если в них есть необходимость; конкретная таблица может не иметь всех приведенных файлов.

Кроме названных файлов, при работе в сети для управления доступом к таблицам Paradox используются файлы с расширением net.

Средства для работы с реляционными базами данных

Хотя система Delphi не имеет своего формата таблиц БД, она тем не менее обеспечивает мощную поддержку большого количества различных СУБД — как локальных (например, dBase или Paradox), так и промышленных (например, Sybase или InterBase). Средства Delphi, предназначенные для работы с БД, можно разделить на два вида:

- инструменты
- компоненты

К *инструментам* относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений.

Компоненты предназначены для создания приложений, осуществляющих операции с БД.

Напомним, что в Delphi имеется окно Обозревателя дерева объектов, которое отображает иерархическую структуру объектов текущей формы. При разработке приложений баз данных это окно удобно использовать для просмотра структуры базы данных и изменения связей между компонентами. Кроме того, в окне Редактора кода имеется вкладка **Diagram**, служащая для отображения и настройки взаимосвязей между элементами баз данных.

Инструменты

Для операций с БД система Delphi предлагает следующий набор инструментов.

- Borland Database Engine (BDE) — процессор баз данных, который представляет собой набор динамических библиотек и драйверов, предназначенных для организации доступа к БД из Delphi-приложений. BDE является центральным звеном при организации доступа к данным.
- BDE Administrator — утилита для настройки различных параметров BDE, настройки драйверов баз данных, создания и удаления драйверов ODBC, создания и обслуживания псевдонимов.
- Database Desktop — программа создания и редактирования таблиц, SQL-запросов и запросов QBE.
- SQL Explorer — Проводник БД, позволяющий просматривать и редактировать БД и словари данных.

- SQL Builder — программа визуального конструирования SQL-запросов.
- SQL Monitor — программа отслеживания порядка выполнения SQL-запросов к удаленным БД.
- Data Pump — программа для переноса данных (схемы базы данных и содержимого) между БД.
- IBConsole — программа для управления удаленными БД.
- InterBase Server Manager — программа для запуска сервера InterBase.
- SQL Links — драйверы для доступа к удаленным промышленным СУБД, таким как Microsoft SQL Server или Oracle. К промышленному серверу InterBase, поставляемому с Delphi, доступ также можно организовать напрямую через BDE, не используя драйвер SQL Links. В Delphi 7 вместо SQL Links для доступа к базам данных SQL Server рекомендуется использовать dbExpress.
- dbExpress — набор драйверов для доступа к базам данных SQL с помощью таких компонентов, как `SQLConnection`, `SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`. dbExpress включает в свой состав следующие драйверы:
  - InterBase (файл dbexpint.dll);
  - DB2 (файл dbexpdb2.dll);
  - Oracle (файл dbexpora.dll);
  - MSSQL (файл dbexpmss.dll);
  - MySQL (файл dbexpmys.dll).
- InterBase Server — клиентская и серверная части сервера InterBase.

Одни инструменты, например, BDE Administrator и SQL Explorer, можно использовать для работы с локальными и удаленными БД, другие, например, IBConsole, — для работы с удаленными БД.

Компоненты

Рассмотрим компоненты, используемые для создания приложений БД. Кроме компонентов, Delphi предоставляет разработчику специальные объекты, например, объекты типа `Field`. Как и другие элементы управления Delphi, связанные с БД компоненты делятся на визуальные и невизуальные.

Невизуальные компоненты предназначены для организации доступа к данным, содержащимся в таблицах. Они представляют собой промежуточное звено между данными таблиц БД и визуальными компонентами.

Визуальные компоненты используются для создания интерфейсной части приложения. С их помощью пользователь может выполнять такие операции с таблицами БД, как просмотр или редактирование данных. Визуальные компоненты также называют *элементами, чувствительными к данным.*

Компоненты, используемые для работы с БД, находятся на страницах **Data Access**, **Data Controls**, **dbExpress**, **DataSnap**, **BDE**, **ADO**, **InterBase**, **Decision Cube**, **Rave** и **InterBase Admin** Палитры компонентов. Некоторые компоненты предназначены специально для работы с удаленными БД в архитектуре "клиент-сервер". Отметим, что в Палитре компонентов Delphi 7 появились новые страницы и по сравнению с предыдущими версиями изменилось распределение компонентов, используемых для работы с БД, по страницам. В частности, вместо страницы **QReport** появилась страница **Rave**.

Замечание

Состав компонентов можно настроить в диалоговом окне **Palette Properties** (Свойства палитры), вызываемом командой **Properties** (Свойства) контекстного меню Палитры компонентов. Здесь приводен состав Палитры компонентов, который получается после установки Delphi 7.

На странице **Data Access** (рис. 2.3) находятся невизуальные компоненты, предназначенные для организации доступа к данным:

- `DataSource` (источник данных);
- `ClientDataSet` (клиентский набор данных);
- `DataSetProvider` (провайдер набора данных);
- `XML Transform` (преобразователь документа XML в пакет данных и обратно);
- `XML TransformProvider` (провайдер данных для преобразования документа XML);
- `XML TransformClient` (адаптер между документом XML и провайдером).

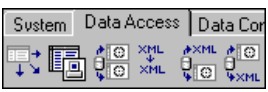


Рис. 2.3. Страница **Data Access**

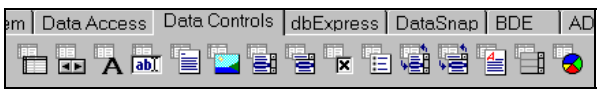


Рис. 2.4. Страница **Data Controls**

На странице **Data Controls** (рис. 2.4) расположены визуальные компоненты, предназначенные для управления данными:

- `DBGrid` (сетка, или таблица);
- `DBNavigator` (навигационный интерфейс);
- `DBText` (надпись);
- `DBEdit` (однострочный редактор, или поле редактирования);
- `DBMemo` (многострочный редактор, или панель редактирования);
- `DBImage` (графическое изображение);
- `DBListBox` (список);
- `DBComboBox` (комбинированный список);

- `DBCheckBox` (флажок);
- `DBRadioGroup` (группа переключателей);
- `DBLookupListBox` (список, формируемый по полю другого набора данных);
- `DBLookupComboBox` (комбинированный список, формируемый по полю другого набора данных);
- `DBRichEdit` (полнофункциональный тестовый редактор, или поле редактирования);
- `DBCtrlGrid` (модифицированная сетка);
- `DBChart` (диаграмма).

На странице **dbExpress** (рис. 2.5) размещены компоненты, предназначенные для соединения приложений с базами данных с помощью bdExpress:

- `SQLConnection` (инкапсуляция bdExpress-соединения с сервером БД);
- `SQLDataSet` (однонаправленный набор данных);
- `SQLQuery` (однонаправленный набор данных `Query`);
- `SQLStoredProc` (вызов хранимой процедуры сервера);
- `SQLTable` (набор данных `Table`);
- `SQLMonitor` (монитор выполнения SQL-запросов);
- `SimpleDataSet` (клиентский набор данных).

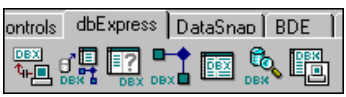


Рис. 2.5. Страница **dbExpress**

Рис. 2.6. Страница **DataSnap**

Страница **DataSnap** (рис. 2.6) содержит компоненты, предназначенные для создания многоуровневых приложений:

- `DCOMConnection` (управление соединением клиентских приложений DCOM с удаленным сервером);
- `SocketConnection` (сокет Windows для управления соединением с сервером приложения);
- `SimpleObjectBroker` (обслуживание списка доступных серверов приложения для компонента соединения);
- `WebConnection` (управление соединением с сервером приложения по протоколу HTTP);
- `ConnectionBroker` (централизация соединения с сервером приложения множества клиентов);

- `SharedConnection` (разделяемое соединение, управляющее соединением с дочерним удаленным модулем данных);
- `LocalConnection` (соединение между клиентским набором данных или посредником XML и провайдером).

Страница **BDE** (рис. 2.7) стала включать на один компонент меньше и содержит компоненты, предназначенные для управления данными с использованием BDE:

- `Table` (набор данных, основанный на таблице БД);
- `Query` (набор данных, основанный на SQL-запросе);
- `StoredProc` (вызов хранимой процедуры сервера);
- `DataBase` (соединение с БД);
- `Session` (текущий сеанс работы с БД);
- `BatchMove` (выполнение операций над группой записей);
- `UpdateSQL` (изменение набора данных, основанного на SQL-запросе или хранимой процедуре);
- `NestedTable` (вложенная таблица).

На странице **ADO** (рис. 2.8) расположены компоненты, предназначенные для управления данными с использованием технологии ADO (Active Data Objects):

- `ADOConnection` (соединение);
- `ADOCommand` (команда);
- `ADODataSet` (набор данных);
- `ADOTable` (набор данных `Table`);
- `ADOQuery` (набор данных `Query`);
- `ADOStoredProc` (вызов хранимой процедуры сервера);
- `RDSConnection` (соединение RDS).

Рис. 2.7. Страница **BDE**

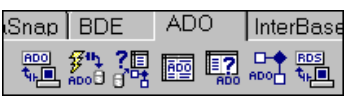


Рис. 2.8. Страница **ADO**

Замечание

Соединение RDS служит для управления передачей объекта `Recordset` от одного процесса (компьютера) к другому при создании серверных приложений.

На странице **InterBase** (рис. 2.9) находятся компоненты, предназначенные для работы с сервером InterBase:

- `IBTable` (набор данных `Table`);
- `IBQuery` (набор данных `Query`);

- `IBStoredProc` (вызов хранимой процедуры);
- `IBDatabase` (соединение с БД);
- `IBTransaction` (транзакция);
- `IBUpdateSQL` (изменение набора данных, основанного на SQL-запросе);
- `IBDataSet` (источник данных);
- `IBSQL` (выполнение SQL-запроса);
- `IBDatabaseInfo` (информация о БД);
- `IBSQLMonitor` (монитор выполнения SQL-запросов);
- `IBEvents` (событие сервера);
- `IBExtract` (извлечение данных);
- `IBClientDataSet` (клиентский источник данных).

Страница **Decision Cube** (рис. 2.10) содержит компоненты, предназначенные для построения систем принятия решений:

- `DecisionCube` (куб многомерных данных);
- `DecisionQuery` (набор, содержащий многомерные данные);
- `DecisionSource` (источник многомерных данных);
- `DecisionPivot` (двумерная проекция многомерных данных);
- `DecisionGrid` (сетка для табличного представления многомерных данных);
- `DecisionGraph` (графическое представление многомерных данных).

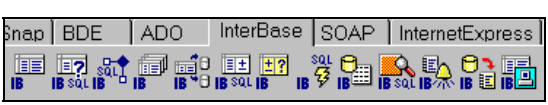


Рис. 2.9. Страница **InterBase**

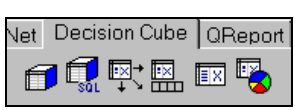


Рис. 2.10. Страница **Decision Cube**

И наконец, на странице **Rave** (рис. 2.11), которая заменила страницу **QReport**, находятся компоненты, предназначенные для построения отчетов:

- `RvProject` (обеспечивает доступ к визуальным отчетам);
- `RvSystem` (обеспечивает просмотр, печать и настройку параметров отчетов);
- `RvNDRWriter` (обеспечивает хранение отчета в двоичном формате);
- `RvCustomConnection` (служит для настройки способа передачи данных в отчет);
- `RvDataSetConnection` (обеспечивает доступ к компонентам, производным от класса `TDataSet`);
- `RvTableConnection` (служит для доступа к компонентам типа `TTable` или их наследникам;
- `RvQueryConnection` (служит для доступа к компонентам типа `TQuery` или их потомкам);

- `RvRenderPreview` (посылает отчет в двоичном формате для предварительного просмотра);
- `RvRenderPrinter` (посылает отчет в двоичном формате на текущее устройство печати);
- `RvRvRenderPDF` (преобразует поток или файл NDR (Rave snapshot report file, файл снимка отчета Rave) в формат PDF);
- `RvRenderHTML` (преобразует поток или файл NDR в формат HTML);
- `RvRenderRTF` (преобразует поток или файл NDR в документ RTF);
- `RvRenderRTF` (преобразует поток или файл NDR в текстовый документ).

Рис. 2.11. Страница **Rave**

Имена многих компонентов, предназначенных для работы с данными, содержат префиксы, например, `DB`, `IB` или `Rv`. Префикс `DB` означает, что визуальный компонент связан с данными и используется для построения интерфейсной части приложения. Такие компоненты размещаются в форме и предназначены для управления данными со стороны пользователя. Префикс `Rv` означает, что компонент используется для построения отчетов Rave. Префикс `IB` означает, что компонент предназначен для работы с сервером InterBase.

Исключения баз данных

В дополнение к основным исключениям, специально для операций, связанных с БД, система Delphi предоставляет следующие классы исключений:

- `EDatabaseError` — ошибка БД; потомки этого класса:
  - `EDBEngineError` — ошибка BDE (для локальных БД и сетевых БД архитектуры "файл-сервер");
  - `EDBClient` — ошибка в приложении клиента (для сетевых БД архитектуры "клиент-сервер"); код ошибки возвращается BDE, ADO, dbExpress или другим механизмом доступа к данным;
  - `EUpdateError` — ошибка, возникающая при обновлении записей;
- `EDBEditError` — введенное в поле значение не соответствует типу поля.

Класс `EDatabaseError` предназначен для обработки ошибок, возникающих при выполнении операций с набором данных (класс `TDataSet` и его потомки, в первую очередь, `TTable` и `TQuery`). Этот класс производится непосредственно от

класса `Exception`. Исключение класса `EDatabaseError` генерируется, например, при попытке открыть набор данных, связанный с отсутствующей таблицей, или изменить запись набора данных, который находится в режиме просмотра.

Класс `EDBEngineError` предназначен для обработки ошибок, возникающих при работе с процессором баз данных BDE на локальном компьютере. В дополнение к свойствам, имеющимся у класса `EDatabaseError`, у класса `EDBEngineError` есть два новых свойства:

- `ErrorCount` типа `Integer` — содержит количество возникших исключений;
- `Errors [Index: Integer]` типа `TDBError` — представляет собой список исключений, общее число которых указано свойством `ErrorCount`. Индекс `Index` позволяет получить доступ к возникшему исключению по его номеру (нумерация начинается с нуля). На практике обычно анализируется первое из исключений (`Errors[0]`), указывающее основную причину ошибки.

Класс `TDBError` содержит информацию об исключении, определяемую следующими свойствами:

- `Message` типа `String` (текст сообщения, характеризующего возникшую ошибку);
- `ErrorCode` типа `DBIResult` (код ошибки), тип `DBIResult` соответствует типу `Word`;
- `Category` типа `Byte` (категория исключения);
- `SubCode` типа `Byte` (группа, или подкод, исключения);
- `NativeError` типа `Longint` (код ошибки, возвращаемой сервером) — если значение этого свойства равно нулю, то исключение произошло не на сервере.

Код ошибки, определяемый значением свойства `ErrorCode`, имеет две составляющие: категорию и группу исключения, определяемые значениями свойств `Category` и `SubCode` соответственно. В *категорию* объединяются по признаку схожести несколько исключений. *Группа* уточняет исключение внутри категории.

Отметим, что для программной генерации исключений, обслуживаемых классами `EDatabaseError` и `EDBEngineError`, используются специальные методы `DatabaseError` и `DBIError`, рассматриваемые ниже.

Класс `EDBClient` отличается от класса `EDBEngineError` в основном тем, что предназначен для обработки ошибок, возникающих при работе с различными механизмами доступа к данным (не только с процессором баз данных BDE) в операциях с сетевыми базами архитектуры "клиент-сервер".

Генерируемые при работе с БД исключения обрабатывают глобальные и локальные обработчики, с которыми мы познакомимся немного позже. Кроме того, используемые для доступа к данным компоненты имеют специальные события для обработки исключений.

Например, для набора данных `Table` такими событиями являются:

- `OnEditError` (ошибка редактирования или вставки записи);
- `OnPostError` и `OnUpdateError` (ошибка закрепления изменений в записи);
- `OnDeleteError` (ошибка удаления записи).

Использование подобных событий и программирование их обработчиков будут рассмотрены при изучении соответствующих компонентов.

Класс `EDBEditError` используется в случаях, когда вводимые в поле данные несовместимы с маской ввода, заданной с помощью свойства `EditMask` этого поля. Отметим, что для проверки вводимых в поле значений можно также использовать события `OnSetText`, `OnValidate` и `OnChange`. Примеры использования этих событий при программировании обработчиков приводятся в *главе 5*.

Приведем в качестве примера процедуру, в которой проводится анализ исключения, возникшего при работе с базой данных с помощью BDE.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
try
 if OpenDialog1.Execute then begin
  Table1.Active := False;
  Table1.TableName := OpenDialog1.FileName;
  Table1.Active := True;
  end;
 except
 on EO: EDatabaseError do MessageDlg('Ошибка EDatabaseError!',
    mtError, [mbOK], 0);
 on EO: EDBEngineError do MessageDlg('Ошибка EDBEngineError!',
    mtError, [mbOK], 0);
  else MessageDlg('Неопознанная ошибка!', mtError, [mbOK], 0);
 end;
end;
```

При смене таблицы, связанной с набором данных `Table1`, выполняются анализ и обработка возможного исключения. Исключение проверяется на принадлежность к одному из двух классов: `EDatabaseError` и `EDBEngineError`. Обработка заключается в выдаче сообщения о принадлежности исключения к одному из этих классов. Если исключение носит другой характер, то выдается сообщение о том, что оно не распознано.

Глава 3

Проектирование баз данных

Проектирование реляционной БД заключается главным образом в разработке структуры данных, т. е. в определении состава таблиц и связей между ними. При этом структура должна быть эффективной и обеспечивать:

- быстрый доступ к данным;
- отсутствие дублирования (повторения) данных;
- целостность данных.

Проектирование структуры данных (структуры БД) также называют *логическим* проектированием, или проектированием на логическом уровне.

При проектировании структур данных можно выделить три основных подхода.

- Сбор информации об объектах решаемой задачи в рамках одной таблицы (одного отношения) и последующее разбиение ее на несколько взаимосвязанных таблиц на основе нормализации отношений.
- Формулирование знаний о системе (определение типов исходных данных и их взаимосвязей) и требований к обработке данных, а затем получение с помощью CASE-средств схемы БД или готовой прикладной информационной системы.
- Структурирование информации в результате системного анализа на основе совокупности правил и рекомендаций.

Проектирование может выполняться классическим способом, когда разработчик собирает и выделяет объекты системы и их характеристики, после чего вручную приводит их к требуемой структуре данных. Кроме того, для проектирования можно использовать так называемые CASE-системы, которые автоматизируют процесс разработки не только БД, но и информационной системы в целом.

Нормализация базы данных

Нормализация БД — это процесс уменьшения избыточности информации в БД. Метод нормализации основан на достаточно сложной теории реляционных моделей данных. Рассмотрим основные особенности и технику нормализации, не вдаваясь в теоретическое обоснование этих вопросов.

Избыточность данных и аномалии

При разработке структуры БД могут возникнуть проблемы, связанные:

- с избыточностью данных;
- с аномалиями.

Под *избыточностью данных* понимают дублирование данных, содержащихся в БД. При этом различают простое (не избыточное) дублирование и избыточное дублирование данных.

Избыточность данных при выполнении операций с ними может приводить к различным аномалиям — нарушению целостности БД. Выделяют аномалии:

- удаления;
- обновления;
- ввода.

Не избыточное дублирование является естественным и допустимым, его примером является список телефонов (местных) сотрудников организации, показанный в табл. 3.1.

Таблица 3.1. *Пример не избыточного дублирования данных*

Сотрудник	Телефон	Сотрудник	Телефон
Иванов П. Л.	123	Кузнецова В. А.	789
Петров А. Ф.	123	Васин И. Г.	123
Сидоров О. Е.	456		

Три сотрудника имеют одинаковый номер телефона 123, что возможно, например, в случае, когда они находятся в одной комнате. Таким образом, номер телефона в таблице дублируется, однако для каждого сотрудника этот номер является уникальным. В случае удаления одного из дублированных значений номера телефона (удаления соответствующей строки таблицы) будет потеряна информация о сотруднике — Иванове П. Л., Петрове А. Ф. или Васине И. Г., — что является *аномалией удаления*.

При смене номера телефона в комнате его необходимо изменить для всех сотрудников, которые в ней находятся. Если для какого-либо сотрудника этого не сделать, например, для Петрова А. Ф., то возникает несоответствие данных, связанное с *аномалией обновления*.

Аномалия ввода заключается в том, что при вводе в таблицу новой строки для ее полей могут быть введены недопустимые значения. Например, значение не входит в заданный диапазон или не задано значение поля, которое в обязательном порядке должно быть заполнено (не может быть пустым).

Теперь приведем пример *избыточного дублирования* данных. Список телефонов дополнен номерами комнат, в которых находятся сотрудники (табл. 3.2). При этом номер телефона указан только для одного из сотрудников — в примере для Иванова П. Л., который находится в списке первым. Вместо номеров телефонов других сотрудников этой комнаты поставлен прочерк. На практике кодировка прочерка зависит от особенностей конкретной таблицы, например, можно обозначать прочерк значением `Null`.

Таблица 3.2. *Пример избыточного дублирования данных*

Сотрудник	Комната	Телефон
Иванов П. Л.	17	123
Петров А. Ф.	17	—
Сидоров О. Е.	22	456
Кузнецова В. А.	8	789
Васин И. Г.	17	—

Однако при таком построении таблицы появляются проблемы.

- Номер телефона произвольного сотрудника можно получить только путем поиска в другом столбце таблицы (по номеру комнаты).
- При запоминании таблицы для каждой строки отводится одинаковая память вне зависимости от наличия или отсутствия прочерков.
- При удалении строки с данными сотрудника, для которого указан номер телефона комнаты, будет утеряна информация о номере телефона для всех сотрудников из этой комнаты.

Если вместо прочерков указать номер телефона, то избыточное дублирование все равно остается. От него можно избавиться, например, с помощью разбиения таблицы на две новых (табл. 3.3 и 3.4). Разбиение — это процесс деления таблицы на несколько таблиц с целью поддержания целостности данных, т. е. устранения избыточности данных и аномалий. Таблицы связаны между собой по номеру комнаты. Для получения информации о номере телефона сотрудника из первой таблицы по фамилии сотрудника нужно считать номер его комнаты, после чего из второй таблицы по номеру комнаты считывается номер телефона.

Рассмотренное разбиение таблицы на две является примером нормализации отношений. При этом избыточность данных уменьшилась, однако одновременно увеличилось *время доступа* к ним. Для получения номера телефона сотрудника теперь необходимо работать с двумя таблицами, а не с одной, как в предыдущем случае.

Таблица 3.3. *Список сотрудников и номеров комнат*

Сотрудник	Комната
Иванов П. Л.	17
Петров А. Ф.	17
Сидоров О. Е.	22
Кузнецова В. А.	8
Васин И .Г.	17

Таблица 3.4. *Список номеров комнат и телефонов*

Комната	Телефон
17	123
8	789
22	456

Приведение к нормальным формам

Процесс проектирования БД с использованием метода нормальных форм является итерационным (пошаговым) и заключается в последовательном переводе по определенным правилам отношений из первой нормальной формы в нормальные формы более высокого порядка. Каждая следующая нормальная форма ограничивает определенный тип функциональных зависимостей, устраняет соответствующие аномалии при выполнении операций над отношениями БД и сохраняет свойства предшествующих нормальных форм.

Выделяют следующую последовательность нормальных форм:

- первая нормальная форма;
- вторая нормальная форма;
- третья нормальная форма;
- усиленная третья нормальная форма, или нормальная форма Бойса-Кодда;
- четвертая нормальная форма;
- пятая нормальная форма.

Проектирование начинается с определения всех объектов, информация о которых должна содержаться в БД, и выделения атрибутов (характеристик) этих объектов. Атрибуты всех объектов сводятся в одну таблицу, которая является исходной. Затем эта таблица последовательно приводится к нормальным формам в соответствии с их требованиями. На практике обычно используются первые три нормальные формы.

Для примера спроектируем БД для хранения информации о футбольном чемпионате страны. Будем запоминать информацию о дате матча, игравших командах и забитых голах. Сначала объединим все данные в одну исходную таблицу, имеющую следующую структуру (поля):

- дата матча;
- команда хозяев;
- команда гостей;
- игрок, забивший гол;

- признак команды;
- время гола.

В качестве данных о команде (хозяев и гостей) укажем ее название, город и фамилию тренера, что однозначно идентифицирует любую команду. Для игрока, забившего гол, будем указывать фамилию, а для обозначения его принадлежности к той или иной команде используем признак, например, `Х` — для команды хозяев, а `Г` — для команды гостей. Время гола представляет число минут, прошедших с начала матча.

Приведенная таблица имеет относительно простую структуру, на практике подобные таблицы содержат также данные о составах команд, арбитрах, времени начала игры, числе зрителей, нарушениях правил, заменах, предупреждениях и удалениях игроков и другую информацию. В структуре таблицы указаны только названия полей, поскольку тип и размерность полей на этом этапе большого значения не имеют.

Созданную таблицу можно рассматривать как однотабличную БД. Ее главным недостатком является значительная избыточность данных. Так, для каждого игрока, забившего гол, указываются дата матча и информация о команде хозяев и гостей. Дублирование данных способно привести к аномалиям, а также к существенному увеличению размера базы.

Отметим, что в таблице отсутствует информация о счете матча, т. к. ее можно получить, подсчитав голы, забитые хозяевами и гостями. Побочным явлением такого подхода является то, что если в матче не было забитых голов, то информация о соответствующем матче в таблице отсутствует. Эта ситуация исправляется автоматически при последующем разбиении исходной таблицы на таблицы матчей и голов.

Первая нормальная форма

Приведем исходную таблицу к первой нормальной форме, для которой должны выполняться следующие условия:

- поля содержат неделимую информацию;
- в таблице отсутствуют повторяющиеся группы полей.

Во втором и третьем полях исходной таблицы содержится информация о названии команды, городе и тренере, например, `Зенит Санкт-Петербург Ю.Морозов`. Это противоречит первому требованию — информация о команде, городе и тренере является делимой. Эта информация может и должна быть разделена на три отдельных поля — название команды, город, фамилия тренера.

Согласно второму требованию, в таблице должны отсутствовать повторяющиеся группы полей, т. е. группы, содержащие одинаковые функциональные значения. В исходной таблице таких групп нет, поэтому второму требованию она удовлетворяет. Может показаться, что повторяющимися группами полей являются поля с информацией о командах хозяев и гостей. Несмотря на то, что состав и определения названных полей полностью совпадают, эти поля имеют различное функциональное назначение и не считаются повторяющимися.

Примером таблицы, имеющей повторяющиеся группы полей, может служить табл. 3.5 с результатами экзаменационной сессии. В этой таблице для оценок, полученных студентами на экзаменах, необходимо создать столько полей, сколько может быть различных предметов. В связи с тем, что студент сдает не все экзамены (для них проставлены прочерки), размер записей и таблицы в целом неоправданно увеличивается. Кроме того, значительные трудности создает изменение состава предметов. Если организовать переименование предмета достаточно просто, то его удаление или добавление требует изменения структуры таблицы, что, вообще говоря, крайне нежелательно, поскольку структура таблицы обычно определяется еще на этапе проектирования БД.

Таблица 3.5. *Результаты экзаменационной сессии*

Студент	Математика	Информатика	Физика	История	Английский язык
Семенов Р. О.	4	4	4	—	4
Костина В. К.	4	3	3	—	4
Папаев Д. Г.	5	5	5	—	—
Симонов П. С.	—	—	—	4	4

Вторая и третья нормальные формы касаются отношений между ключевыми и не ключевыми полями.

Вторая нормальная форма

Ко *второй нормальной форме* предъявляются следующие требования:

- таблица должна удовлетворять требованиям первой нормальной формы;
- любое не ключевое поле должно однозначно идентифицироваться ключевыми полями.

Записи таблицы, приведенной к первой нормальной форме, не являются уникальными и содержат дублированные данные. Так, если за одну минуту футболист забил несколько голов, то таблица будет содержать одинаковые записи. Чтобы обеспечить уникальность записей, введем в таблицу поле ключа — код матча. При этом значение ключа будет однозначно определять каждую запись таблицы.

Тогда структура таблицы примет такой вид:

- Код матча (уникальный ключ)
- Дата матча
- Название команды хозяев
- Город команды хозяев
- Фамилия тренера команды хозяев
- Название команды гостей
- Город команды гостей

- Фамилия тренера команды гостей
- Игрок
- Признак команды
- Время гола

Записи этой таблицы имеют значительное избыточное дублирование данных, т. к. дата матча, а также информация о командах указываются для каждого гола. Поэтому разобьем таблицу на две таблицы, одна из которых будет содержать данные о матчах, а вторая — о голах, забитых в каждом конкретном матче. Структуры этих таблиц будут следующими.

Поля таблицы матчей:

- Код матча (уникальный ключ)
- Дата матча
- Название команды хозяев
- Город команды хозяев
- Фамилия тренера команды хозяев
- Название команды гостей
- Город команды гостей
- Фамилия тренера команды гостей

Поля таблицы голов:

- Код гола (уникальный ключ)
- Код матча
- Игрок
- Признак команды
- Время гола

Таблицы связаны по полю "Код матча", которое для таблицы матча имеет уникальное значение. Чтобы обеспечить уникальность записей таблицы голов, в нее введено ключевое поле "Код гола".

Третья нормальная форма

Требования *третьей нормальной формы*:

- таблица должна удовлетворять требованиям второй нормальной формы;
- ни одно из не ключевых полей не должно однозначно идентифицироваться значением другого не ключевого поля (полей).

Приведение таблицы к третьей нормальной форме предполагает выделение в отдельную таблицу (таблицы) тех полей, которые не зависят от ключа. В таблице матчей такими являются поля с фамилиями тренеров команд, которые однозначно определяются значениями названия и города команды. (Предполагается,

что в течение сезона у команды тренер не меняется, на практике это часто не выполняется, но не относится к сути рассматриваемого вопроса.) Разобьем таблицу матчей на две таблицы: одну с данными о матчах и вторую — с данными о командах. Их структура будет такой:

Таблица матчей:

- Код матча (уникальный ключ)
- Дата матча
- Код команды хозяев
- Код команды гостей

Таблица команд:

- Код команды (уникальный ключ)
- Название команды
- Город команды
- Фамилия тренера команды

Информация о команде хранится в строковых полях, которые имеют достаточно большой размер — в нашем случае приблизительно 60 символов. На практике о каждой команде необходимо запоминать больше данных, чем просто ее название, город приписки и фамилия тренера, что существенно увеличивает объем информации. Поэтому для уменьшения размера записей таблицы матчей не только фамилия тренера, но и вся подробная информация о командах вынесена в таблицу команд. Вместо этого команды хозяев и гостей в таблице матчей идентифицируются кодом команды, который является уникальным ключевым значением в таблице команд.

После приведения к третьей нормальной форме база данных с информацией о футбольном чемпионате страны будет иметь структуру, показанную на рис. 3.1, где кроме описания таблиц обозначены связи между ними.

Следование требованиям нормализованных форм не всегда является обязательным. Как уже отмечалось, с ростом числа таблиц структура БД усложняется и возрастает время доступа к данным. В ряде случаев для упрощения структуры БД можно позволить частичное дублирование данных, не допуская, однако, нарушения их целостности и сохраняя их непротиворечивость.

В рассмотренной БД это относится, например, к информации о счете матча. Чтобы исключить дублирование данных, для него не отведено отдельное поле. Узнать счет матча можно с помощью двух запросов к БД, возвращающих число мячей, забитых в указанном матче хозяевами и гостями соответственно. Запрос может иметь следующий вид:

```
SELECT COUNT(G_OwnerSign)
    FROM Goal
        WHERE Goal.G_Match = :MatchCode
            AND Goal.G_OwnerSign = :OwnerSign
```

Номер матча и признак команды, игрок которой забил гол, передаются в запрос через параметры `MatchCode` и `OwnerSign`.

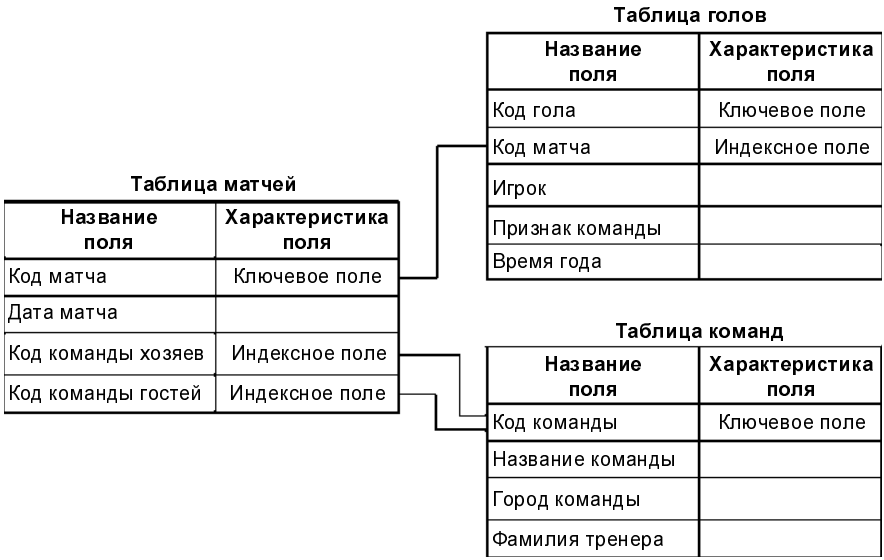


Рис. 3.1. Структура БД "Чемпионат по футболу"

Поскольку процесс обработки данных усложнился, а время доступа к ним возросло, возможным вариантом для структуры рассмотренной БД является такой, когда счет матча запоминается в отдельном поле таблицы матчей. При этом необходимо обеспечить целостность данных при изменении БД с учетом того факта, что часть ее данных дублирована.

Средства CASE

В предыдущем разделе проектирование БД выполнено вручную. Разработчик сам осуществляет такие операции, как определение состава полей, распределение их по таблицам, а также установление связей между таблицами. Ручное проектирование применяется для разработки БД самого различного назначения и для относительно небольших БД вполне приемлемо. Однако с ростом размера базы данных, когда в нее включаются от нескольких десятков до сотен различных таблиц, возникает проблема сложности организации данных, в том числе установления взаимосвязей между таблицами. Для облегчения решения этой проблемы предназначены системы автоматизации разработки приложений, или средства CASE (Computer Aided Software Engineering).

Средства CASE представляют собой программы, поддерживающие процессы создания и/или сопровождения информационных систем, такие как анализ и

формулировка требований, проектирование БД и приложений, генерация кода, тестирование, обеспечение качества, управление конфигурацией и проектом. То есть средства CASE позволяют решать более масштабные задачи, чем просто проектирование БД. Кстати, согласно предлагаемой ниже классификации, система Delphi также относится к типу CASE, т. к. позволяет автоматизировать разработку приложений.

Систему CASE можно определить как набор средств CASE, имеющих определенное функциональное предназначение и выполненных в рамках единого программного продукта.

Классификация средств (систем) CASE, используемых для разработки баз данных, производится по следующим признакам:

- ориентация на этапы жизненного цикла;
- функциональная полнота;
- тип используемых моделей;
- степень независимости от СУБД;
- платформа.

По *ориентации на этапы жизненного цикла* можно выделить следующие основные типы систем CASE (в скобках приведены названия фирм-разработчиков):

- системы анализа, предназначенные для построения и анализа моделей предметной области, например, Design/IDEF (Meta Software) и BPWin (Logic Works);
- системы анализа и проектирования, поддерживающие и обеспечивающие создание проектных спецификаций, например, Vantage Team Builder (Cayenne), Silverrun (Silverrun Technologies), PRO-I (McDonnell Douglas);
- системы проектирования БД, обеспечивающие моделирование данных и разработку схем баз данных для основных СУБД, например, ERwin (Logic Works), SDesigner (SPD), DataBase Designer (Oracle);
- системы разработки приложений, например, Uniface (Compuware), JAM (JYACC), PowerBuilder (Sybase), Developer/2000 (Oracle), New Era (Informix), SQL Windows (Centura), Delphi (Borland).

По *функциональной полноте* системы CASE условно делятся на следующие группы:

- системы, предназначенные для решения частных задач на одном или нескольких этапах жизненного цикла, например, ERwin (Logic Works), S-Designer (SPD), CASE.Аналитик (МакроПроджект) и Silverrun (Silverrun Technologies);
- интегрированные системы, поддерживающие весь жизненный цикл информационной системы и связанные с общим репозиторием (хранилищем), например, система Vantage Team Builder (Cayenne) и система Designer/2000 с системой разработки приложений Developer/2000 (Oracle).

По *типу используемых моделей* системы CASE делятся на три разновидности: структурные, объектно-ориентированные и комбинированные.

Исторически первыми появились структурные системы CASE, которые основываются на методах структурного и модульного программирования, структурного анализа и синтеза, например, Vantage Team Builder (Cayenne).

Объектно-ориентированные системы CASE получили массовое распространение с начала 90-х годов XX века. Они позволяют сократить сроки разработки, а также повысить надежность и эффективность функционирования информационной системы. Примерами объектно-ориентированных систем CASE являются Rational Rose (Rational Software) и Object Team (Cayenne).

Комбинированные системы CASE поддерживают одновременно и структурное, и объектно-ориентированное программирование, например, Designer/2000 (Oracle).

По *степени независимости от СУБД* системы CASE делятся на две группы:

- независимые системы;
- встроенные в СУБД системы.

Независимые системы CASE поставляются в виде автономных систем, не входящих в состав конкретной СУБД. Обычно они поддерживают несколько форматов баз данных через интерфейс ODBC. К числу независимых систем относятся SDesigner (SDP, Powersoft), ERwin (LogicWorks), Silverrun (Silverrun Technologies).

Встроенные системы CASE обычно поддерживают главным образом формат базы данных, в состав системы управления которой они входят. При этом возможна поддержка форматов и других баз данных. Примером встроенной системы является система Designer/2000, входящая в состав СУБД Oracle.

Платформа определяет компьютер и операционную систему, на которых допускается использовать продукт, созданный с помощью данной системы CASE.

Перечислим средства CASE, которые можно применять при разработке БД и приложений с помощью Delphi.

- ModelMaker — продукт, поставляемый вместе с Delphi 7. Служит для разработки классов и пакетов компонентов для Delphi. Представляет собой CASE-средство, ориентированное на генерацию кода Delphi. Позволяет хранить и обслуживать отношения между классами и их членами, поддерживает построение UML-диаграмм. По сравнению с другими генераторами кода ModelMaker позволяет разрабатывать сложные проекты.
- Data Module Designer — позволяет проектировать БД с таблицами формата Paradox. Программа обеспечивает достаточно удобный и наглядный интерфейс. Структура БД, в том числе связи между таблицами, отображается в графическом виде.

- Cadet — независимый продукт, позволяющий проектировать БД с таблицами форматов dBase, Paradox и InterBase. С учетом того, что указанные форматы являются родными для Delphi, программу Cadet удобно использовать при разработке информационных систем.

Программы Data Module Designer и Cadet предназначены для моделирования структур данных и автоматизации проектирования БД. Возможности, предоставляемые этими средствами, меньше, чем возможности таких мощных систем, как, например, Sdesigner, однако доступность делает их привлекательными для использования. Так, программа Cadet является условно бесплатной, а Data Module Designer входит в состав СУБД Paradox 7.0. Впрочем, с появлением ModelMaker использование других CASE-средств может не потребоваться.

Глава 4

Технология создания информационной системы

Продемонстрируем возможности Delphi по работе с БД на примере создания простой информационной системы. Эту информационную систему можно разработать даже без написания кода: все необходимые операции выполняются с помощью программы Database Desktop, Конструктора формы и Инспектора объектов. Работа над информационной системой состоит из следующих основных этапов:

- создание БД;
- создание приложения.

В простейшем случае БД состоит из одной таблицы. Если таблицы уже имеются, то первый этап не выполняется. Отметим, что совместно с Delphi поставляется большое количество примеров приложений, в том числе и приложений БД. Файлы таблиц для этих приложений находятся в каталоге c:\Program Files\Common Files\Borland Shared\Data. Готовые таблицы также можно использовать для своих приложений.

Создание таблиц базы данных

Для работы с таблицами БД при проектировании приложения удобно использовать программу Database Desktop, которая позволяет:

- создавать таблицы;
- изменять структуры;
- редактировать записи.

Кроме того, с помощью Database Desktop можно выполнять и другие действия над БД (создание, редактирование и выполнение визуальных и SQL-запросов,

операции с псевдонимами), которые будут рассматриваться позже (в главе, посвященной инструментам).

Замечание

Почти все рассматриваемые далее действия по управлению структурой таблицы можно выполнить также программно, что описано в *главах 7 и 8*, посвященных навигационному и реляционному способам доступа.

Процесс создания новой таблицы начинается с вызова команды **File\New\Table** (Файл\Новая\Таблица) и происходит в интерактивном режиме. При этом разработчик должен:

- выбрать формат (тип) таблицы;
- задать структуру таблицы.

В начале создания новой таблицы в окне **Create Table** (Создание таблицы) (рис. 4.1) выбирается ее формат. По умолчанию предлагается формат таблицы Paradox версии 7, который мы и будем использовать. Для таблиц других форматов, например, dBase IV, действия по созданию таблицы практически не отличаются.

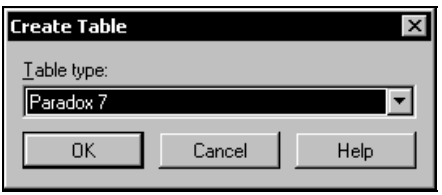


Рис. 4.1. Выбор формата таблицы

После выбора формата таблицы появляется окно определения структуры таблицы (рис. 4.2), в котором выполняются следующие действия:

- описание полей;
- задание ключа;
- задание индексов;
- определение ограничений на значения полей;
- определение условий (ограничений) ссылочной целостности;
- задание паролей;
- задание языкового драйвера;
- задание таблицы для выбора значений.

В этом списке обязательным является только первое действие, т. е. каждая таблица должна иметь хотя бы одно поле. Остальные действия выполняются при необ-

ходимости. Часть действий, таких как задание ключа и паролей, производится только для таблиц определенных форматов, например, для таблиц Paradox.

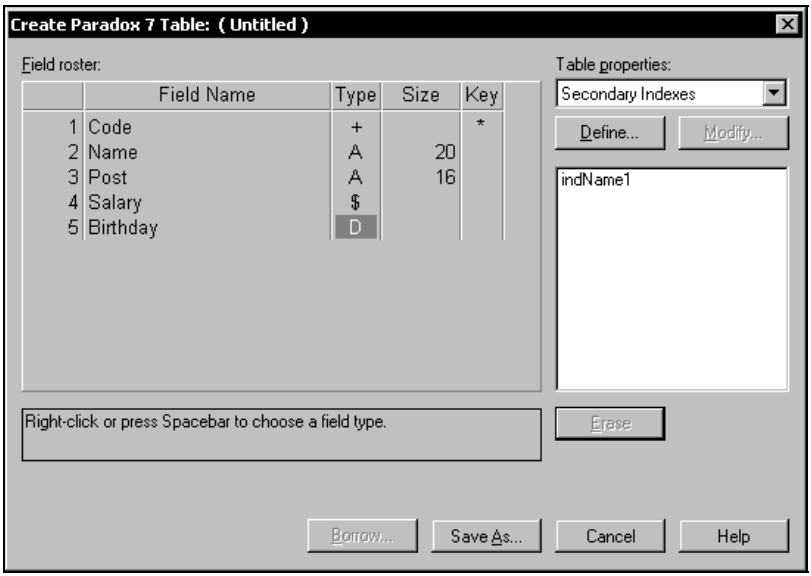


Рис. 4.2. Определение структуры таблицы

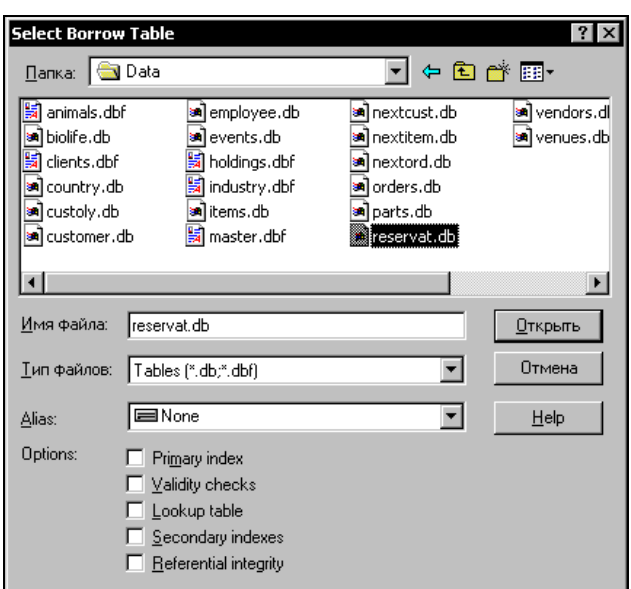


Рис. 4.3. Выбор таблицы для заимствования ее структуры

При создании новой таблицы, сразу после выбора ее формата, можно не создавать структуру таблицы, а скопировать ее из другой таблицы: при нажатии на кнопку **Borrow** (Взаймы) открывается окно **Select Borrow Table** (Выбор таблицы для заимствования) (рис. 4.3).

В этом окне можно выбрать таблицу (ее главный файл) и указать копируемые элементы структуры, установив соответствующий флажок, например, **Primary index** (Первичный индекс) для ключа. После нажатия кнопки **Open** из выбранной таблицы в новую таблицу копируются описания полей, а также те элементы, для которых установлен флажок. Если какой-либо элемент в структуре копируемой таблицы отсутствует, то состояние флажка не имеет значения. Например, если в выбранной таблице не определены ограничения ссылочной целостности, то в новой таблице они не появятся, даже если установлен флажок **Referential integrity** (Ссылочная целостность).

Впоследствии скопированную структуру можно настраивать, изменяя, добавляя или удаляя отдельные элементы.

После определения структуры таблицы ее необходимо сохранить, нажав кнопку **Save As** и указав расположение таблицы на диске и ее имя. В результате на диск записывается новая таблица, первоначально пустая, при этом все необходимые файлы создаются автоматически.

Описание полей

Центральной частью окна определения структуры таблицы является список **Field roster** (Список полей), в котором указываются поля таблицы. Для каждого поля задаются:

- имя поля — в столбце **Field Name**;
- тип поля — в столбце **Type**;
- размер поля — в столбце **Size**.

Имя поля вводится по правилам, установленным для выбранного формата таблиц. Правила именования и допустимые типы полей таблиц Paradox описаны в *главе 2*.

Тип поля можно задать, непосредственно указав соответствующий символ, например, `A` для символьного или `I` для целочисленного поля, или выбрать его в списке (рис. 4.4), раскрываемом нажатием клавиши <пробел> или щелчком правой кнопки мыши в столбце **Type**. Список содержит все типы полей, допустимые для заданного формата таблицы. В списке подчеркнуты символы, используемые для обозначения соответствующего типа, при выборе типа эти символы автоматически заносятся в столбец **Type**.

Размер поля задается не всегда, необходимость его указания зависит от типа поля. Для полей определенного типа, например, автоинкрементного (`+`) или целочисленного (`I`), размер поля не задается. Для поля строкового типа размер определяет максимальное число символов, которые могут храниться в поле.

Добавление к списку полей новой строки выполняется переводом курсора вниз на несуществующую строку, в результате чего эта строка появляется в конце списка. Вставка новой строки между существующими строками с уже описанными полями выполняется нажатием клавиши <Insert>. Новая строка вставляется перед строкой, в которой расположен курсор. Для удаления строки необходимо установить курсор на эту строку и нажать комбинацию клавиш <Ctrl>+<Delete>.

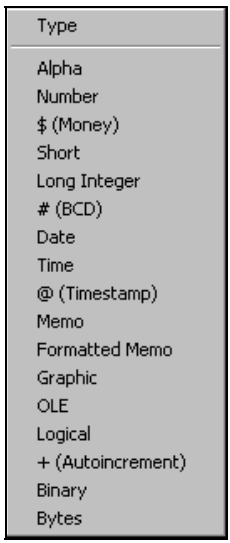


Рис. 4.4. Список типов для полей таблицы Paradox 7

Ключ создается указанием его полей. Для указания ключевых полей в столбце ключа (**Key**) нужно установить символ *, переведя в эту позицию курсор и нажав любую алфавитно-цифровую клавишу. При повторном нажатии клавиши отметка принадлежности поля ключу снимается. В структуре таблицы ключевые поля должны быть первыми, т. е. верхними в списке полей. Часто для ключа используют автоинкрементное поле (см. рис. 4.2).

Напомним, что для таблиц Paradox ключ также называют первичным индексом (Primary Index), а для таблиц dBase ключ не создается, и его роль выполняет один из индексов.

Для выполнения остальных действий по определению структуры таблицы используется комбинированный список **Table properties** (Свойства таблицы) (см. рис. 4.2), содержащий следующие пункты:

- **Secondary Indexes** (Вторичные индексы);
- **Validity Checks** (Проверка правильности ввода значений полей) — выбирается по умолчанию;
- **Referential Integrity** (Ссылочная целостность);

- **Password Security** (Пароли);
- **Table Language** (Язык таблицы, языковой драйвер);
- **Table Lookup** (Таблица выбора);
- **Dependent Tables** (Подчиненные таблицы).

После выбора какого-либо пункта этого списка в правой части окна определения структуры таблицы появляются соответствующие элементы, с помощью которых выполняются дальнейшие действия.

Состав данного списка зависит от формата таблицы. Так, для таблицы dBase он содержит только пункты **Indexes** и **Table Language**.

Задание индексов

Задание индекса сводится к определению:

- состава полей;
- параметров;
- имени.

Эти элементы устанавливаются или изменяются при выполнении операций создания, изменения и удаления индекса.

Замечание

Напомним, что для таблиц Paradox индекс называют также вторичным индексом.

Для выполнения операций, связанных с заданием индексов, необходимо выбрать пункт **Secondary Indexes** (Вторичные индексы) списка **Table properties** (Свойства таблицы), при этом под списком появляются кнопки **Define** (Определить) и **Modify** (Изменить), список индексов и кнопка **Erase** (Удалить). В списке индексов выводятся имена созданных индексов, на рис. 4.2 это индекс `indName1`.

Создание нового индекса начинается с нажатия кнопки **Define**, которая всегда доступна. Она открывает окно **Define Secondary Index** (Задание вторичного индекса), в котором задаются состав полей и параметры индекса (рис. 4.5).

В списке **Fields** окна выводятся имена всех полей таблицы, включая и те, которые нельзя включать в состав индекса, например, графическое поле или поле комментария. В списке **Indexed fields** (Индексные поля) содержатся поля, которые включаются в состав создаваемого индекса. Перемещение полей между списками выполняется выделением нужного поля (полей) и нажатием расположенных между этими списками кнопок с изображением горизонтальных стрелок. Имена полей, которые нельзя включать в состав индекса, выделяются в левом списке серым цветом. Поле не может быть повторно включено в состав индекса, если оно уже выбрано и находится в правом списке.

Замечание

При работе с записями индексные поля обрабатываются в порядке следования этих полей в составе индекса. Это нужно учитывать при указании порядка полей в индексе.

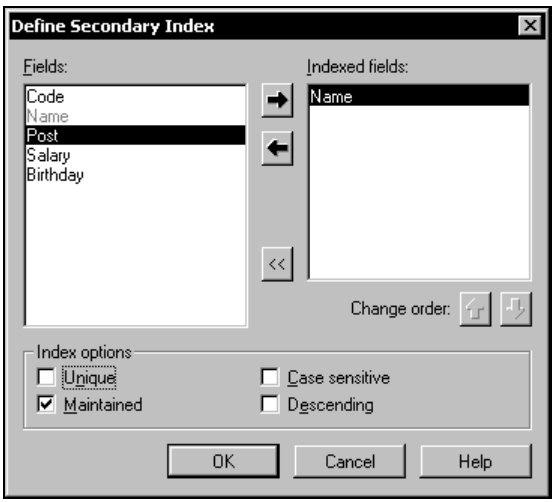


Рис. 4.5. Окно задания индекса

Изменить порядок следования полей в индексе можно с помощью кнопок с изображением вертикальных стрелок, имеющих общее название **Change order** (Изменить порядок). Для перемещения поля (полей) необходимо его (их) выделить и нажать нужную кнопку.

Флажки, расположенные в нижней части окна задания индекса, позволяют указать следующие параметры индекса:

- **Unique** — индекс требует для составляющих его полей уникальных значений;
- **Maintained** — задается автоматическое обслуживание индекса;
- **Case sensitive** — для полей строкового типа учитывается регистр символов;
- **Descending** — сортировка выполняется в порядке убывания значений.

Так как у таблиц dBase нет ключей, для них использование параметра **Unique** является единственной возможностью обеспечить уникальность записей на физическом уровне (уровне организации таблицы), не прибегая к программированию.

После задания состава индексных полей и нажатия кнопки **OK** появляется окно **Save Index As**, в котором нужно указать имя индекса (рис. 4.6). Для удобства обращения к индексу в его имя можно включить имена полей, указав какой-нибудь префикс, например, `ind`. Нежелательно образовывать имя индекса только из имен полей, т. к. для таблиц Paradox подобная система именования используется при автоматическом образовании имен для обозначения ссылочной цело-

стности между таблицами. После повторного нажатия кнопки **OK** сформированный индекс добавляется к таблице, и его имя появляется в списке индексов.

Созданный индекс можно изменить, определив новый состав полей, параметров и имени индекса. Изменение индекса практически не отличается от его создания. После выделения индекса в списке и нажатия кнопки **Modify** снова открывается окно задания индекса (см. рис. 4.5). При нажатии кнопки **OK** появляется окно сохранения индекса (см. рис. 4.6), содержащее имя изменяемого индекса, которое можно исправить или оставить прежним.

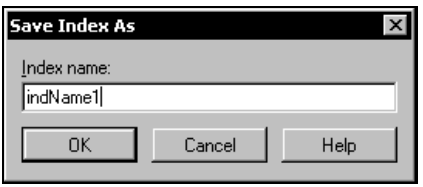


Рис. 4.6. Задание имени индекса

Для удаления индекса его нужно выделить в списке индексов и нажать кнопку **Erase**. В результате индекс удаляется без предупреждающих сообщений.

Кнопки **Modify** и **Erase** доступны, только если индекс выбран в списке.

Задание ограничений на значения полей

Задание ограничений на значения полей заключается в указании для полей:

- требования обязательного ввода значения;
- минимального значения;
- максимального значения;
- значения по умолчанию;
- маски ввода.

Замечание

Установленные ограничения задаются на физическом уровне (уровне таблицы) и действуют для любых программ, выполняющих операции с таблицей: как для программ типа Database Desktop, так и для приложений, создаваемых в Delphi. Дополнительно к этим ограничениям или вместо них в приложении можно задать *программные* ограничения.

Для выполнения операций, связанных с заданием ограничений на значения полей, нужно выбрать пункт **Validity Checks** (Проверка значений) комбинированного списка **Table properties** (см. рис. 4.2), при этом ниже списка появляются флажок **Required Field** (Обязательное поле), поля редактирования **Minimum Value**, **Maximum Value**, **Default Value**, **Picture** (Маска ввода) и кнопка **Assist** (По-

мощь). Флажок и поля редактирования отображают установки для поля таблицы, которое выбрано в списке (курсор находится в строке этого поля).

Требование обязательного ввода значения означает, что поле не может быть пустым (иметь значение `Null`). Это требование действует при добавлении к таблице новой записи. До того как изменения в таблице будут подтверждены, поле должно получить какое-либо непустое значение, в противном случае генерируется ошибка. Ошибка может также возникнуть при редактировании записи, когда будет удалено старое значение поля и не присвоено новое.

Данное требование удобно использовать для так называемых *обязательных полей* таблиц, например, для поля фамилии в таблице сотрудников организации.

Замечание

Обязательность ввода значения не действует на автоинкрементное поле, которое и без того является обязательным и автоматически заполняемым.

Для указания обязательности ввода значения в поле необходимо установить флажок **Required Field**, который по умолчанию снят.

Для полей некоторых типов, в первую очередь числовых, денежных, строковых и даты, иногда удобно задавать диапазон возможных значений, а также значение по умолчанию. Диапазон определяется минимальным и максимальным возможными значениями, которые вводятся в полях редактирования **Minimum Value** и **Maximum Value**. После их задания выход значения поля за указанные границы не допускается при вводе и редактировании любым способом.

Значение поля по умолчанию указывается в поле **Default Value**. Это значение устанавливается при добавлении новой записи, если при этом для поля не указано какое-либо значение.

Замечание

Задание диапазона и значений по умолчанию возможно не для всех полей, например, они не определяются для графического поля и поля комментария. Для этих полей соответствующее поле ввода в диалоговом окне определения структуры таблицы (см. рис. 4.2) блокируется.

В поле ввода **Picture** можно задать маску (шаблон) для ввода значения поля. Ввод по маске поддерживается, например, для таких типов полей, как числовой или строковый. Его удобно использовать для ввода информации определенных форматов, например, телефонных номеров или почтовых индексов.

Для маски используются следующие символы:

- `#` (цифра);
- `?` (любая буква; регистр не учитывается);
- `&` (любая буква; преобразуется к верхнему регистру);
- `~` (любая буква; преобразуется к нижнему регистру);
- `@` (любой символ);

- `!` (любой символ; преобразуется к верхнему регистру);
- `;` (за этим символом следует буква);
- `*` (число повторов следующего символа);
- `[abc]` или `{a,b,c}` (любой из приведенных символов — `a`, `b`, или `c`; во втором случае значения перечисляются через запятую без пробелов).

Маску можно ввести в поле ввода **Picture** вручную или использовать для этого диалоговое окно **Picture Assistance** (Помощник представления), вызываемое нажатием кнопки **Assist** (рис. 4.7).

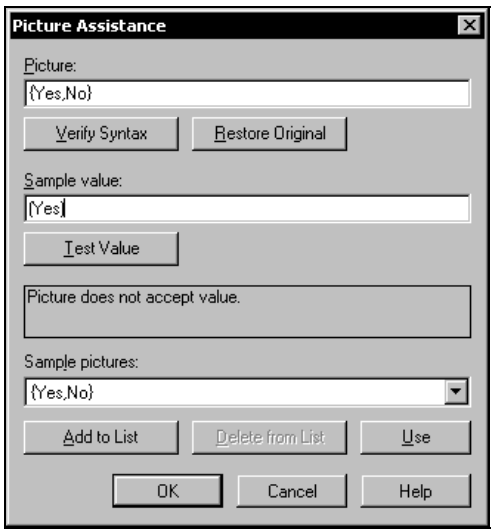


Рис. 4.7. Диалоговое окно формирования маски

Указанное окно помогает ввести, выбрать или откорректировать маску, а также проверить ее функционирование.

Список **Sample pictures** содержит образцы масок, которые выбираются нажатием кнопки **Use**. Выбранная маска помещается в поле ввода **Picture** и доступна для изменения. Для модификации списка образцов масок служат кнопки **Add to List** и **Delete from List**: первая добавляет к списку маску, содержащуюся в поле ввода **Picture**, а вторая удаляет из списка **Sample pictures** выбранную маску.

Проверка синтаксиса маски выполняется по нажатию кнопки **Verify Syntax**, результат проверки выводится в информационной панели. Кнопка **Restore Original** (Вернуть исходную) служит для восстановления начального (т. е. до начала редактирования) значения маски.

Функционирование маски можно проверить, введя в поле ввода **Sample value** значение поля таблицы. По нажатию кнопки **Test Value** выполняется проверка введенного значения, результат проверки выводится в информационной панели.

Задание ссылочной целостности

Понятие ссылочной целостности относится к связанным таблицам и проявляется в следующих вариантах взаимодействия таблиц:

- запрещается изменять поле связи или удалять запись главной таблицы, если для нее имеются записи в подчиненной таблице;
- при удалении записи в главной таблице автоматически удаляются соответствующие ей записи в подчиненной таблице (каскадное удаление).

Для выполнения операций, связанных с заданием ссылочной целостности, необходимо выбрать пункт **Referential Integrity** комбинированного списка **Table properties** (см. рис. 4.2). При этом, как и в случае задания индексов, появляются кнопки **Define**, **Modify**, **Erase** и список, в котором выводятся имена созданных условий ссылочной целостности.

Условие ссылочной целостности задается для подчиненной таблицы и определяется следующими элементами:

- полями связи подчиненной таблицы;
- именем главной таблицы;
- полями связи главной таблицы;
- параметрами.

Разработчик может создать, изменить или удалить условие ссылочной целостности.

Для задания условия ссылочной целостности нужно нажать кнопку **Define**, после чего появляется окно **Referential Integrity** (рис. 4.8).

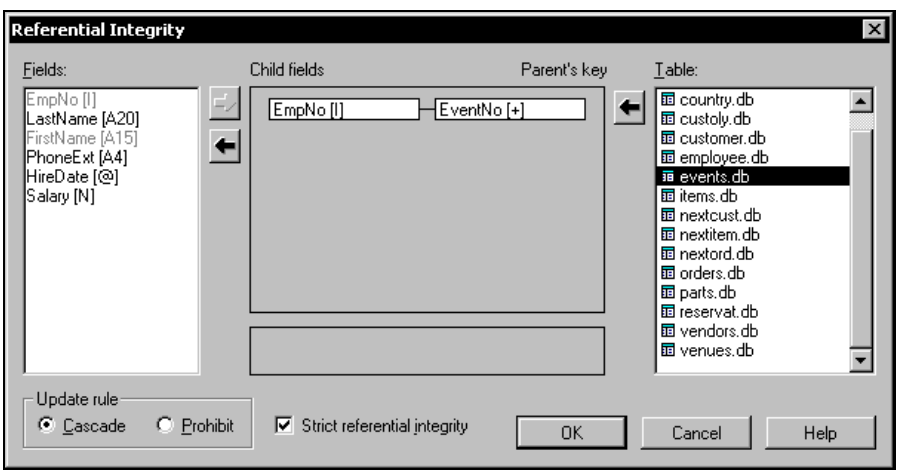


Рис. 4.8. Задание условия ссылочной целостности

В списке **Fields** следует выбрать поле связи и нажатием кнопки со стрелкой вправо перевести его в список **Child fields** (Дочерние поля). Если полей связи

несколько, то эти действия выполняются для каждого из них. Кнопка со стрелкой влево удаляет выбранное поле из списка дочерних полей.

В списке **Table** указывается главная таблица, имена таблиц выбираются в рабочем каталоге программы Database Desktop (каталог, определенный как Working Directory, его задание описано в *главе 12*). После выбора таблицы и нажатия кнопки со стрелкой влево (рядом со списком таблиц) в поле **Parent's key** автоматически заносятся ключевые поля главной таблицы.

Замечание

Главная таблица должна быть закрыта и не использоваться другими программами, включая Database Desktop. В противном случае при попытке сохранить структуру таблицы возникает ошибка.

Параметры ссылочной целостности выбираются переключателями. Группа **Update rule** (Правила изменения) определяет вид взаимодействия таблиц при изменениях в главной таблице. Переключатель **Cascade** устанавливает режим каскадного удаления записей в подчиненной таблице при удалении соответствующей записи главной таблицы. Переключатель **Prohibit** устанавливает режим запрещения изменения поля связи или удаления записи главной таблицы, для которой имеются записи в подчиненной таблице.

Флажок **Strict referential integrity** (Жесткая ссылочная целостность) устанавливает защиту таблиц от модификации с использованием ранних версий программы Database Desktop (под DOS), которые не поддерживают ссылочную целостность.

После установки нужных режимов и нажатия кнопки **OK** появляется окно **Save Referential Integrity As**, в котором нужно указать имя условия (рис. 4.9). Напомним, что для таблиц Paradox условия ссылочной целостности именуются. Для удобства обращения к условию в его имя можно включить имена полей и таблиц, задав при этом некоторый префикс, например, `ri`. После нажатия кнопки **OK** сформированное условие ссылочной целостности добавляется к таблице, и его имя появляется в списке условий.

Замечание

Условия ссылочной целостности задаются на физическом уровне и действуют для любых программ, выполняющих операции с таблицей: как для программ типа Database Desktop, так и для приложений, создаваемых в Delphi.

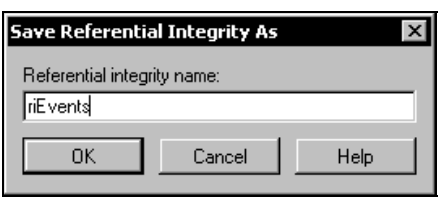


Рис. 4.9. Задание имени для условия ссылочной целостности

Созданное условие ссылочной целостности можно изменить, определив новый состав полей и новые значения параметров. Изменение условия ссылочной целостности практически не отличается от его создания: после выделения имени условия в списке и нажатия кнопки **Modify** открывается окно определения ссылочной целостности (см. рис. 4.8). При нажатии кнопки **OK** измененное условие ссылочной целостности сохраняется под тем же именем.

Для удаления условия ссылочной целостности нужно выделить его в списке и нажать кнопку **Erase**. Удаление производится без выдачи предупреждающих сообщений.

Кнопки **Modify** и **Erase** доступны, только если выбрано условие в списке.

Задание паролей

Пароль позволяет задать права доступа пользователей (приложений) к таблице. Если для таблицы установлен пароль, то он будет автоматически запрашиваться при каждой попытке открытия таблицы.

Замечание

Пароль действует на физическом уровне и его действие распространяется на все программы, выполняющие доступ к таблице: как на программы типа Database Desktop, так и на создаваемые приложения Delphi.

Для выполнения операций, связанных с заданием пароля, нужно выбрать строку **Password Security** в комбинированном списке **Table properties** окна определения структуры таблицы (см. рис. 4.2). При этом под списком становятся доступными кнопки **Define** и **Modify**. Нажатие кнопки **Define** открывает окно **Password Security** (рис. 4.10), в котором задается пароль.

Рис. 4.10. Задание главного пароля

Пароль таблицы вводится дважды — в полях **Master password** (Главный пароль) и **Verify master password** (Подтвердить главный пароль). При нажатии кнопки

OK оба значения сверяются, и при их совпадении пароль принимается. Когда пароль определен, кнопка **Define** блокируется и становится доступной кнопка **Modify** изменения пароля. Ее нажатие снова вызывает окно задания пароля (см. рис. 4.10), в котором появляются кнопки **Change** и **Delete**, а поля ввода заблокированы.

Нажатие кнопки **Delete** удаляет пароль, после чего его можно ввести заново. Если в качестве значения пароля указана пустая строка, то пароль для таблицы не задан. Нажатием кнопки **Change** поля ввода разблокируются, и значение пароля можно изменить (это нужно сделать в обоих полях). При этом название кнопки **Change** изменяется на **Revert** (Возврат), и ее повторное нажатие возвращает значение пароля, которое было до начала редактирования.

Рассмотренный пароль считается главным паролем, который предоставляет пользователю *полные права доступа* к таблице, включая изменение записей и структуры таблицы, в том числе смену пароля. Кроме главного пароля, можно задать для таблицы дополнительные пароли, устанавливающие пользователю *ограниченные права доступа* к таблице. Для задания дополнительных паролей нажатием кнопки **Auxiliary Passwords** открывается одноименное окно (рис. 4.11).

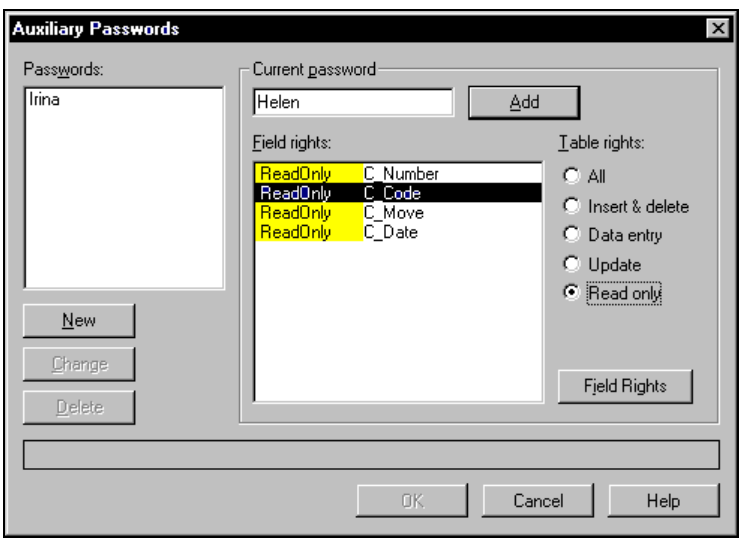


Рис. 4.11. Задание дополнительных паролей

В списке **Passwords** выводятся действующие дополнительные пароли. Группа переключателей **Table rights** определяет для пароля права доступа к таблице в целом. Они могут быть следующими:

- **All** — полные права, включая изменение записей и структуры таблицы;
- **Insert & delete** — разрешены вставка и удаление, а также редактирование записей, запрещено изменение структуры таблицы;

- **Data entry** — разрешены редактирование и вставка записей, запрещены изменение структуры таблицы и удаление записей;
- **Update** — разрешены только просмотр (чтение) записей и редактирование неключевых полей;
- **Read only** — разрешен только просмотр (чтение) записей.

Права доступа к таблице действуют на все ее поля; кроме того, для каждого поля можно установить отдельные права доступа, не зависящие от прав доступа к другим полям. Права доступа к полям выводятся слева от имени поля в списке **Fields rights** и могут иметь следующие значения:

- **All** (Чтение и изменение значения поля);
- **ReadOnly** (Только чтение значения поля);
- **None** (Доступ к полю запрещен).

Смена права доступа к полю выполняется выбором поля в списке и нажатием кнопки **Fields Rights**, при котором право циклически устанавливается очередным значением списка (**All**, **ReadOnly** и **None**).

Создание пароля начинается нажатием кнопки **New**, после чего его имя указывается в поле **Current password** (Текущий пароль) и устанавливаются права доступа к таблице и ее полям. Нажатие кнопки **Add** заносит пароль в список дополнительных паролей.

Нажатие кнопки **Change** переводит выбранный в списке пароль в режим редактирования, при этом появляются кнопки **Accept** (Подтвердить) и **Revert** (Возврат), а также разблокируется кнопка **Delete**. В процессе редактирования пароль можно изменить, удалить или оставить без изменений. После смены пароля и прав доступа внесенные изменения утверждаются нажатием кнопки **Accept**. Для выхода из режима редактирования и возврата к прежним установкам пароля необходимо нажать кнопку **Revert**. Удаляется пароль нажатием кнопки **Delete**.

Отметим, что из приложения паролями можно управлять с помощью методов компонента `Session`. Управление паролями заключается в их добавлении и удалении. Процедура `AddPassword (const Password: String)` добавляет новый пароль, заданный параметром `Password`; процедура `RemovePassword (const Password: String)` удаляет указанный пароль, а процедура `RemoveAllPasswords` удаляет все пароли.

Задание языкового драйвера

Для задания языкового драйвера нужно выбрать пункт **Table Language** (Язык таблицы) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 4.2). При этом под списком становится доступной кнопка **Modify**, открывающая окно **Table Language** (рис. 6.12). Под "языком" таблицы понимается языковой драйвер, используемый для этой таблицы. В поле списка **Language** отображается текущий драйвер.

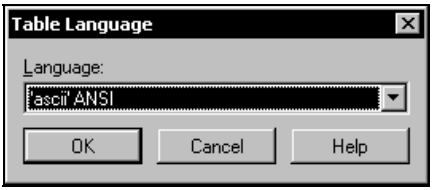


Рис. 4.12. Выбор языкового драйвера

В списке можно выбрать драйвер нужного языка, для русского языка это драйвер Pdox ANSI Cyrillic, который корректно отображает символы русского алфавита и выполняет с ними операции сортировки.

По умолчанию языковой драйвер определяется установками процессора баз данных BDE, поэтому целесообразно установить его (параметр `LANGDRIVER`) в нужное значение, например, с помощью программы Administrator BDE. Параметры BDE и вопросы, связанные с их настройкой, рассматриваются в *главе 12*.

Задание таблицы для выбора значений

Часто возникает ситуация, когда в поле должны заноситься значения из какого-либо набора, который может формироваться различными способами. Одним из часто используемых является вариант, когда эти значения содержатся в поле другой таблицы, а совокупность значений всех записей этого поля образует набор допустимых значений.

Для полей некоторых типов, например, строкового, числового или даты, можно определить некоторую таблицу *(таблицу выбора)*, поля которой будут использоваться для формирования набора допустимых значений. Если для поля задана таблица выбора, то в него можно ввести только значение, содержащееся в таблице выбора (в указанном поле любой записи). Задание таблицы выбора гарантирует, что в поле не будет введено недопустимое значение.

Действие набора допустимых значений распространяется также на редактирование записей таблицы программным способом: при попытке присвоить полю недопустимое значение генерируется исключение.

Замечание

При вводе строковых значений учитывается регистр букв, поэтому, например, `Водитель` и `ВОДИТЕЛЬ` являются разными значениями.

В любом случае допустимым является пустое значение (`Null`), если, конечно, не задано ограничение, что поле не может быть пустым.

Для выполнения операций, связанных с полями выбора, предназначен пункт **Table Lookup** (Таблица выбора) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 4.2). При этом под списком становит-

ся доступной кнопка **Define**, нажатие которой открывает окно **Table Lookup** (рис. 4.13).

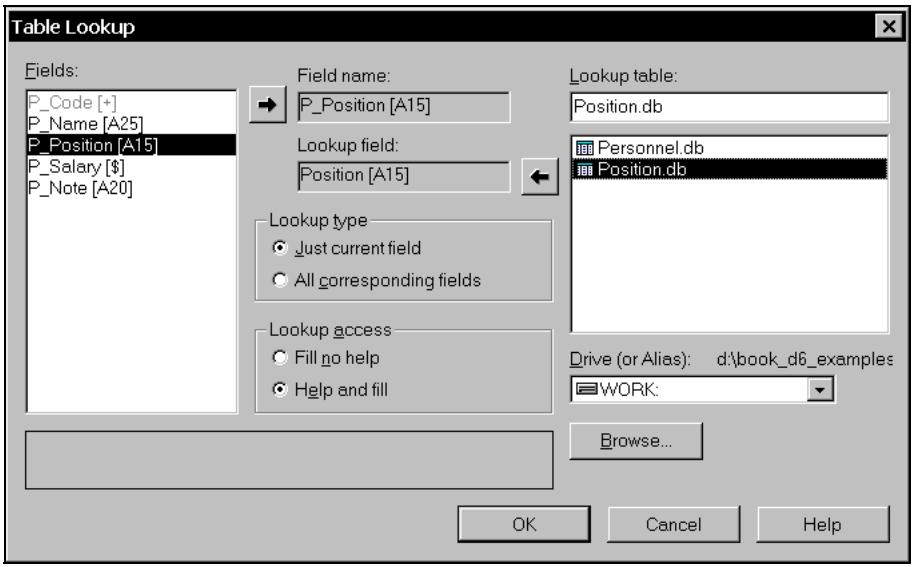


Рис. 4.13. Окно задания таблицы выбора

В списке **Fields** выводятся имена всех полей таблицы, при этом имена полей, которые не допускают создание таблицы выбора, выделены серым цветом (например, поле автоинкрементного типа).

Имя поля, для которого задается таблица выбора, отображается в области **Field name**, для его указания следует выделить в списке **Fields** нужную строку и нажать кнопку с изображением стрелки вправо. Если указать другое поле и нажать кнопку, то имя этого поля перейдет в область **Field name** и заменит имя предыдущего поля.

В списке **Lookup table** задается таблица выбора, имя которой (имя главного файла) появляется в поле редактирования над списком. В списке **Drive (or Alias)** задается диск или псевдоним, которые определяют местоположение таблицы выбора. Название каталога, таблицы которого содержатся в списке **Lookup table**, отображается справа от названия списка **Drive (or Alias)**. Таблицу также можно задать в окне **Select File**, открываемом нажатием кнопки **Browse** (Просмотр).

После задания таблицы выбора нажатие кнопки с изображением стрелки влево переводит имя *ее первого поля*, значения которого будут использованы для формирования набора допустимых значений, в область **Lookup field** (Поле выбора). Типы полей обеих таблиц должны совпадать, в противном случае в информационной панели выдается сообщение об ошибке.

С помощью группы переключателей **Lookup type** (Тип выбора) можно задать способ взаимодействия обеих таблиц. Если включен переключатель **Just current field** (Только текущее поле), то таблица выбора задается только для поля, указанного в области **Field name**. Переключатель **All corresponding fields** назначает таблицу выбора не только указанному полю, но и всем соответствующим полям. Имена и типы этих полей должны совпадать с соответствующими полями таблицы выбора.

Группа переключателей **Lookup access** (Доступ к таблице выбора) определяет, как пользователь использует значения из таблицы выбора. Если включен переключатель **Fill no help** (Вставка без помощи), то при редактировании поля, для которого определена таблица выбора, пользователь должен знать допустимые значения этого поля. При этом термин "выбор" не совсем точно отражает взаимосвязь полей таблиц, т. к. пользователю не предлагается список возможных значений, из которых он может выбрать нужное ему. Пользователь сам вводит значение в поле, при этом на уровне таблицы выполняется проверка, является ли это значение допустимым. Если значение отсутствует в указанном поле таблицы выбора, то оно не принимается.

В этом случае может помочь отображение рядом с редактируемой таблицей таблицы выбора, например, как показано на рис. 4.14.

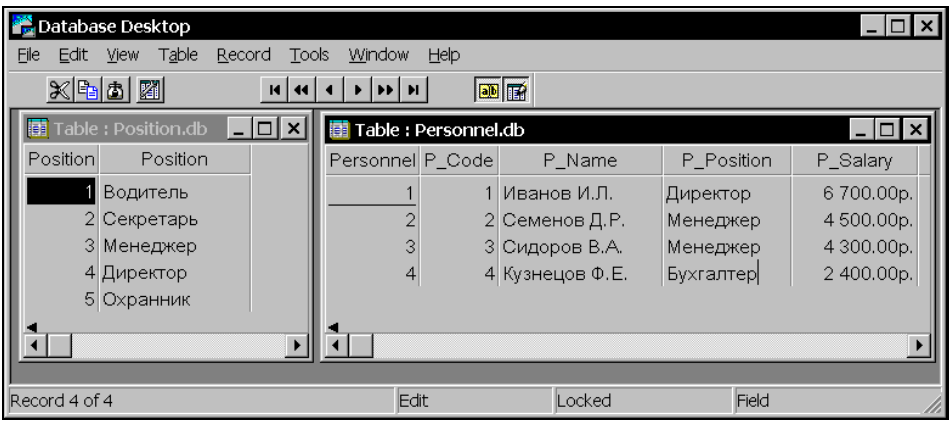


Рис. 4.14. Функционирование поля, для которого задана таблица выбора

В приведенном на этом рисунке примере в окне программы Database Desktop таблица `Personnel` содержит данные о сотрудниках организации, в ней определены поля для кода (`P_Code`), фамилии (`P_Name`), должности (`P_Position`), оклада (`P_Salary`) и примечания (`P_Note`). В строковое поле должности можно вводить только допустимое название должности. Перечень всех должностей содержится в единственном поле `Position` таблицы `Position`. Для предотвращения ввода в таблицу `Personnel` неправильных значений ее полю `P_Position` назначена таблица выбора `Position` с полем `Position`. В таблице `Personnel` для

первых трех сотрудников введены должности, имеющиеся в таблице `Position`. Попытка задать для четвертого сотрудника должность `Бухгалтер` блокируется, т. к. эта должность не является разрешенной для ввода. При необходимости эту должность сначала нужно ввести в таблицу `Position`, после чего она станет доступной для таблицы `Personnel`.

Установка переключателя **Help and fill** (Помощь и вставка) позволяет не только ввести значение в поле, как описано выше, но и действительно выбрать значение в списке. Список, сформированный на основании значений поля (полей) таблицы выбора (рис. 4.15), появляется при нажатии комбинации клавиш <Ctrl>+<пробел> в редактируемом поле. После выбора нужного значения и нажатия кнопки **OK** оно заносится в поле.

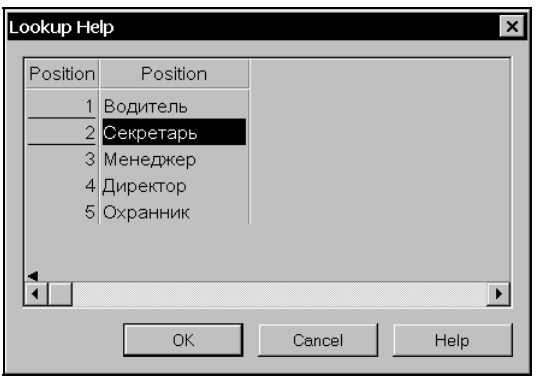


Рис. 4.15. Список выбора

Замечание

Из таблицы выбора могут быть удалены значения. Если эти значения содержатся в полях таблицы, использующей таблицу выбора, то при переходе в режим ее редактирования возникает ошибка. При использовании программы типа Database Desktop таблицу нельзя вывести из режима редактирования, пока все ее значения не приведены в соответствие с новыми значениями таблицы выбора. Во время выполнения приложения при доступе к таблице, использующей удаленные из таблицы выбора значения, генерируется исключение.

После задания для поля таблицы выбора в окне определения структуры таблицы (см. рис. 4.2) становятся доступными кнопки **Modify** и **Erase**, а также список, в котором отображается имя этой таблицы.

Нажатие кнопки **Modify** вновь открывает окно задания таблицы выбора, в котором можно ввести новые данные. Кнопка **Erase** служит для отмены использования таблицы выбора и связанных с ней ограничений на значения поля.

Отметим, что объекты поля типа `TField` наборов данных `Table` и `Query`, а также компонент `DBGrid` позволяют определить для поля список выбора, который

также дает возможность пользователю выбирать значения при редактировании. Они задаются на программном уровне и действуют только в своем приложении. Определение и использование подобных списков выбора рассматриваются в *следующих двух главах.*

Просмотр списка подчиненных таблиц

Таблица, связанная с другими таблицами, является либо главной, либо подчиненной. Выше мы рассмотрели задание ссылочной целостности для подчиненной таблицы. Для главной таблицы можно просмотреть список подчиненных таблиц, отображаемый при выборе пункта **Dependent Table** (Подчиненная таблица) все того же комбинированного списка **Table properties** (см. рис. 4.2). Этот список содержит имена всех таблиц, имеющих условия ссылочной целостности с участием данной таблицы.

Изменение структуры таблицы

Структуру существующей таблицы можно изменить, выполнив команду **Table\Restructure** после предварительного выбора таблицы в окне программы Database Desktop. В результате открывается окно определения структуры таблицы, и дальнейшие действия не отличаются от действий, выполняемых при создании таблицы.

Замечание

При изменении структуры таблицы с ней не должны работать другие приложения, в том числе Delphi. Поэтому предварительно необходимо закрыть Delphi или приложение, в котором компоненты `Table` связаны с перестраиваемой таблицей. Другим вариантом является отключение активности компонентов `Table`, связанных с перестраиваемой таблицей, для чего свойству `Active` этих компонентов через Инспектора объектов устанавливается значение `False`.

Переименование таблицы следует выполнять из среды программы Database Desktop, а не из среды Windows, например, с помощью Проводника. Для этого при работе со структурой таблицы можно нажать кнопку **Save As** и задать новое имя таблицы. В результате в указанном каталоге диска появятся все необходимые файлы таблицы. При этом старая таблица также сохраняется. Информация о названии таблицы используется внутри ее файлов, поэтому простое переименование всех файлов таблицы приведет к ошибке при попытке доступа к ней.

Если необходимо просто ознакомиться со структурой таблицы, то выполняется команда **Table\Into Structure1**. В результате появляется окно определения структуры таблицы, но элементы, с помощью которых в структуру таблицы могут быть внесены изменения, заблокированы. Просмотр структуры возможен также для таблицы, с которой связаны другие приложения.

Создание приложения BDE

Для примера рассмотрим создание приложения, использующего механизм доступа BDE и позволяющего перемещаться по записям таблицы БД, просматривать и редактировать поля, удалять записи из таблицы, а также вставлять новые. Файл проекта приложения обычно не требует от разработчика выполнения каких-либо действий. Поэтому при создании приложения главной задачей является конструирование форм, в простейшем случае — одной формы.

Вид формы приложения на этапе проектирования показан на рис. 4.16, где в форме размещены компоненты `Table1`, `DataSource1`, `DBGrid1` и `DBNavigator1`.

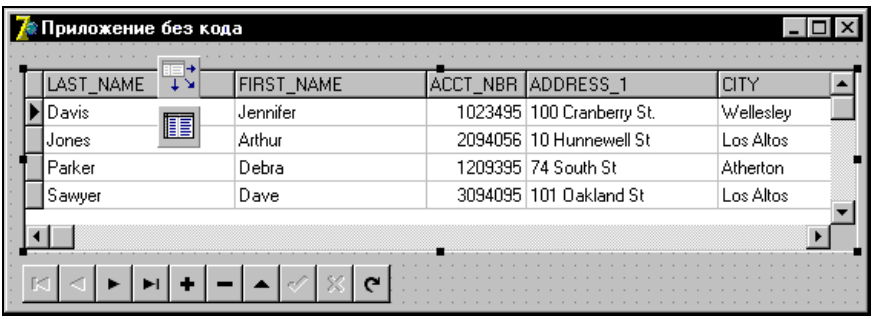


Рис. 4.16. Форма приложения для работы с БД

Компонент `Table1` обеспечивает взаимодействие с таблицей БД. Для связи с требуемой таблицей нужно установить в соответствующие значения свойство `DataBaseName`, указывающее путь к БД, и свойство `TableName`, указывающее имя таблицы. После задания таблицы для открытия набора данных свойство `Active` должно быть установлено в значение `True`.

Замечание

Значение `True` свойства `Active` нужно устанавливать после задания таблицы БД, т. е. после установки нужных значений свойств `DataBaseName` и `TableName`.

Имя таблицы лучше выбирать в раскрывающемся списке в поле значения свойства `TableName`. Если путь к БД (свойство `DataBaseName`) задан правильно, то в этом списке отображаются главные файлы всех доступных таблиц.

В рассматриваемом приложении использована таблица клиентов, входящая в состав поставляемых с Delphi примеров, ее главный файл — Clients.dbf. Файлы этой и других таблиц примеров находятся в каталоге, путь к которому указывает псевдоним `dbdemos`. Настройка псевдонима может быть выполнена с помощью программы BDE Administrator.

Компонент `DataSource1` является промежуточным звеном между компонентом `Table1`, соединенным с реальной таблицей БД, и визуальными компонентами

DBGrid1 и DBNavigator1, с помощью которых пользователь взаимодействует с этой таблицей. На компонент Table1, с которым связан компонент DataSource1, указывает свойство DataSet последнего.

Компонент DBGrid1 отображает содержимое таблицы БД в виде сетки, в которой столбцы соответствуют полям, а строки — записям таблицы. По умолчанию пользователь может просматривать и редактировать данные. Компонент DBNavigator1 позволяет пользователю перемещаться по таблице, редактировать, вставлять и удалять записи. Компоненты DBGrid1 и DBNavigator1 связываются со своим источником данных — компонентом DataSource1 — через свойства DataSource.

Взаимосвязь компонентов приложения и таблицы БД и используемые при этом свойства компонентов показаны на рис. 4.17.

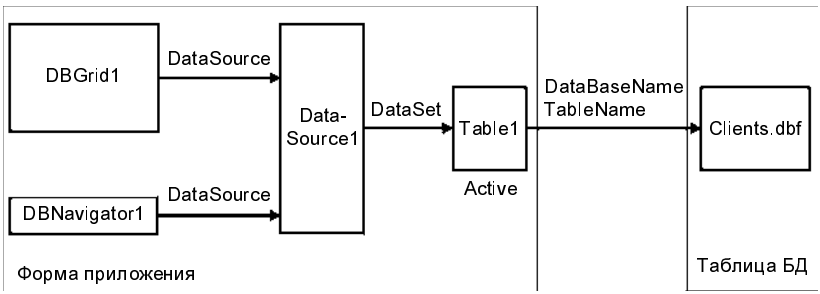


Рис. 4.17. Взаимосвязь компонентов приложения и таблицы БД

Разрабатывая приложение, можно задавать значения всех свойств компонентов с помощью Инспектора объектов. При этом требуемые значения либо непосредственно вводятся в поле, либо выбираются в раскрывающихся списках. В последнем случае приложение создается с помощью мыши и не требует набора каких-либо символов на клавиатуре. В табл. 4.1 приведены компоненты, используемые для работы с таблицей БД, их основные свойства и значения этих свойств.

Таблица 4.1. *Значения свойств компонентов*

Компонент	Свойства	Значения
Table1	DataBaseName	dbdemos
	TableName	Clients.dbf
	Active	True
DataSource1	DataSet	Table1
DBGrid1	DataSource	DataSource1
DBNavigator1	DataSource	DataSource1

В дальнейшем при организации приложений, использующих механизм доступа BDE, предполагается, что названные компоненты связаны между собой именно таким образом, и свойства, с помощью которых эта связь осуществляется, не рассматриваются.

Для автоматизации процесса создания формы, использующей компоненты для операций с БД, можно вызвать **Database Form Wizard** (Мастер форм баз данных), показанный на рис. 4.18. Этот Мастер расположен на странице **Business** Хранилища объектов.

Мастер позволяет создавать формы для работы с отдельной таблицей и со связанными таблицами, при этом можно использовать наборы данных `Table` или `Query`.

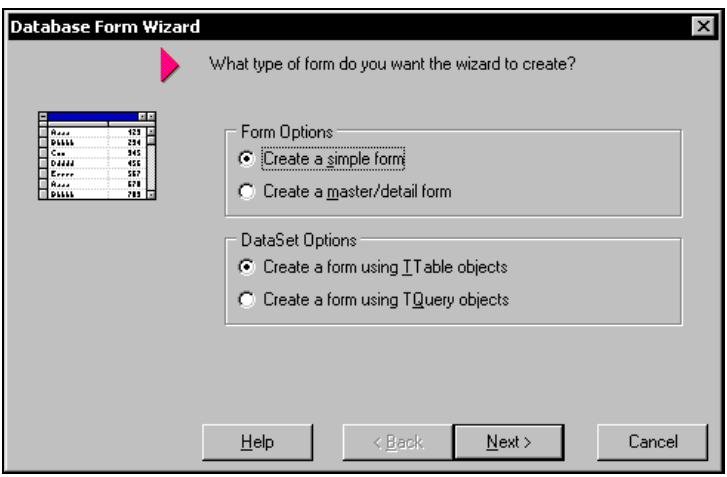


Рис. 4.18. Окно Мастера Database Form Wizard

Использование модуля данных

При конструировании формы невизуальные компоненты, используемые для доступа к данным, такие как `DataSource` или `Table`, размещаются в форме, но при выполнении приложения эти компоненты не видны. Поэтому их можно размещать в любом удобном месте формы, которая для них является контейнером — модулем. Кроме того, для размещения невизуальных компонентов, через которые осуществляется доступ к данным, предназначен специальный объект — *модуль данных*.

Есть три типа модулей данных:

- простой модуль данных;
- удаленный модуль данных;
- Web-модуль.

Ниже рассматривается *простой модуль* данных, который представлен объектом `DataModule`. Использование *удаленного модуля* данных и *Web-модуля* подробно изучается при рассмотрении трехуровневых приложений и публикации БД в Интернете *(см. главы 17, 18, 20)*.

Если применяется простой модуль данных, то взаимосвязь компонентов приложения и таблицы БД имеет вид, показанный на рис. 4.19.

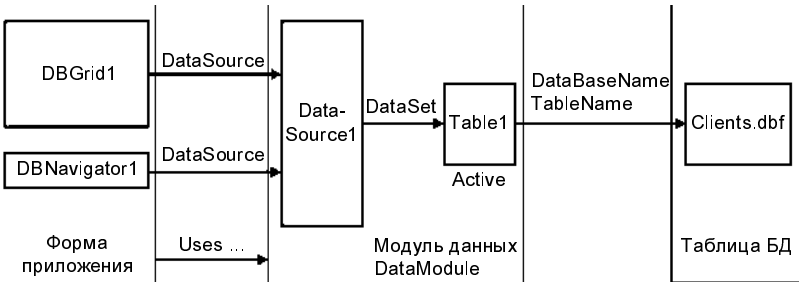


Рис. 4.19. Взаимосвязь компонентов приложения и таблицы БД при использовании модуля данных

Модуль данных, как и форма, является контейнером для своих невизуальных компонентов, и для него создается модуль кода с расширением pas. Добавление модуля данных к проекту выполняется командой **File\New\DataModule** главного меню Delphi. В окне модуля компоненты размещаются так же, как и в форме (рис. 4.20). При выборе объекта в окне Инспектора объектов отображаются его свойства, значения которых можно просматривать и изменять.

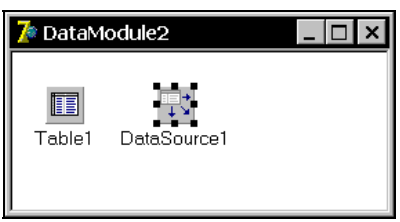


Рис. 4.20. Модуль данных

При обращении к содержащимся в модуле данных компонентам для них указывается составное имя, в которое, кроме имени компонента, входит имя модуля данных. Составное имя имеет формат

```
<Имя модуля данных>.<Имя компонента>
```

Ниже приводится пример кода, в котором осуществляется обращение к компонентам модуля данных.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
DataModule2.Table1.DatabaseName := 'dbdemos';
DataModule2.Table1.TableName := 'Clients.dbf';
DataModule2.DataSource1.DataSet := DataModule2.Table1;
DBGrid1.DataSource := DataModule2.DataSource1;
DBNavigator1.DataSource := DataModule2.DataSource1;
DataModule2.Table1.Active := True;
end;
```

Для компонентов устанавливаются значения свойств, связывающих между собой эти компоненты и таблицу БД. Значения свойств устанавливаются *динамически* в процессе выполнения приложения, для чего использован обработчик события создания главной формы приложения. В составных именах компонентов доступа к данным, которыми являются источник данных `DataSource1` и набор данных `Table1`, указывается имя модуля данных `DataModule2`.

Чтобы обеспечить возможность доступа к компонентам модуля данных в модуле формы, в список `uses` раздела `implementation` необходимо включить ссылку на модуль данных:

```
uses unit2;
```

Ссылку на другой модуль можно написать самостоятельно, но Delphi позволяет вставить ее автоматически. При выборе команды **File\Use Unit** появляется диалоговое окно **Use Unit**. После выбора нужного модуля и нажатия кнопки **OK** он добавляется в список.

Если ссылка на требуемый модуль отсутствует, но в коде используется имя модуля данных, то при компиляции приложения появляется диалоговое окно **Information** (рис. 4.21). Оно сообщает, что форма ссылается на другую форму, объявленную в модуле, отсутствующем в списке `uses` формы. Для автоматического добавления модуля в список достаточно нажать кнопку **Yes**.

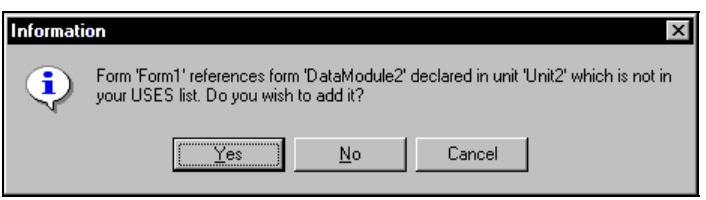


Рис. 4.21. Диалоговое окно **Information**

Помимо компонентов доступа к данным, которыми являются `Session`, `Database`, `Table`, `Query`, `StoredProc`, `BatchMove` и др., в модуле данных можно размещать невизуальные компоненты, не имеющие прямого отношения к БД, например, `ImageList`, `OpenDialog` или `Timer`.

Замечание

При работе с модулем данных в Палитре компонентов доступны только невизуальные компоненты.

Модуль данных позволяет:

- отделить управление БД от обработки данных;
- создать модуль, совместно используемый несколькими приложениями.

Основным назначением модуля данных является *централизованное хранение* компонентов доступа к данным, а также кода для этих компонентов, в частности, обработчиков событий. В модуле данных удобно размещать код, выполняющий управление БД, например, реализацию бизнес-правил.

Использование простого модуля данных несколькими приложениями позволяет ускорить разработку приложений, т. к. готовый модуль данных впоследствии можно включать в новые приложения. Кроме того, управление БД через общий модуль дает возможность определить для всех пользователей одинаковые режимы и правила работы с базой, а также делает более простым изменение этих режимов и правил.

Однако для небольших приложений использование простого модуля данных не всегда оправданно, т. к. может затруднить, а не облегчить разработку приложения.

Удаленный модуль данных предназначен для работы с удаленными БД в трехуровневой архитектуре "клиент-сервер" и используется для создания *сервера приложения* — промежуточного уровня между приложением и сервером БД.

Web-модуль предназначен для работы с БД в сети Интернет и является посредником между обозревателем (программой просмотра Web-документов) и сервером БД. Использование Web-модуля рассмотрено в *главе 20,* посвященной публикации баз данных средствами Delphi.

Глава 5

Компоненты доступа к данным

Компоненты доступа к данным являются невизуальными. В этой главе мы рассмотрим основные компоненты доступа к данным, которые используются при работе с локальными и удаленными БД с помощью механизма BDE. Компоненты `Session` и `Database`, применяемые для управления соединениями с БД и транзакциями, будут изучены в *главах 7 и 13*.

Наборы данных

Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. В терминах системы Delphi *набор данных* представляет собой совокупность записей, взятых из одной или нескольких таблиц БД. Записи, входящие в набор данных, отбираются по определенным правилам, при этом в частных случаях набор данных может включать в себя все записи из связанной с ним таблицы или не содержать ни одной записи. Набор данных является *логической таблицей*, с которой можно работать при выполнении приложения. Взаимодействие таблицы и набора данных напоминает взаимодействие физического файла и файловой переменной.

Замечание

В отличие от Delphi, многие СУБД вместо термина *набор данных* используют термины *выборка* или *таблица*.

В Delphi для работы с наборами данных при использовании механизма BDE служат такие компоненты, как `Table`, `Query`, `UpdateSQL`, `DecisionQuery` или `StoredProc`. В случае других механизмов доступа для работы с наборами данных служат аналогичные компоненты `ADOTable`, `ADOQuery` и `ADOStoredProc` (для механизма ADO), `SQLTable`, `SQLQuery` и `SQLStoredProc` (для механизма dbExpress), `IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc` (для механизма InterBease). Отличительные особенности последних мы рассмотрим при описании соответствующих механизмов доступа. Здесь остановимся на изучении компонентов для механизма доступа BDE.

Компонент `StoredProc` используется для вызова хранимых процедур при организации взаимодействия с удаленными БД, а компонент `UpdateSQL` обеспечивает работу с кэшированными изменениями в записях. Эти компоненты рассматриваются при описании удаленных БД. Компонент `DecisionQuery` применяется при построении систем принятия решений. Наиболее универсальными и, соответственно, часто используемыми являются компоненты `Table` и `Query`, задающие наборы данных. Они будут подробно описаны немного ниже.

Базовые возможности доступа к БД обеспечивает класс `TDataSet`, представляющий наборы данных в виде совокупности строк и столбцов (записей и полей). Этот класс содержит основные средства навигации (перемещения) и редактирования наборов данных.

Компоненты `Table` и `Query` являются производными от класса `TDBDataSet` — потомка класса `TDataSet` (через класс `TBDEDataSet`). Они демонстрируют схожие с базовыми классами характеристики и поведение, но каждый из них имеет и свои особенности. Здесь мы рассмотрим наиболее общие характеристики наборов данных. Большая часть свойств, методов и событий изучается на примере операций с наборами данных.

Расположение БД, с таблицами которой выполняются операции, указывает свойство `DatabaseName` типа `String`. Значением свойства является имя каталога, в котором находится БД (файлы ее таблиц), или псевдоним, ссылающийся на этот каталог. Если для БД определен псевдоним, то его можно выбрать в раскрывающемся списке окна Инспектора объектов.

Замечание

Желательно задавать имя БД через псевдоним. Это заметно облегчает перенос приложения и файлов БД в другие каталоги и на другие компьютеры, т. к. для обеспечения работоспособности приложения после изменения расположения БД достаточно изменить название каталога, на который ссылается псевдоним БД.

Для компонента `Table` использование свойства `DatabaseName` является единственной возможностью задать местонахождение таблиц БД. Для компонента `Query` дополнительно можно указать в запросе SQL путь доступа к каждой таблице.

Замечание

При задании расположения БД программным способом набор данных предварительно необходимо закрыть, установив его свойство `Active` в значение `False`. В противном случае генерируется исключение.

Вот пример, иллюстрирующий, как задается расположение БД:

```
Table1.Active := False;
Table1.DatabaseName := 'BDPlace';
Table2.Active := False;
Table2.DatabaseName := 'C:\SALE\BD';
```

Для набора данных `Table1` таблицы БД расположены в каталоге, на который указывает псевдоним `BDPlace`. Таблицы БД для набора данных `Table2` расположены в каталоге C:\SALE\BD. Для определения и изменения псевдонима и его параметров удобно использовать такие программы, как Database Desktop или BDE Administrator (рассматриваются в *главе 12*).

В зависимости от ограничений и критерия фильтрации один и тот же набор данных в разные моменты времени может содержать различные записи. Число записей, составляющих набор данных, определяет свойство `RecordCount` типа `Longint`. Это свойство доступно для чтения при выполнении приложения. Управление числом записей в наборе данных осуществляется косвенно — путем отбора записей тем или иным способом, например, с помощью фильтрации или SQL-запроса (для компонента `Query`).

В приводимом примере производится перебор всех записей набора данных:

```
var i: integer;
...
Table1.First;
for i := 1 to Table1.RecordCount do begin
    // Здесь можно расположить инструкции, выполняющие
    // обработку очередной записи
    Table1.Next;
    end;
```

Перебор всех записей набора данных осуществляется в цикле, для чего переменная `i` цикла последовательно принимает значения от `1` до `RecordCount`. Перед началом цикла вызовом метода `First` выполняется переход к первой записи набора данных. В цикле для перехода к следующей записи вызывается метод `Next`.

Для локальных таблиц dBase или Paradox составляющие набор данных записи последовательно нумеруются, отсчет начинается с единицы. Номер записи в наборе данных определяет свойство `RecNo` типа `Longint`, которое доступно во время выполнения программы.

Номер текущей записи можно узнать, например, так:

```
Edit1.Text := IntToStr(Table1.RecNo);
```

Замечание

При изменении порядка записей при сортировке или фильтрации нумерация записей также изменяется.

Для таблиц Paradox свойство `RecNo` можно использовать для перехода к требуемой записи, установив в качестве значения свойства номер записи. Так, в операции

```
Table1.RecNo := StrToInt(Edit1.Text);
```

выполняется переход к записи, номер которой содержится в поле ввода `Edit1`. Указанная запись становится текущей.

Для выполнения операций с наборами данных используются два способа доступа к данным:

- навигационный;
- реляционный.

Навигационный способ доступа заключается в обработке каждой *отдельной записи* набора данных. Этот способ обычно используется в локальных БД или в удаленных БД небольшого размера. При навигационном способе доступа каждый набор данных имеет невидимый указатель текущей записи. Указатель определяет запись, с которой могут выполняться такие операции, как редактирование или удаление. Поля текущей записи доступны для просмотра. Например, компоненты `DBEdit` и `DBText` отображают содержимое соответствующих полей именно текущей записи. Компонент `DBGrid` указывает текущую запись с помощью специального маркера.

В разрабатываемом приложении навигационный способ доступа к данным можно реализовать, используя любой из компонентов `Table` или `Query`.

Реляционный способ доступа основан на обработке *группы записей*. Если требуется обработать одну запись, все равно обрабатывается группа, состоящая из одной записи. При реляционном способе доступа используются SQL-запросы, поэтому его называют также *SQL-ориентированным*. Реляционный способ доступа ориентирован на работу с удаленными БД и является для них предпочтительным. Однако его можно использовать и для локальных БД.

Реляционный способ доступа к данным в приложении можно реализовать с помощью компонента `Query`.

Состояния наборов данных

Наборы данных могут находиться в *открытом* или *закрытом* состояниях, на что указывает свойство `Active` типа `Boolean`. Если свойству `Active` установлено значение `True`, то набор данных открыт. Открытый компонент `Table` содержит набор данных, соответствующий данным таблицы, связанной с ним через свойство `TableName`. Для открытого компонента `Query` набор данных соответствует результатам выполнения SQL-запроса, содержащегося в свойстве `SQL` этого компонента. Если свойство `Active` имеет значение `False` (по умолчанию), то набор данных закрыт, и его связь с БД разорвана.

Набор данных может быть открыт на этапе разработки приложения. Если при этом к набору данных через источник данных `DataSource` подключены визуальные компоненты, например, `DBGrid` или `DBEdit`, то они отображают соответствующие данные таблицы БД.

Замечание

На этапе проектирования приложения визуальные компоненты отображают данные записей набора данных, но перемещение по набору данных и редактирование записей невозможны. Исключение составляет возможность перемещения текущего указателя с помощью полос прокрутки компонента `DBGrid`.

Если по каким-либо причинам открыть набор данных невозможно, то при попытке установить свойство Active в значение True выдается сообщение об ошибке, а свойство Active сохраняет значение False. Одной из причин невозможности открытия набора данных может являться неправильное значение свойства TableName или SQL.

Замечание

На этапе проектирования свойство Active наборов данных автоматически устанавливается в значение False при изменении значения свойств DataBaseName, TableName или SQL.

Приводимый пример демонстрирует управление состоянием набора данных с помощью свойства Active, которое используется для открытия и закрытия набора данных Query1:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Active := False;
Query1.SQL.Clear;
Query1.SQL.Add('select * from Example1.db');
Query1.Active := True;
end;
```

Управлять состоянием набора данных можно также с помощью методов Open и Close.

Процедура Open открывает набор данных, ее вызов эквивалентен установке свойства Active в значение True. При вызове метода Open генерируются события BeforeOpen и AfterOpen, а также вызываются процедуры-обработчики этих событий.

Процедура Close закрывает набор данных, ее вызов эквивалентен установке свойства Active в значение False. При вызове метода Close генерируются события BeforeClose и AfterClose, а также вызываются процедуры-обработчики этих событий.

В примере показывается управление состоянием набора данных (открытие и закрытие) с помощью методов Open и Close:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('select * from Example2.db');
Query1.Open;
end;
```

События `BeforeOpen` и `AfterOpen` имеют тип `TDataSetNotifyEvent`, который описывается так:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
```

В этом описании параметр `DataSet` определяет набор данных, для которого произошло событие.

Событие `BeforeOpen` возникает непосредственно перед открытием набора данных. В обработчике этого события можно выполнить проверку определенных условий, и если они не соблюдаются, то открытие набора данных может быть запрещено.

Замечание

Обработчик события не содержит специального параметра, с помощью которого можно запретить открытие набора данных. Одним из вариантов запрета открытия является принудительное возбуждение исключения.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.Table1BeforeOpen(DataSet: TDataSet);
begin
if not CheckBox1.Checked then Abort;
end;
```

Если флажок `CheckBox1`, управляющий возможностью открытия набора данных, не установлен, то открытие набора данных `Table1` блокируется. Для этого с помощью вызова процедуры `Abort` генерируется "тихое" исключение. В результате операции, связанные с открытием набора данных, отменяются, а пользователю не выдается никаких сообщений об ошибках. В подобных случаях выдачу предупреждающих сообщений должен обеспечивать программист, как это реализуется, например, в приведенной ниже процедуре:

```
procedure TForm1.Table1BeforeOpen(DataSet: TDataSet);
begin
if not CheckBox1.Checked then begin
 MessageDlg('Данные таблицы ' + Table1.TableName + ' недоступны!',
       mtError, [mbOK], 0);
 Abort;
 end;
end;
```

Событие `AfterOpen` генерируется сразу после открытия набора данных. Это событие можно использовать, например, для выдачи пользователю сообщения о возможности работы с данными.

В примере демонстрируется открытие набора данных с выдачей соответствующего сообщения:

```
procedure TForm1.Table1AfterOpen(DataSet: TDataSet);
begin
if CheckBox2.Checked then
   MessageDlg('Данные таблицы ' + Table1.TableName + '
      доступны для работы.',
               mtWarning, [mbOK], 0);
end;
```

Если флажок `CheckBox2`, управляющий возможностью выдачи сообщений, установлен, то при открытии набора данных `Table1` пользователю выдается сообщение.

Как уже говорилось, при закрытии набора данных возникают события `BeforeClose` и `AfterClose`. Они, как и события `BeforeOpen` и `AfterOpen`, имеют тип `TDataSetNotifyEvent`.

Отметим, что закрытие набора данных автоматически не сохраняет текущую запись, т. е. если набор данных при закрытии находился в режимах редактирования или вставки, то произведенные изменения данных в текущей записи будут потеряны. Поэтому перед закрытием набора данных нужно проверять его режим и при необходимости принудительно вызывать метод `Post`, сохраняющий сделанные изменения. Одним из вариантов сохранения изменений является вызов метода `Post` в обработчике события `BeforeClose`, возникающего непосредственно перед закрытием набора данных.

Рассмотрим следующий пример:

```
procedure TForm1.Table1BeforeClose(DataSet: TDataSet);
begin
if (Table1.State = dsEdit) or (Table1.State = dsInsert) then Table1.Post;
end;
```

Если набор данных `Table1` находится в режиме редактирования или вставки, то перед его закрытием внесенные изменения сохраняются.

Замечание

При закрытии приложения событие `BeforeClose` не генерируется, и несохраненные изменения теряются.

Событие `AfterClose` возникает сразу после закрытия набора данных, и его можно использовать для выдачи пользователю соответствующих сообщений, как это сделано в приведенной ниже процедуре:

```
procedure TForm1.Table1AfterClose(DataSet: TDataSet);
begin
if CheckBox2.Checked then begin
```

```
 Beep;
 MessageDlg('Таблица ' + Table1.TableName + ' закрыта.',
      mtWarning, [mbOK], 0);
 end;
end;
```

Если установлен флажок `CheckBox2`, управляющий режимом выдачи сообщений, то после закрытия набора данных `Table1` выдается сообщение о закрытии таблицы, связанной с этим набором данных.

Необходимо иметь в виду, что если при работе приложения используется большое число таблиц, то выдача подобных сообщений может затруднять действия пользователя. Поэтому программист должен предусмотреть возможность отключения выдачи сообщений. В приведенных примерах для этой цели предназначен флажок `CheckBox2`.

Режимы наборов данных

Наборы данных могут находиться в различных режимах. Текущий режим набора данных определяется свойством `State` типа `TDataSetState`. Оно доступно для чтения во время выполнения приложения и может быть использовано только для текущего режима. Для перевода набора данных в требуемый режим используются специальные методы. Они могут вызываться явно (указанием имени метода) или косвенно (путем управления соответствующими визуальными компонентами, например, навигатором `DBNavigator` или сеткой `DBGrid`).

Набор данных может находиться в одном из перечисленных ниже режимов.

- `dsInactive` (неактивен) — набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после своего закрытия, когда свойство `Active` установлено в значение `False`.
- `dsBrowse` (навигация по записям набора данных и просмотр данных) — в этот режим набор данных переходит так:
  - из режима `dsInactive` — при установке свойства `Active` в значение `True`;
  - из режима `dsEdit` — при вызове метода `Post` или `Cancel`;
  - из режима `dsInsert` — при вызове метода `Post` или `Cancel`.
- `dsEdit` (редактирование текущей записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове метода `Edit`.
- `dsInsert` (вставка новой записи) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов `Insert`, `InsertRecord`, `Append` или `AppendRecord`.
- `dsSetKey` (поиск записи, удовлетворяющей заданному критерию) — в этот режим набор данных переходит из режима `dsBrowse` при вызове методов

SetKey, SetRangeXXX, FindKey, GotoKey, FindNearest или GotoNearest. Возможен только для компонента TTable, т. к. для компонента Query отбор записей осуществляется средствами языка SQL.

- dsCalcFields (расчет вычисляемых полей) — используется обработчик события OnCalcFields.
- dsFilter (фильтрация записей) — в этот режим набор данных автоматически переходит из режима dsBrowse каждый раз, когда выполняется обработчик события OnFilterRecord. В режиме блокируются все попытки изменения записей. После завершения работы обработчика события OnFilterRecord набор данных автоматически переводится в режим dsBrowse.
- dsNewValue (обращение к значению свойства TField.NewValue).
- dsOldValue (обращение к значению свойства TField.OldValue).
- dsCurValue (обращение к значению свойства TField.CurValue).
- dsBlockRead (запрет изменения элементов управления и генерации событий при вызове метода Next).
- dsInternalCalc (указание на необходимость вычислять значения полей, свойство TField.FieldKind которых имеет значение fkInternalCalc).
- dsOpening (открытие набора данных).

Взаимосвязи между основными режимами наборов данных показаны на рис. 5.1, где приведены также некоторые методы и свойства, с помощью которых набор данных переходит из одного режима в другой.

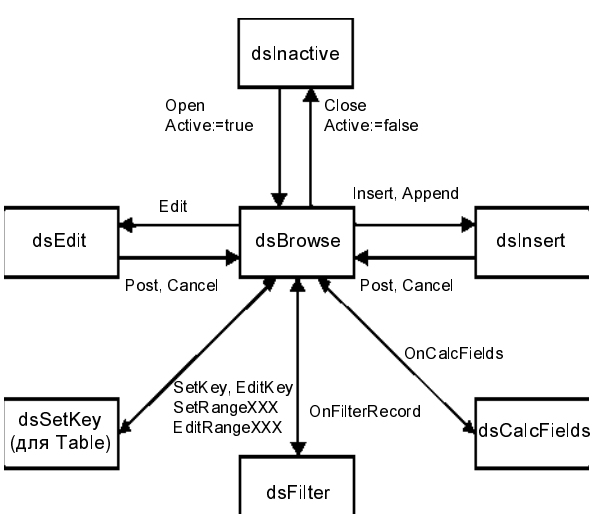


Рис. 5.1. Взаимосвязи режимов наборов данных

Иногда при описании операций, выполняемых с записями набора данных, под режимом *редактирования* подразумевается не только режим `dsEdit` изменения полей текущей записи, но и режим `dsInsert` вставки новой записи. Тем самым режим редактирования понимается в широком смысле слова как режим *модификации* набора данных.

Набор данных использует режимы `dsNewValue`, `dsOldValue`, `dsCurValue`, `dsBlockRead` и `dsInternalCalc` для собственных нужд, обычно программист не анализирует их.

При выполнении программы можно определить режим набора данных с помощью одноименных свойств `State` типа `TDataSetState` самого набора данных и связанного с ним источника данных `DataSource`. При изменении режима набора данных для источника данных `DataSource` генерируется событие `OnStateChange` типа `TNotifyEvent`.

Рассмотрим пример:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
begin
case DataSource1.State of
dsInactive: Label1.Caption := 'Набор данных закрыт';
dsBrowse:   Label1.Caption := 'Просмотр набора данных';
dsEdit:     Label1.Caption := 'Редактирование набора данных';
dsInsert:   Label1.Caption := 'Вставка записи в набор данных'
else        Label1.Caption := 'Режим набора данных не определен';
end;
end;
```

В приведенной процедуре определяется режим набора данных, связанного с источником данных `DataSource1`, и информация об этом режиме выводится в надписи `Label1`. При этом используется свойство `State` источника данных. Код, выполняющий анализ режима, помещен в обработчик события `OnStateChange` компонента `DataSource1`.

Доступ к полям

Каждое поле набора данных представляет собой отдельный столбец, для работы с которым в Delphi служат объект `Field` типа `TField` и объекты производных от него типов, например, `TIntegerField`, `TFloatField` или `TStringField`. Для доступа к этим объектам и, соответственно, к полям записей у набора данных есть специальные методы и свойства, доступные при выполнении приложения.

Свойство `FieldCount` типа `Integer` указывает количество полей набора данных. Это свойство доступно только для чтения. Количество полей набора данных может отличаться от физического числа полей таблицы БД, поскольку в набор данных не обязательно включаются все поля таблицы. Состав полей формируется при разработке приложения с помощью Редактора полей набора данных и Редактора столбцов сетки `DBGrid`. Кроме того, возможно динамическое

изменение состава полей во время выполнения приложения. Для компонента `Query` состав полей набора данных зависит также от SQL-запроса.

Значение свойства `Fields [Index: Integer]` типа `TField` представляет собой поле (столбец) набора данных. К отдельному полю можно обратиться, указав его номер `Index` в массиве `Fields`; номера полей находятся в пределах от нуля до `FieldCount - 1`. Номер объекта поля в массиве полей определяет свойство `Index` типа `Integer`. В отличие от Редакторов полей и столбцов, применяемых на этапе разработки приложения, свойство `Index` можно использовать для определения и изменения порядка полей набора данных во время выполнения приложения.

В качестве примера рассмотрим чтение полей текущей записи:

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
for n := 0 to Table1.FieldCount - 1 do
    ListBox1.Items.Add(Table1.Fields[n].AsString);
end;
```

Здесь содержимое каждого поля текущей записи интерпретируется как строковое значение и добавляется к списку `ListBox1`.

Номер поля в наборе данных не является заранее фиксированным и известным числом и зависит от порядка полей в таблице БД, от текущего состава полей набора данных, а также от значений свойств некоторых визуальных компонентов, связанных с этим набором, например, сетки `DBGrid`. Так, при изменении порядка расположения столбцов в компоненте `DBGrid` соответственно изменяется порядок следования полей набора данных. В связи с этим обращение к какому-либо полю по его номеру в массиве полей может вызвать обращение совсем к другому полю. Поэтому для доступа к полям чаще используются методы `FindField` и `FieldByName`.

Функция `FindField (const FieldName: String): TField` возвращает для набора данных поле, имя которого указывает параметр `FieldName`. В отличие от номера в массиве полей, имя поля более статично и изменяется реже. Кроме того, имя поля несет бóльшую смысловую нагрузку, чем номер. Если заданное параметром `FieldName` поле не найдено, то метод `FindField` возвращает значение `Nil`. На практике чаще используется метод `FieldByName`, который отличается от метода `FindField` тем, что если заданное поле не найдено, то генерируется исключение.

Замечание

Имя поля, определяемое параметром `FieldName`, является именем физического поля таблицы БД, заданным при создании таблицы, а не именем (свойством `Name`) объекта `Field`, которое создано для этого поля.

Для набора данных `Query` имя `FieldName` физического поля можно переопределить в тексте SQL-запроса.

Свойство `Fields` и методы `FindField` и `FieldByName` наиболее часто используются для доступа к значению поля текущей записи совместно с такими свойствами объекта `Field`, как `AsString`, `AsInteger`, `AsFloat` или `AsBoolean`, которые позволяют обращаться к значению поля как к строковому, целочисленному, вещественному или логическому значению соответственно.

Так, в коде

```
Var x: integer;
...
Label1.Caption := Table1.FieldByName('Name').AsString;
x := Table1.FieldByName('Number').AsInteger;
```

строковое значение поля `Name` отображается в надписи `Label1`, а переменной `x` присваивается целочисленное значение поля `Number`. Если же поле `Number` содержит значение, которое нельзя интерпретировать как целое число, то генерируется исключение.

В рассмотренных примерах выполнялось *чтение* содержимого полей текущей записи. Аналогичным образом можно присваивать произвольному полю текущей записи *новое значение*, при этом набор данных должен находиться в режиме редактирования или вставки.

Например, с помощью следующих инструкций:

```
Var x: integer;
. . .
// Перевод набора данных в режим редактирования
Table1.Edit;
// Изменение значений полей
Table1.FieldByName('Name').AsString := Edit1.Text;
Table1.FieldByName('Number').AsInteger := x;
// Сохранение изменений
Table1.Post;
```

выполняется присвоение новых значений полям `Name` и `Number` текущей записи после того, как набор данных `Table1` переведен в режим редактирования. Произведенные изменения сохраняются, а набор данных переводится в режим просмотра.

Особенности набора данных *Table*

Компонент `Table` представляет собой набор данных, который в текущий момент времени может быть связан только с *одной* таблицей БД. Этот набор данных формируется на базе навигационного способа доступа к данным, поэтому

компонент `Table` рекомендуется использовать для локальных БД, таких как dBase или Paradox. При работе с удаленными БД следует использовать компонент `Query`. В дальнейшем при рассмотрении вопросов, связанных с локальными БД, мы обычно будем работать с компонентом `Table`.

Связь между таблицей и компонентом `Table` устанавливается через его свойство `TableName` типа `TFileName`, которое задает имя таблицы (и имя файла с данными таблицы). При задании свойства `TableName` указываются имя файла и расширение имени файла.

На этапе разработки приложения имена всех таблиц доступны в раскрывающемся списке Инспектора объектов. В этот список попадают таблицы, файлы которых расположены в каталоге, указанном свойством `DatabaseName`.

Замечание

При смене имени таблицы на этапе проектирования приложения свойство `Active` набора данных автоматически устанавливается в значение `False`. При задании имени таблицы программным способом набор данных предварительно необходимо закрыть, установив его свойство `Active` в значение `False`. В противном случае генерируется исключение.

Приведем пример, иллюстрирующий, как задается имя таблицы БД:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if OpenDialog1.Execute then begin
 Table1.Active := False;
 Table1.TableName := OpenDialog1.FileName;
 Table1.Active := True;
 end;
end;
```

Здесь нажатие кнопки `Button1` приводит к появлению диалогового окна выбора имени файла. При выборе файла таблицы его имя устанавливается в качестве значения свойства `TableName`. Набор данных `Table1` предварительно закрывается и снова открывается уже после смены таблицы. Тип таблицы определяется автоматически по расширению имени файла. При наличии ошибок, например, связанных с нарушением структуры таблицы, выдается соответствующее сообщение, а набор данных остается закрытым.

Свойство `TableType` типа `TTtableType` определяет тип таблицы. Для локальных таблиц это свойство может принимать следующие значения:

- `ttDefault` — тип таблицы автоматически определяется по расширению файла;
- `ttParadox` — таблица Paradox;
- `ttDBase` — таблица dBASE;

- `ttFoxPro` — таблица FoxPro;
- `ttASCII` — текстовый файл, содержащий данные в табличном виде (таблица ASCII).

Если свойство `TableType` имеет значение `ttDefault` (по умолчанию), то тип таблицы определяется по расширению файла:

- db или отсутствует — таблица Paradox;
- dbf — таблица dBASE;
- txt — текстовый файл (таблица ASCII).

По умолчанию в состав набора данных `Table` попадают все записи связанной с ним таблицы. Для отбора записей, удовлетворяющих определенным условиям, используются фильтры.

Delphi через BDE автоматически поддерживает многопользовательский доступ к локальным таблицам. При этом по умолчанию все пользовательские приложения имеют равные права и могут редактировать содержащиеся в таблицах данные. Чтобы запретить пользователям изменять содержание записей, можно использовать свойство `ReadOnly` типа `Boolean`. По умолчанию оно имеет значение `False`, что предоставляет пользователю право изменения записей.

Если приложению требуется получить к таблице *монопольный доступ*, то это можно осуществить через свойство `Exclusive` типа `Boolean`. По умолчанию свойство имеет значение `False`, и для таблицы, с которой связан набор данных, действует многопользовательский режим доступа. Если установить свойство `Exclusive` в значение `True`, то приложение получает монопольный доступ к соответствующей таблице, при этом другим приложениям доступ к таблице запрещается. Перед заданием значения свойства `Exclusive` набор данных должен быть закрыт, т. е. разорвана его связь с таблицей. Если какое-либо приложение или набор данных этого же приложения уже взаимодействуют с таблицей, то попытка установить режим монопольного доступа к ней вызовет исключение.

Монопольный режим доступа необходим при выполнении таких операций, как добавление или удаление индекса методами `AddIndex` и `DeleteIndex` или очистка таблицы методом `EmptyTable`.

Замечание

Такие приложения, как Delphi и Database Desktop, также могут осуществлять доступ к таблицам. Поэтому перед отладкой приложения, которое устанавливает монопольный доступ к таблице, необходимо проверить, не работают ли с этой таблицей Delphi и/или Database Desktop. В приложении Database Desktop таблицу нужно закрыть, а в среде Delphi достаточно через Инспектор объектов установить в значение `False` свойство `Active` набора данных, связанного с таблицей.

В наборе данных `Table` можно указать текущий индекс для:

- сортировки записей;
- поиска записей;
- установления связей между таблицами.

Текущий индекс устанавливается с помощью свойства `IndexName` или `IndexFieldNames` типа `String`. На этапе разработки приложения текущий индекс выбирается в списке индексов, заданных при создании таблицы. Все возможные значения свойств `IndexName` и `IndexFieldNames` содержатся в раскрывающихся списках, доступных через Инспектор объектов. Оба свойства во многом схожи, и их использование практически одинаково. Значением свойства `IndexName` является *имя индекса*, заданное при создании таблицы, а значением свойства `IndexFieldNames` является *имя поля*, для которого был создан индекс. Если индекс состоит из нескольких полей, то для свойства `IndexName` по-прежнему задается имя этого индекса, а для свойства `IndexFieldNames` через точку с запятой перечисляются имена полей, входящие в этот индекс.

Вот как текущий индекс задается в программе:

```
Table1.IndexName := 'indName';
Table2.IndexFieldNames := 'Name';
```

Здесь компоненты `Table1` и `Table2` связаны с одной таблицей, для поля `Name` которой определен индекс `indName`. Этот индекс устанавливается в качестве текущего для обоих наборов данных.

Для таблиц Paradox сделать текущим индексом ключ (главный индекс) можно только с помощью свойства `IndexFieldNames`, перечислив ключевые поля таблицы, т. к. ключ не имеет имени и поэтому недоступен через свойство `IndexName`.

Задать ключ в качестве текущего индекса можно так:

```
Table1.IndexFieldNames := 'Name;Post;BirthDay';
```

Здесь для таблицы Paradox, с которой связан компонент `Table1`, определен ключ, в который входят поля `Name`, `Post` и `BirthDay`. Этот ключ устанавливается в качестве текущего индекса таблицы.

Замечание

Свойства `IndexName` и `IndexFieldNames` взаимозависимы. При установке значения одного из них другое автоматически очищается.

Индекс, устанавливаемый текущим, должен существовать. Если индекс, задаваемый как значение свойства `IndexName` или `IndexFieldNames`, для таблицы не существует, то возникает исключение.

При смене таблицы, с которой ассоциирован компонент `Table`, значения свойств `IndexName` и `IndexFieldNames` не изменяются автоматически, поэтому программист должен установить нужные значения самостоятельно.

Получить доступ к полям в составе текущего индекса можно с помощью свойств `IndexFieldCount` и `IndexFields`.

Свойство `IndexFieldCount` типа `Integer` содержит число полей в текущем индексе и доступно для чтения при выполнении приложения.

Свойство `IndexFields [Index: Integer]` типа `TField` позволяет обращаться к полям текущего индекса, при этом переменная `Index` задает номер индекса в

массиве полей этого индекса (отсчет начинается с нуля). Класс `TField` представляет собой поле набора данных и имеет большое число свойств и методов.

Чаще всего индексы определяются при создании таблицы и в дальнейшем при работе с таблицей не изменяются. Однако у программиста есть возможность изменять определенные для таблицы индексы динамически, т. е. в процессе выполнения приложения, с помощью методов `AddIndex` и `DeleteIndex`.

Замечание

Допускается изменять индексы для таблицы, открытой в режиме монопольного доступа, когда свойство `Exclusive` имеет значение `True`.

Процедура `AddIndex (const Name, Fields: String; Options: TIndexOptions)` добавляет к таблице индекс, имя которого задано параметром `Name`. Входящие в состав индекса поля указываются в параметре `Fields`; если индекс состоит из нескольких полей, то они разделяются точкой с запятой. Указывать можно только поля, которые входят в структуру таблицы, в противном случае генерируется исключение, а индекс не создается. Параметр `Options` содержит параметры индекса. Он имеет множественный тип и может принимать комбинации следующих значений:

- `ixPrimary` (первичный индекс);
- `ixUnique` (уникальный индекс) — для этого индекса не допускается повторение значений полей в его составе;
- `ixDescending` (сортировка в порядке убывания значений) — по умолчанию строится индекс, определяющий сортировку по возрастанию;
- `ixExpression` (индекс создается на основе выражения) — только для таблиц dBase;
- `ixCaseInsensitive` (сортировка не зависит от регистра букв);
- `IxNonMaintained` (индекс автоматически не изменяется, если таблица открыта).

Приведем пример, демонстрирующий добавление индекса к таблице:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
// Перевод таблицы в режим монопольного доступа
Table1.Close;
Table1.Exclusive := True;
Table1.Open;
// Добавление индекса
Table1.AddIndex('indPosition', 'Position',
   [ixDescending, ixCaseInsensitive]);
// Закрытие режима монопольного доступа к таблице
Table1.Close;
```

```
Table1.Exclusive := False;
Table1.Open;
end;
```

Здесь к таблице, с которой связан набор данных `Table1`, добавляется индекс `indPosition`, построенный по полю `Position`. Для нового индекса определяется порядок сортировки по убыванию значений, не зависящий от регистра букв.

Процедура `DeleteIndex (const Name: String)` удаляет из таблицы индекс, имя которого задано параметром `Name`. При попытке удаления несуществующего индекса генерируется исключение.

В программе удаление индекса из таблицы можно реализовать так:

```
Table1.DeleteIndex('indPosition');
```

Перед удалением индекса таблица должна быть переведена в режим монопольного доступа аналогично тому, как это происходило при добавлении индекса.

Для доступа к информации обо *всех* индексах, определенных для таблицы и, соответственно, для связанного с ней набора данных, можно использовать свойство `IndexDefs` типа `TIndexDefs`. Это свойство доступно как на этапе разработки, так и на этапе выполнения приложения. Класс `TIndexDefs` в свою очередь также имеет полезные свойства и методы, которые мы сейчас и рассмотрим.

Перед работой с индексами всегда рекомендуется вызывать метод `UpDate`, который производит обновление информации об индексах так, чтобы значение свойства `IndexDefs` отражало реальное состояние с учетом сделанных изменений.

Свойство `Count` типа `Integer` содержит количество индексов и доступно для чтения. Управление количеством индексов осуществляется косвенно, т. е. через другие свойства и методы.

Свойство `Items [Index: Integer]` типа `TIndexDef` представляет собой список, содержащий информацию обо всех индексах таблицы. Переменная `Index` указывает номер индекса в списке, отсчет начинается с нуля. Тип `TIndexDef` (не путать с классом `TIndexDefs`) является классом и, в свою очередь, также имеет свойства, главные из которых — `Name`, `Fields` и `Options`.

Замечание

При доступе к информации об индексах можно не указывать свойство `Items`. Например, чтобы считать в редактор `Edit1` список полей, составляющих второй индекс, можно использовать следующие эквивалентные по действию инструкции:

```
Edit1.Text := Table1.IndexDefs[1].Fields;
```

или

```
Edit1.Text := Table1.IndexDefs.Items[1].Fields;
```

Свойство `Name` типа `String` содержит имя индекса. Для индексов, построенных на основе выражений (таблицы dBase), и для главного индекса таблиц Paradox вместо имени возвращается пустая строка.

Свойство `Fields` типа `String` содержит имена полей, по которым построен индекс. Если в индексе используется несколько полей, то их имена разделяются точкой с запятой.

Свойство `Options` типа `TIndexOption` содержит параметры индекса, заданные при его создании.

Рассмотрим работу со списком индексов и индексных полей на примере.

В форме расположены следующие компоненты: сетка `DBGrid`, в которой отображаются записи таблицы БД, два списка `ListBox` с соответствующими надписями, а также две кнопки `Button`. При нажатии кнопки **Индексы** (`Button1`) в список `ListBox1` загружается список индексов, определенных для набора данных `Table1`, а в список `ListBox2` — список полей, образующих эти индексы (рис. 5.2). Так как компонент `Table1` связан с таблицей Paradox, то имя главного индекса, построенного по полю `P_Code`, содержит пустую строку.

Обработчик события нажатия кнопки `Button1` выглядит так:

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
for n := 0 to Table1.IndexDefs.Count - 1 do begin
    ListBox1.Items.Add(Table1.IndexDefs[n].Name);
    ListBox2.Items.Add(Table1.IndexDefs[n].Fields);
    end;
end;
```

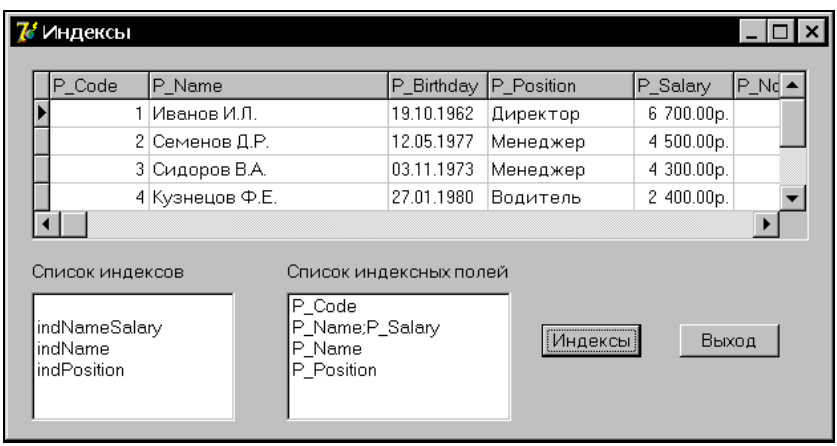


Рис. 5.2. Получение списка индексов и полей

Для получения списка имен индексов можно использовать метод `GetItemNames(List: TStrings)`. Он возвращает список имен индексов через

параметр `List`, в качестве которого можно использовать любой объект типа `TStrings`, например, `ListBoxItems`.

Получение списка имен индексов в программе может выглядеть так:

```
ListBox1.Items.Clear;
Table1.GetItemNames(ListBox1.Items);
```

Для добавления и удаления индексов используются методы `Add` и `Clear`. Процедура `Add (const Name, Fields: String; Options: TIndexOptions)` добавляет новый индекс. Параметры этой процедуры не отличаются от параметров метода `AddIndex` набора данных `Table`, однако для метода `Add` не требуется перевод таблицы в режим монопольного доступа, что делает его использование более удобным.

Замечание

Для таблиц Paradox при определении ключа (главного индекса) имя индекса не задается, т. к. этот индекс не именуется.

Процедура `Clear` удаляет все индексы, содержащиеся в списке индексов.

Вот как удаление и добавление индексов к таблице реализуется в программе:

```
Table1.IndexDefs.Clear;
Table1.IndexDefs.Add('', 'Number', [ixPrimary, ixUnique]);
Table1.IndexDefs.Add('indMain', 'Code;Position',
   [ixDescending, ixCaseInsensitive]);
```

В этом примере из набора данных `Table1`, связанного с таблицей Paradox, удаляются все индексы и создаются два новых индекса: первичный индекс, построенный по полю `Number`, и вторичный индекс `indMain`, построенный по полям `Code` и `Position`.

Ряд методов предназначен для поиска индексов в списке, например, `IndexOf`, `Find` и `FindIndexForFields`. Их удобно использовать в случае, когда индексы или их поля вводит пользователь на этапе выполнения приложения. Перед использованием таких индексов целесообразно проверять, существуют ли они в списке индексов, т. к. в противном случае мы рискуем получить исключение.

Функция `IndexOf (const AName: String): Integer` осуществляет поиск индекса по имени, заданному параметром `AName`. В качестве результата возвращается номер индекса в свойстве `Items`. В случае неудачного поиска возвращается значение `-1`.

Рассмотрим пример поиска индекса:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
if Table1.IndexDefs.IndexOf(Edit1.Text) = -1 then begin
   MessageDlg('Индекс отсутствует!', mtError, [mbOK], 0);
```

```
    if Edit1.CanFocus then Edit1.SetFocus;
    end;
end;
```

Здесь при нажатии кнопки `Button2` выполняется проверка существования индекса, имя которого введено в поле ввода `Edit1`. При отсутствии в наборе данных `Table1` указанного индекса выдается сообщение об ошибке, и фокус ввода устанавливается на `Edit1`.

Функция `FindIndexForFields(const Fields: String): TIndexDef` осуществляет поиск индекса по списку полей, заданному параметром `Fields`. В случае успешного поиска функция возвращает информацию об индексе, если же индекс отсутствует, то генерируется исключение.

Замечание

Если не найден индекс, поля которого полностью совпадают с заданными, но есть индексы, которые включают в себя все заданные поля, то в качестве результата возвращается первый такой индекс.

При запуске приложения информация об индексах таблицы считывается с диска из соответствующих индексных файлов. При динамическом, т. е. в процессе выполнения приложения, изменении индексов содержимое индексных файлов остается прежним. При необходимости программист должен предусмотреть сохранение новой информации на диске для ее последующего применения. Для этого можно использовать свойство `StoreDefs` типа `Boolean`, которое указывает, нужно ли сохранять информацию об индексах на диске. При любых изменениях в индексах это свойство нужно установить в значение `True`, тогда информация об индексах будет автоматически сохранена на диске. Если изменения в индексах сохранять не нужно, то достаточно установить свойство `StoreDefs` в значение `False`, и изменения в индексах будут действовать только при выполнении приложения.

Особенности набора данных *Query*

Компонент `Query` представляет собой набор данных, записи которого формируются в результате выполнения SQL-запроса и основаны на реляционном способе доступа к данным. При работе с удаленными БД рекомендуется использовать именно набор данных `Query`.

Замечание

При работе с удаленными БД следует обращаться к средствам языка SQL. Это относится и к таким операциям, как перемещение по набору данных или вставка в него записей. Если же для компонента `Query` используются методы типа `Next` или `Insert`, то вместо реляционного способа доступа к удаленным данным будет применен навигационный. В результате набор данных `Query` будет мало отличаться от набора данных `Table`.

В отличие от компонента `Table`, набор данных `Query` может включать в себя записи более чем одной таблицы БД.

Текст запроса, на основании которого в набор данных отбираются записи, содержится в свойстве `SQL` типа `TStrings`. Запрос включает в себя команды на языке SQL и выполняется при открытии набора данных. Запрос SQL иногда называют SQL-программой.

При формировании запроса на этапе разработки приложения можно использовать текстовый редактор (рис. 5.3), вызываемый через Инспектор объектов двойным щелчком в области значения свойства `SQL`.

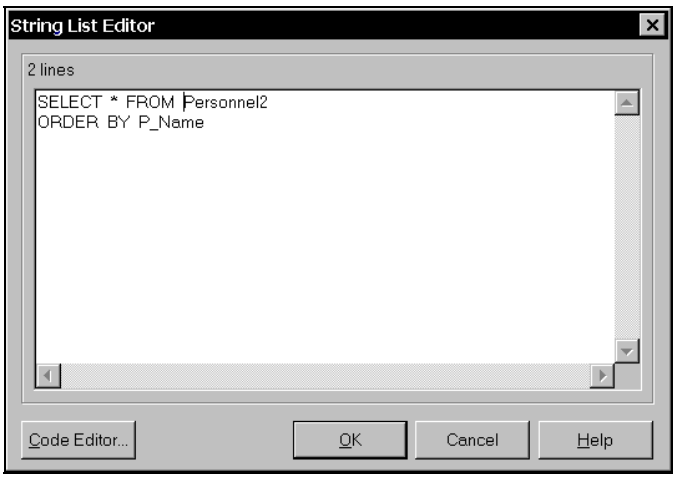


Рис. 5.3. Редактирование запроса SQL

SQL-запрос также можно формировать и изменять динамически, внося изменения в его текст (значение свойства `SQL` компонента `Query`) непосредственно при выполнении приложения.

Замечание

В процессе формирования SQL-запроса проверка его правильности не производится, и если в запросе имеются ошибки, то они выявляются только при открытии набора данных. Одним из вариантов предотвращения ошибок в SQL-запросе является его предварительная отладка, например, с помощью программы Database Desktop.

Рассмотрим пример приложения — простейшего редактора, позволяющего подготавливать и выполнять SQL-запросы. На рис. 5.4 показана форма приложения при его выполнении. Кроме визуальных компонентов, форма содержит два компонента доступа к данным `Query1` и `DataSource1`, которые при выполнении приложения не видны.

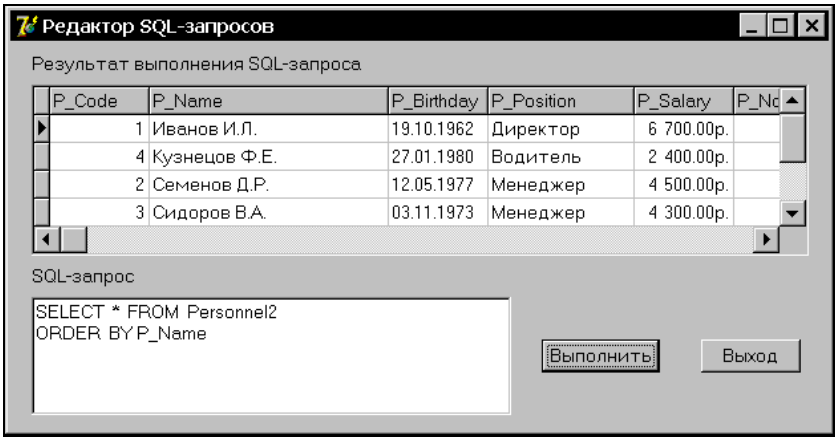


Рис. 5.4. Приложение-редактор SQL-запросов

Редактирование SQL-запроса осуществляется с помощью компонента `Memo1`. Набранный запрос выполняется при нажатии кнопки **Выполнить** (`Button1`), а результат выполнения отображается в компоненте `DBGrid1`.

При наличии в тексте SQL-запроса ошибки генерируется исключение и выдается сообщение об ошибке (рис. 5.5), а результат запроса оказывается не определен. При этом набор данных `Query1` автоматически закрывается.

Значения свойств `DataSet` источников данных `DataSource1` и `DataSource` компонента `DBGrid1`, с помощью которых организуется взаимодействие компонентов `Query1`, `DataSource1` и `DBGrid1`, устанавливаются при создании формы. В последующих примерах приложений значения этих свойств задаются через Инспектор объектов, поэтому инструкции, присваивающие свойствам необходимые значения, в модуле формы отсутствуют.

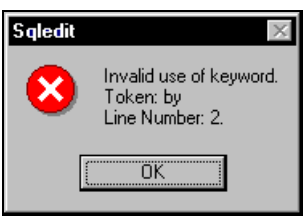


Рис. 5.5. Сообщение об ошибке в тексте SQL-запроса

Приведем код модуля `uSQLEdit` формы `Form1` приложения:

```
unit uSQLEdit;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
     Forms, Dialogs, StdCtrls, Grids, DBGrids, Db, DBTables;

type
  TForm1 = class(TForm)
          Memo1: TMemo;
    DataSource1: TDataSource;
         Query1: TQuery;
        DBGrid1: TDBGrid;
        Button1: TButton;
         Label1: TLabel;
         Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
     end;

var
     Form1: TForm1;

   implementation

   {$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
DataSource1.DataSet := Query1;
DBGrid1.DataSource := DataSource1;
   end;

procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Close;
Query1.SQL.Assign(Memo1.Lines);
Query1.Open;
   end;

end.
```

Метод `Assign` выполняет присваивание одного объекта другому, при этом объекты должны иметь совместимые типы. Применительно к списку строк (класс `TStrings`), которому принадлежат свойства `SQL` компонента `Query1` и `Lines`

компонента Memo1, подобное присваивание означает копирование информации из одного списка в другой с заменой содержимого последнего. Если размеры списков (число элементов) не совпадают, то после замены число элементов заменяемого списка становится равным числу элементов копируемого списка.

Компонент Query обеспечивает выполнение SQL-запроса и является набором данных, который формируется на основе этого запроса. Формирование набора данных выполняется при активизации компонента Query вызовом метода Open или установкой свойства Active в значение True. В ряде случаев при выполнении SQL-запроса не требуется возвращать набор данных, например, при удалении, вставке или изменении записей. В этом случае предпочтительнее выполнять запрос компонента Query не его открытием, а вызовом метода ExecSQL. При работе в сети вызов метода ExecSQL выполняет требуемую модификацию набора данных, не передавая в вызывающее приложение (компьютер) записи набора данных, что заметно снижает нагрузку на сеть.

Компонент Query может быть связан с таблицей БД или напрямую, или содержать *копии* отобранных записей таблицы, доступные для чтения. Вид взаимодействия определяется свойством RequestLive типа Boolean. По умолчанию свойство имеет значение False, и набор данных Query доступен только для чтения. Если пользователю или программисту требуется возможность редактирования записей, то свойство RequestLive нужно установить в значение True. В этом случае набор данных Query напрямую связывается с соответствующей таблицей, аналогично набору данных Table.

Замечание

Влияние свойства RequestLive зависит от текста выполняемого SQL-запроса. Если в результате выполнения запроса не может быть получен редактируемый набор данных, то установка свойства RequestLive в значение True игнорируется.

Чтобы проверить результат установки значения свойства RequestLive, можно воспользоваться свойством CanModify типа Boolean. Если оно имеет значение True, то набор данных является редактируемым, если False — то нередактируемым.

Чтобы получить в результате выполнения SQL-запроса редактируемый набор данных, кроме установки свойства RequestLive в значение True, должны быть выполнены следующие условия:

- данные отбираются только из одной таблицы или просмотра (view);
- таблица или просмотр допускают модификацию;
- в запросе не используется операнд DISTINCT и агрегатные (статистические) функции;
- в запросе не применяется соединение таблиц;
- в запросе отсутствуют подзапросы и вложенные запросы;
- не используется группирование данных;
- сортировка применяется только к индексированным полям.

Для локальных БД вместо компонента `Table` можно также использовать компонент `Query`. Если установить свойство `SQL` в значение `SELECT * FROM NameTableBD`, а свойство `RequestLive` — в значение `True`, то набор данных `Query` будет аналогичен набору данных `Table`. (Здесь `NameTableBD` является именем таблицы БД, которое для компонента `Table` задается в свойстве `TableName`.) Однако набор данных `Query`, в отличие от `Table`, не имеет системы индексов, поэтому к `Query` неприменимы методы, использующие индексирование, например, методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior`.

Более подробно компонент `Query` и язык SQL рассматриваются в *главе 8,* посвященной реляционному доступу к данным.

Объекты поля

Объект поля `Field` имеет тип `TField` и служит полем набора данных. Тип `TField` является абстрактным классом и непосредственно не используется. Вместо него применяются производные классы (табл. 5.1), соответствующие типу данных, размещаемых в рассматриваемом поле набора данных. Производные классы отличаются от базового класса `TField` некоторыми особенностями, связанными с манипулированием конкретным типом данных, например, символьным, числовым или логическим. Далее под объектами типа `TField` мы будем понимать либо сам объект типа `TField`, либо один из производных от него объектов, например, типа `TStringField` (строковое значение) или `TIntegerField` (целочисленное значение).

Таблица 5.1. *Типы объектов* `Field`

Тип объекта	Вид поля
`TBLOBField`	Поле BLOB-значения (BLOB, Binary Large Object — большой двоичный объект)
`TMemoField`	Мемо-поле (поле комментария)
`TGraphicField`	Графическое поле
`TBooleanField`	Поле логического значения
`TBinaryField`	Поле двоичного значения
`TBytesField`	Поле байтового значения фиксированной длины
`TVarBytesField`	Поле байтового значения переменной длины
`TDateTimeField`	Поле значения даты и времени
`TDateField`	Поле значения даты
`TTimeField`	Поле значения времени
`TNumericField`	Поле числового значения

Таблица 5.1 *(окончание)*

Тип объекта	Вид поля
`TBCDField`	Поле BCD-значения
`TFloatField`	Поле вещественного значения
`TCurrencyField`	Поле значения денежной суммы
`TIntegerField`	Поле целочисленного значения (32 разряда)
`TAutoincField`	Поле автоинкрементного значения (32 разряда)
`TSmallintField`	Поле целочисленного короткого значения (16 разрядов)
`TLargeintField`	Поле целочисленного длинного значения (64 разряда)
`TWordField`	Поле целочисленного значения без знака (16 разрядов)
`TStringField`	Поле строкового значения

Объекты типа `TField` являются невизуальными и служат для доступа к данным соответствующих полей записей. Управляя объектами типа `TField`, можно управлять поведением полей, при этом все объекты полей являются независимыми друг от друга. Например, разработчик может запретить изменять значение отдельного поля, несмотря на то что набор данных в целом является модифицируемым и допускает изменение значений других полей. Кроме того, можно скрыть то или иное поле от пользователя, сделав его невидимым.

Задать состав полей набора данных можно двумя способами:

- по умолчанию (динамические поля);
- с помощью редактора полей (статические поля).

По умолчанию при каждом открытии набора данных как на этапе проектирования, так и на этапе выполнения приложения для каждого поля набора автоматически создается свой объект типа `TField`. В этом случае мы имеем дело с *динамическими полями*, достоинством которых является корректность отображения структуры набора данных даже при ее изменении. Напомним, что для компонента `Table` состав полей определяется структурой таблицы, с которой этот компонент связан, а для компонента `Query` состав полей зависит от SQL-запроса.

Однако у использования динамических полей есть и существенные недостатки, связанные с тем, что для полученного набора данных нельзя выполнить такие действия, как ограничение состава полей или определение вычисляемых полей. Поэтому при необходимости этих операций следует использовать второй способ задания состава полей.

На этапе разработки приложения с помощью Редактора полей можно создавать для набора данных *статические* (устойчивые) поля, основные достоинства которых состоят в реализации следующих возможностей:

- определение вычисляемых полей, значения которых рассчитываются с помощью выражений, использующих значения других полей;

- ограничение состава полей набора данных;
- изменение порядка полей набора данных;
- скрытие или показ отдельных полей при выполнении приложения;
- задание формата отображения или редактирования данных поля на этапе разработки приложения.

Необходимо отметить, что при модификации структуры таблицы, например, удалении поля или изменении его типа, открытие набора данных, имеющего статические поля, может привести к возникновению исключения.

Редактор полей

По умолчанию для каждого физического поля при открытии набора данных автоматически создается объект типа `TField`, а все поля в наборе данных являются динамическими и доступными. Для создания статических (устойчивых) полей используется специальный Редактор полей. В случае, если хотя бы одно поле набора данных является статическим, динамические поля больше создаваться не будут. Таким образом, в наборе данных будут доступны только статические поля, а все остальные считаются отсутствующими. Определить или отменить состав статических полей можно с помощью Редактора полей на этапе разработки приложения.

Замечание

Для компонента `Query` состав полей определяется также в тексте SQL-запроса, с помощью которого можно задать или изменить состав полей набора данных, несмотря на то что эти поля являются динамическими.

Во время выполнения приложения можно определить вид полей набора данных с помощью свойства `DefaultFields` типа `Boolean`. Если его значение равно `True`, то набор данных имеет поля по умолчанию, т. е. динамические, в противном случае для набора данных заданы статические поля.

Для запуска Редактора полей (рис. 5.6) следует сделать двойной щелчок на компоненте `Table` или `Query` или вызвать для этих компонентов контекстное меню правой кнопкой мыши и выбрать пункт **Fields Editor**. В заголовке Редактора полей выводится составное имя набора данных, например, `Form1.Table1`. Для перемещения по полям используются четыре кнопки навигации Редактора или мышь. Бóльшую часть Редактора занимает список статических полей, при этом поля перечисляются в порядке их создания, который может отличаться от порядка полей в таблице БД.

Первоначально список статических полей пуст, указывая, что все поля набора данных являются динамическими. С помощью Редактора полей разработчик может выполнить следующие операции:

- создать новое статическое поле;
- удалить статическое поле;
- изменить порядок следования статических полей.

Кроме того, для любого выбранного в редакторе статического поля с помощью Инспектора объектов можно задать или изменить свойства этого поля (объекта типа `TField`) и определить обработчики его событий. Подобные действия разрешается производить благодаря тому, что соответствующие статическим полям объекты типа `TField` доступны на этапе разработки приложения.

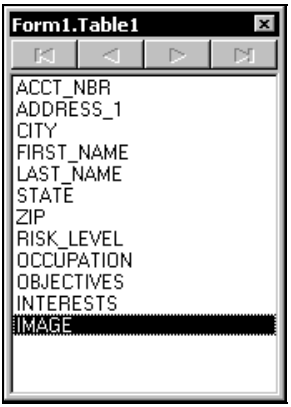


Рис. 5.6. Редактор полей

Рис. 5.7. Добавление новых статических полей

Для создания статического поля следует вызвать контекстное меню Редактора полей и выбрать пункт **Add Fields** (Добавить поля), в результате чего появляется диалоговое окно добавления новых полей (рис. 5.7). В списке **Available fields** (Доступные поля) окна содержатся все те поля набора данных, которые еще не являются статическими. После выбора одного или нескольких полей и нажатия кнопки **OK** эти поля добавляются в состав статических полей набора данных. Добавленное статическое поле является полем *данных* и связано с конкретным физическим полем таблицы БД.

Для добавления в список всех физических полей таблицы (для набора данных `Table`) или результата выполнения SQL-запроса (для набора данных `Query`) нужно выбрать в контекстном меню Редактора полей пункт **Add all Fields** (Добавить все поля).

При каждом открытии (в том числе при разработке приложения) набора данных проверяется возможность создания экземпляров класса `TField` для статических полей. Если поле вследствие какой-нибудь ошибки является недопустимым, и создание объекта типа `TField` невозможно, то генерируется исключение, а набор данных закрывается.

Для удаления статического поля нужно выбрать пункт **Delete** контекстного меню или выделить поле в списке и нажать клавишу <Delete>. После удаления статического поля оно становится недоступным для операций в программе, однако в случае необходимости его снова можно сделать статическим, добавив в

список Редактора полей. При этом следует иметь в виду, что все свойства этого поля устанавливаются заново, а все сделанные ранее изменения теряются.

Замечание

Если удалены все статические поля, то все поля набора данных становятся динамическими и доступными при выполнении приложения.

Порядок следования полей определяется их местом в списке Редактора полей. По умолчанию порядок полей соответствует порядку физических полей в таблицах БД. Его можно изменить, перемещая поля в списке с помощью мыши или комбинаций клавиш <Ctrl>+<Page Up> и <Ctrl>+<Page Down>.

Замечание

Если для набора данных определены статические поля, то изменение значения свойства `TableName` этого набора данных может привести к ошибке, что обычно и происходит. Это связано с тем, что в новой таблице, связываемой с набором данных, могут отсутствовать физические поля, для которых были созданы статические поля. В таких случаях программист должен предусматривать соответствующие операции, например, формирование нового состава статических полей.

Есть три типа статических полей:

- поле данных, связанное с соответствующим физическим полем таблицы;
- вычисляемое поле, значение которого рассчитывается в обработчике события `OnCalcFields` во время выполнения приложения;
- поле выбора, значение которого можно выбирать из списка, формируемого на основе заданных критериев и правил.

Для создания нового статического поля любого типа нужно выбрать в контекстном меню Редактора полей пункт **New Field**, в результате чего открывается одноименное диалоговое окно (рис. 5.8).

Для задания общих свойств (параметров) нового поля используется группа элементов управления **Field properties** (Свойства поля). В поле ввода **Name** задается значение свойства `FieldName` (имя поля), а в поле ввода **Component** — значение свойства `Name` (имя компонента поля — объекта типа `TField`). При программировании обычно используется имя поля. Delphi автоматически формирует значение в поле ввода **Component**, и попытка изменить его не приводит к желаемому результату. В списке **Type** и поле ввода **Size** указываются тип данных и размер поля. Тип данных обязательно задается для всех полей, а необходимость задания размера зависит от типа данных. Например, для поля с типом данных `Integer` задание размера не имеет смысла, а для типа `String` размер поля ограничивает максимальную длину строки.

Тип нового поля выбирается в группе переключателей **Field type** из следующих вариантов:

- **Data** (поле данных);
- **Calculated** (вычисляемое поле);
- **Lookup** (поле выбора).

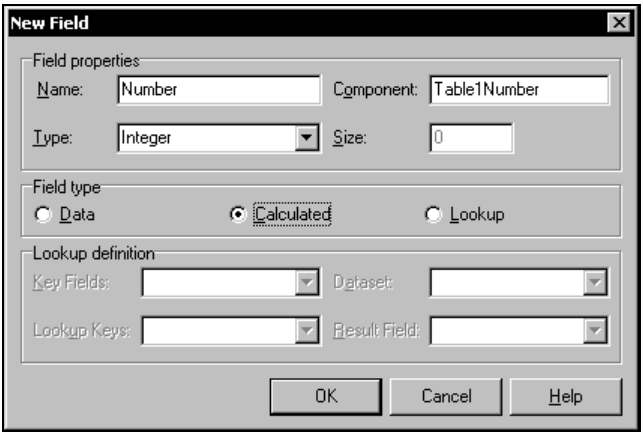


Рис. 5.8. Окно создания статического поля

В группе **Lookup definition** (Определение выбора) для поля выбора устанавливаются такие параметры, как набор данных и поля связи, а также поля для формирования списка выбора и результата.

После создания нового статического поля его свойства становятся доступными через Инспектор объектов и могут быть изменены. При этом каждому параметру, задаваемому с помощью поля ввода или переключателя окна **New Field** (см. рис. 5.8), соответствует определенное свойство объекта типа `TField` (табл. 5.2). Все свойства объекта доступны через Инспектор объектов, за исключением свойства `FieldName`, которое доступно только во время выполнения приложения, однако при разработке приложения значение этого свойства видно в окне Редактора полей. Значение параметра `Type` определяет класс объекта `Field`, который будет соответствовать статическому полю, например, для типа `String` это `TStringField`, а для типа `Float` — это `TFloatField`.

Таблица 5.2. *Свойства объекта поля*

Свойство объекта поля	Элемент управления
`FieldName`	Поле ввода **Name**
`Name`	Поле ввода **Component**
`Size` (для строковых полей)	Поле ввода **Size**
`Precision` (для числовых полей)	
`FieldKind`	Группа переключателей **Field type**
`KeyFields`	Комбинированный список **Key Fields**
`LookupKeyFields`	Комбинированный список **Lookup Keys**
`LookupDataset`	Комбинированный список **Dataset**
`LookupResultField`	Комбинированный список **Result Field**

Свойство `FieldKind` типа `TFieldKind` определяет тип поля и принимает следующие значения:

- `fkData` (поле данных);
- `fkCalculated` (вычисляемое поле);
- `fkLookupField` (поле выбора);
- `fkInternalCalc` (вычисляемое поле, которое сохраняется в наборе данных);
- `fkAggregate` (поле, содержащее агрегированный результат).

Статическое поле данных создается для соответствующего физического поля таблицы рассмотренным выше способом, в то время как создание вычисляемого поля или поля выбора является более сложной задачей.

Для создания статического вычисляемого поля нужно выполнить следующие действия:

1. В окне создания статического поля задать имя и тип поля, а также выбрать переключатель **Calculated**.
2. Для набора данных, содержащего поле, подготовить код обработчика события `OnCalcFields` типа `TDataSetNotifyEvent`, в котором этому полю присваивается требуемое значение, при этом для расчета значения вычисляемого поля можно использовать значения других полей, а также переменные и константы программы.

После создания вычисляемого поля его свойство `FieldKind` автоматически получает значение `fkCalculated`. Кроме того, также автоматически свойство `Calculated` типа `Boolean` устанавливается в значение `True`, указывающее на то, что это поле является вычисляемым. Для полей другого типа свойство `Calculated` имеет значение `False`.

Событие `OnCalcFields` предназначено для определения значений всех вычисляемых полей набора данных. Оно генерируется каждый раз при считывании записи из таблицы, а также при изменении значений невычисляемых полей, если свойство `AutoCalcFields` типа `Boolean` установлено в значение `True` (по умолчанию). На время выполнения обработчика события `OnCalcFields` набор данных переводится в режим `dsCalcFields` расчета вычисляемых полей, а затем возвращается в предыдущий режим.

Замечание

В обработчике события `OnCalcFields` можно присваивать значения вычисляемым полям, а также полям выбора (в том числе не входящим в состав списка выбора). Попытка изменить значение поля данных (физического поля таблицы) вызывает исключение.

Рассмотрим пример, иллюстрирующий использование вычисляемых полей.

Пусть в состав таблицы входят три поля: `Name` (название товара), `Price` (цена единицы товара, в рублях) и `Number` (количество товара). Набор данных должен

содержать для каждого товара также данные об общей стоимости товара в рублях и в Евро. Общая стоимость определяется на основании цены единицы товара и количества товара, поэтому в набор данных добавляются два вычисляемых поля: `Total` и `TotalEvro` для рублей и Евро соответственно (рис. 5.9).

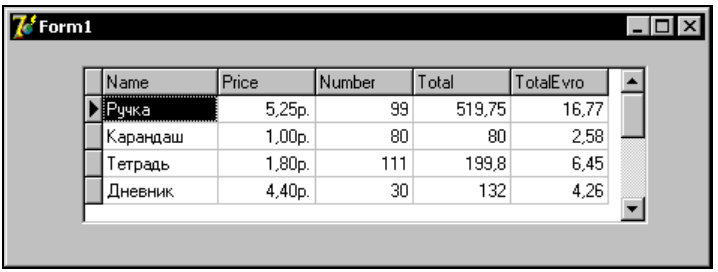


Рис. 5.9. Использование вычисляемых полей

Для вычисления значений новых полей используется код обработчика события `OnCalcFields`, приведенный ниже. Обращение к вычисляемым полям и полям данных выполнено разными способами: по имени компонента (для вычисляемых полей `Total` и `TotalEvro`) и по имени поля (для полей данных `Price` и `Number`). Следует иметь в виду, что поля `Price` и `Number` тоже должны быть статическими, в противном случае при попытке обращения к ним произойдет ошибка. В примере статическим является также поле `Name`.

При расчете стоимости в условных единицах предполагается, что обменный курс составляет 31. В принципе значение этого коэффициента должно быть переменным значением с возможностью его редактирования пользователем.

```
// Отформатировать значения поля можно следующим образом
// Table1TotalEvro.DisplayFormat:='#####.##'
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
Table1Total.AsFloat := Table1.FieldByName('Price').AsFloat*
                        Table1.FieldByName('Number').AsInteger;
Table1TotalEvro.AsFloat := Table1Total.AsFloat/31;
end;
```

Если в качестве обменного курса взять некоторое реальное значение, например, `31.75`, то значения поля `TotalEvro` будут содержать большое количество дробных разрядов. Поэтому необходимо округлять получаемые значения или выводить полученные значения в требуемом формате. Для форматирования значений можно использовать свойство `DisplayFormat` этого поля, установив его через Инспектор объектов или при создании формы, например, следующей инструкцией:

```
Table1TotalEvro.DisplayFormat := '#####.##';
```

Поле выбора предоставляет возможность выбора одного значения из предлагаемого списка и автоматического занесения информации в заданное поле модифицируемой записи. С полем выбора связывается специальный список, заполняемый значениями указанного поля из второго набора данных. Оба набора данных связываются по полю (полям) связи.

Необходимость использования поля выбора может возникнуть в случае, когда при поступлении товара ведется таблица реестра, в которую заносится такая, например, информация (в скобках приведены названия полей таблицы):

- уникальный номер записи (`R_N`);
- дата поступления товара (`R_Date`);
- уникальный код товара (`R_Code`);
- количество поступившего товара (`R_Number`);
- примечание (`R_Note`).

Данные о каждом конкретном товаре хранятся в другой таблице, имеющей поля:

- уникальный код товара (`G_Code`);
- название товара (`G_Name`);
- единица измерения товара (`G_Unit`);
- цена единицы товара (`G_Price`);
- примечание (`G_Note`).

В таблице данных о товаре поле уникального кода товара (`G_Code`) является автоинкрементным — его значение формируется автоматически при добавлении нового товара. При добавлении новой записи в таблицу реестра в поле кода товара (`R_Code`) должен заноситься код поступившего товара. Для установления соответствия между поступившими товарами и их характеристиками обе таблицы связываются по этим полям. Однако при занесении нового товара в реестр пользователь должен самостоятельно задавать код, соответствующий поступившему товару. Эта операция неудобна для пользователя, т. к. нужно знать эти коды, и может привести к ошибкам.

Более удобным вариантом в данном случае будет формирование поля выбора, содержащего товары, из которого пользователь выбирает одно из наименований. Тогда после выбора товара соответствующий код будет заноситься в поле `R_Code` таблицы реестра автоматически. Окно создания поля выбора показано на рис. 5.10.

Поле выбора названо `R_Name` и принадлежит набору данных `Table1`, содержащему записи таблицы реестра. Для формирования списка выбора используется поле `G_Name` набора данных `Table2`, содержащего записи таблицы данных товара. Связь между двумя наборами данных `Table1` и `Table2` осуществляется через их поля кода товара `R_Code` и `G_Code` соответственно (рис. 5.11).

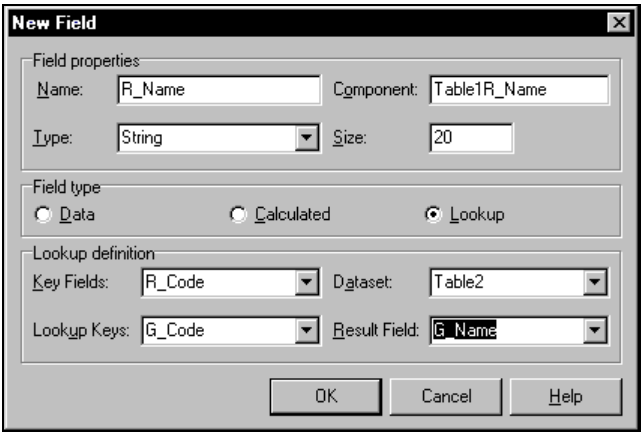


Рис. 5.10. Окно создания поля выбора

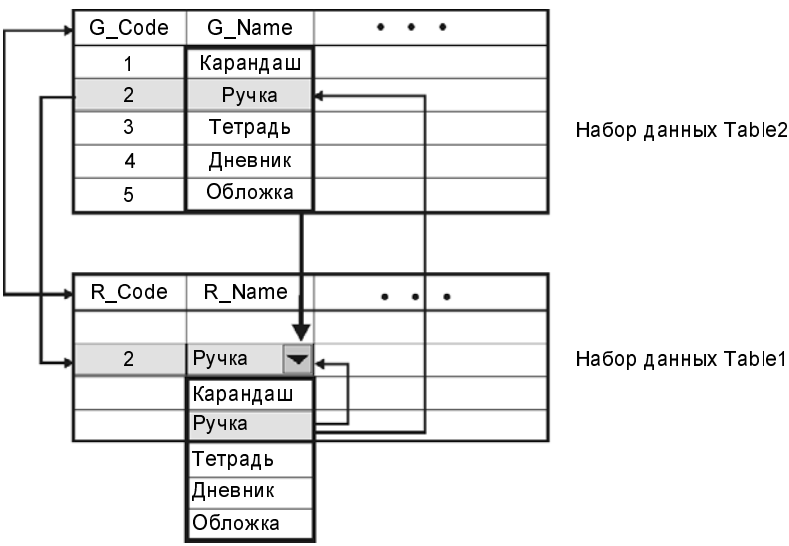


Рис. 5.11. Схема использования поля выбора

Использование поля выбора заключается в том, что пользователь выбирает значение в поле `R_Name`, содержащем список, который построен на основании значений поля `G_Name`. После выбора значения для поля `R_Name` из поля связи `G_Code` автоматически заносится соответствующее значение в поле `R_Code`. Таким образом, поле `R_Name`, содержащее список, используется для выбора, а поле связи `R_Code` — для занесения в него значения.

Замечание

Поля связи не обязательно должны быть индексными.

В частном случае поле выбора и поле связи могут быть одним и тем же полем. Это может потребоваться, например, в случае, когда нет необходимости автоматически вводить код для выбранного значения, а достаточно просто выбрать значение в списке.

На рис. 5.12 демонстрируется использование поля выбора. В верхней части формы отображается таблица реестра, в нижней — таблица данных, которая приведена для наглядности. На практике связанный набор данных обычно не показывается. При выборе значения в списке поля выбора для связанного набора данных текущий указатель устанавливается на запись, значение из которой было выбрано.

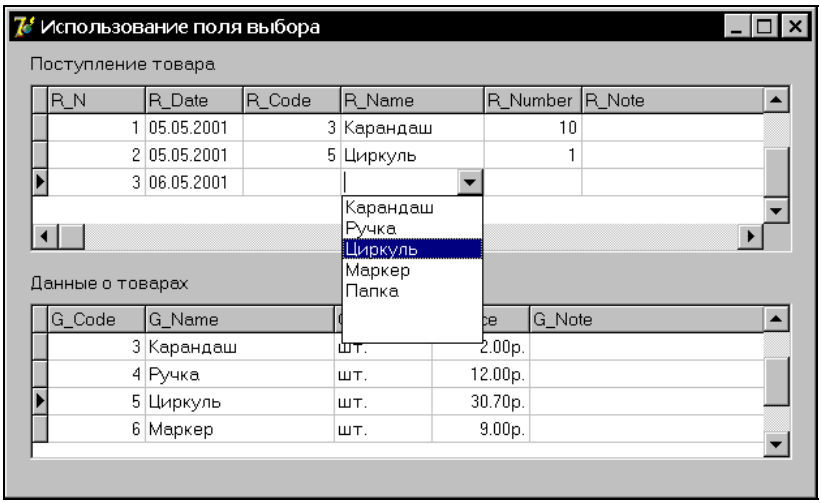


Рис. 5.12. Использование поля выбора

При изменении набора данных, по полю которого построен список выбора, автоматического изменения списка не происходит. Обновление списка выбора является обязанностью программиста, его удобно выполнять с помощью метода `RefreshLookupList`.

Операции с полями

Через объект типа `TField` разработчик может:

- обратиться к полю и его значению;
- проверить тип и значение поля;

- отформатировать значение поля, отображаемое или редактируемое в визуальных компонентах.

При этом динамические и статические поля имеют одинаковые свойства, события и методы, с помощью которых можно управлять этими объектами при выполнении приложения. В связи с тем, что статические поля определяются на этапе разработки, многие их свойства доступны через Инспектор объектов.

Доступ к значению поля

Объект поля, как и любой другой объект, имеет имя (название), определяемое его свойством `Name` типа `String`. Имя объекта `Field` зависит от того, является ли поле динамическим или статическим. По умолчанию для динамического поля имя объекта `Field` совпадает с именем соответствующего физического поля таблицы БД, для которого создан объект, и не может быть изменено. Имя статического поля является *составным* и по умолчанию образуется путем слияния имен набора данных и имени физического поля таблицы БД. Например, если для физического поля `Name` набора данных `Table1` с помощью Редактора полей создано статическое поле, то оно получит имя `Table1Name`. Программист может изменить это имя через Инспектор объектов, когда соответствующее статическое поле выбрано в Редакторе полей.

В отличие от имени объекта `Field`, свойство `FieldName` типа `String` содержит имя *физического поля*, заданное при создании таблицы. Не следует путать свойства `Name` и `FieldName`, они обозначают разные объекты и в общем случае могут не совпадать.

Пример обращения к полю:

```
Table1.FieldByName('Number').DisplayLabel := 'Количество';
Table1Number.DisplayLabel := 'Количество';
```

Здесь для статического поля `Number` возможны два способа обращения: по имени поля в наборе данных и по имени объекта `Field` поля.

Для определения порядкового номера поля в наборе данных можно использовать свойство `FieldNo` типа `Integer`, например, так:

```
var x: integer;
...
x := Table1.FieldByName('Date').FieldNo;
```

Для доступа к значению поля служат свойства `Value` и `AsXXX`. Свойство `Value` типа `Variant` представляет собой фактические данные в объекте типа `TField`. При выполнении приложения это свойство используется для чтения и записи значений в поле. Если программист обращается к свойству `Value`, то он должен самостоятельно обеспечивать преобразование и согласование типов значений полей и читаемых или записываемых значений. В табл. 5.3 приводятся возможные типы свойства `Value` для различных объектов типа `TField`.

Таблица 5.3. *Возможные типы свойства* `Value`

Тип объекта поля	Тип свойства `Value`
`TField`	`Variant`
`TStringField`, `TBLOBField`	`String`
`TIntegerField`, `TSmallntField`, `TWordField`, `TAutoincField`	`Longint`
`TBCDField`, `TFloatField`, `TCurrencyField`	`Double`
`TBooleanField`	`Boolean`
`TDateTimeField`, `TDateField`, `TTimeField`	`TDateTime`

Рассмотрим пример, в котором доступ к значению поля осуществляется с помощью свойства `Value`:

```
procedure TForm1.Button1Click(Sender: TObject);
var s: string;
    x: real;
begin
// Доступ к полю по его имени в наборе данных
s := Table1.FieldByName('Salary').Value;
x := Table1.FieldByName('Salary').Value;
Label1.Caption := s;
Label2.Caption := FloatToStr(x);
// Доступ к полю как к отдельному компоненту
x := Table1Salary.Value;
Label3.Caption := FloatToStr(x);
// Поле вещественного типа,
// в связи с чем следующая инструкция присваивания недопустима
//  s := Table1Salary.Value;
end;
```

Здесь чтение значения поля `Salary` текущей записи набора данных `Table1` выполняется несколькими способами. При доступе к полю по имени в наборе данных значение вещественного поля `Salary` можно читать и использовать и как строковое, и как вещественное значение. При доступе к полю как к отдельному компоненту тип переменной, в данном случае `x`, должен соответствовать типу поля.

Замечание

Доступ к полю по имени объекта типа `TField`, например, `Table1Salary`, возможен только для статических полей, которые существуют на этапе разработки приложения. Попытка использовать имя объекта динамического поля приводит к ошибке при компиляции, т. к. объект поля еще не создан.

Поскольку при доступе к полю с помощью свойства `Value` программист должен обеспечивать преобразование и согласование типов значений, то часто более удобно использовать варианты свойства `AsXXX`:

- `AsVariant` типа `Variant`;
- `AsString` типа `String`;
- `AsInteger` типа `Longint`;
- `AsFloat` типа `Double`;
- `AsCurrency` типа `Currency`;
- `AsBoolean` типа `Boolean`;
- `AsDateTime` типа `TdateTime`.

При использовании любого из этих свойств выполняется *автоматическое преобразование типа* значения поля к типу, соответствующему названию свойства. При этом преобразование должно быть допустимо, в противном случае возникает ошибка компиляции по несоответствию типов, например, при попытке прочитать логическое значение как целочисленное.

Приведем теперь пример, где доступ к значению поля происходит с помощью свойств `AsXXX`:

```
procedure TForm1.Button2Click(Sender: TObject);
var s: string;
    x: real;
begin
// Доступ к полю по его имени в наборе данных
s := Table1.FieldByName('Salary').AsString;
x := Table1.FieldByName('Salary').AsFloat;
Label1.Caption := s;
Label2.Caption := FloatToStr(x);
// Доступ к полю как к отдельному компоненту
s := Table1Salary.AsString;
x := Table1Salary.AsFloat;
Label3.Caption := s;
Label4.Caption := FloatToStr(x);
end;
```

Как и в предыдущем примере, чтение значения поля `Salary` осуществляется несколькими способами. Доступ к полю выполняется по имени поля и по имени объекта поля, а значение поля интерпретируется как строковое или как вещественное.

Замечание

Для того чтобы записать значение в поле, оно должно допускать модификацию, а набор данных должен находиться в соответствующем режиме, например, редактирования или вставки.

При необходимости программист может запретить модификацию поля, а также скрыть его, используя свойства `ReadOnly` и `Visible` типа `Boolean`. Сама возможность модификации данных в отдельном поле определяется значением свойства `CanModify` типа `Boolean`. Напомним, что свойства `ReadOnly` и `CanModify` есть также у набора данных: они определяют возможность модификации набора данных (всех его полей) в целом.

Замечание

Даже если набор данных является модифицируемым и его свойство `CanModify` имеет значение `True`, для отдельных полей этого набора редактирование может быть запрещено, и любая попытка изменить значение такого поля вызовет исключение.

Если поле является невидимым (свойство `Visible` установлено в `False`), но разрешено для редактирования (свойство `ReadOnly` установлено в `False`), то можно изменить значения этого поля программно.

Рассмотрим управление видимостью поля и возможностью его модификации на примере:

```
Table1.FieldByName('Number').ReadOnly := True;
Table1.FieldByName('Salary').Visible := False;
```

Здесь для поля `Number` запрещаются любые изменения, а поле `Salary` скрывается, однако для него по-прежнему допускаются чтение и изменение значения.

Для полей, имеющих типы `TBLOBField` (BLOB-объект), `TGraphicField` (графическое изображение) и `TMemoField` (текст), доступ к их содержимому выполняется обычными для объектов данного типа способами. Например, для загрузки содержимого из файла можно использовать метод `LoadFromFile`.

Проверка типа и значения поля

При выполнении программы можно выяснить тип данных конкретного поля. Это удобно делать в случае, когда, например, типы данных полей таблиц БД заранее неизвестны.

Тип данных поля таблицы определяет свойство `DataType` типа `TFieldType`, принимающее следующие значения:

- `ftUnknown` (тип неизвестен или неопределен);
- `ftString` (короткая строка длиной не более 255 символов);
- `ftSmallInt` (короткое целое число);
- `ftInteger` (целое число);
- `ftWord` (целое число без знака);
- `ftBoolean` (логическое значение);
- `ftFloat` (вещественное число);
- `ftCurrency` (денежное значение);

- `ftBCD` (число в формате BCD (двоично-десятичный формат));
- `ftDate` (дата);
- `ftTime` (время);
- `ftDateTime` (дата и время);
- `ftBytes` (байтовое значение фиксированной длины);
- `ftVarBytes` (байтовое значение переменной длины);
- `ftAutoInc` (автоинкрементное значение);
- `ftBlob` (BLOB-объект);
- `ftMemo` (текст Memo);
- `ftGraphic` (графический объект);
- `ftFmtMemo` (форматированный текст Memo);
- `ftParadoxOle` (поле OLE для таблицы Paradox);
- `ftDBaseOle` (поле OLE для таблицы dBase);
- `ftTypedBinary` (типированное двоичное значение);
- `ftCursor` (курсор для хранимой процедуры Oracle);
- `ftFixedChar` (фиксированное количество символов);
- `ftWideString` (строка);
- `ftLargeInt` (длинное целое число);
- `ftADT` (значение абстрактного типа);
- `ftArray` (массив);
- `ftReference` (REF-поле);
- `ftDataSet` (DataSet-поле).

Программист с помощью специальных свойств может задать ограничения для вводимых в поля значений, а также проверить введенные значения.

Набор символов, допускаемых при вводе значений поля, зависит от типа данных поля:

- `ftBoolean` — все символы;
- `ftSmallInt` — цифры `0 .. 9`, знаки `+` и `-`;
- `ftWord` — цифры `0 .. 9`, знаки `+` и `-`;
- `ftAutoInc` — цифры `0 .. 9`, знаки `+` и `-`;
- `ftDate` — все символы;
- `ftInteger` — цифры `0 .. 9`, знаки `+` и `-`;
- `ftTime` — все символы;
- `ftCurrency` — цифры `0 .. 9`, знаки `+` и `-`, символы `E` или `e`, разделитель разрядов целой и дробной части;

- `ftDateTime` — все символы;
- `ftFloat` — цифры `0 .. 9`, знаки `+` и `-`, символы `E` или `e`, разделитель разрядов целой и дробной части;
- `ftBCD` — цифры `0 .. 9`, знаки `+` и `-`, символы `E` или `e`, разделитель разрядов целой и дробной части;
- `ftString`, `ftVarBytes`, `ftBytes`, `ftBlob`, `ftDBaseOle`, `ftFmtMemo`, `ftGraphic`, `ftMemo`, `ftParadoxOle`, `ftTypedBinary`, `ftUnknown` и `ftCursor` — все символы.

Отметим, что разделитель разрядов целой и дробной части числа (десятичный разделитель) зависит от установок Windows, выполненных через Панель управления.

Набор допустимых для поля символов содержит свойство `ValidChars` типа `TFieldChars`, который описан как множество символов:

```
type TFieldChars = set of Char;
```

Чтобы проверить, разрешен ли символ для ввода в качестве значения поля, удобно использовать метод `IsValidChar (InputChar: Char): Boolean`. Он возвращает значение `True`, если заданный параметром `InputChar` символ является для данного поля допустимым, и `False` — если символ не допускается.

Вот как осуществляется проверка допустимости символа в программе:

```
if not Table1.Fields[2].IsValidChar(Edit1.Text[1]) then
  MessageDlg('Символ "' + Edit1.Text[1] + '" не допустим!',
        mtError, [mbOK], 0);
```

Если первый символ текста, введенного в компонент `Edit1`, не является допустимым для поля (третьего в списке полей) набора данных `Table1`, то выдается сообщение об ошибке.

Замечание

Проверка на корректность символа не гарантирует отсутствие ошибок при вводе значений полей. Например, для числового поля типа `TIntegerField` не подойдет значение `12++3`, хотя каждый символ является допустимым.

Для полей числовых типов, например, объектов типа `TIntegerField` и `TFloatField` свойства `MinValue` и `MaxValue` позволяют установить минимальное и максимальное возможные значения, которые могут быть введены в соответствующее поле. Тип этих свойств определяется конкретным объектом типа `TField`.

Так устанавливается ограничение вводимых значений в программе:

```
Table1Price.MinValue  := 0;
Table1Price.MaxValue  := 9999.99;
Table1Number.MinValue := 0;
Table1Number.MaxValue := 1000;
```

Здесь для поля цены (вещественный тип) устанавливается допустимый диапазон `0 .. 9999.99`, а для поля количества (целый тип) — диапазон `0 .. 1000`.

Ограничения на вводимые значения можно определить также с помощью свойства `CustomConstraint` типа `String`. В этом свойстве допустимый диапазон задается с помощью конструкций, похожих на команды SQL или фильтры для отбора данных. Отличие заключается в том, что команды SQL и фильтры действуют при формировании наборов данных, а конструкция свойства `CustomConstraint` — при задании ограничений на вводимые значения. Например:

```
Table1Price.CustomConstraint := 'Price >= 0 and Price <= 9999.99';
Table1Number.CustomConstraint := 'Number >= 0 and Number <= 1000';
```

Здесь для полей цены и количества устанавливаются те же диапазоны допустимых значений, что и в предыдущем примере.

Если для поля некоторым способом задан диапазон допустимых значений, то при любой попытке установить поле в значение, выходящее за границы этого диапазона, вызывается исключение. При этом пользователю выдается соответствующее сообщение, а значение отвергается. Контроль значений относится не только к вводу со стороны пользователя, он также действует и при изменении значения поля программным способом.

С помощью свойства `ConstraintErrorMessage` типа `String` разработчик при желании может определить собственное сообщение, которое будет выдаваться при нарушении границ диапазона, например:

```
Table1Number.CustomConstraint := 'Number >= 0 and Number <= 10';
Table1Number.ConstraintErrorMessage :=
          'Количество должно находиться в диапазоне 0 .. 10';
```

Разработчик может не только ограничивать значения, вводимые в поля, но и задавать через свойство `DefaultExpression` типа `String` значение по умолчанию, которое автоматически заносится в поле при добавлении новой записи. Отметим, что это свойство строкового типа, поэтому его значения заключаются в кавычки.

Пример установки значения по умолчанию:

```
Table1Number.DefaultExpression := '100';
```

Чтобы проверить, действуют ли для поля ограничения, заданные в свойствах `CustomConstraint`, `DefaultExpression`, а также в `ImportedConstraint` (ограничения, налагаемые сервером), используется свойство `HasConstraints` типа `Boolean`. Это свойство, действующее во время выполнения приложения, принимает значение `True`, если установлено хотя бы одно из перечисленных ограничений.

Кроме проверки выхода за границы допустимых значений, программист может выполнить и более сложную проверку, используя обработчики событий `OnSetText` (типа `TFieldSetTextEvent`, `OnValidate`) и `OnChange` (типа `TFieldNotifyEvent`),

возникающих в указанном здесь порядке при изменении значения поля. Типы этих событий описаны так:

```
type TFieldSetTextEvent = procedure (Sender: TField; const Text: string)
  of object;
type TFieldNotifyEvent = procedure (Sender: TField) of object;
```

Все упомянутые события возникают до изменения значения поля, поэтому программист может в необходимых случаях отказаться от внесения изменения. Отказ от изменения значения поля для этих событий выполняется различными способами. При использовании событий `OnValidate` и `OnChange` программист возбуждает исключение, в результате чего запись значения в поле не происходит. Если исключение не генерируется, то новое значение заносится в поле. Так как обработчик не получает нового значения поля, для доступа к нему нужно использовать свойства `Value` или `AsXXX`. Следует иметь в виду, что эти свойства возвращают новое (предлагаемое для утверждения) значение поля, которое может быть и не принято, в этом случае происходит возврат к старому значению.

При редактировании поля пользователем события `OnValidate` и `OnChange` не вызываются до тех пор, пока не будет выполнена попытка сохранить изменения, например, нажатием клавиши <Enter>.

Рассмотрим на примере, как осуществляется проверка вводимых значений.

В поле `Code` запрещается ввод значений `1` и `3`, которые зарезервированы для специальных целей. При попытке ввода неправильных значений после соответствующей проверки генерируется "тихое" исключение (посредством вызова процедуры `Abort`). В результате попытка ввода отвергается, при этом никакие сообщения пользователю не выдаются.

```
procedure TForm1.Table1NumberValidate(Sender: TField);
begin
if  (Table1.FieldByName('Code').AsString = '1') or
      (Table1.FieldByName('Code').AsString = '3')
    then Abort;
end;
```

Для выдачи сообщения о возникшем исключении в дополнение к процедуре `Abort` можно использовать, например, функцию `MessageDlg`, выводящую диалоговое окно. Кроме того, вместо вызова с помощью `Abort` "тихого" исключения можно вызвать обычное исключение, например, следующим образом:

```
Raise Exception.Create('Код не может равняться значениям 1 и 3!');
```

Для события `OnSetText` действует противоположный порядок принятия изменений и отказа от них. Параметр `Text` обработчика этого события содержит новое значение, которое предложено для занесения в поле, однако запись этого значения не осуществляется автоматически. Поэтому программист должен "вручную" занести предложенное значение в поле: если этого не сделать, то поле остается без изменений. Таким образом, по умолчанию в обработчике события `OnSetText` новое значение *отвергается*.

Вот еще один пример проверки вводимых значений:

```
procedure TForm1.Table1NumberSetText(Sender: TField; const Text: String);
begin
if not ((Text = '1') or (Text = '3'))
   then Table1.FieldByName('Code').AsString := Text;
end;
```

Ряд полей, например, индексное поле, не могут быть пустыми, в этом случае у поля свойство `Required` типа `Boolean` имеет значение `True`. Для проверки полей, содержат ли они отличное от пустого значение, используется свойство `IsNull` типа `Boolean`. Это свойство возвращает значение `True`, если значение поля не задано, и значение `False` — в противном случае. Свойство `IsNull` можно использовать, например, в обработчиках событий `OnSetText`, `OnValidate` и `OnChange`:

```
procedure TForm1.Table1CodeValidate(Sender: TField);
begin
if (Table1Code.Required) and (Table1Code.isNull)
    then Raise Exception.Create('Поле кода не может быть пустым!');
end;
```

Иногда перед занесением значения в некоторое поле необходимо проверить, является ли символ допустимым. Для этого можно использовать метод `IsValidChar`, проверяющий допустимость заданного символа, и свойство `ValidChars`, возвращающее набор допустимых для данного поля символов.

Замечание

Некоторые из описанных ограничений на значения полей также могут быть заданы при создании таблицы БД (запрет пустого значения и задание диапазона допустимых значений). В этом случае ограничения действуют на физическом уровне и не могут быть отменены.

Форматирование отображаемого значения поля

То, как выглядит значение поля в визуальных компонентах, зависит от типа поля, а также от системных установок Windows и BDE. Часто возникает необходимость изменить этот вид при отображении значений или их редактировании. Для этого используется *форматирование выводимой информации.*

Замечание

Названное форматирование относится только к отображению значений в визуальных компонентах и не затрагивает вид и размещение данных в таблице БД.

Свойство `DisplayFormat` типа `String` управляет отображением значений полей в визуальных компонентах, например, `DBGrid` и `DBEdit`. Это свойство дей-

ствует для числовых полей, а также для полей даты и времени. В качестве значения свойства `DisplayFormat` задается маска, определяющая формат отображения поля.

Эта маска состоит из трех секций, разделенных символом `;`. Секции задают форму отображения положительных, отрицательных и нулевых значений соответственно. Если задана только одна секция, то она применяется для вывода всех значений. В маске можно использовать следующие управляющие символы:

- `0` — цифра числа, незначащие нули отображаются;
- `#` — цифра числа, незначащие нули гасятся;
- `.` — разделитель целой и дробной части числа;
- `,` — разделитель тысяч;
- `E+` или `e+` — разделитель мантиссы и порядка для чисел в форме с плавающей точкой, отображаются положительный и отрицательный знаки порядка;
- `E-` или `e-` — разделитель мантиссы и порядка для чисел в форме с плавающей точкой, отображается только отрицательный знак порядка;
- `"xx"` и `'xx'` — символы, выводимые без изменений; символы, заключенные в кавычки (двойные или апострофы), включаются в отображаемое значение;
- `;` — разделитель секций маски для положительных, отрицательных и нулевых значений.

Приведем примеры масок:

- `000E-00` или `000E+00` — для чисел в форме с плавающей точкой;
- `#####0.00` или `000000.00` — для чисел в форме с фиксированной точкой;
- `#####0.00' рублей'` — для денежных сумм;
- `####0;-####0;0` — отдельно для положительных, отрицательных и нулевых значений.

Свойство `EditFormat` типа `String` задает маску, используемую при форматировании значения числового поля перед его отображением для редактирования в визуальном компоненте. Формирование этой маски не отличается от формирования маски, задаваемой в свойстве `DisplayFormat`.

Свойство `EditMask` типа `String` задает маску, используемую при редактировании значения в визуальном компоненте. Маска позволяет ограничить число вводимых пользователем символов и тип вводимых символов (алфавитный, цифровой и т. д.). Кроме того, во вводимую информацию можно вставить дополнительные символы (разделители при вводе даты, времени и т. п.). С помощью свойства `EditMask` удобно вводить телефонные номера, даты, почтовые индексы и другую информацию подобного рода, имеющую заранее определенный формат.

Способы задания маски, содержащейся в свойстве `EditMask`, рассмотрены в *главе 6*, посвященной визуальным компонентам.

Приведенные свойства DisplayFormat и EditMask независимо друг от друга управляют форматированием значения поля при отображении и при редактировании. Кроме них, для объекта типа TField есть событие OnGetText типа TFieldGetTextEvent, в обработчике которого на программном уровне можно *одновременно* управлять форматом данных и при отображении, и при редактировании. Это событие генерируется при каждом обращении к свойствам DisplayText и Text. Тип события OnGetText описан так:

```
type TFieldGetTextEvent = procedure (Sender: TField; var Text: string;
  DisplayText: Boolean) of object;
```

Параметр Text содержит значение, которое выводится в визуальном компоненте. Программист должен задать это значение. Логический параметр DisplayText указывает, для каких целей выводится значение: для отображения (DisplayText = True) или для редактирования (DisplayText = False).

Рассмотрим пример, в котором используется форматирование отображаемых и редактируемых значений:

```
procedure TForm1.Table1NameGetText(Sender: TField; var Text: String;
DisplayText: Boolean);
begin
if DisplayText then Text := Table1Name.AsString
       else Text := AnsiUpperCase(Table1Name.AsString);
   end;

procedure TForm1.Table1PercentGetText(Sender: TField; var Text: String;
DisplayText: Boolean);
begin
if DisplayText then Text := Table1Percent.AsString + '%'
       else Text := Table1Percent.AsString;
end;
```

Значение поля Percent отображается в визуальных компонентах с знаком %. При редактировании этого поля знак процента отсутствует. При редактировании значения поля Name оно выводится строчными буквами. Отметим, что если для перевода символов строки в верхний регистр вместо функции AnsiUpperCase использовать UpperCase, то символы русского алфавита будут преобразовываться некорректно.

Замечание

Если определен обработчик события OnGetText, то свойства DisplayFormat и EditMask не действуют.

Свойство DisplayWidth типа Integer определяет число символов, предназначенных для вывода значения поля в визуальном компоненте (обычно в сетке DBGrid), который связан с этим полем. По умолчанию значение этого свойства определяется шириной физического поля таблицы, заданной при его создании.

Замечание

Ширина столбца компонента-сетки DBGrid также зависит от значения свойства Width этого столбца, задающего ширину в пикселах.

Программист может изменить не только значение, но и заголовок поля, используемый как название столбца, например, в компоненте DBGrid. Заголовок поля определяется свойством DisplayLabel типа String. По умолчанию значением этого свойства является имя поля (свойство FieldName), однако, в отличие от последнего, свойство DisplayLabel доступно и для записи, и его можно использовать, например, для настройки пользователем интерфейса приложения.

Пример настройки заголовка поля:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1Name.DisplayLabel := Edit1.Text;
end;
```

Здесь при нажатии кнопки заголовку поля Name присваивается текст, введенный в компонент Edit1. Если это поле отображается в компоненте DBGrid, то соответствующий столбец сетки также изменяет свое значение на указанное.

Источник данных

Источник данных используется как промежуточное звено между набором данных и визуальными компонентами, с помощью которых пользователь управляет этим набором данных. В Delphi источник данных представлен компонентом DataSource.

Для указания набора данных, с которым связан источник данных, служит свойство DataSet типа TDataSet последнего. Визуальные компоненты связаны с источником данных через свои свойства DataSource. Обычно связь между источником и набором данных устанавливается на этапе проектирования в Инспекторе объектов, однако при необходимости эту связь можно установить или разорвать динамически. При смене у компонента DataSource набора данных визуальные компоненты автоматически подключаются к новому набору данных.

Указать набор данных можно, например, так:

```
DataSource2.DataSet := Nil;
DataSource1.DataSet := Table2;
```

Здесь для компонента DataSource2 связь с набором данных разрывается, а для компонента DataSource1 назначается набор данных Table2.

Для анализа состояния, в котором находится набор данных, можно использовать свойство State типа TDataSetState. При каждом изменении состояния набора данных для связанного с ним источника данных DataSource генерируется событие OnStateChange типа TNotifyEvent.

Если набор данных является редактируемым, то свойство AutoEdit типа Boolean определяет, может ли он автоматически переводиться в режим модификации при выполнении пользователем определенных действий. Например, для компонентов DBGrid и DBEdit таким действием является нажатие алфавитно-цифровой клавиши, когда компонент находится в фокусе ввода. По умолчанию свойство AutoEdit имеет значение True, и автоматический переход в режим модификации разрешен. Если необходимо защитить данные от случайного изменения, то одной из предпринимаемых мер является установка свойства AutoEdit в значение False.

Замечание

Значение свойства AutoEdit влияет на возможность редактирования набора данных только со стороны пользователя. Программно можно изменять записи независимо от значения этого свойства.

Без учета значения свойства AutoEdit пользователь может переводить набор данных в режим модификации путем нажатия кнопок компонента DBNavigator.

При изменении данных текущей записи генерируется событие OnDataChange типа TDataChangeEvent, описанного так:

```
type TDataChangeEvent = procedure (Sender: TObject; Field: TField) of object;
```

Параметр Field указывает на измененное поле; если данные изменены более чем в одном поле, то этот параметр имеет значение Nil. Следует иметь в виду, что в большинстве случаев событие OnDataChange генерируется и при переходе к другой записи. Это происходит, если хотя бы одно поле записи, ставшей текущей, содержит значение, отличное от значения этого же поля для записи, которая была текущей. Событие OnDataChange можно использовать, например, для контроля положения указателя текущей записи и выполнения действий, связанных с его перемещением. Это событие генерируется также при открытии набора данных.

Вот как осуществляется контроль перемещения указателя текущей записи:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
Label1.Caption := 'Запись номер ' + IntToStr(Table1.RecNo);
end;
```

При модификации текущей записи, кроме события OnDataChange, генерируется еще событие OnUpdateData типа TNotifyEvent. Оно возникает непосредственно перед записью данных в БД, поэтому в его обработчике можно предусмотреть дополнительный контроль и обработку введенных в поля значений, а также некоторые другие действия, например, отказ от изменения записи.

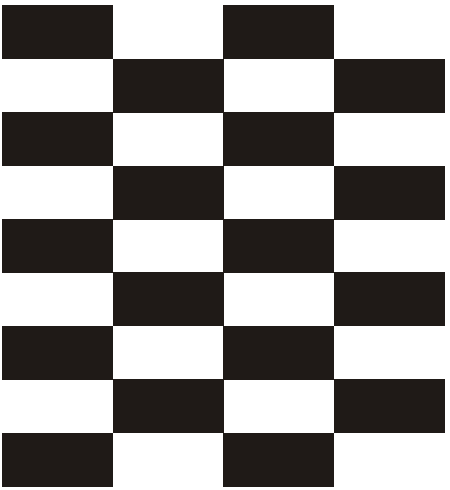
Иногда в визуальных компонентах требуется отключать отображение полей записей набора данных, например, при переборе записей в цикле их обработки,

поскольку при этом возникает мелькание данных вследствие их быстрой смены. Для управления отображением записей можно использовать свойство `Enabled` типа `Boolean` источника данных.

Рассмотрим пример, демонстрирующий, как производится такое управление:

```
procedure TForm1.Button1Click(Sender: TObject);
Var n: integer;
begin
// Отключение отображения записей в визуальных компонентах
DataSource1.Enabled := False;
Table1.First;
for n := 1 to Table1.RecordCount do begin
 // Обработка записи набора данных Table1
   ...
 Table1.Next;
 end;
// Включение отображения записей в визуальных компонентах
DataSource1.Enabled := True;
end;
```

В цикле перебираются все записи набора данных `Table1`. Перед началом цикла отображение записей в визуальных компонентах отключается, а после цикла — включается. Эти действия также можно выполнить, используя методы `DisableControls` и `EnableControls` набора данных.



ЧАСТЬ II

Технологии доступа к данным

Глава 6

Визуальные компоненты для работы с данными

Визуальные компоненты для работы с данными расположены на странице **Data Controls** Палитры компонентов и предназначены для построения интерфейсной части приложения. Они используются для навигации по набору данных, а также для отображения и редактирования записей. Часто эти компоненты называют *элементами, чувствительными к данным.*

Одни визуальные компоненты для работы с данными предназначены для выполнения операций с полями отдельной записи, они отображают и позволяют редактировать значение поля текущей записи. К таким компонентам относятся, например, однострочный редактор `DBEdit` и графическое изображение `DBImage`.

Другие компоненты служат для отображения и редактирования сразу нескольких записей. Примерами таких компонентов являются сетки `DBGrid` и `DBCtrlGrid`, выводящие записи набора данных в табличном виде.

Визуальные компоненты для работы с данными похожи на соответствующие компоненты страниц **Standard**, **Additional** и **Win32** и отличаются, в основном, тем, что ориентированы на работу с БД и имеют дополнительные свойства `DataSource` и `DataField`. Первое из них указывает на источник данных — компонент `DataSource`, второе — на поле набора данных, с которым связан визуальный компонент. Например, однострочный редактор `DBEdit` работает так же, как однострочный редактор `Edit`, отображая строковое значение и позволяя пользователю изменять его. Отличие компонентов состоит в том, что в редакторе `DBEdit` отображается и изменяется значение определенного поля текущей записи набора данных.

Отметим, что для всех визуальных компонентов, предназначенных для отображения и редактирования значений полей, при изменении пользователем их содержимого набор данных автоматически переводится в режим редактирования. Произведенные с помощью этих компонентов изменения также автоматически сохраняются в связанных с ними полях при наступлении определенных событий, таких, например, как потеря фокуса визуальным компонентом или переход к другой записи набора данных.

При программном изменении содержимого этих визуальных компонентов набор данных не переводится в режим редактирования автоматически. Для этой цели в коде должен предварительно вызываться метод `Edit` набора данных. Чтобы сохранить изменения в поле (полях) текущей записи, программист также должен предусмотреть соответствующие действия, например, вызов метода `Post` или переход к другой записи набора данных.

В табл. 6.1 приводятся так называемые стандартные, дополнительные и 32-разрядные визуальные компоненты, расположенные на страницах **Standard**, **Additional** и **Win32** Палитры компонентов, а также соответствующие им визуальные компоненты для работы с данными (страница **Data Controls**).

Таблица 6.1. *Соответствие визуальных компонентов, расположенных на разных страницах Палитры компонентов*

Комоненты страниц *Standard*, *Additional* и *Win32*	**Компоненты страницы *Data Controls***
`Label`	`DBText`
`Edit`	`DBEdit`
`Memo`	`DBMemo`
`RichEdit`	`DBRichEdit`
`ListBox`	`DBListBox`
`ComboBox`	`DBComboBox`
`CheckBox`	`DBCheckBox`
`RadioGroup`	`DBRadioGroup`
`Image`	`DBImage`
`StringGrid`	`DBGrid`
`Chart`	`DBChart`

Замечание

Часть компонентов (`QRLabel`, `QRRichEdit`, `QRDBRichEdit`, `QRImage`, `QRDBImage`, `QRShape`, `QRChart`), используемых для формирования отчетов в предыдущих версиях Delphi (изъятая страница **QReport** Палитры компонентов), тоже имеет свои аналоги среди других визуальных компонентов. Для обеспечения обратной совместимости компоненты страницы **QReport** размещены в пакете dclqrt70.bpl в папке \Delphi 7\Bin и при необходимости могут быть установлены в Палитру компонентов.

В этой главе мы рассмотрим особенности отдельных визуальных компонентов, предназначенных для работы с данными.

Отображение и редактирование значения логического поля

Логическое поле (поле логического типа) может содержать одно из двух значений: `True` (истина) или `False` (ложь). Разрешается использование прописных букв и сокращение вводимого значения, т. е. допустимы значения `True`, `true`, `tru`, `Tr`, `t` и т. д.

Для отображения и изменения значения логического поля можно использовать редактор `DBEdit`. Однако удобнее выполнять эти действия с помощью флажка (независимого переключателя) `DBCheckBox`, который позволяет "включить" или "выключить" значение логического поля.

Флажок `DBCheckBox` является аналогом рассмотренного ранее компонента `CheckBox`, поэтому здесь мы остановимся только на свойствах, характерных именно для этого флажка.

Компонент `DBCheckBox` выглядит на экране как квадратик (флажок) с текстовым заголовком (см. рис. 6.1). Если в нем находится галочка (при этом говорят, что флажок "включен" или "установлен"), то связанное с этим флажком логическое поле текущей записи содержит значение `True`. Если же квадратик пуст (флажок снят), то логическое поле текущей записи содержит значение `False`.

Флажок `DBCheckBox` можно применять также для отображения и редактирования строковых полей, если воспользоваться свойствами `ValueChecked` и `ValueUnChecked`.

Свойство `ValueChecked` типа `String` содержит строковые значения, которые устанавливают связанный с этим полем флажок во включенное состояние. Отдельные значения разделяются точкой с запятой. В качестве значений допускаются любые алфавитно-цифровые символы, в том числе русские буквы. Регистр алфавитных символов не различается, т. е. значения `ДА` и `да` считаются одинаковыми. Например:

```
DBCheckBox1.ValueChecked := 'True;T;Yes;Y;Да;Д';
```

Свойство `ValueUnChecked` типа `String` содержит строковые значения, которые устанавливают связанный с этим полем флажок в выключенное состояние. Значения задаются таким же образом, как и для свойства `ValueChecked`:

```
DBCheckBox1.ValueUnChecked := 'False;F;No;N;Нет;Н';
```

Если поле не содержит ни одного из значений, указанных в свойствах `ValueChecked` и `ValueUnChecked`, то флажок устанавливается в неопределенное состояние.

Отметим, что несмотря на наличие приведенных свойств, возможности флажка `DBCheckBox` по редактированию строковых полей намного меньше, чем у элемента `DBEdit`.

Отображение и выбор значения поля

В ряде случаев возникает необходимость ввода или отображения в поле одного из значений, входящих в состав фиксированного набора. С этой целью удобно использовать компонент `DBRadioGroup` (рис. 6.1), представляющий собой группу зависимых переключателей, из которых в каждый момент времени может быть включен (выбран) только один. Зависимые переключатели также называют просто переключателями; в форме они отображаются в виде кружка с текстовой надписью.

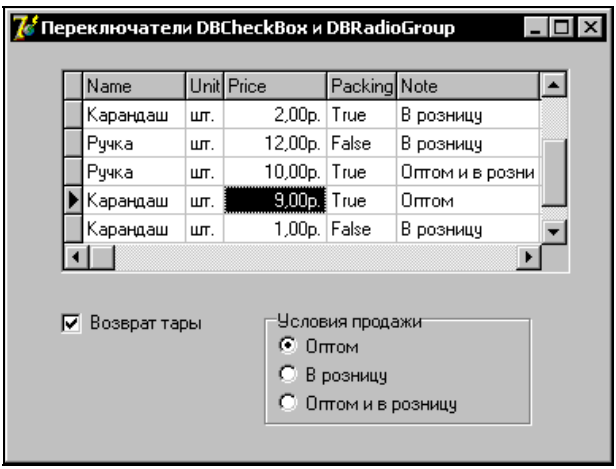


Рис. 6.1. Использование переключателей

Отметим, что выбрать для поля одно из значений можно также с помощью списков, которые представляют для этих целей более широкие возможности.

Управление числом и названиями переключателей производится с помощью свойства `Items` типа `TStrings`, позволяющего получить доступ к отдельным переключателям в группе. Это свойство содержит строки, отображаемые как заголовки переключателей. Отсчет строк в массиве начинается с нуля: `Items[0]`, `Items[1]` и т. д. Обычно задание значений этого свойства выполняется при разработке приложения с помощью Редактора строк. При выполнении приложения для манипуляции со строками (заголовками) можно использовать такие методы, как `Add` и `Delete`.

Свойство `Values` типа `TStrings` содержит список значений поля, на которые должны реагировать переключатели группы. Управление этим списком осуществляется аналогично управлению списком `Items`. Если возможные значения свойства `Values` не заданы, то они выбираются из значений свойства `Items`, т. е. в этом случае значение, соответствующее переключателю, совпадает с названием этого переключателя.

Группа переключателей работает следующим образом. Переключатель включается при переходе к очередной записи, если значение связанного с ним поля содержит одно из значений, присутствующих в списке `Values`. Если же поле содержит значение, отсутствующее в списке возможных, то ни один переключатель не выбирается. Изменение значения поля происходит при выборе другого переключателя группы.

Замечание

Для поля, с которым связана группа переключателей, пользователь может выбрать значение только из списка. Попытки ввести в поле произвольное значение, например с помощью компонента `DBGrid`, блокируются.

В примере, приведенном на рис. 6.1, группа переключателей имеет заголовок **Условия продажи** и связана с полем `Note` набора данных. При перемещении на шестую запись в группе автоматически выбирается третий переключатель, имеющий название (и такое же значение) **Оптом и в розницу**. Если выбрать другой переключатель, например первый, то в поле `Note` шестой записи автоматически занесется новое значение — **Оптом**.

Замечание

Новое значение поля будет зафиксировано в наборе данных и отображено другими визуальными компонентами (`DBGrid` и подобными) после потери фокуса группой переключателей, например, в связи с переходом к другой записи.

Доступ к отдельному переключателю можно получить через свойство `ItemIndex` типа `Integer`, содержащее позицию (номер) переключателя, выбранного в группе в текущий момент. Это свойство используется для выбора отдельного переключателя или для определения, какой из переключателей является выбранным. Если свойство `ItemIndex` имеет значение `-1`, то не выбран ни один из переключателей.

Замечание

При программном выборе переключателя необходимо переводить набор данных в режим редактирования, а после установки нового значения поля закреплять сделанные изменения, например, вызовом метода `Post`.

Вот пример, иллюстрирующий выбор переключателя на программном уровне:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.Edit;
DBRadioGroup1.ItemIndex := 1;
Table1.Post;
end;
```

В приведенной процедуре при нажатии кнопки `Button1` в группе `DBRadioGroup1` выбирается второй переключатель. Предварительно набор данных переводится в режим редактирования, а после переключения и смены значения поля сделанные изменения сохраняются.

Отображение и выбор значения поля в списке

Чтобы отобразить поле текущей записи и выбрать этому полю новое значение, служат списки — компоненты `DBListBox`, `DBComboBox`, `DBLookupComboBox` и `DBLookupListBox`. По своей функциональности списки напоминают группу переключателей `DBRadioGroup`, однако предоставляют большие возможности. Список `DBComboBox` также обеспечивает возможность ввода в поле *произвольного значения*.

Простой и комбинированный списки

Компоненты `DBListBox` и `DBComboBox` позволяют пользователю выбирать один из строковых элементов. При выборе нужной строки простого списка `DBListBox` содержащееся в ней значение автоматически заносится в поле, с которым связан этот список. В дополнение к этому комбинированный список `DBComboBox` позволяет вводить произвольное значение.

Замечание

Новое значение не фиксируется в поле и, соответственно, не отображается в других визуальных компонентах, связанных с этим же набором данных. На рис. 6.2 в списке для поля `Unit` выбрано новое значение `Упаковка`, однако в сетке `DBGrid` это не отображается. Закрепление в поле нового значения, а также отображение этого значения в других визуальных компонентах, связанных с данным полем, произойдет при переходе к другой записи.

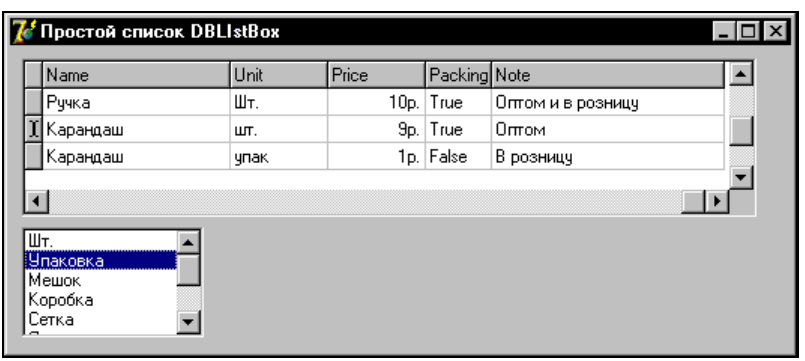


Рис. 6.2. Использование простого списка

Списки, сформированные по значениям поля набора данных

В простом и комбинированном списках перечень возможных значений поля формируется программистом и в общем случае не зависит от значений записей наборов данных. Иногда это может оказаться неудобным и привести к ошибкам ввода или выбора. В Delphi имеются компоненты `DBLookupListBox` и `DBLookupComboBox`, которые также предназначены для выбора значения в списке или непосредственного ввода значения в поле редактирования (только для `DBLookupComboBox`). Во многом эти компоненты похожи на простой список `DBListBox` и комбинированный список `DBComboBox` соответственно и отличаются от них только способом формирования списка возможных значений.

Принципы работы с компонентами `DBLookupComboBox` и `DBLookupListBox` практически такие же, как с полем выбора, которое было рассмотрено в *главе 5*. Основное отличие заключается в том, что поле выбора создается с помощью Редактора полей, и его основные свойства задаются в специальном окне, а в случае указанных компонентов для списка используется один из них (`DBLookupComboBox` или `DBLookupListBox`), а свойства устанавливаются, как правило, через Инспектор объектов.

Компонент `DBLookupComboBox` (или `DBLookupListBox`) связывается с полем "своего" набора данных через свойства `DataSource` и `DataField`, а список формируется с помощью свойств `ListSource` (типа `TDateSource`) и `DataField` (типа `String`), указывающих на второй набор данных и его поле, используемое для заполнения списка.

В свойстве `KeyField` типа `String` указывается поле второго набора данных, значение которого заносится в поле, связанное с компонентом списка.

Представление записей в табличном виде

Для вывода записей набора данных в табличном виде удобно использовать сетку, представленную в Delphi компонентом `DBGrid`. Внешний вид сетки соответствует внутренней структуре таблицы БД и набора данных, при этом строке сетки соответствует запись, а столбцу — поле.

С помощью сетки пользователь управляет набором данных, поля которого в ней отображаются. Для навигации по записям и их просмотра используются полосы прокрутки и клавиши перемещения курсора. Для перехода в режим редактирования поля достаточно установить на него курсор и нажать любую алфавитно-цифровую клавишу. Переход в режим вставки новой записи выполняется нажатием клавиши <Insert>, после чего можно заполнять поля. Вставка записи происходит в том месте, где находится указатель текущей записи. Изменения, сделанные при редактировании или добавлении записи, подтверждаются нажатием клавиши <Enter>, или переходом к другой записи, или отменяются нажатием

клавиши <Esc>. Для удаления записи следует нажать комбинацию клавиш <Ctrl>+<Delete>.

Характеристики сетки

Несмотря на то что по своему виду сетка `DBGrid` похожа на сетку `StringGrid`, между ними есть значительные различия. Так, у сетки `StringGrid` можно устанавливать через соответствующие свойства число ее строк и столбцов. У сетки `DBGrid` числом строк управлять нельзя, т. к. она отображает все записи, имеющиеся в наборе данных.

Основным свойством сетки является свойство `Columns` типа `TDBGridColumns`, которое представляет собой массив (коллекцию) объектов `Column` типа `TColumn`, описывающих отдельные столбцы сетки.

Свойство `SelectedIndex` типа `Integer` задает номер текущего столбца в массиве `Columns`, а свойство `SelectedField` указывает на объект типа `TField`, которому соответствует текущий столбец сетки.

Свойство `FieldCount` типа `Integer`, доступное во время выполнения программы, содержит число видимых столбцов сетки, а свойство `Fields [Index: Integer]` типа `TField` позволяет получить доступ к отдельным столбцам. Индекс определяет номер столбца в массиве столбцов и принимает значения в интервале `0..FieldCount - 1`.

Свойства `Color` и `FixedColor` типа `TColor` задают цвета сетки и ее фиксированных элементов соответственно. По умолчанию свойство `Color` имеет значение `clWindow` (цвет фона Windows), а свойство `FixedColor` — значение `clBtnFace` (цвет кнопки).

Свойство `TitleFont` типа `TFont` определяет шрифт, используемый для вывода заголовков столбцов.

Доступ к параметрам сетки (например, для настройки) возможен через свойство `Options` типа `TGridOptions`. Это свойство представляет собой множество и принимает комбинации следующих значений:

- `dgEditing` (пользователю разрешается редактирование данных в ячейках);
- `dgAlwaysShowEditor` (сетка не блокирует режим редактирования);
- `dgTitles` (отображаются заголовки столбцов);
- `dgIndicator` (для текущей записи в начале строки выводится указатель);
- `dgColumnResize` (пользователь может с помощью мыши изменять размер столбцов и перемещать их);
- `dgColLins` (между столбцами выводятся разделительные вертикальные линии);
- `dgRowLines` (между строками выводятся разделительные горизонтальные линии);
- `dgTabs` (для перемещения по сетке можно использовать клавиши <Tab> и <Shift>+<Tab>);

- dgRowSelect (пользователь может выделить целую строку); при установке этого значения игнорируются значения dgEditing и dgAlwaysShowEditor;
- dgAlwaysShowSelection (ячейка остается выделенной, даже если сетка теряет фокус);
- dgConfirmDelete (при удалении строки выдается запрос на подтверждение операции);
- dgCancelOnExit (добавленные к сетке пустые строки (записи) при потере сеткой фокуса не сохраняются в наборе данных);
- dgMultiSelect (в сетке можно одновременно выделить несколько строк).

По умолчанию свойство Options содержит комбинацию значений [dgEditing, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgConfirmDelete, dgCancelOnExit].

При щелчке на ячейке с данными генерируется событие OnCellClick, а щелчок на заголовке столбца вызывает событие OnTitleClick. Оба события имеют тип TDBGridClickEvent, описываемый так:

```
type TDBGridClickEvent = procedure (Column: TColumn) of object;
```

Параметр Column представляет собой столбец, на котором был произведен щелчок.

При перемещении фокуса между столбцами сетки инициируются события OnColEnter и OnColExit типа TNotifyEvent, первое из которых возникает при получении столбцом фокуса, а второе — при его потере.

Если свойство Options содержит значение dgColumnResize, то пользователь может с помощью мыши перемещать столбцы сетки. При таком перемещении генерируется событие OnColumnMoved типа TMovedEvent, описываемого как:

```
type TMovedEvent = procedure (Sender: TObject; FromIndex,
     ToIndex: Longint) of object;
```

Параметры FromIndex и ToIndex указывают индексы в массиве столбцов сетки, соответствующие предыдущему и новому положению перемещенного столбца соответственно.

Сетка DBGrid способна автоматически отображать в своих ячейках информацию, но при необходимости программист может выполнить и собственное отображение сетки. Это может понадобиться в случае, когда желательно выделить ячейку или столбец с помощью цвета или шрифта, а также вывести в ячейке, кроме текстовой, и графическую информацию, например, небольшой рисунок. Для программной реализации отображения сетки используется обработчик события OnDrawColumnCell типа TDrawColumnCellEvent, которое возникает при прорисовке любой ячейки. Тип события OnDrawColumnCell описан так:

```
type  TDrawColumnCellEvent = procedure (Sender: TObject;
    const Rect: TRect;
  DataCol: Integer; Column: TColumn;
    State: TGridDrawState) of object;
```

Параметр `Rect` содержит координаты ограничивающего ячейку прямоугольника, параметр `DataCol` определяет номер прорисовываемого столбца в массиве столбцов сетки, а параметр `Column` является объектом прорисовываемого столбца. Параметр `State` задает состояние ячейки и принимает комбинации следующих значений:

- `gdSelected` (ячейка находится в выбранном диапазоне);
- `gdFocused` (ячейка имеет фокус ввода);
- `gdFixed` (ячейка находится в фиксированном диапазоне).

Порядок вызова события `OnDrawColumnCell` зависит от значения свойства `DefaultDrawing` типа `Boolean`. Если свойство имеет значение `True` (по умолчанию), то перед генерацией события `OnDrawColumnCell` в ячейке отображается фон и выводится информация. Затем вокруг выбранной ячейки рисуется прямоугольник выбора. Если свойство `DefaultDrawing` имеет значение `False`, то сразу вызывается событие `OnDrawColumnCell`, в обработчике которого следует разместить операции по прорисовке области сетки.

Рассмотрим пример программной прорисовки сетки:

```
// Свойство DefaultDrawing должно быть установлено в значение True
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
   const Rect: TRect;
DataCol: Integer; Column: TColumn; State: TGridDrawState);
var r :TRect;
    s :string;
begin
s := Table1.FieldByName(Column.FieldName).AsString;
r := Rect;
if (Column.FieldName = 'Debt') then begin
 DBGrid1.Canvas.Brush.Color := clRed;
 DBGrid1.Canvas.Font.Color := clYellow;
 DBGrid1.Canvas.Font.Style := [fsItalic];
 DBGrid1.Canvas.FillRect(Rect);
 if Table1.FieldByName(Column.FieldName).AsCurrency > 1000
  then begin
  ImageList1.Draw(DBGrid1.Canvas, Rect.Left + 2, Rect.Top + 2, 2);
  r.Left := r.Left + ImageList1.Width + 4;
  end;
 DBGrid1.Canvas.TextOut(r.Left, r.Top + 2, s);
 end
 else begin
 DBGrid1.Canvas.Brush.Color := clWhite;
 DBGrid1.Canvas.FillRect(Rect);
 DBGrid1.Canvas.Font.Color := clBlack;
```

```
 DBGrid1.Canvas.Font.Style := [];
 DBGrid1.Canvas.TextOut(r.Left, r.Top + 2, s);
 end;
end;
```

Здесь для столбца сетки, который соответствует полю `Debt` (Задолженность) набора данных, устанавливается красный цвет фона, а данные выводятся желтым цветом и курсивом. Кроме того, если задолженность превышает `1000` (рублей или других денежных единиц), то в поле `Debt` соответствующей записи слева от числа выводится рисунок, находящийся в компоненте `ImageList1` (третий по счету). Список с рисунками можно подготовить заранее при разработке приложения и загрузить динамически во время выполнения программы.

При прорисовке ячеек используется свойство `Canvas` элемента `DBGrid1`, а также параметр `Rect`. Если обработчик события `OnDrawColumnCell` является общим для нескольких компонентов `DBGrid`, то при отображении их ячеек вместо названия конкретного компонента (в примере это `DBGrid1`) необходимо ставить конструкцию `(Sender as TDBGrid)` или `TDBGrid(Sender)`. Например, инструкция

```
(Sender as TDBGrid).Canvas.Font.Color := clRed;
```

устанавливает красный цвет для символов ячеек сетки, указываемой параметром `Sender`.

Столбцы сетки

Отдельный столбец `Column` сетки представляет собой объект типа `TColumn`. По умолчанию для каждого поля набора данных, связанного с компонентом `DBGrid`, автоматически создается отдельный столбец, и все столбцы в сетке доступны. Такие столбцы являются *динамическими*. Для создания *статических* столбцов используется специальный Редактор столбцов. Если хотя бы один столбец сетки является статическим, то динамические столбцы уже не создаются ни для одного из оставшихся полей набора данных. Причем в наборе данных доступными являются статические столбцы, а остальные столбцы считаются отсутствующими. Изменить состав статических столбцов можно с помощью Редактора столбцов на этапе разработки приложения.

Взаимодействие между динамическими и статическими столбцами, а также Редактором столбцов аналогично взаимодействию между динамическими и статическими полями набора данных и Редактором полей.

Характеристики и поведение сетки и ее отдельных столбцов во многом определяются полями набора данных (а также соответствующими объектами типа `TField`), для которых создаются объекты типа `TColumn`.

Функционирование динамических столбцов зависит от свойств объекта поля: при изменении свойств объекта типа `TField` соответственно изменяются свой-

ства объекта типа `TColumn`. К примеру, динамический столбец получает от поля такие характеристики, как имя и ширину.

Достоинством статических столбцов является то, что для их объектов можно установить значения свойств, отличные от свойств соответствующего поля и не зависящие от него. Например, если для некоторого статического столбца установить свое имя, то оно не будет меняться даже в случае, если с этим столбцом связывается другое поле набора данных. Кроме того, объекты типа `TColumn` статических столбцов создаются на этапе разработки приложения, и их свойства доступны через Инспектор объектов.

Для запуска Редактора столбцов (рис. 6.3) можно вызвать контекстное меню компонента `DBGrid` и выбрать в нем пункт **Columns Editor**. Редактор столбцов можно вызвать также щелчком мыши на свойстве `Columns` в окне Инспектора объектов.

В заголовке Редактора столбцов выводится составное имя массива столбцов, например, `DBGrid1.Columns`. Под заголовком находится панель инструментов, видимостью которой можно управлять с помощью пункта **Toolbar** контекстного меню Редактора столбцов. Бóльшую часть Редактора столбцов занимает список статических столбцов, при этом столбцы перечисляются в порядке их создания (этот порядок может отличаться от исходного порядка полей в наборе данных).

Замечание

При изменении порядка столбцов сетки автоматически изменяется порядок связанных с ними полей набора данных, что необходимо учитывать при доступе к полям по номерам объектов типа `TField`, а не по именам объектов.

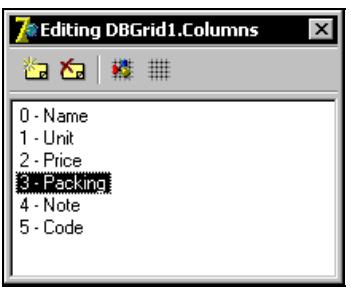


Рис. 6.3. Диалоговое окно Редактора столбцов

Первоначально список статических столбцов пуст, показывая тем самым, что все столбцы сетки являются динамическими. Редактор столбцов позволяет:

- создать статический столбец;
- удалить статический столбец;
- изменить порядок следования статических столбцов.

Кроме того, для любого выбранного в Редакторе статического столбца (объекта типа `TColumn`) через Инспектор объектов можно задать или изменить его свой-

ства и определить обработчики его событий. Это допустимо потому, что соответствующие статическим столбцам объекты типа `TColumn` доступны уже на этапе разработки приложения.

Статический столбец можно создать следующими способами:

- нажатием кнопки панели инструментов Редактора столбцов;
- выбором пункта **Add** контекстного меню Редактора столбцов;
- нажатием клавиши <Insert>.

В любом случае к списку добавляется строка, соответствующая новому статическому столбцу. В левой части строки содержится номер этого столбца в массиве столбцов, в правой — имя поля набора данных, с которым связан столбец. Сразу после добавления к списку столбец не связан ни с одним полем и вместо имени поля указывается `TColumn`. При выполнении приложения подобный столбец окажется пустым. Чтобы связать столбец с каким-либо полем, необходимо установить значение его свойства `FieldName`.

Для добавления в список статических столбцов, соответствующих всем полям набора данных, следует нажать кнопку панели инструментов Редактора столбцов или выбрать в его контекстном меню пункт **Add All Fields**.

Для удаления из списка статических столбцов необходимо их выделить, после чего выполнить одно из следующих действий:

- нажать кнопку панели инструментов Редактора столбцов;
- выбрать пункт **Delete** контекстного меню Редактора столбцов;
- нажать клавишу <Delete>.

Вновь создаваемые статические столбцы получают значения свойств по умолчанию, зависящие также от полей набора данных, с которыми эти столбцы связаны.

Если значения свойств были изменены через Инспектор объектов и требуется восстановить их первоначальные значения, то это можно выполнить нажатием кнопки или выбором пункта **Restore Defaults** (Восстановить параметры по умолчанию) контекстного меню Редактора столбцов.

Объект столбца доступен через свойство `Columns` типа `TDBGridColumns`. При проектировании приложения свойства этого объекта (т. е. столбца, выбранного в списке Редактора столбцов) доступны через Инспектор объектов.

Перечислим наиболее важные свойства объекта столбца.

- `Alignment` типа `TAlignment` — управляет выравниванием значений в ячейках столбца и может принимать следующие значения:
  - `taLeftJustify` (выравнивание по левой границе);
  - `taCenter` (выравнивание по центру);
  - `taRightJustify` (выравнивание по правой границе).
- `Count` типа `Integer` — указывает число столбцов сетки.

- `Field` типа `TField` — определяет объект поля набора данных, связанный со столбцом.
- `FieldName` типа `String` — указывает имя поля набора данных, с которым связан столбец. При установке этого свойства с помощью Инспектора объектов значение можно выбирать в списке.
- `PickList` типа `TStrings` — представляет собой список для выбора заносимых в поле значений. Текущая ячейка совместно со списком `PickList` образуют своего рода компонент `ComboBox` или `DBComboBox`. Если для столбца сформирован список выбора, то при попытке редактирования ячейки этого столбца справа появляется стрелка, при нажатии которой список раскрывается и позволяет выбрать одно из значений. При этом можно ввести в ячейку любое допустимое значение.
- `Title` типа `TColumnTitle` — представляет собой объект заголовка столбца. В свою очередь этот объект имеет такие свойства, как `Caption`, `Alignment`, `Color` и `Font`, определяющие название, выравнивание, цвет и шрифт заголовка соответственно.

Свойства столбца, управляющие форматированием, видимостью или возможностью модификации значений, не отличаются от соответствующих свойств поля.

Рассмотрим пример по установке свойств для столбцов сетки.

В наборе данных определено пять полей, для каждого из которых с помощью Редактора столбцов создан статический столбец. Текст процедуры, выполняемой при создании формы приложения, имеет следующий вид:

```
// Значения этих свойств можно установить также через Инспектор объектов
procedure TForm1.FormCreate(Sender: TObject);
begin
// Скрытие первого столбца
DBGrid1.Columns[0].Visible := False;
// Установка параметров второго столбца
DBGrid1.Columns[1].Title.Caption := 'Дата поступления';
DBGrid1.Columns[1].Title.Alignment := taCenter;
DBGrid1.Columns[1].Alignment := taCenter;
// Скрытие третьего столбца
DBGrid1.Columns[2].Visible := False;
// Установка параметров четвертого столбца
DBGrid1.Columns[3].Title.Caption := 'Количество';
DBGrid1.Columns[3].Title.Alignment := taCenter;
DBGrid1.Columns[3].Alignment := taRightJustify;
// Устан*овка параметров пятого столбца
DBGrid1.Columns[4].Title.Caption := 'Примечание';
DBGrid1.Columns[4].Title.Alignment := taCenter;
DBGrid1.Columns[4].Title.Alignment := taLeftJustify;
```

```
DBGrid1.Columns[4].PickList.Clear;
DBGrid1.Columns[4].PickList.Add('Товар на складе');
DBGrid1.Columns[4].PickList.Add('Некондиция');
DBGrid1.Columns[4].PickList.Add('Срок реализации не лимитирован');
end;
```

Первый и третий столбцы устанавливаются невидимыми, для остальных столбцов задаются название заголовка и его выравнивание, а также выравнивание значений. Кроме того, для пятого столбца, соответствующего полю примечания, создан список выбора. Установка свойств столбцов произведена при создании формы, эти же действия могут быть проделаны и через Инспектор объектов. При выполнении приложения форма имеет вид, показанный на рис. 6.4.

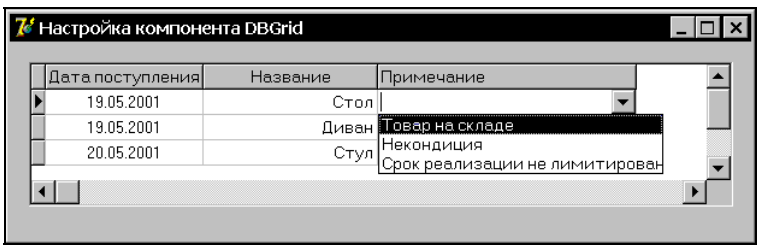


Рис. 6.4. Установка свойств для столбцов сетки

В дальнейшем при использовании компонента `DBGrid` свойства его столбцов изменяться не будут, при этом состав и порядок следования столбцов сетки будут соответствовать составу и порядку следования полей набора данных, а в качестве заголовков столбцов сетки будут отображаться названия полей набора данных.

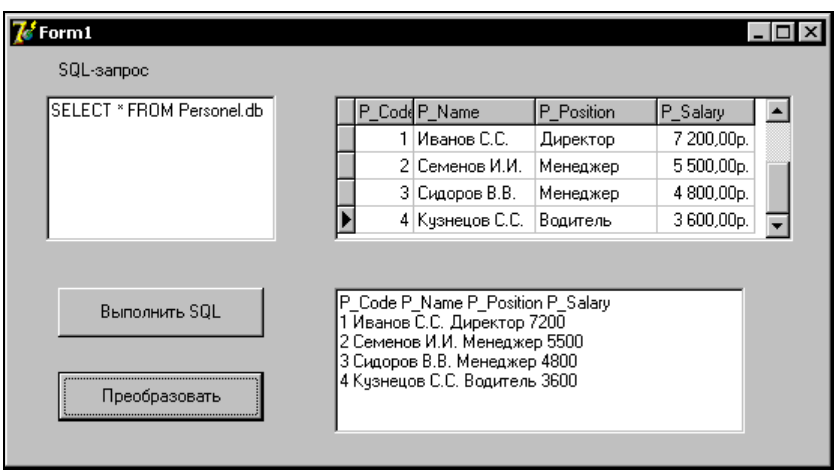


Рис. 6.5. Преобразование значений записей набора данных в текст

Рассмотрим еще один пример, в котором осуществляется преобразование значений записей набора данных в текст.

В качестве набора данных используется компонент `Query1`, SQL-запрос для которого вводится в многострочное поле редактирования `Memo1`. Выполнение запроса происходит при нажатии кнопки **Выполнить SQL** (`Button1`). Полученные в результате выполнения запроса записи отображаются в сетке `DBGrid1`. При нажатии кнопки **Преобразовать** (`Button2`) происходит последовательный просмотр полей всех записей набора данных (сетки) и преобразование их в текст, который помещается в многострочное поле редактирования `Memo2` (рис. 6.5).

Ниже приведен код обработчиков событий нажатия кнопок.

```
// Выполнение SQL-запроса
procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Close;
Query1.SQL.Assign(Memo1.Lines);
Query1.Open;
   end;

// Преобразование значений записей набора данных в текст
procedure TForm1.Button2Click(Sender: TObject);
var c, n   :integer;
    s, rs  :string;
begin
Memo2.Clear;
Query1.First;
// Перебор всех записей набора данных
for n := 1 to Query1.RecordCount do begin
    rs := ''; s := '';
    // Чтение названий столбцов сетки
    if n = 1 then begin
    for c := 0 to DBGrid1.Columns.Count - 1 do begin
    s := DBGrid1.Columns[c].FieldName + ' ';
    rs := rs + s;
    end;
Memo2.Lines.Add(rs);
rs := ''; s := '';
end;
// Чтение значений полей текущей записи
for c := 0 to DBGrid1.Columns.Count - 1 do begin
    s := DBGrid1.Columns[c].Field.AsString + ' ';
    rs := rs + s;
    end;
```

```
Memo2.Lines.Add(rs);
Query1.Next;
end;
end;
```

Для доступа к названиям и значениям полей набора данных использованы свойства `FieldName`, `Count` и `Field` столбцов сетки. При необходимости текст из поля редактирования `Memo2` можно скопировать в буфер.

Использование модифицированной сетки

Кроме компонента `DBGrid`, для управления записями таблицы предназначен компонент `DBCtrlGrid` — модифицированная сетка. Компонент `DBCtrlGrid` представляет собой несколько отдельных панелей, на которых располагаются визуальные компоненты (рис. 6.6).

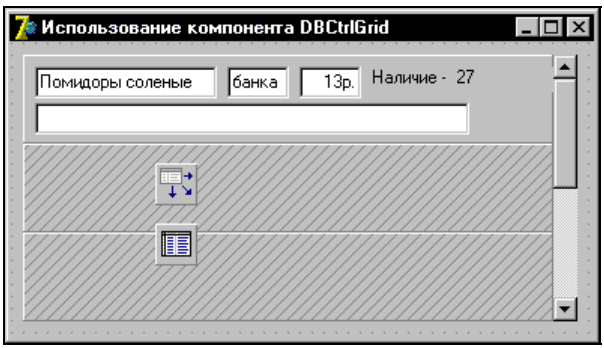


Рис. 6.6. Вид компонента `DBCtrlGrid` при проектировании приложения

Модифицированная сетка `DBCtrlGrid` может отображать несколько одинаковых панелей, но текущей является только одна из них. При проектировании приложения визуальные компоненты, например, `DBEdit` или `DBtext`, располагаются на одной (верхней) панели. Когда визуальные компоненты помещаются на панель модифицированной сетки, у них автоматически устанавливается нужное значение свойства `DataSource`, взятое из аналогичного свойства модифицированной сетки. При выполнении приложения компоненты, размещенные на одной панели, дублируются на другие панели сетки. На приведенной на рис. 6.7 сетке расположены четыре компонента `DBEdit` — для названия, единицы измерения, цены товара и примечания, компонент `DBtext` — для количества товара на складе и надпись `Label` с текстом `Наличие -`.

Число панелей, одновременно видимых в модифицированной сетке, определяется свойством `PanelCount` типа `Integer`, доступным для чтения во время выполнения

приложения. Свойство `PanelIndex` типа `Integer` указывает текущую панель, на которой находится просматриваемая запись набора данных. Установив это свойство в соответствующее значение, можно сделать текущей нужную панель.

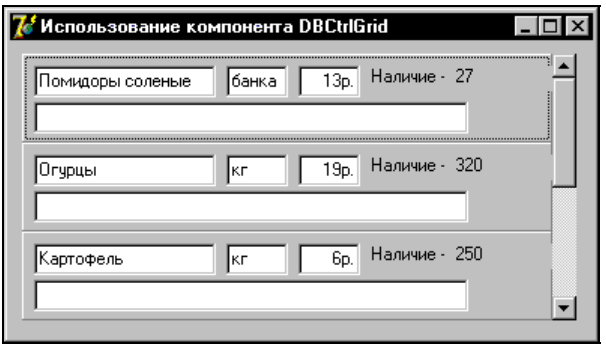


Рис. 6.7. Вид компонента `DBCtrlGrid` при выполнении приложения

Все панели сетки имеют одинаковые *размеры*, определяемые свойствами `PanelHeight` (высота) и `PanelWidth` (ширина) типа `Integer`. Каждая панель может иметь *рамку*, что определяется свойством `PanelBorder` типа `TDBCtrlGridBorder`, принимающим следующие значения:

- `gbNone` (рамки нет);
- `gbRaised` (трехмерная приподнятая рамка) — по умолчанию.

Число одновременно *видимых строк* и *столбцов* сетки задают свойства `RowCount` и `ColCount` типа `Integer`, значения которых по умолчанию равны трем и одному. Это соответствует трем панелям в одном столбце и наличию вертикальной полосы прокрутки. При задании более одного столбца у сетки появляется горизонтальная полоса прокрутки (рис. 6.8).

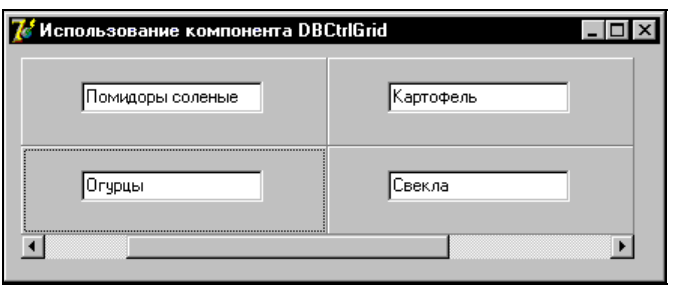


Рис. 6.8. Вид сетки с панелями в две строки и два столбца

Свойство `Orientation` типа `TDBCtrlGridOrientation` при наличии нескольких столбцов определяет *порядок размещения записей* на панелях. Это свойство принимает следующие значения:

- `goVertical` (записи выводятся по горизонтали: слева направо и сверху вниз) — по умолчанию;
- `goHorizontal` (записи выводятся по вертикали: сверху вниз и слева направо).

На панели может отображаться *прямоугольник фокуса* (прямоугольная рамка), указывающий на текущую запись. Его отображением управляет свойство `ShowFocus` типа `Boolean`, которое по умолчанию имеет значению `True`, что соответствует отображению прямоугольника фокуса. Значение `False` свойства `ShowFocus` скрывает прямоугольник фокуса.

Свойства `AllowDelete` и `AllowInsert` типа `Boolean` управляют возможностями *удаления* текущей и *вставки* новой записи в набор данных, записи которого отображаются на панелях сетки. По умолчанию оба свойства имеют значение `True`, и пользователь имеет возможность удалять и вставлять записи.

Свойство `EditMode` типа `Boolean` определяет, находится ли набор данных в *режиме редактирования* (значение `True`). Пользователь управляет его значением с помощью действий с визуальными компонентами, разработчик может устанавливать его программно.

С помощью мыши и клавиатуры пользователь управляет записями набора данных посредством компонента `DBCtrlGrid` так же, как и в случае компонента `DBGrid`.

Программисту предоставляются дополнительные возможности управления набором данных, связанные с использованием метода `DoKey`. Процедура `DoKey (Key: TDBCtrlGridKey)` выполняет различные операции, которые задает параметр `Key`, принимающий следующие значения:

- `gkNull` (пустое действие);
- `gkEditMode` (переключение значения свойства `EditMode`);
- `gkPriorTab` (передача фокуса ввода следующему элементу формы);
- `gkNextTab` (передача фокуса ввода предыдущему элементу формы);
- `gkLeft` (переход на один столбец влево);
- `gkRight` (переход на один столбец вправо);
- `gkUp` (переход на одну панель вверх);
- `gkDown` (переход на одну панель вниз);
- `gkScrollUp` (переход на одну строку вверх);
- `gkScrollDown` (переход на одну строку вниз);
- `gkPageUp` (переход вперед на число записей, равное произведению числа строк и столбцов панелей сетки `ColCount * RowCount`);

- `gkPageDown` (переход назад на число записей, равное произведению числа строк и столбцов панелей сетки);
- `gkHome` (переход на первую запись);
- `gkEnd` (переход на последнюю запись);
- `gkInsert` (вставка новой записи методом `Insert` и переход в режим редактирования);
- `gkAppend` (вставка новой записи методом `Append` и переход в режим редактирования);
- `gkDelete` (удаление текущей записи);
- `gkCancel` (отмена режима редактирования).

При необходимости разработчик может выполнить *программную прорисовку* панелей, использовав для этого событие `OnPaintPanel` типа `TPaintPanelEvent`, возникающее непосредственно перед отображением панелей. Тип этого события описан как

```
type TPaintPanelEvent = procedure (DBCtrlGrid: TDBCtrlGrid;
     Index: Integer) of object;
```

Параметр `Index` указывает номер панели, отображение элементов которой выполняется в настоящий момент. Кроме прорисовки, в обработчике события `OnPaintPanel` можно выполнить другие действия, например, обработку связанных с панелью данных.

Использование навигационного интерфейса

Для управления набором данных можно использовать навигатор, который обеспечивает соответствующий интерфейс пользователя. По внешнему виду и организации работы навигатор похож на мультимедийный проигрыватель. В Delphi навигатор представлен компонентом `DBNavigator` (рис. 6.9).

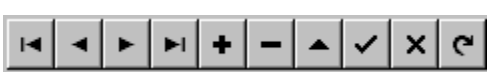


Рис. 6.9. Навигатор

Навигатор содержит кнопки, обеспечивающие выполнение различных операций с набором данных путем автоматического вызова соответствующих методов. *Состав видимых кнопок* определяет свойство `VisibleButtons` типа `TButtonSet`, принимающее комбинации следующих значений (в скобках указан вызываемый метод):

- `nbFirst` — перейти к первой записи (`First`);
- `nbPrior` — перейти к предыдущей записи (`Prior`);
- `nbNext` — перейти к следующей записи (`Next`);

- nbLast — перейти к последней записи (Last);
- nbInsert — вставить новую запись (Insert);
- nbDelete — удалить текущую запись (Delete);
- nbEdit — редактировать текущую запись (Edit);
- nbPost — утвердить результат изменения записи (Post);
- nbCancel — отменить изменения в текущей записи (Cancel);
- nbRefresh — обновить информацию в наборе данных (Refresh).

По умолчанию в навигаторе видимы все кнопки.

Метод BtnClick(Index: TNavigateBtn) служит для имитации нажатия кнопки, заданной параметром Index. Тип TNavigateBtn этого параметра идентичен типу TButtonSet, возможные значения соответствующего параметра которого перечислены выше. В качестве примера приведем строку кода:

```
DBNavigator.BtnClick(nbNext);
```

В ней имитируется нажатие кнопки nbNext, вызывающей переход к следующей записи набора данных.

При нажатии кнопки nbDelete может появляться диалоговое окно, в котором пользователь должен подтвердить или отменить удаление текущей записи. Появлением *окна подтверждения* управляет свойство ConfirmDelete типа Boolean, по умолчанию имеющее значение True, т. е. окно подтверждения выводится. Если при отладке приложения установить это свойство в значение False, то запись будет удаляться без запроса подтверждения.

Свойство Flat типа Boolean управляет *внешним видом* кнопок. По умолчанию оно имеет значение False, и кнопки отображаются в объемном виде. При установке свойства Flat в значение True кнопки приобретают плоский вид, соответствующий современному стилю.

Подсказку для отдельной кнопки можно установить с помощью свойства Hints типа TString. По умолчанию список подсказок содержит текст на английском языке, который можно заменить на русский, вызвав Строковый редактор (String list editor). Подсказка для навигатора устанавливается через свойство Hint типа String. Напомним, что для отображения подсказок нужно присвоить значение True свойству ShowHint, по умолчанию имеющему значение False.

На практике часто вместо навигатора используются отдельные кнопки Button или BitBtn, при нажатии которых вызываются соответствующие методы управления набором данных. Например, в процедурах:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.Next;
   end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Table1.Prior;
end;
```

при нажатии кнопок `Button1` и `Button2` выполняется переход к следующей и предыдущей записям набора данных `Table1` соответственно.

Вывод графических изображений

Компонент `DBImage` (графическое изображение) предназначен для вывода изображений, содержащихся в графических полях БД. Если компонент `DBImage` связать с полем, не содержащим изображение, например, с числовым, то в области компонента выводится имя этого поля. В случае, когда компонент `DBImage` не связан ни с одним полем, он отображает свое собственное имя (значение свойства `Name`).

Кроме вывода изображения, компонент также позволяет изменить (заменить) его, вставив нужный рисунок из буфера обмена Windows.

Основные свойства графического изображения мы рассматривать не будем, а рассмотрим только дополнительные характеристики компонента `DBImage`.

Свойство `AutoDisplay` типа `Boolean` указывает, каким способом в компоненте `DBImage` отображаются *изменения* в связанном с ним поле. По умолчанию свойство имеет значение `True`, и содержимое графического поля отображается. Если свойство `AutoDisplay` установить в значение `False`, то при изменении значения поля, например, из-за перехода к другой записи, вместо изображения выводится имя поля. В этом случае для вывода графики нужно выполнить двойной щелчок на графическом компоненте или нажать клавишу <Enter>, когда на нем находится фокус ввода. Можно также вывести содержимое поля программно с помощью метода `LoadPicture`. Например:

```
DBImage1.AutoDisplay := False;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
DBImage1.LoadPicture;
end;
```

В данной процедуре при нажатии кнопки `Button1` в графическом компоненте `DBImage1` выводится содержимое графического поля текущей записи.

Компонент `DBImage` поддерживает работу с *буфером обмена* Windows, позволяя копировать изображение в буфер и вставлять изображение из буфера. Для выполнения этих действий используются обычные для Windows-программ комбинации клавиш: копирование в буфер — <Ctrl>+<Insert>, удаление в буфер — <Shift>+<Delete>, вставка из буфера — <Shift>+<Insert>.

Указанные действия также могут быть выполнены программно. Метод `CopyToClipboard` копирует изображение в буфер обмена, метод `CutToClipboard` вырезает (перемещает) изображение в буфер обмена, а метод `PasteFromClipboard` вставляет изображение из буфера обмена.

При использовании любого способа вставки из буфера новое изображение автоматически заменяет предыдущее содержимое компонента `DBImage`.

Рассмотрим в качестве примера программу для работы с фотоальбомом.

Пусть информация о фотографиях хранится в таблице Paradox, представляющей собой "электронный фотоальбом" и включающей следующие поля:

- код (автоинкрементное поле);
- название (символьное поле);
- дата (поле даты);
- изображение (графическое поле);
- описание (Memo-поле).

Для просмотра и редактирования альбома создано приложение, форма которого показана на рис. 6.10. Содержимое альбома выводится в соответствующих компонентах. В верхней части расположены поля редактирования `DBEdit`, одно из которых содержит название, а другое — дату фотографии. Снимок выводится в графическом компоненте `DBImage1`, а описание снимка — в многострочном редакторе `DBMemo1`. Для навигации по альбому используется навигатор `DBNavigator1`, расположенный в нижней части формы. Флажок `CheckBox1` управляет масштабированием изображения по размеру компонента `DBImage1`. По умолчанию этот флажок снят, и изображение не подстраивается под размеры графического компонента.

Название, дата и описание фотографии редактируются обычным способом с помощью соответствующих компонентов `DBEdit` и `DBMemo`. Для редактирования снимка имеются кнопки, позволяющие вставлять изображение из файла и сохранять его в файле, а также обмениваться изображениями через буфер.

Нажатие кнопки **Открыть** вызывает появление диалога `OpenPictureDialog1` выбора файла для открытия. После выбора нужного файла содержащееся в нем изображение загружается в компонент `DBImage1`. Фильтр диалога настроен на выбор графических файлов формата BMP, а также форматов ICO, EMF и WMF, которые при загрузке автоматически преобразуются в формат BMP.

Нажатие кнопки **Сохранить** открывает диалог `SavePictureDialog1` выбора файла для сохранения. После выбора файла в него записывается изображение из компонента `DBImage1`. Фильтр диалога настроен на выбор графического файла формата BMP. Настройка фильтров обоих диалогов произведена при создании формы.

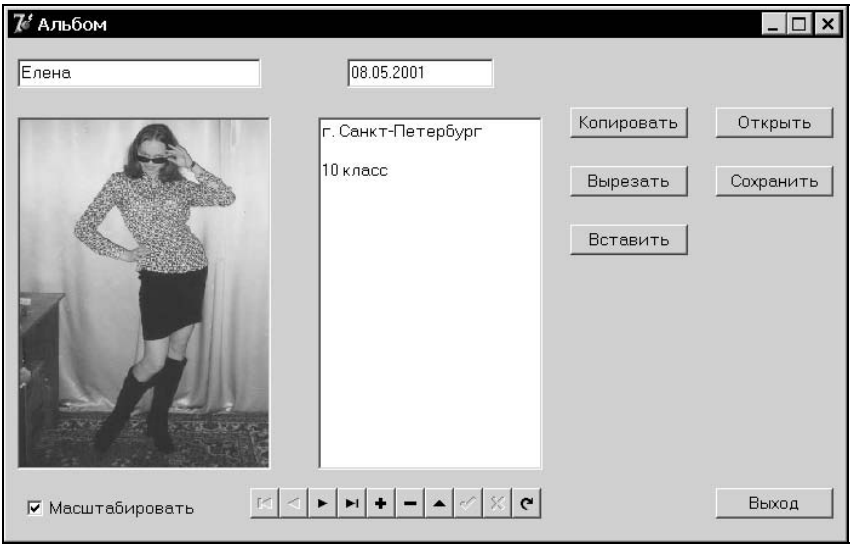


Рис. 6.10. Просмотр и редактирование электронного фотоальбома

Кнопки с названиями **Копировать**, **Вырезать** и **Вставить** выполняют соответствующий обмен между графическим компонентом `DBImage1` и буфером. Кнопка **Закрыть** прекращает работу приложения.

Ниже приведен код модуля `uAlbum` формы `Form1` приложения.

```
   unit uAlbum;

   interface

uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
   StdCtrls, DBCtrls, ExtCtrls, Db, DBTables, Grids, DBGrids, ExtDlgs,
   Mask;

type
TForm1 = class(TForm)
       DataSource1: TDataSource;
            Table1: TTable;
      DBNavigator1: TDBNavigator;
           Button1: TButton;
OpenPictureDialog1: TOpenPictureDialog;
SavePictureDialog1: TSavePictureDialog;
           Button2: TButton;
           Button3: TButton;
           Button4: TButton;
```

```
          Button5: TButton;
          Button6: TButton;
          DBMemo1: TDBMemo;
         DBImage1: TDBImage;
        CheckBox1: TCheckBox;
          DBEdit1: TDBEdit;
          DBEdit2: TDBEdit;

procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button4Click(Sender: TObject);
procedure Button5Click(Sender: TObject);
procedure Button6Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure CheckBox1Click(Sender: TObject);

private
{ Private declarations }
public
{ Public declarations }


   end;

var
   Form1: TForm1;

   implementation

   {$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
OpenPictureDialog1.Filter :=
   'Все файлы (*.bmp;*.ico;*.emf;*.wmf)|*.bmp;*.ico;*.emf;*.wmf';
SavePictureDialog1.Filter := '*.bmp|*.bmp';
   end;

procedure TForm1.Button1Click(Sender: TObject);
begin
if OpenPictureDialog1.Execute then
 DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
   end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
if SavePictureDialog1.Execute then
 DBImage1.Picture.LoadFromFile(SavePictureDialog1.FileName);
   end;

procedure TForm1.Button3Click(Sender: TObject);
begin
DBImage1.CopyToClipboard;
   end;

procedure TForm1.Button4Click(Sender: TObject);
begin
DBImage1.CutToClipboard;
   end;

procedure TForm1.Button5Click(Sender: TObject);
begin
DBImage1.PasteFromClipboard;
   end;

procedure TForm1.Button6Click(Sender: TObject);
begin
Form1.Close;
   end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
DBImage1.Stretch := CheckBox1.Checked;
   end;

end.
```

Отметим, что в отличие от компонента `Image`, у графического компонента `DBImage` нет свойства `Canvas`, позволяющего рисовать на поверхности изображения.

Построение диаграмм

Для построения диаграмм на основании информации, содержащейся в наборе данных, предназначен компонент-диаграмма `DBChart`. Он позволяет выводить диаграммы различных типов, в том числе объемные. Этот компонент является достаточно сложным и имеет большое количество разнообразных свойств, многие из которых являются объектами и также имеют свои свойства. На практике

установка значений этих свойств выполняется при разработке приложения с помощью Редактора диаграмм, окно которого **Editing DBChart1** показано на рис. 6.11. Редактор позволяет оперировать со свойствами-объектами, информация о которых отображается на его страницах, и вызывается двойным щелчком на компоненте `DBChart` или через поле значения свойства-объекта в Инспекторе объектов (в этом случае активной становится страница, соответствующая выбранному свойству).

Важнейшим свойством компонента `DBChart` является свойство `Series [Index:Longint]` типа `TChartSeries`, представляющее собой массив диаграмм, выводимых в области компонента. (Часто компонент `DBChart` содержит только одну диаграмму.)

Для каждой диаграммы можно установить:

- тип;
- название;
- оси;
- описание;
- источник данных

и другие параметры.

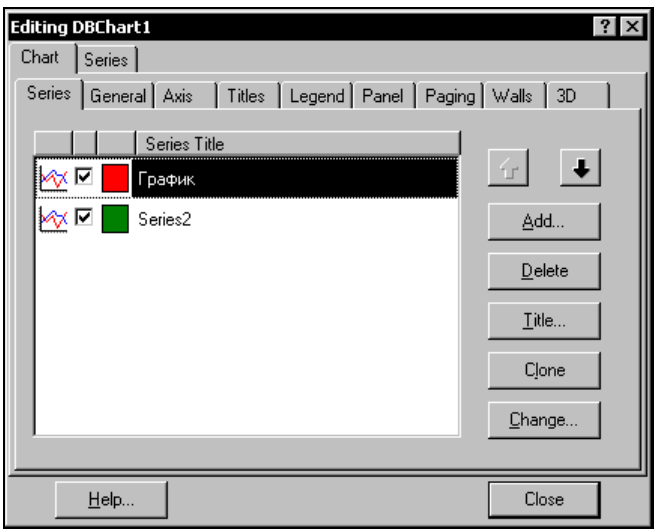


Рис. 6.11. Редактор диаграмм

Разработчик должен как минимум указать тип диаграммы и источник данных. Тип выбранной диаграммы (диаграмм) и ее название отображаются на странице **Chart-Series** Редактора диаграмм (рис. 6.11). Для добавления новой диаграммы нужно нажать кнопку **Add**, в результате чего появляется окно, показанное на рис. 6.12. После выбора типа диаграммы, объемного или плоского варианта ее построения и нажатия кнопки **OK** диаграмма добавляется к значению свойства `Series` и отображается на соответствующей странице Редактора диаграмм.

Для выбранной диаграммы можно выполнить следующие действия:

- изменить имя по умолчанию (`Series1`, `Series2` и т. д.) — кнопка **Title**;
- изменить тип диаграммы — кнопка **Change**;
- скопировать диаграмму — кнопка **Clone**;
- удалить диаграмму — кнопка **Delete**.

Источник данных выбирается на странице **Series-DataSource** из следующих вариантов:

- **No Data** — значения, вводимые программно;
- **Random Values** — случайные числа;
- **Function** — значения, определяемые выбранной функцией;
- **DataSet** — значения набора данных; компонент `DBChart` отличается от компонента `Chart` именно тем, что для него в качестве источника данных можно использовать *набор данных*, т. е. `DBChart` является более универсальным компонентом.

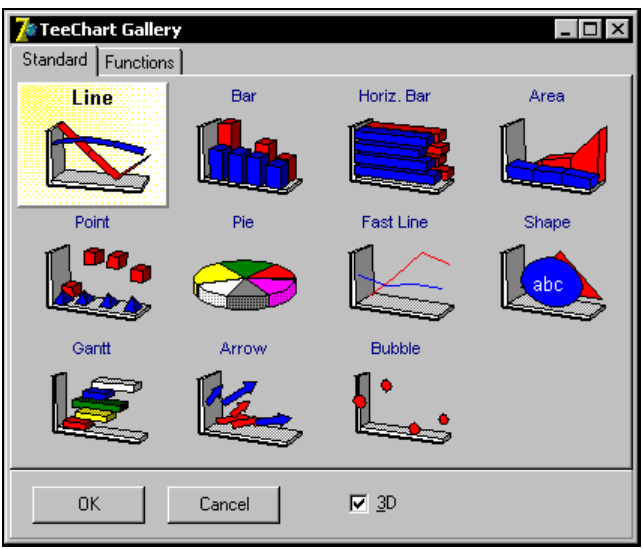


Рис. 6.12. Выбор типа диаграммы

Если выбран программный способ (вариант **No Data**) ввода значений, то при выполнении приложения нужно вызывать соответствующие методы. Для *управления значениями*, по которым строится диаграмма, часто используются методы `Add`, `Delete` или `Clear`.

Функция `Add (Const AValue: Double; Const ALabel: String; AColor: TColor): Longint` добавляет к диаграмме значение, указанное параметром

AValue. Параметры ALabel и AColor содержат соответственно надпись значения и цвет, используемый при выводе. В качестве результата функция возвращает номер значения в массиве значений диаграммы. Кроме Add, есть несколько других методов, также позволяющих добавлять значения.

Процедура Delete (ValueIndex : LongInt) удаляет значение с номером, указанным параметром ValueIndex. Для удаления всех значений удобно использовать процедуру Clear.

В качестве примера рассмотрим, как осуществляется вывод графика.

Тип диаграммы (график) задан при разработке приложения. Вид графика при выполнении приложения показан на рис. 6.13. Используемые при его построении значения вводятся построчно в редакторе Memo1.

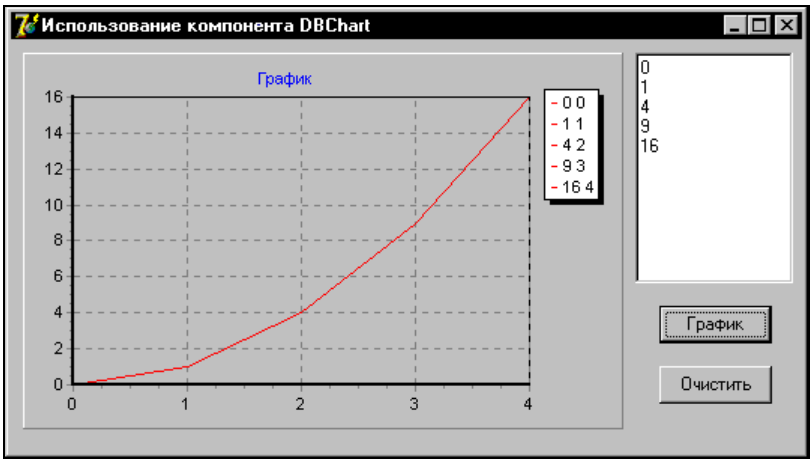


Рис. 6.13. Вывод графика

Ниже приведены обработчики событий для кнопок формы приложения.

```
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
DBChart1.Title.Text.Add('График');
for n := 0 to Memo1.Lines.Count - 1 do
    DBChart1.Series[0].Add(StrToFloat(Memo1.Lines[n]), IntToStr(n), clRed);
   end;

procedure TForm1.Button2Click(Sender: TObject);
begin
DBChart1.Title.Text.Clear;
DBChart1.Series[0].Clear;
end;
```

При нажатии кнопки **График** (`Button1`) значения из редактора `Memo1` заносятся в компонент `DBChart1`, отображающий график. Удаляется график нажатием кнопки **Очистить** (`Button2`).

Задание случайных чисел (вариант **Random Values**) в качестве источника данных для диаграммы бывает полезным при предварительной настройке диаграммы, например, при выборе ее типа или размера.

Использование функции (вариант **Function**) в качестве источника данных для диаграммы заключается в том, что диаграмма строится на основании обработки значений, взятых из двух и более других диаграмм (серий). В качестве функций обработки можно использовать **Copy**, **Average**, **Low**, **High**, **Divide**, **Multiply**, **Subtract** и **Add**. Например, если указать функцию **Subtract** (Вычитание) и две диаграммы-источника `Series1` и `Series2`, то каждое значение нашей диаграммы будет вычисляться как разность между соответствующими значениями диаграмм `Series1` и `Series2`.

При задании набора данных (вариант **Dataset**) в качестве источника данных для диаграммы (рис. 6.14) становится видимой панель для ввода информации о наборе данных.

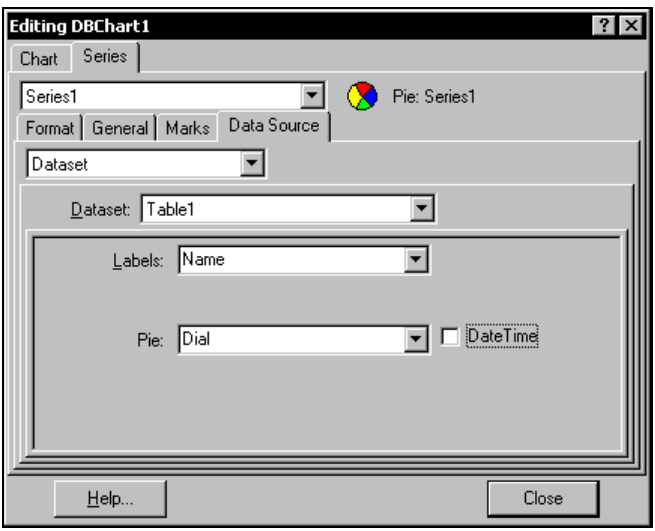


Рис. 6.14. Задание источника данных для диаграммы

В списке **Dataset** содержатся имена наборов данных, доступных в модуле той формы, в которой расположен компонент `DBChart`. В приведенном на рис. 6.14 примере — это набор данных `Table1`. В списке **Labels** выбирается имя поля (в примере `Name`), данные из которого используются в качестве надписей для обо-

значения секторов диаграммы, а в списке **Pie** — имя поля (в примере `Dial`), из которого выбираются данные для построения секторов диаграммы.

После закрытия окна Редактора диаграмма автоматически строится системой Delphi на основании записей, составляющих набор данных (рис. 6.15).

При выполнении приложения диаграмма выглядит так же, как и при проектировании. При этом ее функционирование является динамическим, т. е. при изменении данных, содержащихся в наборе, диаграмма изменяется автоматически.

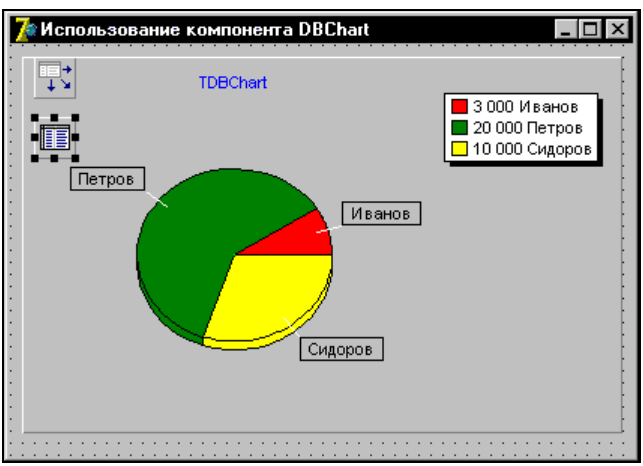


Рис. 6.15. Диаграмма, построенная на основании значений набора данных

Глава 7

Навигационный доступ к данным с помощью BDE

Напомним, что навигационный способ доступа заключается в обработке каждой отдельной записи набора данных. Достоинством этого способа является простота кодирования операций, а основной недостаток состоит в том, что приложение получает все записи набора независимо от того, сколько их требуется обработать на самом деле. Это приводит к большой нагрузке на сеть, особенно при интенсивном обмене данными. Поэтому применение навигационного способа доступа обычно ограничивается локальными БД.

В данной главе мы познакомимся с основными операциями, используемыми в локальных БД. Эти операции также могут быть применены и при организации работы с удаленными БД, если сеть имеет небольшое число пользователей. Операции с таблицами будут рассмотрены по отношению к наборам данных `Table` и `Query`, используемым при механизме BDE. Наряду с навигационным, язык структурированных запросов SQL и набор данных `Query` позволяют реализовать для локальных БД и реляционный доступ к данным, который подробно будет изложен в *главе 8*.

Если нет каких-либо специальных ограничений, то при работе с локальными БД более предпочтительным представляется набор данных `Table`, т. к. он работает несколько быстрее, чем `Query`. Для доступа к удаленным БД, наоборот, лучше использовать набор данных `Query`, т. к. с помощью SQL-запроса он позволяет реализовать реляционный доступ к данным, уменьшающий число передаваемых по сети записей.

При навигационном способе доступа операции выполняются с отдельными записями. Каждый набор данных имеет указатель текущей записи, т. е. записи, с полями которой могут быть выполнены такие операции, как редактирование или удаление. Компоненты `Table` и `Query` позволяют управлять положением этого указателя.

Навигационный способ доступа дает возможность осуществлять следующие операции:

- сортировку записей;
- навигацию по набору данных;

- редактирование записей;
- вставку и удаление записей;
- фильтрацию записей.

Отметим, что аналогичные операции применимы к набору данных и при реляционном способе доступа, реализуемом с помощью SQL-запроса.

Операции с таблицей БД

Кроме действий с отдельными записями, с помощью компонента `Table` можно выполнять также действия с таблицей БД в целом, например, создавать, удалять, переименовывать таблицы или устанавливать режимы доступа к ним.

Создание, удаление и переименование

Обычно таблицы создаются на этапе разработки приложения с помощью соответствующих инструментальных программ типа Database Desktop. Как правило, удаление таблицы также выполняется при разработке приложения, например, с помощью Проводника Windows. Использование инструментов позволяет достаточно удобно создавать таблицы, удалять их и изменять их структуру. Кроме того, программист может создать или удалить таблицу динамически, т. е. в процессе выполнения приложения. Такая потребность может возникнуть, например, при необходимости получить резервную или архивную копию всей таблицы или ее части.

Для *создания таблицы* используется метод `CreateTable`. В результате на диске появляется пустая таблица. Перед вызовом метода нужно подготовить необходимые данные, на основе которых создается таблица. Эти данные следует присвоить в качестве значений соответствующим свойствам набора данных. Перед вызовом метода `CreateTable` набор данных должен быть закрыт и установлены значения следующих свойств:

- `DatabaseName` (путь к файлам базы данных (каталог));
- `TableType` (тип таблицы);
- `FieldDefs` (описание полей);
- `IndexDefs` (описание индексов);
- `TableName` (название таблицы).

Свойство `TableName` задает имя физического файла таблицы, находящегося в каталоге, указанном для размещения БД (свойство `DatabaseName`).

Формат таблицы `TableName` типа `TTableName` может быть следующим:

- `ttDefault` (формат таблицы) — по умолчанию, определяется на основании расширений имен файлов таблиц:
  - db или отсутствует — таблица Paradox;

- dbf — таблица dBase;
- txt — таблица ASCII;

- ttParadox (таблица Paradox);
- ttDBase (таблица dBase);
- ttFoxPro (таблица FoxPro);
- ttASCII (таблица ASCII — текстовый файл, разбитый на столбцы).

Для новой таблицы в свойстве FieldDefs типа TFieldDefs обязательно должно быть определено хотя бы одно поле. Перед тем как приступить к описанию полей новой таблицы, значение этого свойства следует очистить, т. к. в нем может находиться информация о полях предыдущей таблицы, с которой был связан набор данных. Для *очистки значения* свойства FieldDefs можно применить метод Clear, а для *занесения информации о полях* новой таблицы — метод Add.

Процедура Add (const Name: String; DataType: TFieldType; Size: Word; Required: Boolean) добавляет к массиву полей описание нового *поля*. Параметр Name указывает название, а параметр DataType — тип поля, который можно выбрать в следующем списке: ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet. В перечисленных значениях префикс ft является сокращением от Field Type — тип поля, а последующая часть определяет собственно тип.

Параметр Size определяет размер поля; если для полей некоторых типов, например, поля даты (ftDate), размер не задается, то параметр Size принимает значение ноль. Логический параметр Required определяет, должно ли поле обязательно содержать значение (True) или может быть пустым (False).

Замечание

Если в наборе данных определены статические поля, то при вызове метода CreateTable, скорее всего, возникнет исключение. Это происходит из-за того, что в новой таблице задан новый состав полей и отсутствуют физические поля, с которыми были связаны созданные ранее статические поля.

В таблице можно определить *индексы*. Описание индексов заносится в свойство IndexDefs типа TIndexDefs (мы рассматривали его при описании набора данных Table) с помощью метода Add. Отметим, что после создания таблицы в ней можно удалить или создать индекс методами AddIndex и DeleteIndex.

Замечание

После задания новых индексов следует установить нужные значения для свойств IndexName и IndexFieldName, т. к. они могут указывать на индекс предыдущей таблицы, с которой был связан набор данных.

Для компонента `Query` действия, связанные с созданием таблицы, выполняются через запрос SQL.

Рассмотрим следующий пример:

```
procedure TForm1.btnNewTableClick(Sender: TObject);
begin
// Закрытие набора данных
Table1.Active := False;
// Параметры таблицы БД
Table1.DatabaseName := 'BDPlace';
Table1.TableName := 'NewTable';
Table1.TableType := ttParadox;
// Описание полей таблицы
Table1.FieldDefs.Clear;
Table1.FieldDefs.Add('Code', ftAutoinc, 0, True);
Table1.FieldDefs.Add('Name', ftString, 20, True);
Table1.FieldDefs.Add('Date', ftDate, 0, False);
// Описание индексов таблицы
Table1.IndexDefs.Clear;
Table1.IndexDefs.Add('', 'Code', [ixPrimary, ixUnique]);
Table1.IndexDefs.Add('indName', 'Name', [ixCaseInsensitive]);
// Создание таблицы
Table1.CreateTable;
// Установка текущего индекса
Table1.IndexName := 'indName';
// Открытие набора данных, связанного с новой (пустой) таблицей
Table1.Active := True;
end;
```

Здесь в каталоге, указываемом псевдонимом `BDPlace`, создается новая таблица Paradox с именем `NewTable.db`. Для таблицы заданы три поля: код (номер), имя и дата. Поля кода и имени требуют обязательного заполнения при вводе или модификации записей созданной таблицы. По автоинкрементному полю номера построен главный ключ (без имени), а по полю имени — вторичный ключ, который после создания таблицы устанавливается как текущий.

Для *удаления* таблицы используется метод `DeleteTable`, в результате выполнения которого происходит физическое удаление всех файлов указанной таблицы. Путь и имя удаляемой таблицы определяют свойства `DatabaseName` и `TableName` набора данных. Перед удалением таблицы набор данных должен быть закрыт.

Для *переименования* таблиц dBase и Paradox можно использовать метод `RenameTable (const NewTableName: String)`, при выполнении которого переименовываются все файлы, относящиеся к таблице. Параметр `NewTableName`

задает новое название таблицы. Напомним, что имя таблицы совпадает с именами файлов, а расширения имен файлов указывают на содержащуюся в них информацию, например, данные или индексы.

Установка уровня доступа

В случае многопользовательского доступа к БД для таблицы можно задать уровни доступа, которые будут действовать для других приложений. Уровень доступа определяет возможность записи данных в таблицу и их чтения.

Метод `LockTable (LockType: TLockType)` устанавливает *блокировку* для таблицы, при этом параметр `LockType` задает тип блокировки:

- `ltReadLock` (запрещены запись и чтение записей таблицы); этот режим является своего рода режимом монопольного доступа;
- `ltWriteLock` (запрещена запись в таблицу); другие приложения не могут выполнять модификацию записей таблицы, но чтение данных им разрешено.

Метод `UnLockTable (LockType: TLockType)` снимает установленную ранее блокировку.

Рассмотрим на конкретном примере, как осуществляется блокировка таблицы.

```
procedure TForm1.Button1Click(Sender: TObject);
var n: longint;
begin
try
   // Блокировка таблицы от внесения изменений
   Table1.LockTable(ltWriteLock);
   // Пересчет окладов
   Table1.First;
   for n := 1 to Table1.RecordCount do begin
       Table1.Edit;
       Table1.FieldByName('Salary').AsFloat :=
              Table1.FieldByName('Salary').AsFloat + 500;
       Table1.Next;
       Table1.Post;
       end;
   finaly
   // Отмена блокировки таблицы
   Table1.UnLockTable(ltWriteLock);
   end;
end;
```

Оклады всех сотрудников увеличиваются на `500` (рублей). На время пересчета таблица блокируется, что защищает ее от внесения изменений другими прило-

жениями. При операциях с таблицей выполняется локальная обработка исключений, состоящая в отмене блокировки таблицы.

Сортировка набора данных

Порядок расположения записей в наборе данных может быть неопределенным. По умолчанию записи не отсортированы или сортируются, например, для таблиц Paradox по ключевым полям, а для таблиц dBase в порядке их поступления в файл таблицы.

С отсортированными записями набора данных работать более удобно. Сортировка заключается в упорядочивании записей по определенному полю в порядке возрастания или убывания содержашихся в нем значений. Сортировку можно выполнять и по нескольким полям. Например, при сортировке по двум полям сначала записи упорядочиваются по значениям первого поля, а затем группы записей с одинаковым значением первого поля сортируются по второму полю.

Сортировка наборов данных `Table` и `Query` выполняется различными способами. Здесь мы рассмотрим сортировку набора данных `Table`, для компонента `Query` вопросы сортировки рассмотрены в *главе 8*.

Сортировка наборов данных `Table` выполняется автоматически по текущему индексу. При смене индекса происходит автоматическое переупорядочивание записей. Таким образом, сортировка возможна по полям, для которых создан индекс. Для сортировки по нескольким полям нужно создать индекс, включающий эти поля.

Направление сортировки определяет параметр `ixDescending` текущего индекса, по умолчанию он выключен, и упорядочивание выполняется в порядке возрастания значений. Если признак `ixDescending` индекса включен, то сортировка выполняется в порядке убывания значений.

Напомним, что задать индекс (текущий индекс), по которому выполняется сортировка записей, можно с помощью свойств `IndexName` или `IndexFieldNames`. Эти свойства являются взаимоисключающими, и установка значения одного из них приводит к автоматической очистке значения другого. В качестве значения свойства `IndexName` указывается имя индекса, установленное при его создании. При использовании свойства `IndexFieldNames` указываются имена полей, образующих соответствующий индекс.

В связи с тем, что главный индекс (ключ) таблиц Paradox не имеет имени, выполнить сортировку по этому индексу можно только с помощью свойства `IndexFieldNames`.

Вот небольшой пример, иллюстрирующий сортировку с указанием имен индексов:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
     0: Table1.IndexName := 'indName';
```

```
    1: Table1.IndexName := 'indBirthDay';
    end;
end;
```

В качестве набора данных используется компонент `Table1`, а сортировка выполняется двумя способами: по индексу `indName`, созданному для поля `Name`, и по индексу `indBirthDay`, созданному для поля `BirthDay`.

Еще один пример сортировки, на этот раз с указанием имен индексных полей (для связанной с набором данных таблицы поле `Code` задано автоинкрементным и определено в качестве главного индекса):

```
procedure TForm1.Button5Click(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
    0: Table1.IndexFieldNames := 'Name';
    1: Table1.IndexFieldNames := 'Name;BirthDay';
    2: Table1.IndexFieldNames := 'Code';
    end;
end;
```

Здесь сортировка выполняется по следующим полям: `Name` (индекс `indName`), `Name` и `BirthDay` (индекс `indNameBirthDay`), `Code` (главный индекс).

А теперь рассмотрим более сложный пример сортировки. В качестве набора данных снова используется компонент `Table1`. Пользователь может управлять сортировкой его записей с помощью двух групп переключателей: в первой задается вид, а во второй — направление сортировки. Сортировка выполняется после нажатия кнопки **Сортировать** (`btnSort`). Вид формы при проектировании показан на рис. 7.1.

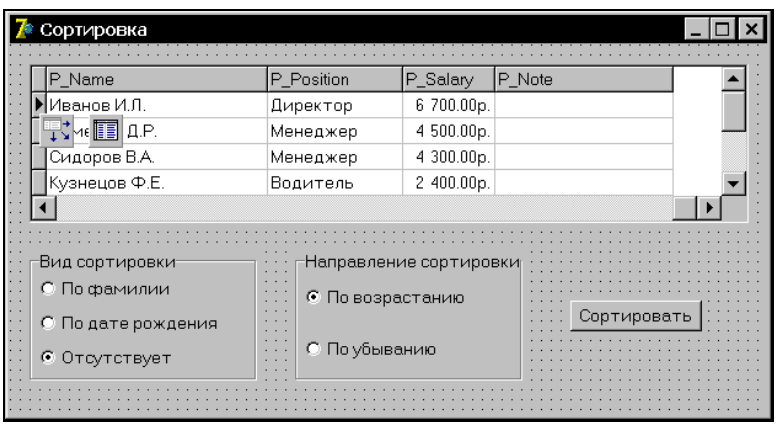


Рис. 7.1. Вид формы для сортировки набора данных

В связи с тем, что компоненты Table и DataSource являются невизуальными и при выполнении приложения не видны, их можно размещать в любом удобном месте формы, где они не мешают другим компонентам. Часто компоненты Table и DataSource помещаются на компоненте DBGrid, как это сделано в рассматриваемом примере.

Ниже приводится обработчик события нажатия кнопки btnSort, вызывающей выполнение сортировки.

```
procedure TForm1.btnSortClick(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
    0: Table1.IndexName := 'indName';
    1: Table1.IndexName := 'indBirthDay';
    2: Table1.IndexName := '';
    end;
case RadioGroup2.ItemIndex of
    0: Table1.IndexDefs
      [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options :=
          Table1.IndexDefs
             [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options +
                        [ixDescending];
    1: Table1.IndexDefs
      [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options :=
          Table1.IndexDefs
             [Table1.IndexDefs.IndexOf(Table1.IndexName)].Options -
                        [ixDescending];
    end;
end;
```

Поля, по которым сортируются записи, устанавливаются через свойство IndexName. При отсутствии сортировки этому свойству присваивается пустая строка. Для таблиц Paradox это означает сортировку по первому полю. Для таблиц dBase записи располагаются в порядке их поступления в файл таблицы.

Управление направлением сортировки осуществляется с помощью параметра ixDescending текущего индекса. Для определения номера текущего индекса в списке IndexDefs используется метод IndexOf.

Навигация по набору данных

Навигация по набору данных заключается в управлении указателем текущей записи (курсором). Этот указатель определяет запись, с которой будут выполняться такие операции, как редактирование или удаление.

Перемещение по записям

Перед перемещением указателя текущей записи набор данных автоматически переводится в режим просмотра. Если текущая запись находилась в режимах редактирования или вставки, то перед перемещением указателя сделанные в записи изменения вступают в силу, для чего набор данных автоматически вызывает метод `CheckBrowseMode`.

Для перемещения указателя текущей записи в наборе данных используются следующие методы:

- процедура `First` — установка на первую запись;
- процедура `Next` — установка на следующую запись (при вызове метода для последней записи указатель не перемещается);
- процедура `Last` — установка на последнюю запись;
- процедура `Prior` — установка на предыдущую запись (при вызове метода для первой записи указатель не перемещается);
- функция `MoveBy (Distance: Integer): Integer` — перемещение на число записей, определяемое параметром `Distance`. Если его значение больше нуля, то перемещение осуществляется вперед, если меньше нуля — то назад. При нулевом значении параметра указатель не перемещается. Если заданное параметром `Distance` число записей выходит за начало или конец набора данных, то указатель устанавливается на первую или на последнюю запись. В качестве результата возвращается число записей, на которое переместился указатель.

При перемещении указателя текущей записи учитываются ограничения и фильтр, определенные для набора данных. Таким образом, перемещение выполняется по записям набора данных, которые он содержит в текущий момент времени. Число записей определяется свойством `RecordCount`.

Значения указателя текущей записи изменяют также методы, связанные с поиском записей, например, метод `FindFirst`. Они рассматриваются ниже в этой главе.

Замечание

При изменении порядка сортировки набора данных расположение его записей может измениться, что чаще всего и происходит, но указатель по-прежнему указывает на первоначальную запись, даже если она находится на другом месте и имеет новое значение свойства `RecNo`.

Рассмотрим следующий пример:

```
procedure TForm1.Button3Click(Sender: TObject);
var s: real;
    n: integer;
begin
s := 0;
```

```
// Установка текущего указателя на первую запись
Table1.First;
for n := 1 to Table1.RecordCount do begin
    s := s + Table1.FieldByName('Salary').AsFloat;
    // Перемещение текущего указателя на следующую запись
    Table1.Next;
    end;
Label2.Caption := FloatToStr(s);
end;
```

В приведенной процедуре перебираются все записи набора данных `Table1`, при этом в переменной `s` накапливается сумма значений, содержащихся в поле `Salary`. Перебор записей осуществляется с помощью метода `Next`, вызываемого в цикле. Предварительно с помощью метода `First` указатель устанавливается на первую запись. После выполнения кода указатель будет установлен на последнюю запись.

Отметим, что используемая для перебора записей переменная `n` имеет тип `integer`, совпадающий с типом `longint` свойства `RecordCount`.

Аналогичным образом можно перебрать все записи набора данных, начиная с последней. Естественно, при этом нужно вызывать методы `Last` и `Prior`.

Для *контроля положения* указателя текущей записи можно использовать свойство `RecNo`, которое содержит номер записи, считая от начала набора данных (для локальных таблиц dBase и Paradox).

Для таблиц Paradox свойство `RecNo` можно использовать еще и для перехода к записи с известным номером: такой переход выполняется установкой свойства `RecNo` в значение, равное номеру нужной записи. Например:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.RecNo := StrToInt(Edit1.Text);
end;
```

При нажатии кнопки `Button1` указатель текущей записи набора данных `Table1` устанавливается на запись, номер которой содержит редактор `Edit1`.

Для определения *начала* и *конца* набора данных при перемещении указателя текущей записи можно использовать свойства `BOF` и `EOF` типа `Boolean` соответственно. Эти свойства доступны для чтения при выполнении приложения. Свойство `BOF` показывает, находится ли указатель на первой записи набора данных. Этому свойству присваивается значение `True` при установке указателя на первой записи, например, сразу после вызова метода `First`. Свойство `EOF` показывает, находится ли указатель на последней записи набора данных. Этому свойству устанавливается значение `True` при размещении указателя на последней записи.

Замечание

Для пустого набора данных свойства BOF и EOF имеют значения True.

При изменении порядка сортировки или фильтрации, а также при удалении или добавлении записей значения свойств BOF и EOF могут изменяться. Например, если направление сортировки изменяется на противоположное, то первая запись становится последней.

При работе с таблицами одновременно нескольких приложений, когда постоянно добавляются или удаляются записи, значения свойств BOF и EOF соответствуют действительному состоянию набора данных в определенные моменты времени. Так, свойство EOF устанавливается в значение True сразу после выполнения метода Last. Если после этого другим приложением в конец набора данных добавлена новая запись, то значение свойства EOF становится неправильным.

Рассмотрим еще один пример, иллюстрирующий работу с указателем текущей записи:

```
procedure TForm1.Button2Click(Sender: TObject);
var s: real;
begin
s := 0;
Table1.First;
while not Table1.EOF do begin
      s := s + Table1.FieldByName('Salary').AsFloat;
      Table1.Next;
      end;
s := s + Table1.FieldByName('Salary').AsFloat;
Label2.Caption := FloatToStr(s);
end;
```

Как и в предыдущем примере, здесь перебираются все записи набора данных Table1 и подсчитывается сумма значений, содержащихся в поле Salary. Отличие заключается в том, что использован итерационный цикл с верхним окончанием, условием выхода из которого является достижение последней записи набора данных.

Иногда в цикле выполняются сложные действия. При этом на перебор записей набора данных может потребоваться достаточно большое время, в течение которого приложение (и компьютер) не реагирует на команды пользователя. Поэтому в циклы, надолго загружающие работой компьютер, рекомендуется вставлять вызов метода Application.ProcessMessages, обеспечивающий системе Windows возможность обрабатывать сообщения. Кроме того, программист должен предусмотреть досрочный выход из длительного цикла, например, с помощью переменной-признака.

Перед началом длительного цикла обработки записей БД и выполнения расчетов целесообразно выдать предупреждающее сообщение пользователю.

При перемещении по записям набора данных связанные с ним визуальные компоненты отображают изменения данных, причем смена отображения может происходить достаточно быстро, вызывая неприятное мелькание на экране. Чтобы избежать этого эффекта, можно программно до начала цикла перебора записей временно отключить набор данных от всех связанных с ним визуальных компонентов, а по окончании цикла снова подключить. Для этого предназначены методы `DisableControls` и `EnableControls`.

Рассмотрим следующий пример. Пусть в цикле перебираются все записи набора данных `Table1`, начиная с первой, и подсчитывается сумма значений, содержащихся в поле `Salary`. Для запуска и остановки процесса расчета используются кнопки **Начать** (`btnWorkStart`) и **Прервать** (`btnWorkBreak`). Поле индикатора `ProgressBar1` служит для показа хода работы, а процент перебранных записей отображается в надписи `lblWorkPercent`, находящейся рядом с индикатором. На рис. 7.2 показан вид формы во время выполнения приложения.

Код модуля приведенной на рис. 7.2 формы имеет вид:

```
   unit uNavig;

   interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, Db, DBTables, Grids, DBGrids, StdCtrls, ExtCtrls,
     DBCtrls, ComCtrls;
type
  TForm1 = class(TForm)
          btnClose: TButton;
       DataSource1: TDataSource;
            Table1: TTable;
           DBGrid1: TDBGrid;
            Panel1: TPanel;
      ProgressBar1: TProgressBar;
      btnWorkStart: TButton;
      btnWorkBreak: TButton;
            Label1: TLabel;
    lblWorkPercent: TLabel;
            lblSum: TLabel;

    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnWorkStartClick(Sender: TObject);
    procedure btnWorkBreakClick(Sender: TObject);
```

```
  private
    { Private declarations }
  public
    { Public declarations }

  end;
var     Form1: TForm1;
       WorkBreak: boolean;

   implementation

   {$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
btnWorkStart.Enabled := True;
btnWorkBreak.Enabled := False;
WorkBreak := False;
   end;

procedure TForm1.btnWorkStartClick(Sender: TObject);
label 10;
var s: real;
begin
btnWorkStart.Enabled := False;
btnWorkBreak.Enabled := True;
s := 0;
ProgressBar1.Max := Table1.RecordCount;
Table1.DisableControls;
Table1.First;
while not Table1.EOF do begin
      s := s + Table1.FieldByName('Salary').AsFloat;
      // Без этого вызова не будет обрабатываться нажатие кнопки
      // btnWorkBreak
      Application.ProcessMessages;
      ProgressBar1.Position := Table1.RecNo;
      lblWorkPercent.Caption :=
          IntToStr(Round(ProgressBar1.Position/ProgressBar1.Max*100)) + '%';
      if WorkBreak then goto 10;
      Table1.Next;
      end;
lblSum.Caption := 'Итого ' + FloatToStr(s);
10:
Table1.EnableControls;
```

```
ProgressBar1.Position := 0;
lblWorkPercent.Caption := '';
btnWorkStart.Enabled := True;
btnWorkBreak.Enabled := False;
WorkBreak := False;
   end;

procedure TForm1.btnWorkBreakClick(Sender: TObject);
begin
WorkBreak := True;
   end;

procedure TForm1.btnCloseClick(Sender: TObject);
begin
Close;
   end;

end.
```

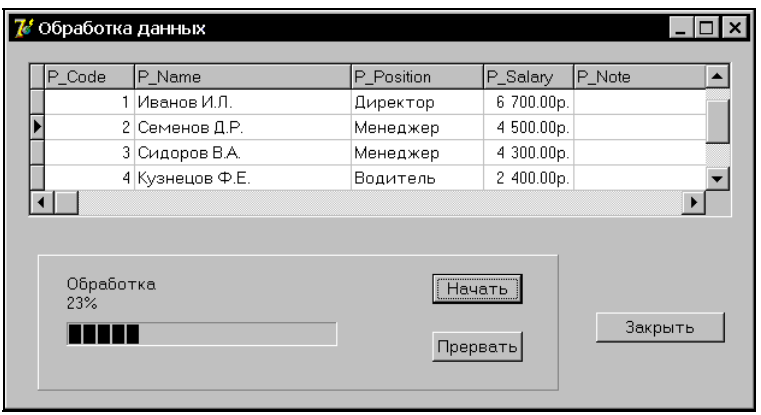


Рис. 7.2. Форма приложения, выполняющего перебор записей набора данных

Цикл обработки запускается нажатием кнопки `btnWorkStart`. Перед началом цикла определяется максимальное значение индикатора `ProgressBar1`, переменная `WorkBreak` устанавливается в `False`, а визуальные компоненты отключаются от набора данных `Table1`. В цикле происходит подсчет суммы, а также отображение хода выполнения работ в поле индикатора `ProgressBar1` и надписи `lblWorkPercent`. По окончании цикла результат отображается в надписи `lblSum`, и визуальные компоненты снова подключаются к набору данных `Table1`.

Для прерывания обработки служит кнопка btnWorkBreak, при ее нажатии переменная-признак WorkBreak принимает значение True. Значение этой переменной проверяется в цикле обработки, и если оно равно True, то цикл прекращается путем вызова процедуры Break. Чтобы при выполнении цикла обрабатывались событие нажатия кнопки btnWorkBreak и другие события, в цикл включен вызов метода ProcessMessages.

При любом изменении положения указателя текущей записи для набора данных генерируются события BeforeScroll и AfterScroll типа TDataSetNotifyEvent.

Пользователь имеет возможность *перемещаться* по набору данных с помощью элементов управления, в качестве которых часто используются компоненты DBGrid и DBNavigator. Управление этими элементами с помощью мыши или клавиатуры приводит к автоматическому вызову соответствующих методов, перемещающих указатель текущей записи в заданное место. Например, после нажатия кнопкок **First record**, **Prior record**, **Next record** или **Last record** компонента DBNavigator1 косвенно вызываются методы First, Prior, Next или Last, перемещающие указатель текущей записи соответственно на первую, предыдущую, следующую или последнюю записи набора данных, с которым связан (через источник данных DataSource) компонент DBNavigator1.

Другим вариантом является размещение в форме элементов управления, например, кнопки Button, предназначенных для навигации по набору данных. Кроме того, часто в форме также размещаются элементы (обычно кнопки) для управления такими операциями, как редактирование, вставка записей, фильтрация и сортировка набора данных.

Рассмотрим в качестве примера форму приложения, в которой осуществляется перемещение по набору данных с помощью пяти кнопок (рис. 7.3). При нажатии какой-либо из этих кнопок вызывается соответствующий метод перемещения текущего указателя в заданном направлении. Например, при нажатии кнопки **Предыдущая запись** (btnPrior) вызывается метод Prior. При перемещении указателя на произвольное число записей дополнительно используется обратный счетчик SpinEdit1, в поле которого вводится это число.

Код модуля uNavig2 рассматриваемой формы приложения имеет следующий вид:

```
  unit uNavig2;

  interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, StdCtrls, Grids, DBGrids, Db, DBTables, Spin;

type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
         Query1: TQuery;
```

```
        DBGrid1: TDBGrid;
       btnFirst: TButton;
        btnNext: TButton;
       btnPrior: TButton;
        btnLast: TButton;
      lblRecord: TLabel;
      btnMoveBy: TButton;
      SpinEdit1: TSpinEdit;

    procedure FormCreate(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPriorClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure Query1AfterScroll(DataSet: TDataSet);
    procedure btnMoveByClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
     Form1: TForm1;

   implementation

   {$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
// Набор данных является редактируемым
Query1.RequestLive := True;
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM Goods.db');
Query1.Open;
SpinEdit1.Value := 0;
   end;

// Переход на первую запись
procedure TForm1.btnFirstClick(Sender: TObject);
```

```
begin
Query1.First;
   end;

// Переход к следующей записи
procedure TForm1.btnNextClick(Sender: TObject);
begin
Query1.Next;
   end;

// Переход к предыдущей записи
procedure TForm1.btnPriorClick(Sender: TObject);
begin
Query1.Prior;
   end;

// Переход на последнюю запись
procedure TForm1.btnLastClick(Sender: TObject);
begin
Query1.Last;
   end;

// Перемещение на число записей, заданное в счетчике SpinEdit1
procedure TForm1.btnMoveByClick(Sender: TObject);
begin
Query1.MoveBy(SpinEdit1.Value);
   end;

// Вывод номера записи и пересчет границ счетчика
procedure TForm1.Query1AfterScroll(DataSet: TDataSet);
begin
lblRecord.Caption := 'Запись номер ' + IntToStr(Query1.RecNo);
SpinEdit1.MinValue := 1 - Query1.RecNo;
SpinEdit1.MaxValue := Query1.RecordCount - Query1.RecNo;
   end;

end.
```

Сразу после того, как положение указателя изменилось, в надписи `lblRecord` выводится номер текущей записи в наборе данных. Пересчитываются минимальное и максимальное значения обратного счетчика, определяющие диапазон допустимых значений, указываемых при вызове метода `MoveBy`. Эти действия кодируются в обработчике события `OnAfterScroll` для того, чтобы информация о текущей записи обновлялась сразу после перемещения текущего указателя. Событие `OnAfterScroll` происходит также при открытии набора данных, когда указатель текущей записи устанавливается на первую запись.

Рис. 7.3. Форма приложения, в которой осуществляется перемещение по набору данных

В качестве набора данных использован компонент `Query1`, задание основных свойств которого выполняется при создании формы. SQL-запрос (`SELECT * FROM Goods.db`) указан в свойстве `SQL`. Он обеспечивает получение всех полей и записей таблицы `Goods.db`. Свойство `RequestLive` установлено в значение `True`, чтобы обеспечить возможность редактирования набора данных. Таким образом, в рассмотренном примере компонент `Query` практически не отличается от компонента `Table` в плане работы с редактируемым набором данных, связанным с одной таблицей.

Иногда при подключении к одной таблице нескольких наборов данных возникает необходимость *синхронизации* их текущих указателей. Для этих целей используется метод `GotoCurrent (Table: TTable)`, устанавливающий указатель на текущую запись в наборе данных `Table`. Например:

```
Table1.GotoCurrent(Table2);
```

Здесь в наборе данных `Table1` текущий указатель устанавливается на запись, которая является текущей в наборе данных `Table2`.

Переход по закладкам

Побочным эффектом выполнения ряда операций с наборами данных является изменение положения указателя текущей записи. Часто этот эффект нежелателен, т. к. после выполнения такой операции указатель находится не в том месте, где был до начала операции. При этом приходится снова отыскивать нужную запись и позиционировать на ней указатель, что неудобно даже при небольшом количестве записей.

Примером такой операции является рассмотренный выше расчет суммы значений по полю `Salary`, при котором выполняется перебор всех записей в прямом или обратном порядке. По окончании цикла указатель находится на первой или последней записи, а не там, где он был до начала суммирования.

Замечание

Положение указателя текущей записи может измениться также при формировании отчета из записей набора данных.

Для восстановления прежнего положения текущего указателя можно использовать, например, поиск записей: до начала операции данные текущей записи запоминаются, а после ее выполнения осуществляется поиск записи по этим данным. Похожим вариантом восстановления прежнего положения указателя является использование номера записи в наборе данных, который доступен через свойство `RecNo`.

Переход на определенную запись может понадобиться также для позиционирования указателя на запись, информация о которой была сохранена. Осуществляемые при этом действия не отличаются от указанных выше. Информация, необходимая для выполнения последующего перехода к требуемой записи, может запоминаться предварительно в удобный момент, например, при просмотре или редактировании записи.

Вот пример, иллюстрирующий, как восстановить положение текущего указателя:

```
var RecordNumber: integer;
...
// Запоминание номера текущей записи
   RecordNumber := Query1.RecNo;

     // Операция, изменяющая положение указателя

// Переход к записи с запомненным номером
Query1.First;
Query1.MoveBy(RecordNumber - 1);
```

Здесь для перемещения текущего указателя используется номер записи, запоминаемый в переменной `RecordNumber`.

В таблицах Paradox для возврата к записи можно установить свойство `RecNo` набора данных в предварительно сохраненное значение. Например:

```
var RecordNumber: integer;
...
// Запоминание номера текущей записи
   RecordNumber := Table1.RecNo;

     // Операция, изменяющая положение указателя

// Переход к записи с запомненным номером
Table1.RecNo := RecordNumber;
```

Вопросы, связанные с поиском записей, рассматриваются в данной главе ниже.

Кроме описанных способов, для перехода на определенную запись можно использовать *закладки* — специальные пометки каких-либо записей. Закладка имеет тип `TBookmark`, представляющий собой обычный указатель `Pointer`, с помощью которого можно ссылаться на различные объекты, в данном случае — на записи набора данных.

Перед использованием закладку нужно *создать*, т. е. определить запись, с которой эта закладка связана. Для этого предназначена функция `GetBookmark: TBookmark`, которая создает закладку на текущей записи набора данных. Возвращаемую закладку обычно запоминают в переменной типа `TBookmark` и впоследствии используют для перехода к помеченной ею записи.

Иногда закладка может неверно указывать позицию связанной с ней записи, при этом говорят о *нестабильности* закладки. Стабильность закладки в значительной степени зависит от форматов таблиц. Так, для таблиц dBase закладка всегда стабильна; для таблиц Paradox закладка стабильна, если у таблицы определен главный ключ. Таким образом, закладка устойчиво указывает на связанную с ней запись в тех случаях, когда существует признак, позволяющий однозначно различать записи. В противном случае закладка может быть нестабильной и ошибочно указывать на другую запись набора данных. Для проверки стабильности закладки используется функция `DBIGetCursorProps` (модуль `DBE`).

На установленную закладку можно перейти с помощью метода `GotoBookmark`. Процедура `GotoBookmark (Bookmark: TBookmark)` позиционирует указатель текущей записи на закладку, указанную параметром `Bookmark`. Если закладка предварительно не создана и ее значение равно `Nil`, то текущий указатель не перемещается.

Для удаления закладки используется метод `FreeBookmark`. Процедура `FreeBookmark (Bookmark: TBookmark)` сбрасывает значение закладки в `Nil` и освобождает занимаемую ею память.

Замечание

Попытка перейти на удаленную закладку путем вызова метода `GotoBookmark` приводит к возникновению исключения. К такому же эффекту приводит попытка повторного освобождения закладки методом `FreeBookmark`.

Целесообразно перед использованием закладки уточнить, существует ли она. Функция `BookmarkValid (Bookmark: TBookmark): Boolean` проверяет наличие закладки, заданной параметром `Bookmark`. Если закладка найдена, то в качестве результата функция возвращает значение `True`, в противном случае — `False`.

Можно устанавливать и использовать несколько закладок.

Вот пример, иллюстрирующий использование закладок для перехода к нужным записям:

```
procedure TForm1.Button1Click(Sender: TObject);
// Описание переменной типа закладка
var bm: TBookmark;
```

```
begin
// Создание закладки bm
bm := Table1.GetBookmark;
Table1.DisableControls;
Table1.First;
   while not Table1.EOF do begin

         // Обработка записей

      Table1.Next;
      end;
// Переход на закладку bm
if Table1.BookmarkValid(bm) then Table1.GotoBookmark(bm);
// Освобождение закладки bm
if Table1.BookmarkValid(bm) then Table1.FreeBookmark(bm);
Table1.EnableControls;
end;
```

Перед началом цикла перебора записей на текущей записи набора данных `Table1` устанавливается закладка `bm`. По окончании цикла для восстановления прежнего положения текущего указателя выполняется переход на закладку, и закладка удаляется. Перед переходом на закладку проверяется ее существование.

Аналогичным образом закладки можно использовать и для перехода к заранее выбранным и отмеченным записям.

Фильтрация записей

Фильтрация — это задание ограничений для записей, отбираемых в набор данных. Напомним, что набор данных представляет собой записи, выбранные из одной или нескольких таблиц. Состав записей в наборе данных в данный момент времени зависит от установленных ограничений, в том числе от фильтров. Система Delphi дает возможность осуществлять фильтрации записей:

- по выражению;
- по диапазону.

По умолчанию фильтрация записей не ведется, и набор данных `Table` содержит все записи связанной с ним таблицы БД, а в набор данных `Query` включаются все записи, удовлетворяющие SQL-запросу, содержащемуся в свойстве `SQL`.

Замечание

При включении фильтрация записей действует в дополнение к другим ограничениям, например, SQL-запросу компонента `Query` или ограничению, налагаемому отношением "главный-подчиненный" между таблицами БД. Отметим, что для компонента `Query` SQL-запрос является средством отбора записей в набор данных, а фильтрация дополнительно ограничивает состав этих записей.

Фильтрация по выражению

При использовании фильтрации по выражению набор данных ограничивается записями, удовлетворяющими выражению фильтра, задающему условия отбора записей.

Достоинством фильтрации по выражению является то, что она применима к любым полям, в том числе к неиндексированным. В связи с тем, что в процессе отбора просматриваются все записи таблицы, фильтрация по выражению эффективна при небольшом количестве записей.

Для задания выражения фильтра используется свойство `Filter` типа `String`. Фильтр представляет собой конструкцию, в состав которой могут входить следующие элементы:

- имена полей таблиц;
- литералы;
- операции сравнения;
- арифметические операции;
- логические операции;
- круглые и квадратные скобки.

Если имя поля содержит пробелы, то его заключают в квадратные скобки, в противном случае квадратные скобки необязательны.

Литерал представляет собой значение, заданное явно, например, число, строка или символ. Отметим, что имена переменных в выражении фильтра использовать нельзя. Если в фильтр требуется включить значение переменной или свойства какого-либо компонента, то это значение должно быть преобразовано в строковый тип.

Операции сравнения представляют собой обычные для языка Pascal отношения <, >, =, <=, >= и <>.

Арифметическими являются операции +, –, * и / (сложения, вычитания, умножения и деления соответственно).

В качестве логических операций можно использовать `AND`, `OR` и `NOT` (логическое умножение, сложение и отрицание соответственно).

Круглые скобки применяются для изменения порядка выполнения арифметических и логических операций.

В качестве примеров задания условий фильтрации приведем следующие выражения:

```
Salary <= 2000
Post = 'Лаборант' OR Post = 'Инженер'
```

Первое выражение обеспечивает отбор всех записей, для которых значение поля оклада (`Salary`) не превышает `2000`. Второе выражение обеспечивает отбор записей, поле должности (`Post`) которых содержит значение `Лаборант` или `Инженер`.

Если выражение фильтра не позволяет сформировать сложный критерий фильтрации, то в дополнение к нему можно использовать обработчик события `OnFilterRecord`.

Для *активизации* и *деактивизации* фильтра применяется свойство `Filter` типа `Boolean`. По умолчанию это свойство имеет значение `False`, и фильтрация выключена. При установке свойства `Filtered` в значение `True` фильтрация включается, и в набор данных отбираются записи, которые удовлетворяют фильтру, записанному в свойстве `Filter`. Если выражение фильтра не задано (по умолчанию), то в набор данных попадают все записи.

Замечание

Активизация фильтра и выполнение фильтрации возможны также на этапе разработки приложения.

Если выражение фильтра содержит ошибки, то при попытке выполнить его генерируется исключение. Если фильтр неактивен (свойство `Filtered` имеет значение `False`), то выражение фильтра не анализируется на корректность.

Параметры фильтрации задаются с помощью свойства `FilterOptions` типа `TFilterOptions`. Это свойство принадлежит к множественному типу и может принимать комбинации двух значений:

- `foCaseInsensitive` — регистр букв не учитывается, т. е. при задании фильтра `Post = 'Водитель'` слова `Водитель`, `ВОДИТЕЛЬ` или `водитель` будут восприняты как одинаковые. Значение `foCaseInsensitive` нужно отключать, если требуется различать слова, написанные в различных регистрах;
- `foNoPartialCompare` — выполняется проверка на полное соответствие содержимого поля и значения, заданного для поиска. Обычно применяется для строк символов. Если известны только первые символы (или символ) строки, то нужно указать их в выражении фильтра, заменив остальные символы на звездочки (`*`) и выключив значение `foNoPartialCompare`. Например, при выключенном значении `foNoPartialCompare` для фильтра `Post = 'В*'` будут отобраны записи, у которых в поле `Post` содержатся значения `Водитель`, `Вод.`, `Вод-ль` или `Врач`.

По умолчанию все параметры фильтра выключены, и свойство `FilterOptions` имеет значение `[]`.

Рассмотрим в качестве примера обработчики событий формы, используемой для фильтрации записей набора данных по выражению. Вид формы приведен на рис. 7.4.

Управление фильтрацией набора данных выполняется с помощью двух кнопок и поля редактирования. При нажатии кнопки **Фильтровать** (`btnFilter`) фильтр активизируется путем присваивания значения `True` свойству `Filtered` набора данных. Редактор `edtFilter` предназначен для задания выражения фильтра. При активизации фильтра происходит отбор записей, которые удовлетворяют заданному в выражении условию. При нажатии кнопки **Все записи** (`btnAllRecord`) фильтр отключается, при этом показываются все записи.

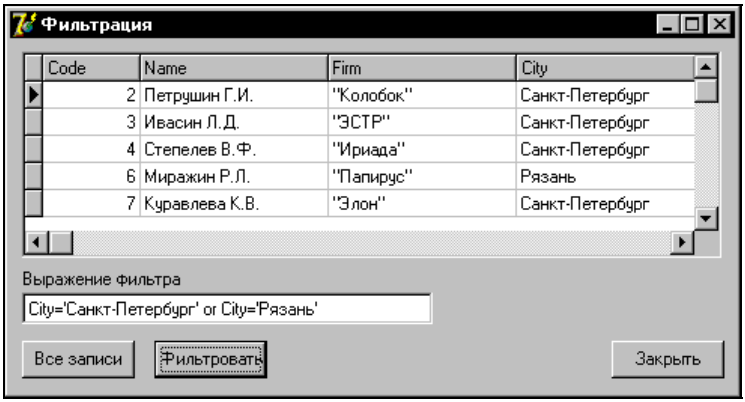


Рис. 7.4. Фильтрация по выражению

Ниже приведены три обработчика событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Table1.FilterOptions := [foCaseInsensitive];
Table1.Filtered := False;
   end;

procedure TForm1.btnFilterClick(Sender: TObject);
begin
Table1.Filtered := True;
Table1.Filter := edtFilter.Text;
   end;

procedure TForm1.btnAllRecordClick(Sender: TObject);
begin
Table1.Filtered := False;
end;
```

Включение и выключение фильтра осуществляется через свойство `Filtered`. Фильтрация выполняется без учета регистра букв.

В приведенном примере пользователь должен самостоятельно набирать выражение фильтра. Это предоставляет пользователю достаточно широкие возможности управления фильтрацией, но требует от него знания правил построения выражений.

Часто удобно предоставить пользователю список готовых выражений (шаблонов) для выбора. При этом пользователь получает также возможность редактировать выбранное выражение и корректировать весь список. Такой режим реализуется, например, с помощью компонентов `ComboBox` и `Memo`.

Если набор условий фильтрации ограничен и не изменяется, то пользователь может управлять отбором записей с помощью таких компонентов, как флажки (`CheckBox`) и переключатели (`RadioButton`).

Рассмотрим в качестве примера обработчики событий формы приложения, в которой пользователю предоставлена возможность управлять фильтрацией по двум полям или по выражению, либо совсем отключать фильтрацию (рис. 7.5).

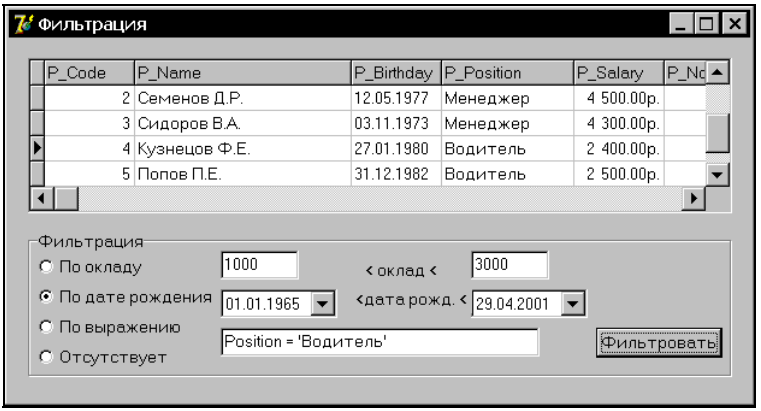


Рис. 7.5. Управление фильтрацией по выражению

Условия фильтрации по полям `Salary` и `BirthDay` заданы в виде двойного неравенства на этапе разработки приложения и при выполнении приложения не могут быть изменены пользователем. Пользователь может задавать в этом неравенстве минимальное и максимальное значения. Фильтрация записей присходит при нажатии кнопки **Фильтровать**. В обработчике события нажатия этой кнопки автоматически формируется выражение фильтра на основании выбранного для фильтрации поля и введенных пользователем значений.

Ниже приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Table1.Filter := '';
Table1.FilterOptions := [foCaseInsensitive];
Table1.Filtered := True;
   end;

procedure TForm1.btnFilterClick(Sender: TObject);
begin
// Фильтровать по окладу
if rbFilterSalary.Checked then Table1.Filter := 'Salary > ' +
   edtSalaryMin.Text + ' AND Salary < ' + edtSalaryMax.Text;
```

```
// Фильтровать по дате рождения
if rbFilterBirthDay.Checked then Table1.Filter :=
      'BirthDay > ' + DateToStr(dtpBirthDayMin.Date) +
      ' AND BirthDay < ' + DateToStr(dtpBirthDayMax.Date);
// Фильтровать по выражению
if rbFilterExpression.Checked then Table1.Filter := edtFilter.Text;
//  Отключить фильтрацию
if rbNoFilter.Checked then Table1.Filter := '';
end;
```

Для ввода значений минимальной и максимальной даты применяются два компонента типа `TDateTimePicker` (`dtpBirthDayMin` и `dtpBirthDayMax`), с помощью которых пользователь может выбирать дату. Для ограничения значения оклада использованы однострочные редакторы `edtSalaryMin` и `edtSalaryMax`.

Поскольку в выражении фильтра нельзя использовать имена переменных и свойств компонентов, то при его формировании набранные пользователем значения должны преобразовываться в строковый тип. В приведенном примере это относится к значению даты.

Аналогично задаются и более сложные условия формирования фильтра, в том числе с помощью логических операций `OR` и `NOT`. Кроме того, пользователь может, как и в предыдущем примере, управлять процессом отбора записей с помощью выражения фильтра, которое вводится в редакторе `edtFilter`.

В данном примере, в отличие от предыдущего, при отключении фильтрации фильтр остается активным, однако его выражение очищается.

Приведем еще один пример на фильтрацию — реализуем электронную записную книжку. Для этого совместно с набором данных будем использовать элемент управления вкладками `TabControl`, расположенный на странице **Win32** Палитры компонентов.

Рассмотрим обрабочики событий формы приложения, имитирующего записную книжку со списком телефонов. Вид книжки с распечаткой фамилий на букву `A` показан на рис. 7.6.

Список владельцев телефонов хранится с таблице `Notebook.db`, с которой связан набор данных `Table1`. Содержимое списка отображается в компоненте `DBGrid1`. Пользователь может отобразить все фамилии или вывести фамилии, начинающиеся с определенной буквы.

Для управления отбором записей используется фильтрация. Флажок **Показывать все записи** (`CheckBox1`) активизирует и деактивизирует фильтр. По умолчанию флажок установлен, при этом фильтр неактивен, и отображаются все фамилии. Если флажок `CheckBox1` сброшен, то выполняется фильтрация. Отбор записей осуществляется по первой букве фамилии, которая выбирается с помощью вкладок компонента `TabControl1`.

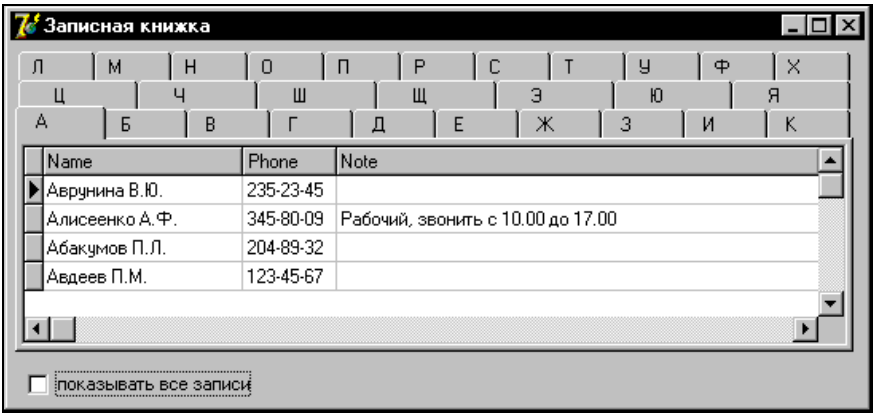


Рис. 7.6. Записная книжка

Ниже приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
DBGrid1.Align := alClient;
// Разрешена фильтрация по частичному совпадению и независимо от регистра
букв
Table1.FilterOptions := [foCaseInsensitive];
// Отключение фильтрации
CheckBox1.Checked := True; CheckBox1Click(Sender);
// Первоначальное формирование выражения фильтра (для буквы А)
TabControl1Change(Sender);
   end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
if CheckBox1.Checked
   then Table1.Filtered := False
   else Table1.Filtered := True;
   end;

// Формирование выражения фильтра
// Фильтрация ведется по совпадению первой буквы фамилии
procedure TForm1.TabControl1Change(Sender: TObject);
begin
Table1.Filter:='Name = '''+TabControl1.Tabs[TabControl1.TabIndex]+'*''';
end;
```

Здесь при выборе закладки автоматически формируется выражение фильтра, включающее в себя название (букву) выбранной закладки и символ `*`. Фильтрация выполняется независимо от регистра букв и по частичному совпадению, для чего параметры фильтра содержат значение `foCaseInsensitive` и не содержат `foNoPartialCompare`.

При включении фильтрации путем установки свойства `Filtered` в значение `True` для каждой отбираемой в набор данных записи генерируется событие `OnFilterRecord` типа `TFilterRecordEvent`. В процессе отбора записей набор данных автоматически переводится в режим `dsFilter`, при этом в нем запрещаются действия, связанные с изменением записей. Тип `TFilterRecordEvent` описан следующим образом:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
var Accept: Boolean) of object;
```

Логический параметр `Accept` указывает, включать ли в определяемый параметром `DataSet` набор данных запись, для которой сгенерировано событие `OnFilterRecord`. По умолчанию параметр `Accept` имеет значение `True`, и в набор данных включаются все записи, удовлетворяющие условию фильтра, определяемому свойством `Filter`. При установке параметра `Accept` в значение `False` запись в набор данных не включается.

В обработчике события `OnFilterRecord` можно определять дополнительные к выражению фильтра условия фильтрации. По своему действию основные и дополнительные условия как бы соединены логической операцией `AND`, т. е. для отбора записи в набор данных требуется соблюдение обоих условий. В отличие от выражения фильтра, в обработчике события `OnFilterRecord` можно кодировать любые сколь угодно сложные проверки с помощью средств языка Delphi.

Таким образом, набор данных `Table` допускает два способа задания условий фильтрации: с помощью выражения фильтра `Filter` и в обработчике события `OnFilterRecord`.

Замечание

Обработчик события `OnFilterRecord` вызывается для каждой записи, считываемой в набор данных, поэтому код его должен быть коротким и оптимизированным, чтобы не снижать производительность приложения в целом.

Событие `OnFilterRecord` генерируется для каждой записи также при использовании методов поиска `FindFirst`, `FindLast`, `FindNext` и `FindPrior` (см. в данной главе ниже), независимо от значения свойства `Filtered`.

В случае набора данных `Query` для отбора записей можно использовать:

- SQL-запрос;
- обработчик события `OnFilterRecord`;
- выражение фильтра.

Напомним, что для связанных таблиц на отбор записей в набор данных также влияет ограничение, налагаемое отношением "главный-подчиненный" между таблицами БД.

Фильтрация по диапазону

При фильтрации по диапазону в набор данных включаются записи, значения полей которых попадают в заданный диапазон, т. е. условием фильтрации является выражение вида `значение > нижней границы AND значение < верхней границы` (вместо операторов сравнения `<` и `>` можно использовать операторы `<=` и `>=`). Такая фильтрация применяется к наборам данных `Table`.

Достоинством фильтрации по диапазону является высокая скорость обработки записей. В отличие от фильтрации по выражению, когда последовательно просматриваются все записи таблицы, фильтрация по диапазону ведется индексно-последовательным методом, поэтому этот способ фильтрации применим только для индексированных полей. Индекс поля, диапазон которого задан в качестве критерия для отбора записей, должен быть установлен как текущий с помощью свойства `IndexName` или `IndexFieldNames`. Если текущий индекс не установлен, то по умолчанию используется главный индекс.

Для *включения* и *выключения* фильтрации по диапазону применяются методы `ApplyRange` и `CancelRange`. Первый из них активизирует фильтр, а второй — деактивизирует. Предварительно для индексного поля (полей), по которому выполняется фильтрация, следует задать диапазон допустимых значений.

Методы `SetRangeStart` и `SetRangeEnd` устанавливают нижнюю и верхнюю границу диапазона соответственно. Названные процедуры не имеют параметров, и для задания границ диапазона используется просто инструкция присваивания. При этом методы `SetRangeStart` и `SetRangeEnd` переводят набор данных в режим `dsSetKey`.

Для изменения предварительно установленных границ диапазона предназначены методы `EditRangeStart` и `EditRangeEnd`, действие которых аналогично действию методов `SetRangeStart` и `SetRangeEnd` соответственно.

Совместно с этими методами используется свойство `KeyExclusive` типа `Boolean`, которое определяет, как учитывается заданное граничное значение при анализе записей. Если свойство `KeyExclusive` имеет значение `False` (по умолчанию), то записи, у которых значения полей фильтрации совпадают с границами диапазона, включаются в состав набора данных, если же свойство имеет значение `True`, то такие записи в набор данных не попадают. Свойство `KeyExclusive` действует отдельно для нижней и верхней границы. Значение этого свойства должно устанавливаться сразу после вызова методов `EditRangeStart`, `EditRangeEnd`, `SetRangeStart` и `SetRangeEnd`.

Вот как в программе устанавливается нижняя граница диапазона:

```
Table1.SetRangeStart;
Table1.KeyExclusive := True;
Table1.FieldByName('Price').AsFloat := 10.75;
```

Индекс по полю `Price` должен быть текущим. Записи, содержащие в поле цены (`Price`) значение `10.75`, не входят в отфильтрованный набор данных, поскольку свойству `KeyExclusive` присвоено значение `True`. Отметим, что инструкция присваивания, выполняемая после вызова метода `SetRangeStart`, не меняет значение поля текущей записи, а устанавливает нижнюю границу диапазона.

Когда одна из границ диапазона не задана, то диапазон открыт, т. е. нижняя граница становится равной минимальному возможному, а верхняя граница — максимальному возможному значению этого поля.

Если фильтрация выполняется одновременно по нескольким полям, то после вызова методов `SetRangeStart` или `SetRangeEnd` должны стоять несколько инструкций присваивания, каждая из которых определяет границу по одному полю. Предварительно в качестве текущего должен быть установлен индекс, построенный по этим полям.

Вот пример процедуры, в которой фильтрация осуществляется по нескольким полям:

```
procedure TForm1.btnFilterClick(Sender: TObject);
begin
with Table1 do begin
     IndexName := 'indNameBirthDay';
     SetRangeStart;
     FieldByName('Name').AsString := 'А';
     FieldByName('BirthDay').AsString := Edit1.Text;
     SetRangeEnd;
     FieldByName('Name').AsString := 'Б';
     ApplyRange;
     end;
end;
```

Здесь для фильтрации одновременно используются поля `Name` и `BirthDay`. Набор данных содержит только записи, соответствующие сотрудникам, которые родились позже даты, введенной в поле редактирования `Edit1`, и с фамилиями, которые начинаются с букв `А` и `Б`. Верхняя граница для даты рождения не задается.

Неправильный ввод даты в поле редактирования `Edit1` является ошибкой и при попытке присвоить это значение полю даты вызывает исключение. Для предотвращения ошибок, связанных с набором даты, можно использовать компонент `DateTimePicker` или проверять вводимые в поле редактирования символы и блокировать неправильный ввод, например, с помощью обработчика события `OnKeyPress`.

В качестве примера рассмотрим обработчики событий модуля формы приложения, обеспечивающего фильтрацию по диапазону (рис. 7.7).

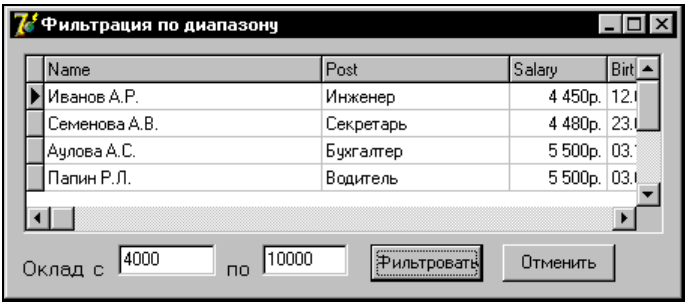


Рис. 7.7. Фильтрация по диапазону

Ниже приведены обработчики событий модуля формы `Form1` приложения.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
Table1.IndexName := 'indSalary';
   end;

// Фильтрация по полю Salary
// Индекс по этому полю должен быть установлен как текущий
procedure TForm1.btnFilterClick(Sender: TObject);
begin
with Table1 do begin
     // Установка нижней границы диапазона
     SetRangeStart;
     FieldByName('Salary').AsString := edtRangeMin.Text;
     // Установка верхней границы диапазона
     SetRangeEnd;
     FieldByName('Salary').AsString := edtRangeMax.Text;
     // Включение фильтрации
     ApplyRange;
     end;
   end;

// Отключение фильтрации
procedure TForm1.btnNoFilterClick(Sender: TObject);
begin
Table1.CancelRange;
end;
```

Пользователь может отбирать записи, задавая нижнюю и верхнюю границы оклада. Фильтрация выполняется по полю `Salary`. Предварительно индекс по этому полю должен быть установлен как текущий, в противном случае при попытке фильтрации по диапазону возникнет исключение.

Метод `SetRange` объединяет возможности методов `SetRangeStart`, `SetRangeEnd` и `ApplyRange`. Процедура `SetRange (const StartValues, EndValues: array of const)` позволяет одновременно задать границы диапазона и выполнить фильтрацию. Параметры `StartValues` и `EndValues` являются массивами констант и содержат значения для нижней и верхней границы диапазона соответственно. Если фильтрация выполняется по нескольким полям, то граничные значения для этих полей перечисляются в параметрах `StartValues` и `EndValues` через запятую. Например:

```
// Должен быть установлен текущий индекс,
// построенный по полям должности и оклада
Table1.SetRange(['Преподаватели', 1200], ['Преподаватели',  1500]);
```

В набор данных попадают записи о преподавателях, имеющих оклад от 1200 до 1500 рублей. Поля, по которым выполняется фильтрация и для значений которых устанавливаются границы диапазона, в явном виде не указываются, а являются полями текущего индекса. Соответствующий текущий индекс должен быть установлен до вызова метода `SetRange`.

Для *отмены фильтрации*, выполненной с помощью методов `ApplyRange` или `SetRange`, используется метод `CancelRange`. Другим вариантом отмены предыдущей фильтрации является задание новых границ диапазона, например, методами `SetRangeStart` и `SetRangeEnd`.

Замечание

Если текущий индекс был изменен, то фильтрация по предыдущему индексу перестает действовать, и в наборе данных будут видимы все записи. При возвращении к прежнему текущему индексу фильтрация также не действует, несмотря на то, что диапазон был задан при предыдущей фильтрации.

При выполнении фильтрации по диапазону возможен отбор записей по частичному совпадению значений символьных полей, когда задаются только начальные символы строки. В отличие от фильтрации по выражению, при фильтрации по диапазону отсутствует свойство, аналогичное `FilterOptions` с параметром `foCaseInsensitive`.

В качестве нижней границы указываются символы, являющиеся началом строки, а для получения значения верхней границы можно добавить справа от них строку `яяя`. Как правило, символьные значения в полях для фильтрации не могут содержать несколько идущих подряд букв `я`, а код прописной буквы `я` больше, чем коды других букв, в результате в набор данных попадут записи, поля которых содержат значения, начинающиеся с заданных символов. Если в полях таблицы могут оказаться строки, включающие в себя добавленную строку `яяя`, то можно увеличить количество букв `я` в строке, добавляемой при формировании верхней границы диапазона. При использовании осмысленных слов вероятность такого совпадения близка к нулю, в крайнем случае, в отфильтрованном наборе данных может оказаться несколько лишних записей.

Чтобы продемонстрировать фильтрацию с проверкой на частичное совпадение значений, изменим обработчики событий для записной книжки (см. рис. 7.7) так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
DBGrid1.Align := alClient;
// Главный ключ (по полю Name) используется как текущий индекс
Table1.IndexFieldNames := 'Name';
// Первоначальное задание границ диапазона (для буквы А)
// TabControl1Change(Sender);
// Отключение фильтрации
CheckBox1.Checked := True; CheckBox1Click(Sender);
   end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
if CheckBox1.Checked
   then Table1.CancelRange
   else TabControl1Change(Sender);
   end;

// Задание границ диапазона
// Фильтрация по совпадению первой буквы фамилии
procedure TForm1.TabControl1Change(Sender: TObject);
var Letter: string[1];
begin
Letter := TabControl1.Tabs[TabControl1.TabIndex];
Table1.CancelRange;
Table1.SetRange([Letter], [Letter + 'яяя']);
end;
```

В случае набора данных `Query`, используя средства SQL, можно отбирать записи по частичному совпадению не только начальных символов строки, но и по вхождению заданных символов в любое место строки.

Навигация с псевдофильтрацией

Система Delphi предоставляет возможность перемещения по набору данных, в котором фильтрация выключена, как по отфильтрованному. Методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior` перемещают указатель текущей записи соответственно на первую, последнюю, следующую и предыдущую записи, удовлетворяющие условиям фильтрации.

Условия фильтрации должны быть заданы предварительно с помощью выражения фильтра `Filter` и/или обработчика события `OnFilterRecord`. При этом фильтр может быть выключен путем установки свойства `Filtered` в значение `False`.

На время действия каждого из названных методов набор данных автоматически переводится в режим `dsFilter`, в результате чего записи *временно* фильтруются, и осуществляется переход на требуемую запись. Связанные с набором данных визуальные компоненты на время фильтрации отображают прежний состав записей набора данных. В результате пользователь не видит хода временной фильтрации, хотя ему запрещено изменять записи, пока набор данных находится в режиме `dsFilter`.

Таким образом, методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior` обеспечивают навигацию по записям, удовлетворяющим условиям фильтра, в неотфильтрованном наборе данных.

Замечание

При действии методов `FindFirst`, `FindLast`, `FindNext` и `FindPrior` используется механизм фильтрации, последовательно перебирающий все записи набора данных, поэтому их применение эффективно только для относительно небольших наборов данных.

На время выполнения названных методов набор данных автоматически переводится в режим `dsFilter`, поэтому если до вызова любого из методов имелись записи, изменения в которых не были подтверждены, то изменения теряются.

Поиск записей

Поиск записи, удовлетворяющей определенным условиям, означает переход на эту запись. Поиск во многом похож на фильтрацию, т. к. в процессе поиска также выполняется проверка полей записей по некоторым условиям. Отличие заключается в том, что в результате поиска количество записей в наборе данных не изменяется.

При организации поиска записей важное значение имеет наличие индекса для полей, по которым ведется поиск. Индексирование значительно повышает скорость обработки данных, кроме того, ряд методов может работать только с индексированными полями.

Далее мы рассмотрим средства, с помощью которых можно выполнить поиск записей в наборах данных `Table` и `Query`. Отметим, что к средствам поиска можно отнести также методы `FindFirst`, `FindLast`, `FindNext` и `FindPrior`, осуществляющие переход на записи, удовлетворяющие условиям фильтра.

Поиск в наборах данных

Для поиска записей по полям служат методы `Locate` и `Lookup`, причем поля могут быть неиндексированными.

Функция `Locate (const KeyFields: String; const KeyValues: Variant; Options: TLocateOptions): Boolean` ищет запись с заданными значениями полей. Если удовлетворяющие условиям поиска записи существуют, то указатель текущей записи устанавливается на первую из них. Если запись найдена,

функция возвращает значение `True`, в противном случае — значение `False`. Список полей, по которым ведется поиск, задается в параметре `KeyFields`, поля разделяются точкой с запятой. Параметр `KeyValues` типа `Variant` указывает значения полей для поиска. Если поиск ведется по одному полю, то параметр содержит одно значение, соответствующее типу поля, заданного для поиска.

Параметр `Options` позволяет задать значения, которые обычно используются при поиске строк. Этот параметр принадлежит к множественному типу `TLocateOptions` и принимает комбинации следующих значений:

- `loCaseInsensitive` (регистр букв не учитывается);
- `loPartialKey` (допускается частичное совпадение значений).

Отметим, что тип `TLocateOptions` по сути похож на тип `TFilterOptions`, определяющий параметры фильтрации по выражению, но значения `loPartialKey` и `foNoPartialCompare` имеют противоположное действие: первое из них допускает, а второе запрещает частичное совпадение значений.

Замечание

При наличии у параметра `Options` значения `loPartialKey` к нему автоматически добавляется значение `loCaseInsensitive`.

Пример поиска по одному полю:

```
Table1.Locate('Number', 123, []);
```

Поиск выполняется по полю `Number` и ищется первая запись, для которой значением этого поля является число `123`. Все параметры поиска отключены. Возвращаемый методом `Locate` результат не анализируется.

При поиске *по нескольким полям* в методе `Locate` параметр `KeyValues` является массивом значений типа `Variant`, в котором содержится несколько элементов. Для приведения к типу вариантного массива используется функция `VarArrayOf`. Значения разделяются запятыми и заключаются в квадратные скобки, порядок значений должен соответствовать порядку полей параметра `KeyFields`. Вот пример:

```
Table1.Locate('Name;Post;', VarArrayOf(['П', 'Инженер']),
[loCaseInsensitive, loPartialKey]);
```

Поиск выполняется по полям `Name` и `Post`, ищется первая запись, для которой значение поля фамилии начинается с букв `п` или `П`, а значение поля должности содержит строку `Инженер`. Регистр букв значения не имеет. Результат поиска не анализируется.

Обычно при разработке приложений пользователю предоставляется возможность влиять на процесс поиска с помощью элементов управления, расположенных в форме. При этом действия пользователя по управлению поиском в наборе данных мало чем отличаются от аналогичных действий при выполнении фильтрации.

Замечание

Если имя поля или тип значения заданы неправильно, то при попытке выполнить метод `Locate` генерируется исключение.

Метод `Locate` позволяет вести поиск по любым полям, однако если поля индексированы или являются частью некоторого индекса, то соответствующий индекс при поиске используется автоматически. При этом также автоматически записи набора данных сортируются по указанному индексу. В результате, если условиям поиска удовлетворяет несколько записей, то будет найдена и установлена текущей первая в порядке сортировки запись. При использовании другого индекса порядок расположения записей, удовлетворяющих условиям поиска, может измениться, и будет найдена другая запись.

В качестве примера рассмотрим обработчик события, обеспечивающий поиск в наборе данных. Вид формы приложения приведен на рис. 7.8.

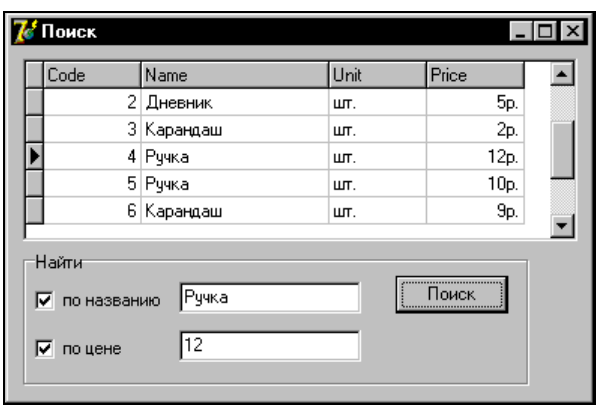


Рис. 7.8. Вид диалогового окна поиска в наборе данных

Пользователь имеет возможность осуществлять поиск по полям названия (`Name`) и цены (`Price`) товара. Процесс поиска начинается при нажатии кнопки **Поиск** (`btnFind`). Ниже приведен текст обработчика события нажатия этой кнопки.

```
procedure TForm1.btnFindClick(Sender: TObject);
var KeyFields:   String;
    KeyValues:   Variant;
      Options:   TLocateOptions;
begin
if not (cbFindName.Checked or cbFindPrice.Checked) then begin
   MessageDlg('Не заданы условия поиска!', mtInformation, [mbOK], 0);
   exit;
   end;
// Поиск одновременно по двум полям Name и Price
```

```
if cbFindName.Checked and cbFindPrice.Checked then begin
   KeyFields := 'Name;Price';
   KeyValues := VarArrayOf([edtFindName.Text, edtFindPrice.Text]);
   end
// Поиск по одному из двух полей
   else begin
      // По полю Name
      if cbFindName.Checked then begin
         KeyFields := 'Name';
         KeyValues := edtFindName.Text;
         end;
      // По полю Price
      if cbFindPrice.Checked then begin
         KeyFields := 'Price';
         KeyValues := edtFindPrice.Text;
         end;
      end;
// Поиск выполняется независимо от регистра букв
// с возможностью частичного совпадения
Options := [loCaseInsensitive, loPartialKey];
// Запись не найдена
if not Table1.Locate(KeyFields, KeyValues, Options) then begin
   Beep;
   MessageDlg('Запись не найдена!', mtInformation, [mbOK], 0);
   exit;
   end;
// Здесь могут быть выполнены действия по обработке найденной записи
end;
```

Включать или не включать поля в поиск — определяется с помощью флажков `cbFindName` и `cbFindPrice`, значения для поиска вводятся в полях редактирования `edtFindName` и `edtFindPrice`.

Если включен параметр `loPartialKey`, то в процессе поиска допускаются частичные совпадения значений. Используя это, можно организовать поиск путем последовательного приближения к требуемой записи. Такой последовательный поиск реализован, например, на страницах **Предметный указатель** и **Поиск** окна справочной системы Windows, где после ввода очередного символа выполняется автоматический переход на подходящую строку. Последовательный поиск можно организовать также с помощью методов `FindNearest`, `SetNearest`, `EditNearest` и `GotoNearest`, предназначенных для поиска с частичным совпадением заданных для поиска значений и значений полей записей.

Рассмотрим в качестве примера обработчики событий формы приложения, используемой для последовательного поиска записей (рис. 7.9).

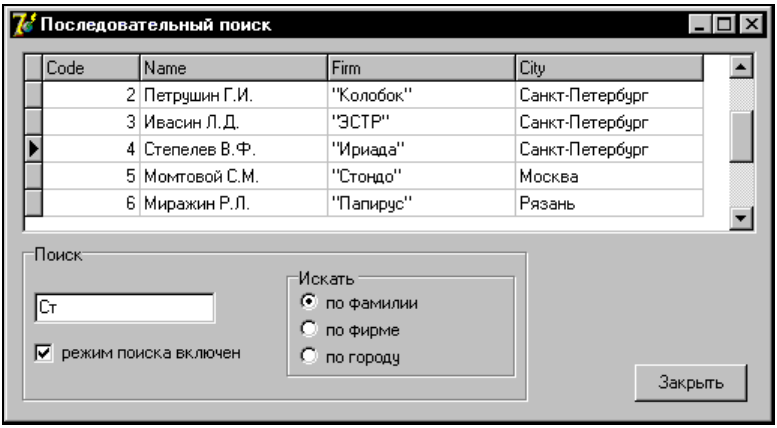


Рис. 7.9. Вид диалогового окна для последовательного поиска

Ниже приведен код обработчиков событий для компонентов формы `Form1` приложения.

```
procedure TForm1.edtSearchChange(Sender: TObject);
var strField: String;
begin
// Если режим поиска выключен, то выйти из процедуры
if not cbSearch.Checked then exit;
// Выбрать поле для поиска
case RadioGroup1.ItemIndex of
     0: strField := 'Name';
     1: strField := 'Firm';
     2: strField := 'City';
     end;
// Выполнить поиск
Table1.Locate(strField, edtSearch.Text,
      [loCaseInsensitive, loPartialKey]);
   end;

procedure TForm1.cbSearchClick(Sender: TObject);
begin
edtSearchChange(Sender);
   end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
edtSearchChange(Sender);
end;
```

Здесь при установке флажка `cbSearch` включается режим поиска, и указатель текущей записи перемещается на запись, удовлетворяющую условиям поиска. Поле, по которому ведется поиск, выбирается с помощью группы переключателей. Процесс поиска запускается каждый раз при изменении содержимого редактора `edtSearch`.

Для поиска в наборе данных также используется метод `Lookup`, который работает аналогично методу `Locate`. Функция `Lookup (const KeyFields: String; const KeyValues: Variant; const ResultFields: String): Variant` осуществляет поиск записи, удовлетворяющей определенным условиям, но, в отличие от метода `Locate`, не перемещает указатель текущей записи на найденную запись, а считывает информацию из полей записи. Еще одно различие между двумя методами заключается в том, что метод `Lookup` осуществляет поиск на *точное* соответствие значений для поиска и значений в полях записей с учетом регистра букв.

Параметры `KeyFields` и `KeyValues` имеют такое же назначение, как и в методе `Locate`, и используются аналогичным образом.

В параметре `ResultFields` через точку с запятой перечисляются названия полей, значения которых будут получены в случае успешного поиска. Эти значения считываются из первой найденной записи, удовлетворяющей условиям поиска. Порядок перечисления полей в `ResultFields` может отличаться от порядка полей в наборе данных. Например, если набор данных имеет поля `Code`, `Name`, `Salary` и `Note`, то в `ResultFields` можно задать `Salary` и `Name`.

В случае удачного поиска метод `Lookup` в качестве результата возвращает значение типа `Variant`, размерность которого зависит от списка полей `ResultFields`. Если список содержит одно значение, то метод возвращает значение одного поля, если в списке задано несколько полей, то метод возвращает массив `Variant`, число элементов которого совпадает с числом полей в списке `ResultFields`.

Замечание

Перед использованием результирующие значения элементов массива `Variant` следует самостоятельно преобразовать к требуемому типу, например, `String`, `Real` или `Integer`.

Для работы с массивом `Variant`, число элементов которого заранее неизвестно, предназначены следующие функции:

- `VarIsArray (const V: Variant): Boolean` — проверяет, является ли параметр `V` массивом типа `Variant`;
- `VarArrayLowBound (const A: Variant; Dim: Integer): Integer` — возвращает нижнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива;
- `VarArrayHighBound (const A: Variant; Dim: Integer): Integer` — возвращает верхнюю границу массива, заданного параметром `A`, параметр `Dim` определяет размерность массива.

При неудачном поиске метод `Lookup` возвращает значение `Null`. Для анализа такого результата можно использовать функцию `VarIsNull (const V: Variant): Boolean`, возвращающую значение `True` при значении параметра `V`, равном `Null`.

Замечание

Не следует путать `Null`, соответствующее пустому (нулевому) значению, и `Nil`, означающее отсутствие значения.

Замечание

Если из найденной записи считываются значения нескольких полей, то даже в случае успешного поиска часть полей может быть не определена и содержать пустые значения. Поэтому при операциях с элементами массива `Variant` следует проверять значение каждого из них на равенство `Null`.

Рассмотрим, как проводится анализ значений полей найденной записи:

```
procedure TForm1.btnFindClick(Sender: TObject);
var KeyFields, ResultFields:    String;
      KeyValues, vrntResult:    Variant;
                  strResult:    String;
begin
KeyFields := 'Name';
KeyValues := edtFindName.Text;
// Порядок полей результата отличается от порядка полей набора данных
ResultFields := 'Name;Price;Code';
// Выполнение поиска
vrntResult := Table1.Lookup(KeyFields, KeyValues, ResultFields);
// Проверка, успешный ли поиск
if not VarIsNull(vrntResult) then begin
// Поиск завершился успешно
   // Анализ поля Name
   if not VarIsNull(vrntResult[0])
      then strResult := 'Название: ' + String(vrntResult[0])
      else strResult := 'Название: -';
   // Анализ поля Price
   if not VarIsNull(vrntResult[1])
      then strResult := strResult + #13#10 + 'Цена: ' +
           String(vrntResult[1])
      else strResult := strResult + #13#10 + 'Цена: -';
   // Анализ поля Code
   if not VarIsNull(vrntResult[2])
      then strResult := strResult + #13#10 + 'Код: ' +
           String(vrntResult[2])
```

```
      else strResult := strResult + #13#10 + 'Код: -';
   // Вывод значений полей найденной записи
   MessageDlg('Данные найденной записи:' + #13#10 + strResult,
                     mtInformation, [mbOK], 0);
   end;
// Поиск завершился неудачно
   else begin
        Beep;
        MessageDlg('Запись не найдена!', mtInformation, [mbOK], 0);
        end;
end;
```

В примере поиск ведется по полю названия товара (`Name`), а из найденной записи считываются значения полей названия (`Name`), цены (`Price`) и кода (`Code`). Порядок считываемых полей отличается от порядка этих полей в таблице БД. Значение, по которому ищется запись, вводится в редакторе `edtFindName`. После поиска выполняется анализ его успешности, и при положительном результате выводится диалоговое окно с данными полей найденной записи. Если запись не найдена, то выводится соответствующее сообщение. Перед анализом каждого из возвращенных значений проверяется его равенство `Null` (пустое значение). Если поле не пустое, то тип `Variant` преобразовывается в строковый.

Поиск по индексным полям

Для набора данных `Table` имеются методы, позволяющие вести поиск записей только по индексным полям. Перед вызовом любого из этих методов следует установить в качестве текущего индекс, построенный по используемым для поиска полям. Методы поиска можно разделить на две группы, в первую из которых входят методы `FindKey`, `SetKey`, `EditKey` и `GotoKey`, предназначенные для поиска на точное соответствие, а другую образуют методы `FindNearest`, `SetNearest`, `EditNearest` и `GotoNearest`, допускающие только частичное совпадение заданных для поиска значений и значений полей записей.

Метод `FindKey (const KeyValues: array of const): Boolean` выполняет поиск в наборе данных `Table` записи, у которой значения полей совпадают со значениями, указанными в параметре `KeyValues`. Список полей для поиска не задается, а берутся индексные поля в соответствии с текущим индексом. Если поиск завершился успешно, то найденная запись становится текущей, а метод возвращает значение `True`. При неудачном поиске указатель текущей записи не перемещается, а метод возвращает значение `False`.

Если заданным условиям поиска соответствует несколько записей, то указатель текущей записи устанавливается на первую из них. Порядок сортировки и, соответственно, порядок расположения записей в наборе данных определяется текущим индексом, по которому и ведется поиск.

Например:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
Table1.IndexFieldNames := 'City;Date';
if not Table1.FindKey ([edtFindCity.Text, edtFindDate.Text])
   then ShowMessage('Запись не найдена!');
end;
```

Здесь в наборе данных выполняется поиск первой записи, поля `City` и `Date` которой содержат значения, введенные в редакторы `edtFindCity` и `edtFindDate`. В случае неудачного поиска пользователю выдается соответствующее сообщение. Перед выполнением поиска текущим устанавливается индекс, построенный по поисковым полям.

Вместо метода `FindKey` можно вызывать методы `SetKey`, `EditKey` и `GotoKey`, которые применяются совместно. Их использование похоже на применение рассмотренных ранее методов `SetRangeStart`, `EditRangeStart` и `ApplyRange` для фильтрации набора данных по диапазону значений.

Метод `SetKey` переводит набор данных в режим поиска записи `dsSetKey`; этот метод вызывается один раз для текущего индекса. Впоследствии для перехода в режим поиска можно вызывать метод `EditKey`. Если набор данных находится в режиме поиска, значения полей устанавливаются с помощью инструкций присваивания. Метод `GotoKey: Boolean` выполняет собственно поиск записи, удовлетворяющей заданному условию. Использовать комбинацию названных методов менее удобно, чем метод `FindKey`. Вот еще один пример, иллюстрирующий точный поиск:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
Table1.IndexFieldNames := 'City';
// Впоследствии вместо этого метода можно вызывать EditKey
Table1.SetKey;
Table1.FieldByName('City').AsString := edtFindCity.Text;
if not Table1.GotoKey
   then ShowMessage('Запись не найдена!');
end;
```

Приведенная процедура обработки события нажатия кнопки выполняет точный поиск записей по полю города (`City`).

Метод `FindNearest`, в отличие от метода `FindKey`, производит поиск значений полей записей набора данных `Table`, которые только частично совпадают со значениями, заданными для поиска. Сравнение проводится, начиная с первого стоящего в поле символа. Поиск по частичному совпадению можно применять к строкам или к данным других типов, например, целым. Поиск с помощью про-

цедуры `FindNearest (const KeyValues: array of const)` всегда является успешным и перемещает указатель текущей записи на запись, в наибольшей степени отвечающую условиям поиска.

Вместо метода `FindNearest` можно использовать комбинацию методов `SetNearest`, `EditNearest` и `GotoNearest`, работа с которыми аналогична работе с соответствующими методами поиска на точное соответствие значений полей.

Например:

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
Table1.IndexFieldNames := 'Name';
Table1.FindNearest (['И']);
end;
```

В приведенной процедуре указатель текущей записи перемещается на запись, поле фамилии которой содержит значение, начинающееся с буквы `И`. Если такой фамилии нет, то будет найдена фамилия, начинающаяся с ближайшей к `И` следующей буквы.

При поиске по нескольким полям текущего индекса используется свойство `KeyFieldCount` типа `Boolean`, указывающее, сколько полей индекса, начиная с первого, участвует в поиске. По умолчанию поиск осуществляется по всем полям текущего индекса.

Модификация набора данных

Модификация (изменение) набора данных представляет собой редактирование, добавление и удаление его записей. Модифицируемость набора данных зависит от различных условий. Разработчик может разрешить или запретить изменение набора данных с помощью соответствующих свойств.

Управлять возможностью изменения набора данных `Table` можно посредством свойства `ReadOnly` типа `Boolean`, при установке которого в значение `True` изменение записей запрещается. По умолчанию свойство `ReadOnly` имеет значение `False`, и набор данных можно модифицировать.

Замечание

Значение свойства `ReadOnly` можно изменять только у закрытого набора данных.

В отличие от многих элементов управления, например, редакторов `Edit` и `Memo`, данный запрет на редактирование относится как к визуальному (пользователем), так и к программному изменению записей набора данных.

Возможность модификации набора данных `Query` определяет свойство `RequestLive` типа `Boolean`. По умолчанию это свойство имеет значение `False`, и набор данных `Query` доступен только для чтения. Чтобы получить разрешение визуально и программно редактировать записи, свойство `RequestLive` нужно

установить в значение `True`. Кроме этого свойства, возможность изменения набора `Query` зависит также от содержания SQL-запроса. Например, если при запросе отбираются записи из нескольких таблиц, то набор данных не может быть модифицируемым, и значение `True` свойства `RequestLive` не учитывается.

Для проверки, можно ли изменять набор данных, предназначено свойство `CanModify` типа `Boolean`, действующее при выполнении приложения и доступное только для чтения. Если это свойство имеет значение `True`, то набор данных изменять можно, а если `False`, то изменения в наборе данных запрещены, и любая попытка сделать это визуально или программно вызовет исключение.

Замечание

Можно запретить редактирование отдельных полей набора данных даже в том случае, если он является модифицируемым.

Для программного изменения набора данных вызываются соответствующие методы, например, метод `Edit` редактирования текущей записи или метод `Append` вставки новой записи.

Пользователь редактирует набор данных с помощью визуальных компонентов, например, редактора `DBEdit` или сетки `DBGrid`, управляя ими с помощью мыши и клавиатуры. Набор данных может автоматически переводиться в режимы редактирования или вставки, для этого свойство `AutoEdit` источника данных `DateSource` для визуальных компонентов должно быть установлено в значение `True` (по умолчанию). Если это свойство установить в значение `False`, то пользователь не сможет изменять набор данных с помощью визуальных компонентов.

Замечание

Свойство `AutoEdit` влияет на визуальные компоненты, подключенные к источнику данных `DateSource`, и не оказывает никакого влияния на другие элементы управления, такие как, например, кнопка `Button` или флажок `CheckBox`.

При модификации набора данных для связанного с ним источника данных `DateSource` генерируется событие `OnUpdateData`.

После модификации набора данных возможна ситуация, когда изменения сделаны, но не отображены визуальными компонентами, связанными с этим набором. В таких случаях нужно вызывать метод `Refresh`, который повторно считывает набор данных и тем самым гарантирует, что визуальные компоненты будут отображать текущие, а не устаревшие данные. При работе в многопользовательской системе рекомендуется периодически вызывать метод `Refresh`, чтобы своевременно учитывать изменения, сделанные другими пользователями.

В однопользовательском приложении обновление набора данных применяется в случае, если с одной таблицей связано *несколько* наборов данных. Например, с таблицей клиентов может быть связан набор данных, с помощью которого выполняется редактирование списка клиентов, а также набор данных, предназначенный для выбора клиента при вводе информации в расходную накладную.

Компоненты Table обоих наборов могут находиться в разных формах. После редактирования списка клиентов следует обновить оба набора данных, в противном случае возможна ситуация, когда данные о новом клиенте не будут доступны для ввода в накладную. Вот фрагмент кода, в котором проводится обновление набора данных:

```
Table1.Edit;
Table1.FieldByName('Name').AsString:=Edit1.Text;
Table1.Post;
Table1.Refresh;
Form2.Table2.Refresh;
```

В этом примере при программном изменении набора данных Table1, находящегося в форме Form1, обновляется он сам, а также набор данных Table2, расположенный в форме Form2. Оба набора связаны с одной и той же физической таблицей. В модуле формы Form1 должна быть ссылка на модуль формы Form2.

Редактирование записей

Редактирование записей заключается в изменении значений их полей. Отредактирована может быть только текущая запись, поэтому перед действиями, связанными с редактированием, обычно выполняются операции поиска и перемещения на требуемую запись. После того как указатель текущей записи установлен на нужную запись, и набор данных находится в режиме просмотра, для редактирования записи следует:

1. Перевести набор данных в режим редактирования.
2. Изменить значения полей записи.
3. Подтвердить сделанные изменения или отказаться от них, в результате чего набор данных снова перейдет в режим просмотра.

Набор данных переводится в режим редактирования вызовом метода Edit, при этом возможны такие ситуации:

- если набор данных немодифицируемый, то возбуждается исключение;
- если набор данных уже находился в режиме редактирования или вставки, то никакие действия не выполняются;
- если набор данных пуст, то он переходит в режим вставки.

Если набор данных является модифицируемым и исключение не возбуждается, то при выполнении метода Edit производятся следующие действия:

1. Для набора данных вызывается обработчик события BeforeEdit типа TdataSetNotifyEvent.
2. Из набора данных заново считывается текущая запись и блокируется так, чтобы другие пользователи могли ее читать, но не могли изменять или блокировать. Если операция блокирования завершается неудачно, например, в случае, когда запись уже редактирует другой пользователь, то генерируется

исключение, а выполнение метода `Edit` прекращается. Возникшее на этом шаге исключение можно обработать с помощью обработчика события `OnEditError` типа `TDataSetErrorEvent`, если он назначен.

3. Если в записи есть вычисляемые поля, то они пересчитываются.
4. Набор данных переходит в режим редактирования.
5. Для связанного с набором данных источника данных `DataSource` вызывается обработчик события `OnDataChange`.
6. Для набора данных вызывается обработчик события `AfterEdit` типа `TDataSetNotifyEvent`.

Указанные действия осуществляются только для модифицируемого набора данных, поэтому перед вызовом метода `Edit` следует выполнять проверку на возможность редактирования записи (например, путем анализа свойства `CanModify`). Например:

```
if Table1.CanModify then Table1.Edit;
```

Пользователь осуществляет управление набором данных с помощью расположенных в форме элементов как связанных, так и не связанных с набором. Для отдельных визуальных компонентов, связанных с набором данных, переход в режим редактирования происходит различными способами. Например, для компонентов `DBGrid` и `DBEdit` надо сделать двойной щелчок на нужном поле или нажать алфавитно-цифровую клавишу, когда курсор находится в этом поле, а для компонента `DBNavigator` требуется нажать кнопку **Edit Record**. Таким образом, при управлении визуальными компонентами метод `Edit` вызывается пользователем косвенно. В случае, когда набор данных является немодифицируемым, блокировка перехода в режим его редактирования выполняется автоматически и не приводит к ошибке.

Для компонентов управления, *не связанных* с набором данных, например, кнопок `Button` или флажков `CheckBox` программист должен самостоятельно кодировать действия по предотвращению попыток редактирования немодифицируемого набора данных. Вот пример осуществления такой блокировки:

```
procedure TForm1.btnEditClick(Sender: TObject);
begin
if not Table1.CanModify then begin
   Beep;
   MessageDlg('Редактирование запрещено!', mtInformation, [mbOK], 0);
   exit;
   end;
Table1.Edit;
end;
```

Здесь переход в режим редактирования осуществляется при нажатии кнопки `btnEdit`, которая может иметь заголовок **Редактировать** или **Edit**. Перед переводом в этот режим выполняется проверка, можно ли изменять записи набора

данных Table1, и если нет, то процедура выдает соответствующее сообщение и завершается.

Другой способ предотвратить редактирование — заблокировать элементы управления, для которых в настоящий момент времени невозможно выполнить обработчики событий. Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1. cbEditBanClick(Sender: TObject);
begin
if cbEditBan.Checked then begin
   Table1.Active := False;
   Table1.ReadOnly := True;
   btnEdit.Enabled := False;
   Table1.Active := True;
   end
   else begin
   Table1.Active := False;
   Table1.ReadOnly := False;
   btnEdit.Enabled := True;
   Table1.Active := True;
   end;
end;
```

В ней флажок cbEditBan (с возможным заголовком **Редактирование запрещено**) указывает, допустимо ли изменять записи набора данных Table1. Если этот флажок установлен, то модификация набора данных запрещается, при этом также блокируется кнопка btnEdit вызова метода Edit.

Для блокирования элементов управления можно использовать компонент ActionList, предназначенный для синхронизации элементов управления.

Блокировка попыток пользователя изменить немодифицируемый набор данных должна выполняться также при добавлении и удалении записей.

При выполнении метода Edit непосредственно перед переводом набора данных в режим редактирования возникает событие BeforeEdit, которое можно использовать для проверки возможности перехода в этот режим. Например, при попытке пользователя редактировать запись ему может быть предложено подтвердить свои действия. Для отмены процесса редактирования в обработчике события BeforeEdit можно сгенерировать "тихое" исключение.

При переходе в режим редактирования с помощью кнопки Button проверку его допустимости можно выполнить в обработчике события ее нажатия. Однако обычно удобнее использовать обработчик события BeforeEdit, т. к. оно генерируется при переводе набора данных в режим редактирования любым способом. Вот пример соответствующей процедуры:

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
```

```
if MessageDlg('Выполнить редактирование?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
end;
```

После перевода набора данных в режим редактирования можно с помощью инструкций присваивания изменять значения полей текущей записи. При этом нужно учитывать тип поля, выполняя при необходимости операции приведения типов. Например:

```
Table1.FieldByName('City').AsString := Edit1.Text;
Table1.FieldByName('Code').AsInteger := StrToInt(Edit2.Text);
Table1.FieldByName('Price').AsFloat := StrToFloat(Edit3.Text);
```

Перед выполнением приведенных инструкций набор данных `Table1` должен находиться в режиме редактирования или вставки. Если редактор `Edit2` или `Edit3` содержит данные в формате, не соответствующем целому и вещественному числам, то генерируется исключение.

Для проверки, вносились ли изменения в запись, можно проанализировать свойство `Modified` типа `Boolean`. Если свойство имеет значение `True`, то было изменено значение как минимум одного поля текущей записи.

После ввода информации сделанные изменения должны быть или подтверждены, или отменены.

Метод `Post` *записывает модифицированную запись* в таблицу БД, снимает блокировку записи и переводит набор данных в режим просмотра. Если набор данных не находился в режиме редактирования, то вызов метода `Post` приведет к генерации исключения. Перед его выполнением автоматически вызывается обработчик события `BeforePost` типа `TDataSetNotifyEvent`, а сразу после выполнения — обработчик события `AfterPost` типа `TDataSetNotifyEvent`. Используя событие `BeforePost`, можно проверить сделанные изменения и при необходимости отменить их, например, прервав выполнение метода с помощью вызова "тихого" исключения.

Рассмотрим процедуру, в которой осуществляется редактирование записей:

```
procedure TForm1.btnPriceChangeClick(Sender: TObject);
     var bm1:  TBookmark;
    coeff, x:  real;
begin
// Проверка, является ли набор данных модифицируемым
if not Table1.CanModify then begin
   MessageDlg('Записи изменять нельзя!', mtError, [mbOK], 0);
   exit;
   end;
// Получение коэффициента
try
   coeff := StrToFloat(Edit1.Text);
```

```
   except
   MessageDlg('Неправильный коэффициент!' + #13#10 +
              'Повторите ввод.', mtError, [mbOK], 0);
   if Edit1.CanFocus then Edit1.SetFocus;
    exit;
   end;
// Запоминание позиции текущего указателя
bm1 := Table1.GetBookmark;
// Отключение отображения записей визуальными компонентами
Table1.DisableControls;
// Перебор всех записей
Table1.First;
while not Table1.EOF do begin
      // Чтение цены из очередной записи
      x := Table1.FieldByName('Price').AsFloat;
      // Пересчет цены
      x := x * Coeff;
      // Изменение цены в текущей записи
      Table1.Edit;
      Table1.FieldByName('Price').AsFloat := x;
      Table1.Post;
      // Переход к следующей записи
      Table1.Next;
      end;
// Восстановление позиции текущего указателя
// и отображение записей визуальными компонентами
if Table1.BookmarkValid(bm1) then Table1.GotoBookmark(bm1);
if Table1.BookmarkValid(bm1) then Table1.FreeBookmark(bm1);
Table1.EnableControls;
end;
```

Здесь для всех записей набора данных `Table1` выполняется пересчет поля цены `Price`. Цены умножаются на коэффициент, который введен в редакторе `Edit1`.

Метод `Post` вызывается автоматически при переходе к другой записи с помощью методов `First`, `Last`, `Next` и `Prior`, если набор данных находится в режиме редактирования, и изменения в записях не подтверждены. Поэтому в приведенном примере метод `Post` можно было не вызывать, т. к. сразу после него вызывается метод `Next`. Однако при использовании методов `FindFirst`, `FindLast`, `FindNext` и `FindPrior` неподтвержденные изменения в записях будут потеряны.

Пользователь подтверждает сделанные в записях изменения, управляя соответствующими компонентами, явно или неявно вызывающими метод `Post`. Конкрет-

ные действия пользователя зависят от используемых компонентов. Например, при работе с компонентом DBGrid изменения подтверждаются (фиксируются) при переходе к другой записи или нажатии клавиши <Enter>, а в компоненте DBNavigator можно нажать кнопку **Post Edit** (Подтвердить изменения).

Независимо от способа вызова, метод Post может завершиться неудачно, например, если не заданы значения полей, которые не могут быть пустыми, или значение выходит за установленные для него допустимые пределы. В этом случае набор данных обычно возвращается в состояние, которое было до перехода в режим редактирования. При *ошибке выполнения* метода Post генерируется событие OnPostError типа TDataSetErrorEvent. Кодируя обработчик этого события, можно попытаться исправить ошибку.

Метод Cancel *отменяет изменения*, выполненные в текущей записи, и возвращает набор данных в режим просмотра. При выполнении метода Cancel вызываются обработчики событий BeforeCancel и AfterCancel типа TDataSetNotifyEvent.

Пользователь может отменить сделанные в записях изменения, используя элементы управления компонентов. Например, при работе с сеткой DBGrid изменения отменяются нажатием клавиши <Esc>, а в компоненте DBNavigator — нажатием кнопки **Cancel Edit**.

В случае применения механизма транзакций для отмены изменений сразу в нескольких записях можно обратиться к методу RollBack класса TDateBase.

При *редактировании* текущей записи последовательность инструкций присваивания и вызовов метода Post можно заменить вызовом метода SetFields.

Процедура SetFields (const Values: array of const) устанавливает все или часть значений полей текущей записи. Параметр Values содержит массив значений, которые присваиваются этим полям, при этом порядок значений соответствует порядку полей в наборе данных. Кроме того, должны соответствовать типы полей и типы присваиваемых им значений. Если значений в массиве меньше, чем полей в наборе данных, то эти значения присваиваются первым полям, а оставшиеся поля не изменяются. Чтобы не присваивать значение какому-либо полю, в качестве соответствующего значения задается Nil, в результате поле не изменяется. Если число значений в массиве превышает число полей, то при выполнении метода SetFields генерируется исключение.

Замечание

Повторим еще раз, что значения Nil и Null не равны друг другу и имеют разный смысл. Nil означает отсутствие значения, а Null — пустое (нулевое) значение. Если вместо значения Nil указать значение Null, то поле будет изменено и содержать значение Null.

Если для набора данных использовался Редактор полей, то при выполнении метода SetFields учитывается порядок полей, установленный с помощью этого Редактора. В противном случае принимается порядок полей, определенный при создании таблицы БД.

Метод `SetFields` удобно использовать для изменения значений нескольких полей. После его выполнения набор данных автоматически возвращается в режим просмотра. Пример использования метода `SetFields` в программе:

```
Table1.Edit;
Table1.SetFields ([Nil, 'Иванов П.О.', Nil, 'Переводчик', 650]);
```

Здесь во второе, четвертое и пятое поля текущей записи набора данных `Table1` заносятся фамилия, должность и оклад соответственно. Значения первого и третьего полей этой записи не изменяются.

Добавление записей

Добавлять записи можно только к модифицируемому набору данных, в противном случае будет сгенерировано исключение.

Для добавления записи нужно выполнить следующие действия:

1. Перевести набор данных в режим вставки.
2. Задать значения полей новой записи.
3. Подтвердить сделанные изменения или отказаться от них, после чего набор данных снова переходит в режим просмотра.

Для добавления записей используются методы `Insert`, `InsertRecord`, `Append` и `AppendRecord`.

Метод `Insert` переводит набор данных в режим вставки и добавляет к нему новую пустую запись. Новая запись вставляется в позицию, на которой находится указатель текущей записи. При необходимости перед вызовом метода `Insert` нужно выполнить перемещение текущего указателя в требуемую позицию набора данных.

После перевода набора данных в режим вставки дальнейшие действия по заданию (изменению) значений полей, подтверждению или отмене сделанных изменений не отличаются от аналогичных действий при редактировании записи. При этом для задания или изменения значений полей используются инструкции присваивания и метод `SetFields`, а для подтверждения или отмены изменений — методы `Post` и `Cancel`. Некоторые поля новой записи могут остаться пустыми, если до подтверждения им не были присвоены значения.

Замечание

При переходе в режим вставки к набору данных добавляется пустая запись, значения полей которой не заданы. Если запись добавляется к подчиненному набору данных, связанному с главным набором связью "главный-подчиненный", то индексные поля автоматически получают корректные значения, и программист может не заботиться об их заполнении.

Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
Table1.Insert;
Table1.FieldByName('Name').AsString := Edit1.Text;
Table1.FieldByName('Group').AsString := Edit2.Text;
Table1.Post;
end;
```

Здесь в новой записи задаются значения полей фамилии (`Name`) и группы (`Group`), остальные поля остаются пустыми. В этом и последующих примерах предполагается, что набор данных не связан с главным набором данных, и полям не были автоматически присвоены значения как индексным полям.

Часто удобно устанавливать значения сразу нескольких полей с помощью метода `SetFields`. Напомним, что после выполнения этого метода набор данных автоматически возвращается в режим просмотра, и запись считается включенной в набор данных. Если требуется изменить часть полей новой записи, то нужно использовать редактирование, а не вставку. Например, так:

```
procedure TForm1.btnInsertClick(Sender: TObject);
begin
if not Table1.CanModify then begin
   MessageDlg('Записи изменять нельзя!', mtError, [mbOK], 0);
   exit;
   end;
Table1.Insert;
Table1.SetFields ([Nil, edtName.Text, Nil, edtPost.Text,
IntToStr(edtCode.Text)]);
end;
```

Здесь значения полей новой записи пользователь вводит в редакторах. Первое и третье поля, а также поля с номером больше пяти остаются пустыми. Перед добавлением записи выполняется проверка, можно ли изменять набор данных `Table1`.

Метод `InsertRecord` объединяет функциональность методов `Insert` и `SetFields`, выполняя те же действия, что и их последовательный вызов. Процедура `InsertRecord (const Values: array of const)` вставляет в позицию указателя текущей записи новую запись, задавая значения всех или части ее полей. Например:

```
procedure TForm1.btnInsertClick(Sender: TObject);
begin
Table1.InsertRecord ([Nil, edtName.Text, Nil, edtPost.Text,
       IntToStr(edtCode.Text)]);
end;
```

Методы `Append` и `AppendRecord` отличаются от методов `Insert` и `InsertRecord` тем, что вставляют запись в конец набора данных, а не в позицию указателя текущей записи.

Пользователь управляет набором данных, в том числе вставкой записи, с помощью элементов управления формы. Для компонента DBGrid новая запись добавляется к набору данных при нажатии клавиши <Insert> или при переходе на последнюю запись. Если в форме находится компонент DBNavigator, то новая запись добавляется при нажатии кнопки **Insert Record**.

При добавлении новой записи любым методом возникают события BeforeInsert и AfterInsert типа TDataSetNotifyEvent, а также событие OnNewRecord типа TDataSetNotifyEvent. В обработчиках событий BeforeInsert и OnNewRecord можно выполнить действия, связанные с проверкой набранных пользователем данных или с заполнением (инициализацией) части полей новой записи. Вот пример соответствующей процедуры:

```
procedure TForm1.Table1NewRecord(DataSet: TDataSet);
begin
Table1.FieldByName('Unit').AsString := 'штука';
Table1.FieldByName('NDS').AsString := '20';
end;
```

При утверждении или отмене изменений, связанных с добавлением новой записи, также генерируются события BeforePost и AfterPost или BeforeCancel и AfterCancel.

Удаление записей

Удаление текущей записи выполняет метод Delete, который работает только с модифицируемым набором данных. В случае успешного удаления записи текущей становится следующая запись, если же удалялась последняя запись, то курсор перемещается на предыдущую запись, которая после удаления становится последней. В отличие от некоторых СУБД, в Delphi удаляемая запись действительно удаляется из набора данных.

Обычно метод Delete вызывается для удаления просматриваемой записи, однако с его помощью можно удалить и редактируемую запись. Если набор данных находится в режиме вставки или поиска, то вызов метода Delete аналогичен вызову метода Cancel, отменяя соответственно вставку или поиск записи.

Замечание

Если набор данных пуст, то вызов метода Delete порождает исключение.

При удалении записи генерируются события BeforeDelete и AfterDelete типа TDataSetNotifyEvent. Используя обработчик события BeforeDelete, можно отменить операцию удаления, если не соблюдаются определенные условия.

Если выполнение метода Delete приводит к ошибке, то возбуждается исключение, и генерируется событие OnDeleteError, в обработчике которого можно

выполнить собственный анализ ошибки. Вот небольшая процедура, в которой перед удалением записи запрашивается подтверждение пользователя:

```
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
if MessageDlg('Удалить запись?',
       mtConfirmation, [mbYes, mbNo], 0) = mrYes then Table1.Delete;
end;
```

Удаление нескольких последовательно расположенных записей имеет особенность, связанную с тем, что при вызове метода `Delete` в цикле по перебору удаляемых записей не нужно вызывать методы, перемещающие указатель текущей записи. После удаления текущей записи указатель автоматически перемещается на соседнюю (обычно следующую) запись. Так можно удалить все записи набора данных:

```
procedure TForm1.btnDeleteAllClick(Sender: TObject);
var n: longint;
begin
Table1.Last;
for n := Table1.RecordCount downto 1 do Table1.Delete;
end;
```

В примере перебор записей выполняется с конца набора данных. После удаления текущей записи указатель снова оказывается на последней записи.

Для набора данных `Table` удалить все записи можно также с помощью метода `EmptyTable`, который вызывается в режиме исключительного доступа к таблице БД.

Перед удалением записи часто предварительно выполняется поиск записи (записей), удовлетворяющей заданным условиям. Для отбора группы удаляемых записей используется фильтрация. Метод `Delete` позволяет удалить записи, видимые в наборе данных. Поэтому с помощью фильтрации можно временно оставить в наборе данных записи, которые подлежат удалению, а после удаления фильтрацию отключить.

Пример формы приложения

Обычно визуальные компоненты, предназначенные для редактирования, добавления и удаления записей, группируются на форме и работают взаимосвязанно. Вместе или рядом с этими компонентами часто располагают элементы управления сортировкой, фильтрацией и поиском. Тем самым для пользователя обеспечивается удобство выполнения различных операций с данными.

Для примера рассмотрим форму (рис. 7.10), с помощью которой можно изменять записи таблицы, содержащей сведения о товарах. В информацию о товаре входят уникальный код записи (поле `Code`), название товара (поле `Name`), единица измерения (поле `Unit`), цена единицы (поле `Price`) и примечание (поле

Note). Поле названия и поле цены товара обязательно должны быть заполненными, поле кода — автоинкрементное, и его значение при добавлении новой записи формируется автоматически.

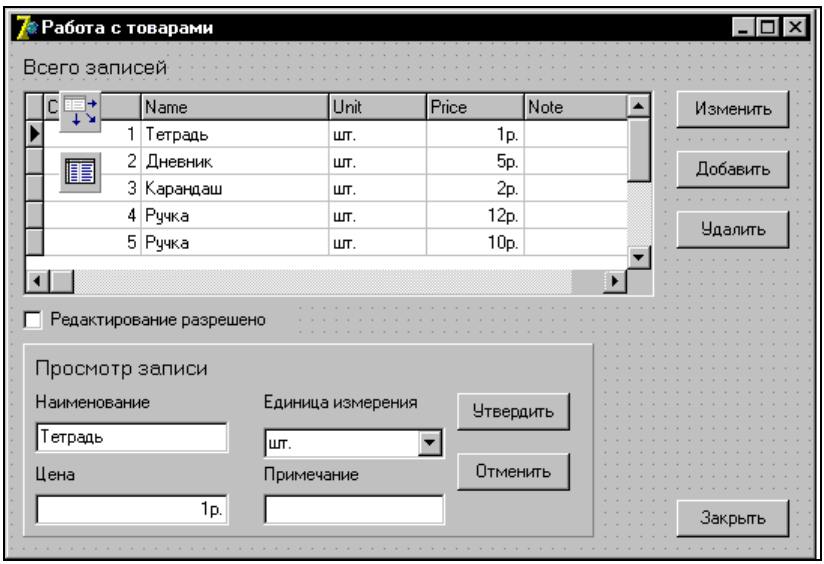


Рис. 7.10. Форма приложения для работы с товарами

Ниже приводится код модуля uChange формы Form1 приложения.

```
  unit uChange;

  interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, Db, DBTables, Grids, DBGrids, StdCtrls, Mask,
     DBCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
        DBGrid1: TDBGrid;
    DataSource1: TDataSource;
         Table1: TTable;
        btnEdit: TButton;
      cbCanEdit: TCheckBox;
      pnlChange: TPanel;
         Label1: TLabel;
        dbeName: TDBEdit;
```

```
       dbePrice: TDBEdit;
        dbeNote: TDBEdit;
         Label2: TLabel;
         Label3: TLabel;
         Label4: TLabel;
  lblChangeKind: TLabel;
    btnChangeOK: TButton;
btnChangeCancel: TButton;
       dbcbUnit: TDBComboBox;
       btnClose: TButton;
 lblRecordCount: TLabel;
      btnInsert: TButton;
      btnDelete: TButton;

    procedure StateBrowse(Sender: TObject);
    procedure StateChange(Sender: TObject);
    procedure btnEditClick(Sender: TObject);
    procedure cbCanEditClick(Sender: TObject);
    procedure Table1BeforeEdit(DataSet: TDataSet);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnChangeOKClick(Sender: TObject);
    procedure btnChangeCancelClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnInsertClick(Sender: TObject);
    procedure Table1BeforePost(DataSet: TDataSet);
    procedure Table1BeforeInsert(DataSet: TDataSet);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
     Form1: TForm1;

   implementation

   {$R *.DFM}

// Режим просмотра
procedure TForm1.StateBrowse(Sender: TObject);
```

```
begin
lblRecordCount.Caption := 'Всего записей — ' + Int-
ToStr(Table1.RecordCount);
cbCanEditClick(Sender);
btnChangeOK.Enabled      := False;
btnChangeCancel.Enabled  := False;
lblChangeKind.Font.Color := clBlack;
lblChangeKind.Caption    := 'ПРОСМОТР ЗАПИСИ';
   end;

// Режимы изменения
procedure TForm1.StateChange(Sender: TObject);
begin
btnEdit.Enabled         := False;
btnInsert.Enabled       := False;
btnDelete.Enabled       := False;
btnChangeOK.Enabled     := True;
btnChangeCancel.Enabled := True;
   end;

procedure TForm1.FormCreate(Sender: TObject);
begin
// Исходное состояние элементов управления
StateBrowse(Sender);
// Первоначально изменение записей запрещено
cbCanEdit.Checked := False;
// Запрет автоматического перехода в режим редактирования
DataSource1.AutoEdit := False;
// Формирование списка единиц измерения товара
dbcbUnit.Items.Clear;
dbcbUnit.Items.Add('шт.');
dbcbUnit.Items.Add('упак.');
dbcbUnit.Items.Add('кор.');
   end;

procedure TForm1.cbCanEditClick(Sender: TObject);
var bm1: TBookmark;
begin
// Запоминание положения текущей записи
bm1 := Table1.GetBookmark;
// Отключение отображения изменений данных в визуальных компонентах
Table1.DisableControls;
if not cbCanEdit.Checked then begin
```

```
   Table1.Active   := False;
   Table1.ReadOnly := True;
   Table1.Active   := True;
   // Блокирование элементов, связанных с переходом
   // в режимы изменения записей
   btnEdit.Enabled   := False;
   btnInsert.Enabled := False;
   btnDelete.Enabled := False;
   end
   else begin
   Table1.Active   := False;
   Table1.ReadOnly := False;
   Table1.Active   := True;
   // Разблокирование элементов, связанных с переходом
   // в режимы изменения записей
   btnEdit.Enabled   := True;
   btnInsert.Enabled := True;
   // Если набор данных пуст, то удаление записей запрещено
   if Table1.RecordCount > 0
      then btnDelete.Enabled := True
      else btnDelete.Enabled := False;
   end;
// Возврат к текущей записи
if Table1.BookmarkValid(bm1) then Table1.GotoBookmark(bm1);
if Table1.BookmarkValid(bm1) then Table1.FreeBookmark(bm1);
// Включение отображения изменений данных в визуальных компонентах
Table1.EnableControls;
   end;

procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
// Запрос на подтверждение перехода в режим редактирования
if MessageDlg('Будете редактировать?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
   end;

// Переход в режим редактирования
procedure TForm1.btnEditClick(Sender: TObject);
begin
Table1.Edit;
lblChangeKind.Font.Color := clRed;
lblChangeKind.Caption    := 'РЕДАКТИРОВАНИЕ ЗАПИСИ';
```

```
StateChange(Sender);
if dbeName.CanFocus then dbeName.SetFocus;
   end;

procedure TForm1.Table1BeforeInsert(DataSet: TDataSet);
begin
// Подтверждение перехода в режим вставки
if MessageDlg('Добавить запись?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Abort;
   end;

// Переход в режим вставки
procedure TForm1.btnInsertClick(Sender: TObject);
begin
Table1.Insert;
lblChangeKind.Font.Color := clBlue;
lblChangeKind.Caption    := 'ВСТАВКА ЗАПИСИ';
StateChange(Sender);
if dbeName.CanFocus then dbeName.SetFocus;
   end;

// Переход в режим просмотра удаляемой записи
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
// Подтверждение перехода в режим просмотра удаляемой записи
if MessageDlg('Удалить запись?',
       mtConfirmation, [mbYes, mbNo], 0) <> mrYes then Exit;
lblChangeKind.Font.Color := clRed;
lblChangeKind.Caption    := 'УДАЛЕНИЕ ЗАПИСИ';
StateChange(Sender);
if btnChangeCancel.CanFocus then btnChangeCancel.SetFocus;
   end;

procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
// Проверка, задано ли значение для названия товара
if dbeName.Text = '' then begin
   Beep;
   MessageDlg('Не задано название товара!', mtError, [mbOK], 0);
   if dbeName.CanFocus then dbeName.SetFocus;
   Abort;
   end;
// Проверка, задано ли значение цены товара
if dbePrice.Text = '' then begin
```

```
   Beep;
   MessageDlg('Не задана цена товара!', mtError, [mbOK], 0);
   if dbePrice.CanFocus then dbePrice.SetFocus;
   Abort;
   end;
   end;

procedure TForm1.btnChangeOKClick(Sender: TObject);
begin
// Утверждение изменений в текущей записи (редактируемой или новой)
// или удаления текущей записи (просматриваемой)
if Table1.State in [dsEdit, dsInsert]
   then Table1.Post
   else if lblChangeKind.Caption = 'УДАЛЕНИЕ ЗАПИСИ' then Table1.Delete;
StateBrowse(Sender);
   end;

procedure TForm1.btnChangeCancelClick(Sender: TObject);
begin
// Если набор данных находился в режиме просмотра (при удалении записи),
// то никаких действий метод Cancel не выполняет
Table1.Cancel;
StateBrowse(Sender);
   end;

// Закрытие формы
procedure TForm1.btnCloseClick(Sender: TObject);
begin
Close;
   end;

end.
```

В качестве набора данных используется компонент `Table1`, записи которого отображает сетка `DBGrid1`. Значения полей текущей записи отображаются в компонентах `edtName`, `dbcbUnit`, `edtPrice` и `edtNote`. Установка свойств, определяющих взаимные отношения компонентов, связанных с набором данных, выполнена при разработке приложения через Инспектор объектов.

Пользователь не может переводить набор данных в режимы изменения с помощью визуальных компонентов, т. к. свойство `AutoEdit` источника данных `DataSource1` установлено в значение `False`.

Для перевода набора данных в режимы редактирования и вставки служат кнопки **Изменить** (`btnEdit`) и **Добавить** (`btnInsert`) соответственно. Переход в эти режимы осуществляется вызовом методов `Edit` и `Insert`, после чего название

режима отображается в надписи `lblChangeKind`, блокируются кнопки, связанные с переходом в режимы изменения, и разблокируются кнопки `btnChangeOK` и `btnChangeCancel`, позволяющие принять или отменить изменения, сделанные при редактировании или вставке записи. Для подтверждения или отмены изменений вызываются методы `Post` или `Cancel`, после чего кнопки подтверждения снова блокируются, а кнопки перехода в режимы изменения разблокируются. Изменение и задание значений полей выполняются с помощью компонентов `edtName`, `dbcbUnit`, `edtPrice` и `edtNote`.

При переходе в режимы, связанные с изменением записей, пользователю предлагается подтвердить свои действия, что программируется в обработчиках событий `OnBeforeEdit` и `OnBeforeInsert`.

Программирование удаления записей отличается от предыдущих действий по изменению набора данных, т. к. режим удаления у набора данных отсутствует. При нажатии кнопки **Удалить** (`btnDelete`) набор данных остается в режиме просмотра текущей записи. Однако надпись `lblChangeKind` указывает на переход в режим удаления, и если пользователь подтвердит удаление, нажав кнопку `btnChangeOK`, то для удаления текущей записи вызывается метод `Delete`. Таким образом, понятие "режим удаления" относится к форме, а не к набору данных. Запрос на подтверждение перехода в режим удаления кодируется в обработчике события нажатия кнопки `btnDelete`.

Флажок `cbCanEdit` с заголовком **Редактирование разрешено** включает и отключает возможность изменения в наборе данных. Управление этой возможностью осуществляется через свойство `ReadOnly` набора данных. Свойство переключается только при закрытом наборе данных. После нового открытия набора данных указатель текущей записи устанавливается на первую запись, поэтому перед переключением свойства `ReadOnly` положение указателя текущей записи запоминается с помощью закладки, а после переключения это положение восстанавливается. При разрешении изменений в наборе данных проверяется число его записей, и если набор данных пуст, то кнопка `btnDelete` блокируется.

Работа со связанными таблицами

Между отдельными таблицами БД может существовать связь, которая организуется через *поля связи* таблиц. Поля связи обязательно должны быть индексированными. Связь между таблицами определяет отношение подчиненности, при котором одна таблица является главной, а вторая — подчиненной. Обычно используется связь "один-ко-многим", когда одной записи в главной таблице может соответствовать несколько записей в подчиненной таблице. Такая связь также называется "мастер-детальный" (Master-Detail). После установления связи между таблицами при перемещении в главной таблице текущего указателя на какую-либо запись в подчиненной таблице автоматически становятся доступными записи, у которых значение поля связи равно значению поля связи текущей записи главной таблицы. Такой отбор записей подчиненной таблицы является своего рода фильтрацией.

Для организации связи между таблицами в подчиненной таблице используются следующие свойства, указывающие:

- `MasterSource` — источник данных главной таблицы;
- `IndexName` — текущий индекс подчиненной таблицы;
- `IndexFieldNames` — поле или поля связи текущего индекса подчиненной таблицы;
- `MasterFields` — поле или поля связи индекса главной таблицы.

Работа со связанными таблицами имеет определенные особенности.

- При изменении (редактировании) поля связи может нарушиться связь между записями двух таблиц. Поэтому при редактировании поля связи записи главной таблицы нужно соответственно изменять и значения поля связи всех подчиненных записей.
- При удалении записи главной таблицы нужно удалять и соответствующие ей записи в подчиненной таблице (каскадное удаление).
- При добавлении записи в подчиненную таблицу значение поля связи формируется автоматически по значению поля связи главной таблицы.

Ограничения по изменению полей связи и каскадному удалению записей могут быть наложены на таблицы при их создании, например, в среде программы Database Desktop, или реализовываться программно. Напомним, что эти ограничения ссылочной целостности относятся к так называемым бизнес-правилам — правилам управления БД и поддержания ее в целостном и непротиворечивом состоянии.

Пример приложения

В качестве примера работы со связанными таблицами рассмотрим приложение, предназначенное для автоматизации складского учета.

При организации складского учета используются две таблицы формата Paradox: `Store` для хранения информации о товарах и `Cards` для хранения карточек товара, в которой отмечается движение (приход и расход) каждого товара. Структура таблиц показана в табл. 7.1 и 7.2. В названия полей включены префиксы `S` и `C` (по первым буквам названий таблиц). Такое обозначение помогает при установлении связи между таблицами — из названия поля сразу видно, к какой таблице оно принадлежит.

Таблица 7.1. *Структура таблицы* `Store`

Имя поля	Тип	Размер	Ключевое поле	Примечание
`S_Code`	+		*	Уникальный код товара. Используется для связи с подчиненной таблицей

Таблица 7.1 *(окончание)*

Имя поля	Тип	Размер	Ключевое поле	Примечание
S_Name	A	20		Название товара. Требует обязательного заполнения
S_Unit	A	7		Единица измерения. Требует обязательного заполнения
S_Price	$	Не задается		Цена единицы товара. Требует обязательного заполнения
S_Quantity	N	Не задается		Количество товара на складе
S_Note	A	30		Примечание

Таблица 7.2. *Структура таблицы* `Cards`

Имя поля	Тип	Размер	Ключевое поле	Примечание
C_Number	+	Не задается	*	Уникальный код записи о движении товара
C_Code	I	Не задается		Код записи о движении товара, используемый для связи с главной таблицей
C_Move	N	Не задается		Приходное или расходное количество. Требует обязательного заполнения
C_Date	D	Не задается		Дата прихода или расхода

Между таблицами устанавливается связь "главный-подчиненный", при которой таблица `Store` склада является главной, а таблица `Cards` движения товара — подчиненной (рис. 7.11). Для организации связи в качестве поля связи главной таблицы берется автоинкрементное поле `S_Code` уникального кода товара. По этому полю построен ключ, значение которого автоматически формируется при добавлении новой записи и в пределах таблицы является уникальным. В подчиненной таблице полем связи (внешним ключом) является целочисленное поле `C_Code`, по которому построен индекс.

Приложение для работы со складом включает главную форму `fmStore` (рис. 7.12) и форму `fmInput` ввода данных о новом товаре.

В верхней части главной формы выводится информация о состоянии склада, в нижней части — сведения о движении товара. При выборе в таблице склада записи о товаре в таблице движения товара автоматически отображаются только записи, соответствующие движению именно этого товара. Для наглядности в наборы данных включены все поля таблиц, которые отображаются в компонентах `DBGrid`. При этом названия заголовков столбцов совпадают с названиями полей.

Рис. 7.11. Связь между таблицами `Store` и `Cards`

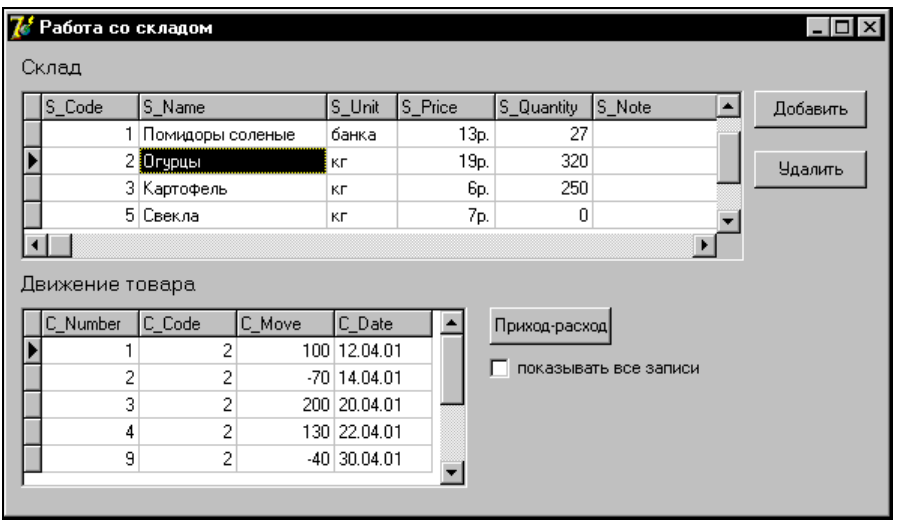


Рис. 7.12. Окно приложения **Работа со складом**

Модификация данных таблиц с помощью компонентов `DBGrid` запрещена, для этого их свойства `AutoEdit` установлены в значение `False`. Для модификации таблиц используются кнопки `Button`, а также отдельная форма `fmInput`. Обработчики событий нажатия кнопок **Добавить** (`btnNew`) и **Удалить** (`btnDelete`) добавляют записи о новом товаре в таблицу склада и удаляют записи о товаре.

При нажатии кнопки `btnNew` выводится в модальном режиме форма `fmInput` (рис. 7.13), содержащая четыре элемента `DBEdit`, которые связаны с полями названия, единицы измерения, цены товара и примечания таблицы `Store`. Связь устанавливается через источник данных `dsStore`, расположенный в главной форме `fmStore`. Чтобы такая связь стала возможной, в модуле `uInput`

формы ввода выполнена ссылка на модуль `uStore` главной формы. В свою очередь, в модуле главной формы есть ссылка на форму ввода.

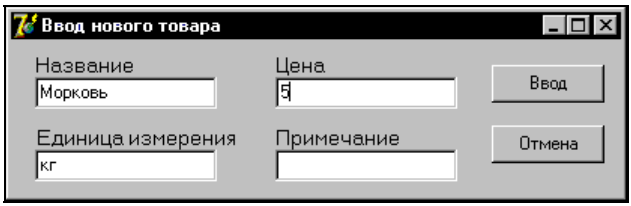


Рис. 7.13. Форма ввода данных о новом товаре

Перед вызовом формы ввода данных о новом товаре в таблицу склада добавляется новая запись, и компоненты-редакторы `DBEdit` этой формы содержат значения полей (первоначально пустые) новой записи. В процессе ввода пользователь может утвердить ввод, нажав кнопку **OK**, или отменить его, нажав кнопку **Отмена**. После закрытия модальной формы ввода проверяется, какая кнопка была нажата: если **OK**, то сделанные изменения принимаются, в противном случае — нет.

Для удаления записи с данными о товаре следует нажать кнопку `btnDelete`, после чего выдается запрос на подтверждение операции. В случае подтверждения сначала в цикле удаляются все записи дочерней таблицы с данными о движении этого товара, а затем происходит удаление записи с данными о товаре.

Добавление новой записи в таблицу движения товара выполняется при нажатии кнопки **Приход-расход** (`btnMove`). При добавлении к таблице движения новой записи поле кода товара, являющееся полем связи, автоматически заполняется правильным значением из текущей записи таблицы склада. В поле даты с помощью инструкции присваивания заносится текущая дата.

Пользователь должен вводить только приходное количество, поэтому для ввода этой информации специальная форма не создавалась, а используется функция `InputQuery`, позволяющая ввести строковое значение. На практике обычно требуется ввод большего количества данных и применяется форма, построенная таким же образом, как и форма `fmInput`. Поступление товара (приход) кодируется положительным числом, расход товара — отрицательным числом. После ввода количества товара выполняются преобразование и проверка формата введенного числа. В случае ошибки выдается соответствующее сообщение, и ввод записи отменяется.

После ввода новой записи о движении товара происходит изменение значения поля `S_Quantity` количества товара в таблице склада.

Для разрыва связи между таблицами используется флажок **показывать все записи** (`cbMoveAll`). По умолчанию он снят, и связь между таблицами существует. После разрыва связи в таблице движения товара отображаются все записи, независимо от положения текущего указателя в таблице склада. При этом блокируется

кнопка btnDelete удаления записей, т. к. при ее нажатии будут удалены все записи о движении товара.

Таким образом, в приложении выполнены следующие действия:

- организована связь между двумя таблицами по полю связи;
- реализовано каскадное удаление записей таблиц;
- запрещено изменение полей связи — пользователь не имеет возможности редактировать их с помощью компонентов DBGrid, а в коде модулей эти поля не затрагиваются.

Ниже приведены коды модулей форм приложения. Установка свойств большинства компонентов выполнена в обработчиках событий создания форм приложения.

```
// Модуль главной формы
   unit uStore;

   interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
     Db, DBTables, ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, DBCGrids;

type
  TfmStore = class(TForm)
        dsStore: TDataSource;
      TableStore: TTable;
     DBGridStore: TDBGrid;
          dsCard: TDataSource;
       TableCard: TTable;
      DBGridCard: TDBGrid;
          Label1: TLabel;
          Label2: TLabel;
         btnMove: TButton;
          btnNew: TButton;
       btnDelete: TButton;
       cbMoveAll: TCheckBox;

    procedure FormCreate(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnMoveClick(Sender: TObject);
    procedure cbMoveAllClick(Sender: TObject);

  private
    { Private declarations }
```

```
  public
    { Public declarations }

  end;

var
     fmStore: TfmStore;

   implementation

   uses uInput;

   {$R *.DFM}

procedure TfmStore.FormCreate(Sender: TObject);
begin
dsStore.AutoEdit := False;
dsCard.AutoEdit := False;
TableCard.MasterSource := dsStore;
cbMoveAll.Checked := False;
cbMoveAllClick(Sender);
   end;

procedure TfmStore.btnNewClick(Sender: TObject);
begin
TableStore.Append;
if fmInput.ShowModal = mrOK
   then begin
   TableStore.FieldByName('S_Quantify').AsFloat := 0;
   TableStore.Post;
   end
   else TableStore.Cancel;
   end;

procedure TfmStore.btnDeleteClick(Sender: TObject);
var n: longint;
begin
if TableStore.RecordCount = 0 then exit;
if MessageDlg('Удалить запись?', mtConfirmation, [mbOK, mbNo], 0) = mrOK
   then begin
   // Удаление записей в карточке движения товара (с конца набора данных)
   TableCard.Last;
   for n := 1 to  TableCard.RecordCount do TableCard.Delete;
   // Удаление карточки движения товара
   TableStore.Delete;
   end;
   end;
```

```
procedure TfmStore.btnMoveClick(Sender: TObject);
var sMove: string;
    nMove: double;
begin
if InputQuery ('Поступление товара +
  TableStore.FieldByName('S_Name').AsString,
              'Приход-расход', sMove) then begin
   // Проверка введенного приходного или расходного количества товара
   try
     nMove := StrToFloat(sMove);
     except
     Beep;
     MessageDlg ('Неправильно введен приход-расход: ' + sMove,
  mtError, [mbOK], 0);
     exit;
     end;
   // Добавление новой записи в карточку движения товара
   TableCard.Append;
   // Поле C_Code заполняется автоматически по полю S_Code
   TableCard.FieldByName('C_Move').AsFloat := nMove;
   TableCard.FieldByName('C_Date').AsDateTime := Now;
   TableCard.Post;
   // Пересчет наличного количества товара
   TableStore.Edit;
   TableStore.FieldByName('S_Quantify').AsFloat :=
        TableStore.FieldByName('S_Quantify').AsFloat + nMove;
   TableStore.Post;
   end;
   end;

procedure TfmStore.cbMoveAllClick(Sender: TObject);
begin
if not cbMoveAll.Checked then begin
   TableCard.IndexName := 'indC_Code';
   TableCard.MasterFields := 'S_Code';
   btnDelete.Enabled := True;
   end
   else begin
   TableCard.IndexName := '';
   TableCard.IndexFieldNames := '';
   TableCard.MasterFields := '';
   btnDelete.Enabled := False;
   end;
   end;

   end.
```

```
// Модуль формы ввода
   unit uInput;

   interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
     Dialogs, StdCtrls, Mask, DBCtrls;

type
  TfmInput = class(TForm)
    dbeName: TDBEdit;
    dbeUnit: TDBEdit;
   dbePrice: TDBEdit;
    dbeNote: TDBEdit;
     Label1: TLabel;
     Label2: TLabel;
     Label3: TLabel;
     Label4: TLabel;
     btnOK: TButton;
 btnCancel: TButton;

    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
     fmInput: TfmInput;

   implementation

   uses uStore;

   {$R *.DFM}

procedure TfmInput.FormCreate(Sender: TObject);
begin
dbeName.DataSource    := fmStore.dsStore;
dbeName.DataField     := 'S_Name';
dbeUnit.DataSource    := fmStore.dsStore;
dbeUnit.DataField     := 'S_Unit';
dbePrice.DataSource   := fmStore.dsStore;
```

```
dbePrice.DataField    := 'S_Price';
dbeNote.DataSource    := fmStore.dsStore;
dbeNote.DataField     := 'S_Note';
btnOK.ModalResult     := mrOK;
btnCancel.ModalResult := mrCancel;
   end;

end.
```

В рассмотренном примере связь между таблицами устанавливалась уже при выполнении приложения. Обычно же таблицы связываются на этапе разработки через Инспектор объектов. При этом для установки свойств `IndexName` и `MasterFields` удобно использовать специальный Редактор полей связи (Field Link Designer), вызываемый двойным щелчком в области значения свойства `MasterFields` в окне Инспектора объектов. В списке **Available Indexes** (Доступные индексы) выбирается индекс подчиненной таблицы, после чего составляющие его поля отображаются в списке **Detail Fields** (Детальные поля). В этом списке необходимо выбрать поле подчиненной таблицы, а в списке **Master Fields** (Главное поле) — поле главной таблицы (рис. 7.14). После нажатия кнопки **Add** выбранные поля связываются между собой, что отображается в списке **Joined Fields** (Связанные поля), например, так: `C_Code -> S_Code`. При этом оба поля пропадают из своих списков. Заполнение свойств `IndexName` и `MasterFields` происходит после закрытия окна при нажатии кнопки **OK**.

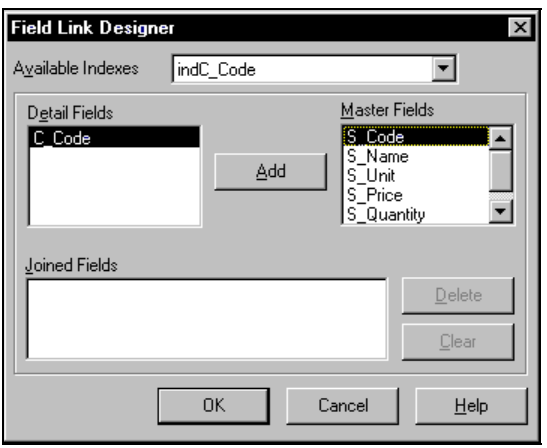


Рис. 7.14. Редактор полей связи

Замечание

Перед открытием окна Редактора полей связи необходимо установить значение свойства `MasterSource` подчиненного набора данных `TableCard`, которое должно указывать на источник данных `dsStore` главной таблицы.

Аналогично можно установить связь между несколькими таблицами, например, таблицей приходной накладной и таблицами поставщиков, покупателей и товаров.

Использование механизма транзакций

При работе с несколькими таблицами БД взаимосвязанные изменения периодически вносятся в разные таблицы. Например, в рассмотренном выше примере добавлялись новые записи о приходе или расходе товара в таблицу движения товара и соответствующим образом изменялось количество товара в таблице склада.

При возникновении какой-либо ошибки, связанной с записью нового количества товара, новое значение может быть не занесено в соответствующую запись, в результате чего целостность БД нарушится, и она будет содержать некорректные значения. Такая ситуация возможна, например, в случае многопользовательского доступа к БД при редактировании этой записи другим приложением. Поэтому в случае невозможности изменить информацию о количестве товара должно блокироваться и добавление новой записи о движении товара.

Для поддержания целостности БД используется так называемый *механизм транзакций*. Транзакция представляет собой последовательность операций, обычно выполняемую для нескольких наборов данных. Транзакция переводит БД из одного целостного состояния в другое. Чтобы транзакция была успешной, в обязательном порядке должны быть выполнены все операции, предусмотренные в ее рамках (принцип "всё или ничего"). В случае возникновения ошибки при выполнении хотя бы одной из операций вся транзакция считается неуспешной, и база данных возвращается в предшествующее транзакции состояние.

Транзакция может быть явной или неявной. *Неявная* транзакция запускается автоматически при модификации набора данных, например, при выполнении методов `Edit`, `Insert`, `Append` и `Delete`. Неявная транзакция утверждается методом `Post` закрепления изменений в наборе данных, а отменяется методом `Cancel`.

Явная транзакция означает, что программист должен самостоятельно организовывать операции изменения наборов данных. Для реализации механизма явных транзакций Delphi предоставляет специальные методы `StartTransaction`, `Commit` и `Rollback` компонента `DataBase`. Метод `StartTransaction` начинает транзакцию, после него должны располагаться инструкции, составляющие транзакцию. При выполнении операций производится обработка возникающих исключений. Если исключения не возникли, то после выполнения всех операций вызывается метод `Commit`, подтверждающий транзакцию, и все изменения вступают в силу. При возникновении исключения должен вызываться метод `Rollback`, котрый отменяет транзакцию и действие всех операций в рамках этой транзакции. Более подробно компонент `DataBase` будет рассмотрен в *главе 14*, посвященной работе с удаленными БД. Здесь же мы немного расширим наше приложение по работе со складом, включив в него механизм транзакций путем изменения кода обработчика события нажатия кнопки `btnMove`, как показано ниже. Для вызова методов, связанных с запуском

ниже. Для вызова методов, связанных с запуском и завершением транзакции, в форме размещен компонент `DataBase1`.

```
// Начало транзакции
Database1.StartTransaction;
try
  TableCard.Append;
  TableCard.FieldByName('C_Move').AsFloat := nMove;
  TableCard.FieldByName('C_Date').AsDateTime := Now;
  TableCard.Post;
  TableStore.Edit;
  TableStore.FieldByName('S_Quantity').AsFloat :=
       TableStore.FieldByName('S_Quantity').AsFloat + nMove;
  TableStore.Post;
  // Транзакция выполнена успешно
  // Утвердить изменения
  Database1.Commit;
  except
  // Транзакция не выполнена
  // Отказаться от изменений
  Database1.Rollback;
  end;
```

В приведенном примере механизм транзакций применяется к связанным таблицам, что в общем случае не обязательно. В одну транзакцию могут быть объединены операции, выполняемые и над отдельными таблицами.

При использовании реляционного способа доступа к данным с помощью инструкций SQL также можно явно управлять транзакциями. Эти возможности рассматриваются в *главе 8*.

Глава 8

Реляционный доступ к данным с помощью BDE

Реляционный способ доступа к данным основан на операциях с группами записей. Для задания операций используются средства языка структурированных запросов SQL (Structured Query Language), поэтому реляционный способ доступа называют также *SQL*-ориентированным. Для его реализации в приложениях Delphi при использовании механизма BDE в качестве набора данных должны применяться такие компоненты, как `Query` или `StoredProc`, позволяющие выполнить SQL-запрос.

Средства SQL применимы для выполнения операций с локальными и удаленными БД. Наиболее полно преимущества реляционного способа доступа и языка SQL проявляются при работе с удаленными БД. Основным достоинством реляционного способа доступа является небольшая загрузка сети, поскольку передаются только запросы и результат их выполнения.

Применение реляционного способа доступа для локальных БД не дает существенного преимущества, но и в этом случае с помощью SQL-запроса можно:

- формировать состав полей набора данных при выполнении приложения;
- включать в набор данных поля и записи из нескольких таблиц;
- отбирать записи по сложным критериям;
- сортировать набор данных по любому полю, в том числе неиндексированному;
- осуществлять поиск данных по частичному совпадению со значениями в поле.

Замечание

Многие из названных действий неприменимы к набору данных `Table`.

Для компонента `Query` реляционный способ доступа реализуется в случае, когда используются только средства SQL-запросов. Если дополнительно применять методы, ориентированные на операции с отдельными записями, например, `Next` или `Edit`, то будет реализован навигационный способ доступа со всеми его недостатками.

При работе с удаленными БД можно также использовать навигационный способ доступа, но только для небольших сетей, чтобы не создавать большой загрузки.

Основные сведения о языке SQL

Язык SQL ориентирован на выполнение действий с таблицами БД и данными в этих таблицах, а также некоторых вспомогательных действий. В отличие от процедурных языков программирования, в нем нет инструкций управления вычислительным процессом (циклов, переходов, ветвления) и средств ввода/вывода. Составленную на языке SQL программу также называют *SQL-запросом*.

Язык SQL обычно интегрируется в другие средства (оболочку) и используется в интерактивном режиме. Так, в системе управления базами данных, имеющей интерактивный интерфейс, пользователь может работать, ничего не зная об языке SQL и независимо от того, какая БД используется: локальная или удаленная. Такие СУБД, как Microsoft Access, Visual FoxPro или Paradox, сами выполняют действия, связанные с программированием запросов на SQL, предлагая пользователю средства визуального построения запросов, например, Query By Example (QBE) — запрос по образцу.

Поскольку SQL не обладает возможностями полнофункционального языка программирования, а ориентирован на доступ к данным, его часто включают в средства разработки программ. Встроен он и в систему Delphi. При этом для работы с командами SQL предлагаются соответствующие средства и компоненты. В Delphi к таким компонентам относятся наборы данных `Query`, `SQLQuery` и `ADOQuery`.

Различают два вида SQL-запросов: статический и динамический. *Статический* SQL-запрос включается в исходный код на этапе разработки и в процессе выполнения приложения не изменяется. Разработчик может изменить SQL-запрос путем использования параметров, если таковые имеются в его тексте.

Код *динамического* SQL-запроса формируется или изменяется при выполнении приложения. Такие запросы обычно применяются в случае, когда при выполнении запроса требуется учитывать действия пользователя.

Замечание

Принятая нами классификация не является однозначной. Так, в некоторых источниках SQL-запросы с параметрами также относят к разряду динамических.

Язык SQL имеет несколько стандартов, из которых наиболее популярными среди производителей программных продуктов являются стандарты SQL-89 и SQL-92. Стандарт SQL-92, поддерживаемый Американским национальным институтом стандартов (ANSI, American National Standards Institute) и Международной организацией по стандартизации (ISO, International Standard Organization), также называют стандартом ANSI или ANSI/ISO. Вследствие наличия нескольких стандартов и их различных интерпретаций появилось множество диалектов языка SQL, более или менее отличающихся друг от друга.

В языке SQL можно выделить следующие основные подмножества инструкций:

- определения данных;
- обработки данных;

- управления привилегиями (доступом к данным);
- управления транзакциями.

Рассмотрим основные возможности версии языка SQL, используемой в Delphi. Эта версия несколько отличается от стандарта SQL-92, например, в ней нельзя работать с просмотрами и управлять привилегиями.

Особенности языка SQL, используемого для работы с удаленными БД (промышленными СУБД), будут рассмотрены в следующих главах.

В приложениях Delphi для выполнения инструкций SQL, применяя механизм BDE, можно использовать набор данных `Query`. Напомним, что текст SQL-запроса является значением свойства `SQL` компонента `Query` и формируется либо при разработке приложения, либо во время его выполнения. Компонент `Query` обеспечивает выполнение SQL-запроса и получение соответствующего набора данных. Формирование набора данных выполняется при активизации компонента `Query` путем вызова метода `Open` или установкой свойства `Active` в значение `True`. Иногда при отработке SQL-запроса нет необходимости получать набор данных, например, при удалении, вставке или изменении записей. В этом случае предпочтительнее выполнять запрос вызовом метода `ExecSQL`. При работе в сети вызов этого метода производит требуемую модификацию набора данных, не передавая в вызывающее приложение (компьютер) записи набора данных, что существенно снижает нагрузку на сеть.

Кроме того, набрать и выполнить в интерактивном режиме текст SQL-запроса позволяют инструментальные программы, поставляемые вместе с Delphi, например, Database Desktop, SQL Explorer и SQL Builder. Отметим, что первые две программы вызываются одноименными командами меню **Tools** и **Database** соответственно, а визуальный построитель запросов SQL Builder вызывается через контекстное меню компонента `Query`.

Проверка синтаксиса и отработка запроса, встроенного в приложение (чаще всего с помощью компонента `Query`), производятся на этапе выполнения приложения. При наличии синтаксических ошибок в тексте SQL-запроса генерируется исключение, интерпретировать которое порой не просто. Для отладки SQL-запросов удобно использовать программы с развитым интерфейсом, например, Database Desktop. После отладки текст запроса вставляется в разрабатываемое приложение. При таком подходе значительно сокращается время создания запросов и существенно уменьшается вероятность появления динамических ошибок.

В качестве результата выполнения SQL-запроса может возвращаться набор данных, который составляют отобранные с его помощью записи. Этот набор данных называют *результирующим*.

В дальнейшем при описании инструкций языка мы будем опускать несущественные операнды и элементы, используя для обозначения отдельных элементов символы < и > (при программировании не указываются), а необязательные конструкции заключать в квадратные скобки. Для наглядности зарезервированные слова языка SQL будут писаться прописными, а имена — преимущественно строчными буквами. Регистр букв не влияет на интерпретацию инструкций языка. Точка с запятой в конце SQL-инструкций необязательна. Элементы в списках, например, имена полей и таблиц, должны быть разделены запятыми.

Имена таблиц и полей (столбцов) заключаются в апострофы или двойные кавычки, например, "`First Name`". Если имя не содержит пробелы и другие специальные символы, то апострофы можно не указывать.

В SQL-запросе допускаются комментарии, поясняющие текст программмы. Комментарий ограничивается символами `/*` и `*/`.

Функции языка

Язык SQL, как и другие языки, предоставляет для использования ряд функций, из которых наиболее употребительны следующие:

- агрегатные, или статистические, функции:
  - `AVG()` (среднее значение);
  - `MAX()` (максимальное значение);
  - `MIN()` (минимальное значение);
  - `SUM()` (сумма);
  - `COUNT()` (количество значений);
  - `COUNT(*)` (количество ненулевых значений);
- функции работы со строками:
  - `UPPER(Str)` (преобразование символов строки `Str` к верхнему регистру);
  - `LOWER(Str)` (преобразование символов строки `Str` к нижнему регистру);
  - `TRIM(Str)` (удаление пробелов в начале и в конце строки `Str`);
  - `SUBSTRING(Str FROM n1 TO n2)` (выделение из строки `Str` подстроки, которая включает в себя символы, начиная с номера (позиции) `n1` и заканчивая номером `n2`);
  - `CAST(<Expression> AS <Type>)` (приведение выражения `Expression` к типу `Type`);
- функции декодирования даты и времени:
  - `EXTRACT(<Элемент> FROM <Выражение>)` (из выражения, содержащего значение даты или времени, извлекается значение, соответствующее указанному элементу); в качестве элемента даты или времени можно указывать значения: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` или `SECOND`.

Определение данных

Определение данных — это манипулирование целыми таблицами. Сюда включаются операции:

- создания новой таблицы;
- удаления таблицы;

- изменения состава полей таблицы;
- создания и удаления индекса.

Эти действия выполняются с помощью подмножества инструкций определения данных языка SQL.

Создание и удаление таблицы

Для создания таблицы служит инструкция `CREATE TABLE`, который имеет следующий формат:

```
CREATE TABLE <Имя таблицы>
       (<Имя поля> <Тип данных>,
        ...
        <Имя поля> <Тип данных>);
```

Обязательными операндами являются имя создаваемой таблицы и имя как минимум одного поля с соответствующим типом данных.

Замечание

В действительности вместо имени таблицы указывается имя главного файла таблицы.

Для локальной таблицы ее формат автоматически определяется по расширению файла: db для таблицы Paradox и dbf для таблицы dBase. Если расширение файла не указано, то тип таблицы определяется драйвером, заданным в BDE для локальных БД *(см. главу 12)*. По умолчанию установлен драйвер Paradox.

Файлы таблицы размещаются в каталоге БД, на который указывает псевдоним БД. Для компонента `Query` псевдоним задается свойством `DatabaseName`.

Порядок следования строк с описаниями полей определяет порядок расположения полей создаваемой таблицы. Отметим, что описания полей могут располагаться подряд, а не занимать отдельные строки инструкции.

Типы данных языка SQL и соответствующие им типы данных для таблиц dBase и Paradox приведены в табл. 8.1.

В приведенной таблице `N` обозначает длину поля в байтах, `X` — число цифр, `Y` — число цифр после десятичной точки. Для типа `CHARACTER` допускается сокращение `CHAR`. Отметим, что в стандарте SQL-92 число допустимых для полей типов данных намного меньше, например, нет автоинкрементного типа.

Таблица 8.1. *Типы данных для таблиц БД*

SQL	dBase	Paradox
SMALLINT	Number (6,10)	Short
INTEGER	Number (20,4)	Long Integer
DECIMAL(X,Y)	—	BCD

Таблица 8.1 *(окончание)*

SQL	dBase	Paradox
NUMERIC(X,Y)	Number (X,Y)	Number
FLOAT(X,Y)	Number	Float (X,Y)
CHARACTER(N)	Character	Alpha
VARCHAR(N)	Character	Alpha
DATE	Date	Date
BOOLEAN	Logical	Logical
BLOB(N,1)	Memo	Memo
BLOB(N,2)	Binary	Binary
BLOB(N,3)	—	Formatted memo
BLOB(N,4)	OLE	OLE
BLOB(N,5)	—	Graphic
TIME	—	Time
TIMESTAMP	—	Timestamp
MONEY	Number (20,4)	Money
AUTOINC	—	Autoincrement
BYTES(N)	—	Bytes

Пример создания таблицы средствами языка SQL:

```
CREATE TABLE NewTable.dbf
      (Number    INTEGER,
       Name      CHAR(20),
       BirthDay  DATE);
```

В каталоге БД создается новая таблица `NewTable` формата dBase, для которой определены целочисленное поле `Number`, символьное поле `Name` и поле даты `BirthDay`.

Если таблица с заданным именем уже существует, то при выполнении инструкции создания таблицы генерируется исключение.

Для таблицы Paradox можно определить ключ (главный, или первичный), указав описатель `PRIMARY KEY` и перечислив в скобках после него поля, образующие этот ключ. Ключевые поля должны быть в списке полей первыми. Вот пример создания таблицы с построением главного ключа:

```
CREATE TABLE Personnel.db
       (Code    AUTOINC,
       Name     CHAR(20),
```

```
        Position CHAR(15),
        Salary   NUMERIC(10,2),
        PRIMARY KEY(Code));
```

Новая таблица `Personnel` имеет формат Paradox, и для нее определены автоинкрементное поле кода `Code`, символьные поля фамилии `Name` и должности `Position`, а также числовое поле оклада `Salary`. По полю кода построен главный ключ.

Для удаления таблицы предназначена инструкция:

```
DROP TABLE <Имя таблицы>;
```

Например:

```
DROP TABLE NewTable.dbf;
```

В результате выполнения этой инструкции с диска удаляются все файлы, относящиеся к таблице с именем `NewTable`. Если таблица не существует или с ней работает другое приложение, то генерируется исключение.

Изменение состава полей таблицы

Изменение состава полей таблицы заключается в добавлении или удалении полей и приводит к изменению ее структуры, при этом таблицу не должны использовать другие приложения. Изменение состава полей таблицы выполняется инструкцией `ALTER TABLE`:

```
ALTER TABLE <имя таблицы>
     ADD   <имя поля1> <тип данных1>,
     DROP  <имя поля1>,
     ...
     ADD   <имя поляN> <тип данныхN>,
     DROP  <имя поляN>;
```

Операнд `ADD` добавляет к таблице новое поле, имя и тип которого задаются так же, как и в инструкции `CREATE TABLE`, а операнд `DROP` удаляет из таблицы поле с заданным именем. Операнды `ADD` и `DROP` не зависят друг от друга и могут следовать в произвольном порядке.

При попытке удалить отсутствующее поле или добавить поле с существующим именем генерируется исключение.

Пример изменения структуры таблицы:

```
ALTER TABLE Personnel.db
      ADD   Section SMALLINT,
      ADD   Note    CHAR(30),
      DROP  Position;
```

К таблице `Personnel` добавляются целочисленное поле номера отдела `Section` и символьное поле примечаний `Note`, поле `Position` удаляется.

Создание и удаление индекса

Напомним, что индекс обеспечивает быстрый доступ к данным, хранимым в поле, для которого он создан. Для ускорения операций с таблицей индексными следует делать поля, по которым часто производятся поиск и отбор записей. Индекс создается инструкцией `CREATE INDEX` следующего формата:

```
CREATE INDEX
      <Имя индекса> ON <Имя таблицы> (<Имя поля1>, ..., [<Имя поляN>]);
```

Одной инструкцией можно создать один индекс, при этом одно поле может входить в состав нескольких индексов. Кроме того, не требуется, чтобы значения составляющих индекс полей были уникальными. При сортировке по индексу записи упорядочиваются в порядке возрастания значений индексных полей.

С помощью инструкции `CREATE INDEX` для таблиц dBase создаются индексы, а для таблиц Paradox — вторичные индексы. Напомним, что первичным индексом таблиц Paradox является ключ, описываемый непосредственно при создании таблицы. Использование инструкции `CREATE INDEX` является единственным способом определения индекса для таблиц dBase.

Так можно создать индекс по одному полю:

```
CREATE INDEX
      indName ON Personnel.db (Name)
```

А так по двум:

```
CREATE INDEX
      indNamePosition ON Personnel.db (Name, Position)
```

Для удаления индекса используется инструкция `DROP INDEX` формата

```
DROP INDEX
      <Имя таблицы>.<Имя индекса>
```

или

```
DROP INDEX
      <Имя таблицы>.PRIMARY
```

Во время удаления индекса таблица не должна использоваться другими приложениями. При выполнении инструкции `DROP INDEX` можно удалить один индекс, обозначив его составным именем, состоящим из имени таблицы и имени собственно индекса. Если удаляется первичный индекс (ключ) таблицы Paradox, то вместо имени индекса указывается описатель `PRIMARY`, поскольку главный ключ не имеет имени.

Например, в инструкции

```
DROP INDEX
      "Personnel.db".indNamePosition
```

из таблицы `Personnel` удаляется индекс `indNamePost`, созданный по полям `Name` и `Position`.

Первичный ключ удаляется так:

```
DROP INDEX
       "Personnel.db".PRIMARY
```

Если удаляемый индекс отсутствует или таблица используется другим приложением, то генерируется исключение.

Отбор данных из таблиц

Отбор данных из таблиц заключается в получении из них полей и записей, удовлетворяющих заданным условиям. Результат выполнения запроса, на основании которого отбираются записи, называют *выборкой*. Данные можно выбирать из одной или нескольких таблиц с помощью инструкции `SELECT`.

Описание инструкции *SELECT*

Инструкция `SELECT` — важнейшая инструкция языка SQL. Она используется для отбора записей, удовлетворяющих сложным критериям поиска, и имеет следующий формат:

```
SELECT [DISTINCT] {* | <Список полей>}
      FROM <Список таблиц>
        [WHERE    <Условия отбора>]
        [ORDER BY <Список полей для сортировки>]
        [GROUP BY <Список полей для группирования>]
        [HAVING   <Условия группирования>]
        [UNION    <Вложенная инструкция SELECT>]
```

Результат выполнения SQL-запроса, заданного инструкцией `SELECT`, представляет собой выборку записей, отвечающих заданным в нем условиям. При рассмотрении инструкции `SELECT` будем предполагать, что SQL-запрос набран и выполнен с помощью компонента `Query`. В этом случае результатом выполнения запроса является соответствующий этому компоненту набор данных.

В таком результирующем наборе данных могут быть разрешены или запрещены повторяющиеся записи (т. е. имеющие одинаковые значения всех полей). Этим режимом управляет описатель `DISTINCT`. Если он отсутствует, то в наборе данных разрешаются повторяющиеся записи.

В инструкцию `SELECT` обязательно включается список полей и операнд `FROM`, остальные операнды могут отсутствовать. В списке операнда `FROM` перечисляются имена таблиц, для которых отбираются записи. Список должен содержать как минимум одну таблицу.

Список полей определяет состав полей результирующего набора данных, эти поля могут принадлежать разным таблицам. В списке должно быть задано хотя бы одно поле. Если в набор данных требуется включить все поля таблицы (таблиц), то вместо перечисления имен полей можно указать символ `*`. Если список содержит поля нескольких таблиц, то для указания принадлежности поля к той

или иной таблице используют составное имя, включающее в себя имя таблицы и имя поля, разделенные точкой: `<Имя таблицы>.<Имя поля>`.

Операнд `WHERE` задает условия (критерии) отбора, которым должны удовлетворять записи в результирующем наборе данных. Выражение, описывающее условия отбора, является логическим. Его элементами могут быть имена полей, операции сравнения, арифметические и логические операции, скобки, специальные функции `LIKE`, `NULL`, `IN` и др.

Операнд `GROUP BY` позволяет выделять группы записей в результирующем наборе данных. Группой являются записи с одинаковыми значениями в полях, перечисленных за операндом `GROUP BY`. Выделение групп требуется для выполнения групповых операций над записями, например, для определения количества какого-либо товара на складе.

Операнд `HAVING` действует совместно с операндом `GROUP BY` и используется для отбора записей внутри групп. Правила записи условий группирования аналогичны правилам формирования условий отбора в операнде `WHERE`.

Операнд `ORDER BY` содержит список полей, определяющих порядок сортировки записей результирующего набора данных. По умолчанию сортировка по каждому полю выполняется в порядке возрастания значений; если необходимо задать для поля сортировку по убыванию, то после имени этого поля указывается описатель `DESC`.

Инструкции `SELECT` могут иметь сложную структуру и быть вложенными друг в друга. Для объединения инструкций используется операнд `UNION`, в котором располагается вложенная инструкция `SELECT`, называемая также *подзапросом*. Результирующий набор данных представляют записи, отобранные с учетом выполнения условий отбора, заданных операндами `WHERE` обеих инструкций.

Инструкция `SELECT` используется также внутри других инструкций, например, инструкций модификации записей, обеспечивая для их выполнения требуемый отбор записей.

Управление полями

Управление полями состоит в указании полей таблицы (таблиц), включаемых в результирующий набор данных. Как отмечалось выше, при отборе всех записей таблицы условия отбора записей не указываются: если вместо списка полей указать `*`, то в наборе данных оказываются все поля записей. При этом можно не задумываться о названиях полей. Порядок следования полей в наборе данных соответствует порядку физических полей таблицы, определенному при ее создании.

Пример отбора всех полей в таблице:

```
SELECT * FROM Personnel.db
```

В результате выполнения этого запроса из таблицы `Personnel` в набор данных попадают все поля и все записи, и набор данных имеет вид:

```
Code   Name          Position      Salary
1      Иванов Р.О.   Директор      6700
```

```
2      Петров А.П.   Менеджер      5200
3      Семенова И.И. Менеджер      5200
4      Кузнецов П.А. Секретарь     3600
5      Попов А.Л.    Водитель      2400
6      Васин Н.Е.    Водитель      2500
```

При необходимости получения данных из нескольких полей таблицы после слова `SELECT` через запятую перечисляются в нужном порядке названия этих полей. Порядок полей в наборе данных будет соответствовать порядку полей в списке. Если имя поля указано в списке неоднократно, то в наборе данных оказывается несколько столбцов с одинаковыми именами и данными. Например:

```
SELECT Name, Salary
      FROM Personnel.db
```

В набор данных, формируемый в результате выполнения этого SQL-запроса, включаются поля `Name` и `Salary` всех записей из таблицы `Personnel`. Набор данных получает вид:

```
Name          Salary
Иванов Р.О.   6700
Петров А.П.   5200
Семенова И.И. 5200
Кузнецов П.А. 3600
Попов А.Л.    2400
Васин Н.Е.    2500
```

Можно указать все поля таблицы — такой список аналогичен использованию знака `*`, но отличается тем, что при явном указании имен полей мы можем управлять порядком их следования в наборе данных.

Приведенный достаточно простой запрос демонстрирует определенные преимущества компонента `Query`, использующего возможности языка SQL, по сравнению с компонентом `Table`. Для компонента `Table` изменить порядок или ограничить состав полей набора данных можно только на этапе разработки приложения, задав с помощью Редактора полей статические поля. Компонент `Query` позволяет осуществлять эти операции и для динамических полей, что достигается небольшим изменением SQL-запроса как при разработке, так и при выполнении приложения.

Кроме физических полей таблиц, в набор данных можно включать *вычисляемые поля*. Для получения вычисляемого поля в списке полей указывается не имя этого поля, а выражение, по которому рассчитывается его значение. Выражение составляется по обычным правилам, при этом в него могут не входить имена полей. Например:

```
SELECT "- " || Name, Salary, Salary * 1.1
     FROM Personnel;
```

Для сотрудников организации (таблица `Personnel`) выводятся старое значение оклада и новое, увеличенное на 10%. К каждой фамилии с помощью операции конкатенации (`||`) добавляются дефис и пробел. Результирующий набор будет таким:

```
Name            Salary  Salary * 1.1
- Иванов Р.О.   6700    7370
- Петров А.П.   5200    5720
- Семенова И.И. 5200    5720
- Кузнецов П.А. 3600    3960
- Попов А.Л.    2400    2640
- Васин Н.Е.    2500    2750
```

Записи могут иметь одинаковые значения некоторых полей. Для того чтобы включить в набор данных только записи с *уникальными* (неповторяющимися) значениями, перед списком полей указывается описатель `DISTINCT`. Например:

```
SELECT DISTINCT Position
      FROM Personnel
```

Сформированный набор данных содержит список занимаемых штатных должностей организации. Записи выбираются из таблицы сотрудников организации, при этом в набор данных каждая должность включается только один раз (поле `Position`):

```
Position
Директор
Менеджер
Секретарь
Водитель
```

А так задается выборка записей с уникальными значениями двух полей:

```
SELECT DISTINCT Position, Salary
      FROM Personnel
```

Сформированный набор данных содержит список занимаемых штатных должностей и окладов. В набор данных попадают записи с уникальной комбинацией значений двух полей (`Position` и `Salary`):

```
Position      Salary
Директор      6700
Менеджер      5200
Секретарь     3600
Водитель      2400
Водитель      2500
```

Еще одним достоинством языка SQL является простота объединения в результирующем наборе данных, содержащихся в *нескольких таблицах*. Для этого после слова `FROM` перечисляются имена таблиц, из записей которых формируется

набор данных. Такое использование данных из различных таблиц называется *соединением*.

Пример запроса на отбор записей из двух таблиц:

```
SELECT * FROM Personnel, Info
```

или

```
SELECT Personnel.*, Info.* FROM Personnel, Info
```

Результирующий набор данных состоит из всех полей и всех записей таблиц `Personnel` и `Info`. Первыми располагаются поля первой таблицы, далее следуют поля второй таблицы. Имена полей набора данных являются составными и включают в себя имена таблиц.

При выполнении запроса к нескольким таблицам в набор данных отбираются записи этих таблиц, удовлетворяющие заданным условиям. В данном случае условия отбора не заданы, поэтому в набор данных попадают все записи из обеих таблиц. Обычно таким образом отбираются записи из таблиц, связанных по определенным полям. Если же сформировать набор данных из таблиц, содержащих не связанные между собой данные, например, список сотрудников организации и список товаров на складе, то получившиеся записи могут содержать такие поля, как фамилия сотрудника и цена товара, что вряд ли имеет какой-либо смысл.

Еще один пример на отбор полей из разных таблиц:

```
SELECT Personnel.Name, Info.I_Code
      FROM Personnel, Info
```

В случае если имя таблицы, приводимое в операнде `FROM`, указывает формат файла таблицы, то в обозначении поля имя этой таблицы заключается в кавычки, например, `"Personnel.db".Name`.

Вместо имени таблицы в тексте SQL-запроса можно задать *псевдоним*, упрощающий указание имени таблицы, например, при обозначении ее полей. После определения псевдонима его можно использовать вместо имени таблицы. Псевдоним задается с помощью описателя `AS`, указываемого после имени таблицы. Например:

```
SELECT P.Name, I.I_Code
      FROM "Personnel.db" As P, "Info.db" As I
```

Для таблицы `Personnel` определен псевдоним `P`, а для таблицы `Info` — псевдоним `I`, которые используются при обозначении полей этих таблиц. Отметим, что в данном примере использование псевдонимов встречается раньше, чем их определение. Язык SQL не является алгоритмическим, и в нем допускается такой порядок описания и использования псевдонимов.

Замечание

Псевдоним таблицы применяется в инструкции SQL вместо имени таблицы и никак не связан с псевдонимом БД, который используется для определения значения свойства `DatabaseName` набора данных `Query` и задает расположение БД.

Если в составе таблиц, из которых отбираются записи, имя некоторого поля является уникальным, то имя содержащей это поле таблицы можно не указывать. Например:

```
SELECT Name, I_Code
       FROM Personnel, Info
```

Имена полей `Name` и `I_Code` уникальны для таблиц `Personnel` и `Info`, поэтому имена таблиц в обозначении полей можно опустить.

Простое условие отбора записей

В предыдущем разделе в набор данных попадали все записи из указанных таблиц, при этом разработчик мог управлять составом полей этих записей. На практике набор данных обычно ограничивается записями, удовлетворяющими каким-либо определенным условиям (критериям) отбора, задаваемым с помощью операнда `WHERE`. Критерий отбора может варьироваться от простейшего, в котором сравниваются два значения, до сложного, когда учитывается много факторов. При этом в набор данных попадают записи, для которых выполняется условие отбора.

Критерий отбора представляет собой логическое выражение, в котором можно использовать операции, перечисленные ниже.

- Сравнение; возможные операторы:
  - `=` (равно);
  - `>` (больше);
  - `<` (меньше);
  - `>=` (больше или равно);
  - `<=` (меньше или равно);
  - `<>` или `!=` (не равно);
  - `!>` (не больше);
  - `!<` (не меньше).
- `LIKE` (сравнение по шаблону).
- `IS NULL` (проверка на нулевое значение).
- `IN` (проверка на вхождение).
- `BETWEEN` (проверка на вхождение в диапазон).

В простом критерии отбора используется одна операция. Для операций сравнения и сравнения по шаблону критерий отбора имеет следующий формат:

```
<Выражение1> <Оператор> <Выражение2>
```

Выражение состоит из имен полей, функций, констант, значений, знаков операций и круглых скобок. В простейшем случае — из имени поля или значения. Например:

```
SELECT Name
      FROM Personnel
      WHERE Salary >= 4000;
```

Эта инструкция задает получение следующего списка сотрудников, имеющих оклад не менее 4000 (рублей):

```
Name
Иванов Р.О.
Петров А.П.
Семенова И.И.
```

Пример отбора записей по значениям символьного поля:

```
SELECT Name
      FROM Personnel
      WHERE Position = 'Водитель';
```

В результате получим выборку с фамилиями водителей:

```
Name
Попов А.Л.
Васин Н.Е.
```

Поля, входящие в набор данных, и поля, используемые в критерии отбора, могут отличаться друг от друга. Так, в приведенном примере в наборе данных присутствует поле фамилии `Name`, в то время как в критерии отбора записей используется поле оклада `Salary`.

В последнем примере в операции сравнения учитывается регистр символов, и слова `Водитель` и `водитель` не равны друг другу. Поэтому различия в регистре символов или наличие ведущих и конечных пробелов приводят к ошибкам отбора записей. В данной ситуации критерий отбора лучше записать в следующем виде:

```
WHERE UPPER(TRIM(Post)) = 'ВОДИТЕЛЬ'
```

Функция `TRIM` удаляет из строкового значения ведущие и конечные пробелы, а функция `UPPER` приводит символы полученной строки к верхнему регистру. В результате значение должности `водитель` независимо от наличия ведущих и конечных пробелов, а также регистра букв будет приведено к значению `ВОДИТЕЛЬ`.

Для сравнения строк вместо операций `=`, `!=` и `<>` можно использовать операцию `LIKE`, выполняющую сравнение по частичному совпадению. Частичное совпадение значений целесообразно проверять, например, в случаях, когда известны только начальная часть фамилии или названия предмета. Вот образец соответствующего запроса:

```
SELECT Name
      FROM Personnel
      WHERE Name LIKE "Ав"
```

Здесь мы получаем список фамилий, начинающихся на `Ав`.

В выражениях операции `LIKE` допускается использование *шаблона*, в котором разрешены все алфавитно-цифровые символы (с учетом регистра).

При этом два символа имеют специальное назначение:

- % (замещение любого количества символов, в том числе и нулевого);
- _ (замещение одного символа).

С помощью шаблона можно выполнить проверку на частичное совпадение не только начальных символов строки, но и найти вхождение заданного фрагмента в любую часть строкового значения. Например:

```
SELECT Name
      FROM Goods
      WHERE Name LIKE "%" || "ка" || "%"
```

В приведенном запросе происходит отбор всех товаров, в названия которых входят символы ка. Набор данных, полученный при таком отборе, может иметь вид:

```
Name
ручка
карандаш
замазка
```

Перед операцией LIKE можно использовать описатель NOT, который изменяет результат выполнения операции на противоположное значение и проверяет значения выражений на несовпадение.

Для проверки нулевого значения выражения служит операция IS NULL, которая имеет следующий формат:

```
<Выражение> IS [NOT] NULL
```

Так, в запросе:

```
SELECT *
      FROM Store
      WHERE S_Price IS NULL
```

выполняется отбор всех полей записей таблицы склада (Store), для которых не определена цена (S_Price) товара.

Проверка на *вхождение* значения выражения *в список* выполняется с помощью операции IN следующего формата:

```
<Выражение> [NOT] IN <Список значений>
```

Эту операцию удобно использовать, если выражение может принимать относительно небольшое количество различных значений. Вот пример соответствующего запроса:

```
SELECT Name, Salary
      FROM Personnel
      WHERE LOWER(Post) IN ("менеджер", "водитель")
```

В результате его выполнения получим выборку фамилий и окладов всех менеджеров и водителей:

```
Name            Salary
Петров А.П.     5200
Семенова И.И.   5200
Попов А.Л.      2400
Васин Н.Е.      2500
```

Операция `BETWEEN` выполняет проверку *вхождения* значения *в диапазон* и имеет формат:

```
<Выражение> [NOT] BETWEEN
      <Минимальное значение> AND <Максимальное значение>
```

При использовании этой операции в набор данных включаются записи, для которых значение выражения больше или равно минимальному, а также меньше или равно максимальному значениям.

Замечание

Описатель `NOT` изменяет значение результата операции на противоположное.

Рассмотрим запрос:

```
SELECT *
      FROM Cards
      WHERE C_Date BETWEEN "21.5.02" AND "27.5.2002"
```

В результате его выполнения получим набор записей, для которых дата (поле `C_Date`) находится в диапазоне с 21 по 27 мая 2002 года:

```
C_Number   C_Code   C_Move   C_Date
3          3        -150     22.05.02
4          3        -30      22.05.02
5          3        270      24.05.02
6          4        200      27.05.02
```

Значение даты заключается в кавычки, в качестве разделителя используется точка. В некоторых других реализациях языка SQL в качестве разделителей допускается использовать символы `-` и `/`. Значение года можно задавать как двумя (`21.5.02`), так и четырьмя (`27.5.2002`) цифрами. В значениях дня и месяца можно опускать незначащий ноль.

Сложные критерии отбора записей

При отборе можно использовать несколько операций, задавая тем самым сложные критерии отбора записей.

Сложный критерий (логическое выражение) состоит из:

- простых условий;
- логических операций:
  - AND (логическое И);
  - OR (логическое ИЛИ);
  - NOT (логическое НЕ);
- круглых скобок.

Замечание

В языке SQL приоритет операций сравнения выше приоритета логических операций. Для изменения порядка выполнения операций используются круглые скобки.

Пример запроса со сложным критерием отбора:

```
SELECT Cards.*
      FROM Store,Cards
      WHERE (S_Quantity > 250) AND
             (C_Date BETWEEN "21.5.2002" AND "27.5.2002")
```

Здесь происходит отбор всех записей о движении товара, количество которого на складе превышает 250 единиц, в течение заданного периода. В данном логическом выражении простые условия заключены в скобки, что не обязательно, т. к. приоритет операций сравнения выше приоритета логических операций. Набор данных, полученный при таком отборе записей, может иметь вид:

C_Number	C_Code	C_Move	C_Date
5	3	270	25.05.02
6	4	200	26.05.02
5	3	270	27.05.02
6	4	200	27.05.02

Отметим, что рассмотренные выше операции BETWEEN и IN также реализуют логические операции: BETWEEN — логическое И, а IN — логическое ИЛИ.

Группирование записей

Записи набора данных могут быть сгруппированы по некоторому признаку. Группу образуют записи с одинаковыми значениями в полях, перечисленных в списке операнда GROUP BY. При группировании записей их проще анализировать и обрабатывать, например, с помощью статистических функций.

Группирование записей автоматически исключает повтор значений в полях, заданных для группирования, т. к. записи с совпадающими значениями этих полей объединяются в одну группу.

Пример запроса с группированием записей:

```
SELECT C_Date, COUNT(C_Date)
      FROM Cards
      WHERE C_Date BETWEEN "01.06.2002" AND "03.06.2002"
      GROUP BY C_Date
```

Для каждой даты из указанного периода выводится количество записей, в которых она встречается. Если не выполнить группирование, то в набор данных попадут все записи, а при использовании группирования все даты для полученного набора данных уникальны. Функция `COUNT` выводит для каждой группы (сформированной по полю даты) число записей в группе. Полученный набор данных может иметь следующий вид:

```
C_Date       COUNT(C_Date)
01.06.02     20
02.06.02     17
03.06.02     8
```

В этом примере поле даты `C_Date` использовано в инструкции `SELECT` трижды, в общем случае можно использовать различающиеся поля.

Совместно с операндом `GROUP BY` можно использовать операнд `HAVING`, с помощью которого задаются дополнительные условия группирования записей.

Рассмотрим пример запроса:

```
SELECT C_Date, COUNT(C_Date)
      FROM Cards
      GROUP BY C_Date
      HAVING COUNT(C_Date) > 50
```

Здесь отбираются данные для дат, когда движение товара было интенсивным — общее число записей в соответствующей группе превышало 50. Так как не указан ограничивающий период времени, при отборе в набор данных будут учитываться записи таблицы `Cards` с любыми датами.

Сортировка записей

Как уже говорилось, сортировка представляет собой упорядочивание записей по возрастанию или убыванию значений полей. Список полей, по которым выполняется сортировка, указывается в операнде `ORDER BY`. Порядок полей в этом операнде определяет порядок сортировки: сначала записи упорядочиваются по значениям поля, указанного в этом списке первым, затем записи, имеющие одинаковое значение первого поля, упорядочиваются по второму полю и т. д.

Поля в списке обозначаются именами или номерами, которые соответствуют номерам в списке полей после слова `SELECT`.

По умолчанию сортировка происходит в порядке возрастания значений полей. Для указания обратного порядка сортировки по какому-либо полю нужно указать после имени этого поля описатель `DESC`.

Замечание

В отличие от набора данных `Table`, средствами языка SQL можно выполнять сортировку для набора данных `Query` и по неиндексированным полям. Однако по индексированным полям таблицы сортировка выполняется быстрее. При этом состав полей индекса должен соответствовать списку полей, указанных в операнде `ORDER BY`.

Пример запроса на сортировку записей:

```
SELECT * FROM Personnel.db
      ORDER BY Name
```

Сортировка записей задана по полю `Name`. Полученный набор данных может иметь вид:

Code	Name	Position	Salary
6	Васин Н.Е.	Водитель	2500
1	Иванов Р.О.	Директор	6700
4	Кузнецов П.А.	Секретарь	3600
2	Петров А.П.	Менеджер	5200
5	Попов А.Л.	Водитель	2400
3	Семенова И.И.	Менеджер	5200

Еще один пример запроса на сортировку — на этот раз по двум полям:

```
SELECT Name, Position, Salary
      FROM Personnel.db
      ORDER BY Position, Salary DESC
```

или

```
SELECT Name, Position, Salary
      FROM Personnel.db
      ORDER BY 2, 3 DESC
```

В набор данных входят поля `Name`, `Position` и `Salary` всех записей. Записи отсортированы по полям `Position` и `Salary`, при этом по полю `Salary` упорядочивание выполняется в порядке убывания значений. Полученный набор данных будет таким:

Name	Position	Salary
Васин Н.Е.	Водитель	2500
Попов А.Л.	Водитель	2400
Иванов Р.О.	Директор	6700
Петров А.П.	Менеджер	5200
Семенова И.И.	Менеджер	5200
Кузнецов П.А.	Секретарь	3600

Если по полям `Position` и `Salary` построен индекс, то операции с набором данных будут выполняться быстрее.

При разработке приложения управление сортировкой осуществляется посредством различных элементов формы, например, кнопок и переключателей.

Рассмотрим следующий пример. Пользователь управляет сортировкой с помощью двух групп переключателей: в первой задается вид, во второй — направление сортировки. Сортировка выполняется после нажатия кнопки **Отсортировать** (`btnSort`). На рис. 8.1 показан вид формы на этапе проектирования.

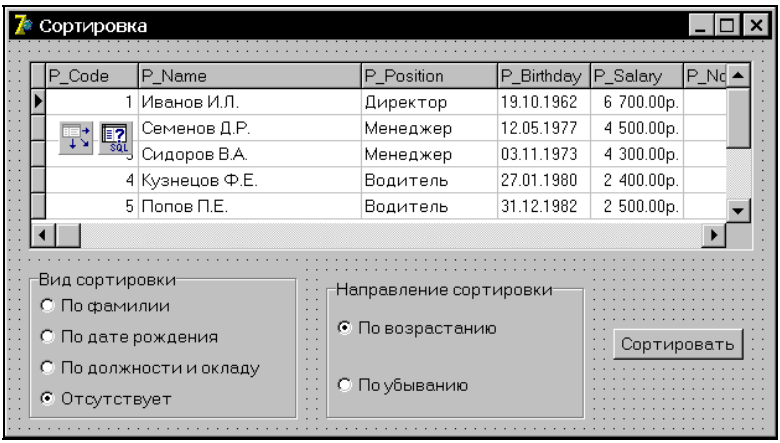


Рис. 8.1. Диалоговое окно сортировки набора данных

Ниже приводится код обработчика события нажатия кнопки `btnSort`.

```
procedure TForm1.btnSortClick(Sender: TObject);
var s: string;
begin
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM Personnel.db');
case RadioGroup2.ItemIndex of
     0: s := '';
     1: s := 'DESC';
     end;
case RadioGroup1.ItemIndex of
     0: s := 'ORDER BY P_Name ' + s;
     1: s := 'ORDER BY P_Birthday ' + s;
     2: s := 'ORDER BY P_Position ' + s + ', P_Salary ' + s;
     3: s := '';
     end;
```

```
Query1.SQL.Add(s);
Query1.Open;
end;
```

При нажатии кнопки `btnSort` для набора данных `Query1` происходят подготовка и выполнение SQL-запроса. Текст запроса формируется в зависимости от состояния переключателей, управляющих сортировкой.

Когда в SQL-запросе отсутствует параметр `ORDER`, по умолчанию записи упорядочиваются по первому полю. Поэтому в рассматриваемом примере отсутствие параметра сортировки и сортировка по полю `Name` приводят к одинаковому результату.

Соединение таблиц

В набор данных можно включать поля из разных таблиц, подобное включение называется *соединением (связыванием)* таблиц. Соединение таблиц может быть внутренним и внешним.

Внутреннее соединение представляет собой простейший случай, когда после слова `SELECT` перечисляются поля разных таблиц. Например:

```
SELECT P.Name, P.Position, P.Salary, A.Birthday, A.Note
      FROM Personnel.db P, Advanced.db A
```

Таблицы `Personnel` и `Advanced` содержат основные и дополнительные сведения о сотрудниках организации. Таблицы связаны отношением "один-к-одному", т. е. каждой записи первой таблицы соответствует одна запись второй таблицы. Результирующий набор данных является объединением полей двух таблиц так, будто дополнительные данные соединены с основными. В таблицах могут быть выбраны не все поля, что не меняет принципа соединения. Результирующий набор будет иметь вид:

Name	Position	Salary	Birthay	Note
Иванов Р.О.	Директор	6700	29.10.51	
Петров А.П.	Менеджер	5200	3.4.62	
Семенова И.И.	Менеджер	5200	12.10.64	
Кузнецов П.А.	Секретарь	3600	7.11.81	Исп. срок до 31.6.2002
Васин Н.Е.	Водитель	2500	20.5.78	
Попов А.Л.	Водитель	2400	3.2.75	

Использование внутреннего соединения применимо не всегда. Например, при использовании внутреннего соединения таблиц с отношением "один-ко-многим" результат выполнения запроса может содержать избыточную информацию.

Рассмотрим в качестве примера запрос, в котором внутреннее соединение таблиц приводит к избыточной информации в результирующей выборке:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
      FROM Store.db S, Cards.db C
```

Набор данных включает поля названия (`S_Name`) и цены (`C_Date`) товара из таблицы `Store` склада, а также поля (`C_Date`) даты и количества (`C_Move`) товара из таблицы `Cards` движения товара. Число записей набора данных является произведением числа записей в таблицах склада и движения товара. Результирующий набор данных может иметь вид:

```
S_Name       C_Date       C_Move   S_Price
Морковь      12.05.02     300      7,5
Морковь      12.05.02     200      7,5
Морковь      13.05.02     -7,50    7,5
Морковь      14.05.02     -30      7,5
Морковь      15.05.02     270      7,5
Морковь      17.05.02     200      7,5
Морковь      18.05.02     -120     7,5
Морковь      20.05.02     -10      7,5
Морковь      21.05.02     250      7,5
Яблоки       12.05.02     300      25
Яблоки       12.05.02     200      25
Яблоки       13.05.02     -7,50    25
...
Помидоры     20.05.02     -10      40
Помидоры     21.05.02     250      40
```

Результирующая выборка содержит избыточную информацию и большое число записей, что не помогает, а, наоборот, мешает пользователю. Поэтому при внутреннем соединении таблиц, связанных соотношением "один-ко-многим", обычно применяются критерии отбора, ограничивающие состав записей. Рассмотрим теперь такой запрос:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
      FROM Store.db S, Cards.db C
      WHERE C.C_Code = S.S_Code
```

В отличие от предыдущего примера, число записей набора данных равно числу записей таблицы движения товара, т. к. отбираются записи, для которых совпадают значения полей кода. Результирующий набор данных будет таким:

```
S_Name       C_Date       C_Move   S_Price
Морковь      12.05.02     300      7,5
Яблоки       12.05.02     200      25
Морковь      13.05.02     -7,50    7,5
Морковь      14.05.02     -30      7,5
Морковь      15.05.02     270      7,5
Яблоки       17.05.02     200      25
Яблоки       18.05.02     -120     25
Морковь      20.05.02     -10      7,5
Помидоры     21.05.02     250      40
```

При внутреннем соединении все таблицы, поля которых указываются в SQL-запросе, являются равноправными.

При *внешнем* соединении таблиц можно указать, какая из таблиц будет *главной*, а какая — *подчиненной*. При использовании внешнего соединения операнд `FROM` имеет следующий формат:

```
FROM <Таблица1> [<Вид соединения>] JOIN <Таблица2>
     ON <Условия отбора>
```

Критерий отбора после слова `ON`, как и ранее, задает условие включения записей в набор данных, связываемые таблицы указываются слева и справа от слова `JOIN`. Какая из двух таблиц будет главной, определяет `Вид соединения`, который может иметь одно из следующих значений:

- `LEFT` (главная таблица указана слева);
- `RIGHT` (главная таблица указана справа) — по умолчанию.

Вот запрос, в котором используется внешнее связывание таблиц:

```
SELECT S.S_Name, C.C_Date, C.C_Move, S.S_Price
       FROM Store.db S LEFT JOIN Cards.db C
            ON C.C_Code = S.S_Code
```

Как и в предыдущем примере, связываются таблицы склада `Store` и движения товара `Cards`. Главной является таблица `Store`.

Модификация записей

Модификация записей заключается в редактировании записей, вставке в набор данных новых записей или удалении существующих записей. При реляционном доступе к БД операции модификации, как и другие операции, выполняются не над одиночными записями, а над группами записей. Группа может состоять и из одной записи.

Для модификации записей используются инструкции `UPDATE`, `INSERT` и `DELETE`, которые соответствующим образом изменяют записи и возвращают в качестве результата набор данных, состоящий из модифицированных записей, удовлетворяющих критерию отбора.

Редактирование записей

Редактирование записей представляет собой изменение значений полей в группе записей. Оно выполняется инструкцией `UPDATE` следующего формата:

```
UPDATE <Имя таблицы>
      SET <Имя поля1> = <Выражение1>,
      ...
      <Имя поляN> = <ВыражениеN>
      [WHERE <Условия отбора>];
```

После выполнения инструкции UPDATE для всех записей, удовлетворяющих условию отбора, изменяются значения полей. Имя поля указывает модифицируемое поле всей совокупности записей, а Выражение определяет значение, которое будет присвоено этому полю. Например:

```
UPDATE Personnel
      SET Salary = Salary + 200
      WHERE Salary < 1500
```

Если сотрудник имеет оклад менее 1500 (рублей), то оклад увеличивается на 200 (рублей).

Критерий отбора, указанный в операнде WHERE, не отличается от критерия, задаваемого в инструкции SELECT. Если он не задан, то изменяются значения всех указанных полей.

Вот соответствующий запрос:

```
UPDATE Store
      SET S_Price = S_Price * 1.28;
```

После его выполнения цена всех товаров увеличивается на 28%.

В одной инструкции UPDATE можно изменить значения *нескольких полей*, в этом случае для каждого из них указывается соответствующее значение. Например:

```
UPDATE Store
      SET S_Quantity = 0,
      S_Note = "Обнулено"
      WHERE S_Quantify BETWEEN - 0.5 AND 0.5
```

Для всех записей, у которых значение поля S_Quantity находится в диапазоне -0,5 .. 0,5, этому полю присваивается значение 0, а в поле S_Note примечания записывается слово Обнулено.

Вставка записей

Вставка записей осуществляется с помощью инструкции INSERT, которая позволяет добавлять к таблицам одну или несколько записей. При добавлении одной записи инструкция INSERT имеет формат:

```
INSERT INTO <Имя таблицы>
      [(<Список полей>)]
      VALUES (<Список значений>);
```

В результате выполнения этой инструкции к таблице, имя которой указано после слова INTO, добавляется одна запись. Для добавленной записи заполняются поля, перечисленные в списке. Значения полей берутся из списка, расположенного после слова VALUES. Список полей и список значений должны соответствовать друг другу по числу элементов и по типу. При присваивании значений для первого поля берется первое значение, для второго — второе и т. д. При этом порядок полей и значений может отличаться от порядка полей в таблице.

Пример запроса на добавление записи:

```
INSERT INTO Store
       (S_Name, S_Price, S_Quantity)
      VALUES ("Торшер", 499.9, 10);
```

Здесь в таблицу склада `Store` добавляется новая запись, в которой присваиваются значения полям названия товара, его цены и количества.

Список полей в инструкции `INSERT` может отсутствовать, в этом случае необходимо указать значения для всех полей таблицы. Порядок и тип этих значений должны соответствовать порядку и типу полей таблицы.

При добавлении к таблице сразу нескольких записей инструкция `INSERT` имеет формат:

```
INSERT INTO <Имя таблицы>
       (<Список полей>)
      Инструкция SELECT;
```

В данном случае значения полей новых записей определяются через значения полей записей, отобранных с помощью инструкции `SELECT`. Число добавленных записей равно числу отобранных записей. Список значений полей, возвращаемых инструкцией `SELECT`, должен соответствовать списку инструкции `INSERT` по числу и типу полей.

С помощью вставки группы записей можно скопировать данные из одной таблицы в другую, например, при резервном копировании или архивировании записей. При этом обе таблицы обычно имеют одинаковую структуру или их структуры частично совпадают.

Вот запрос на добавление нескольких записей:

```
INSERT INTO CardsArchives
       (Code, Move, Date)
       SELECT  C_Code, C_Move, C_Date
       FROM Cards
       WHERE C_Date BETWEEN 1.1.02 AND 31.12.02
```

В архивную таблицу `CardsArchives` добавляется группа записей из таблицы `Cards` движения товара. Для записей, сделанных в 2002 году, в архив копируются код товара, приход или расход и дата.

Если необходимо выполнить не копирование, а перемещение записей в архив, то после успешного копирования можно удалить записи в исходной таблице с помощью инструкции `DELETE`.

Замечание

Перемещение записей целесообразно оформлять в рамках транзакции, чтобы обеспечить надежность выполнения операции и сохранение целостности БД.

Удаление записей

Для удаления группы записей используется инструкция DELETE, имеющая формат:

```
DELETE FROM <Имя таблицы>
       [WHERE <Условия отбора>];
```

В результате выполнения этой инструкции из таблицы, имя которой указано после слова FROM, удаляются все записи, которые удовлетворяют критерию отбора.

Замечание

Если критерий отбора не задан, то из таблицы будут удалены все записи.

Вот соответствующий запрос:

```
DELETE FROM Store
       WHERE S_Quantity = 0
```

Из таблицы склада Store удаляются все записи о товаре, которого нет на складе. Отметим, что если с записями этой таблицы связаны записи другой таблицы, например, движения товара, то может потребоваться их предварительное удаление, что связано с действием бизнес-правил и налагаемыми ими ограничениями.

Статический и динамический запросы

Как отмечалось, в зависимости от способа формирования SQL-запрос может быть

- статическим;
- динамическим.

Текст *статического* SQL-запроса формируется при разработке приложения и в процессе выполнения приложения не может быть изменен. Такой запрос обычно используется в случаях, когда код запроса заранее известен и во время работы приложения не требует модификации.

Динамический SQL-запрос формируется или изменяется при выполнении приложения. Такой запрос обычно применяется, если его текст зависит от действий пользователя, например, при управлении сортировкой набора данных, когда в тексте запроса изменяются названия полей и добавляется или исключается описатель DESC.

Рассмотрим в качестве примера процедуру, в которой осуществляется формирование динамического запроса:

```
procedure TForm1.btnSortClick(Sender: TObject);
var str: string;
begin
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM Personnel.db');
```

```
case RadioGroup1.ItemIndex of
     0: str := 'ORDER BY Name ';
     1: str := 'ORDER BY BirthDay ';
end;
case RadioGroup2.ItemIndex of
     0:;
     1: str := str + ' DESC';
     end;
Query1.SQL.Add(str);
Query1.Open;
end;
```

При нажатии кнопки `btnSort` список сотрудников сортируется по полям `Name` или `BirthDay`. Пользователь управляет выбором поля с помощью группы переключателей `RadioGroup1`. В группе `RadioGroup2` выбирается порядок сортировки.

Для настройки статического SQL-запроса во время выполнения приложения в его тексте можно использовать параметры. *Параметр* это специальная переменная, перед именем которой ставится двоеточие в тексте запроса. Двоеточие не является частью имени параметра и ставится только в тексте запроса. Как и для обычных переменных программы, в процессе выполнения приложения вместо параметра подставляется его значение.

Параметры удобно использовать для передачи в текст SQL-запроса внешних значений. Например, если необходимо вывести фамилии сотрудников с окладом не менее указанного в редакторе `Edit1`, то организовать формирование и выполнение динамического запроса можно так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT Name, Position, Salary');
Query1.SQL.Add('FROM Personnel.db');
Query1.SQL.Add('WHERE Salary >= ' + Edit1.Text);
Query1.Open;
end;
```

С помощью параметров эту задачу можно решить проще, например, через включение в текст запроса параметра `prmSalary`:

```
SELECT Name, Position, Salary
       FROM Personnel.db
       WHERE Salary >= :prmSalary
```

Система Delphi автоматически учитывает все указанные в SQL-запросе параметры в специальном списке параметров, являющемся для набора данных `Query`

значением свойства `Params` типа `TParams`, представляющим собой массив. Это свойство позволяет получить доступ к каждому параметру как при разработке, так и при выполнении приложения. Чтобы обратиться к параметру во время выполнения приложения, следует указать его номер (индекс) в списке параметров. Например, для обращения ко второму параметру указывается `Params[1]`. На этапе разработки приложения можно вызвать Редактор параметров (рис. 8.2) в Инспекторе объектов. Тип каждого параметра необходимо указать в свойстве `DataType` типа `TFieldType`. Например, для параметра `prmSalary` установлен тип `ftFloat`.

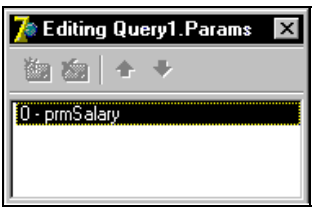


Рис. 8.2. Окно Редактора параметров

В последующем перед выполнением запроса вместо параметра необходимо подставить его значение, в данном случае из редактора `Edit1`. Ниже приведен код обработчика события нажатия кнопки `Button1`, выполняющий указанное действие.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Close;
Query1.ParamByName('prmSalary').AsFloat := StrToFloat(Edit1.Text);
Query1.Open;
end;
```

Для доступа к параметру во время выполнения приложения используется метод `ParamByName`, отличающийся от аналогичного метода `FieldByName` тем, что вместо имени поля указывается имя параметра.

При использовании параметров можно не изменять текст SQL-запроса во время выполнения приложения и тем не менее передавать в него различные значения. То есть статический запрос как бы превращается в динамический.

Замечание

Статические запросы, в которых использованы параметры, иногда также называют *изменяющимися* (т. е. фактически динамическими).

Обычно текст SQL-запроса проверяется и выполняется при каждом открытии набора данных `Query`. Если текст запроса при выполнении приложения не изменяется, то его можно предварительно подготовить, а после этого только ис-

пользовать такой подготовленный к выполнению запрос. Это позволяет ускорить обработку статических запросов, в том числе имеющих параметры.

Подготовку запроса к выполнению осуществляет метод `Prepare`, который можно вызывать при создании формы. Чтобы определить, была ли произведена предварительная подготовка запроса, необходимо проанализировать свойство `Prepared` типа `Boolean`, которое после вызова метода `Prepare` устанавливается в значение `True`.

Если текст подготовленного к выполнению запроса изменился (к изменению значений параметров это не относится), то автоматически вызывается метод `UnPrepare`, и свойство `Prepared` устанавливается в `False`.

Глава 9

Технология dbExpress

Общая характеристика

В основе технологии dbExpress лежит использование множества легковесных драйверов, компонентов, объединяющих соединения, транзакции, запросы и наборы данных, а также интерфейсов, реализующих универсальный доступ к соответствующим функциям.

По сравнению с использованием механизма BDE технология dbExpress обеспечивает построение более легковесных (по объему кода) приложений для работы с базами данных. При ее применении для доступа к данным используются SQL-запросы. Технология dbExpress обеспечивает легкую переносимость приложений, допускает работу приложений баз данных под управлением Windows и Linux.

Для использования технологии dbExpress достаточно включить в распространяемое приложение динамически подключаемую библиотеку с драйвером, взаимодействующим с клиентским программным обеспечением для нужного сервера базы данных (рис. 9.1).

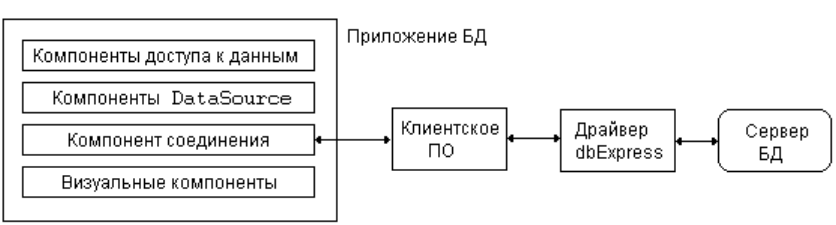


Рис. 9.1. Схема доступа к данным с помощью dbExpress

Драйверы, используемые для доступа к различным серверам баз данных по технологии dbExpress, реализованы в виде динамически подключаемых библиотек (DLL) (табл. 9.1). Драйверы dbExpress в составе Delphi размещаются в папке ..\Borland\Delphi7\BIN.

Таблица 9.1. *Драйверы БД*

Драйвер сервера БД	Динамическая библиотека
Interbase dbExpress	Dbexpint.dll
MySQL dbExpress	Dbexpmysql.dll
Old MySQL dbExpress	Dbexpmys.dll
DB2 dbExpress	Dbexpdb2.dll
Oracle dbExpress	Dbexpora.dll
dbExpress for Microsoft SQL Server	Dbexpmss.dll
Informix dbExpress	Dbexpinf.dll

На странице **dbExpress** Палитры компонентов Delphi находятся компоненты, используемые в технологии dbExpress: `SQLConnection` (`Database`), `SQLDataSet`, `SQLQuery` (`Query`), `SQLStoredProc` (`StoredProc`), `SQLTable` (`Table`), `SQLMonitor` (утилита SQL Monitor) и `SimpleDataSet` (`BDEClientDataSet`) *(см. главу 2)*. Для наглядности в круглых скобках указаны аналоги этих компонентов, используемые в случае механизма BDE. Как видим из приведенного списка, у компонента `SQLDataSet` нет аналога для механизма BDE. Кроме того, ряд компонентов из механизма BDE не имеет аналогов для технологии dbExpress.

Определенным недостатком технологии dbExpress является то, что несколько из перечисленных компонентов (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) являются однонаправленными наборами данных, в которых отсутствует буферизация. Эти наборы данных обеспечивают более быстрый доступ к данным и предъявляют меньшие требования к ресурсам, но при этом на них накладываются заметные ограничения. Для компонента `SimpleDataSet` большинство из этих ограничений не действует.

Установление соединения с сервером

Для установления соединения с сервером базы данных служит компонент `SQLConnection`, который представляет собой аналог компонента `DataBase` в BDE.

Этот компонент взаимодействует с двумя файлами, расположенными в каталоге ..\Common Files\Borland Shared\DBExpress. Файл dbxdrivers.ini содержит список инсталлированных драйверов серверов БД и для каждого драйвера список динамически подключаемых библиотек и параметров соединений, установленных по умолчанию. Список соединений с параметрами соединений содержится в файле dbxconnections.ini.

Поместив компонент `SQLConnection` в форму (или модуль данных) на этапе разработки приложения, можно выбрать одно из существующих соединений,

либо создать новое соединение с помощью диалогового окна Редактора соединений (рис. 9.2). Вызов указанного окна можно выполнить выбором пункта **Edit Connection Properties** контекстного меню компонента.

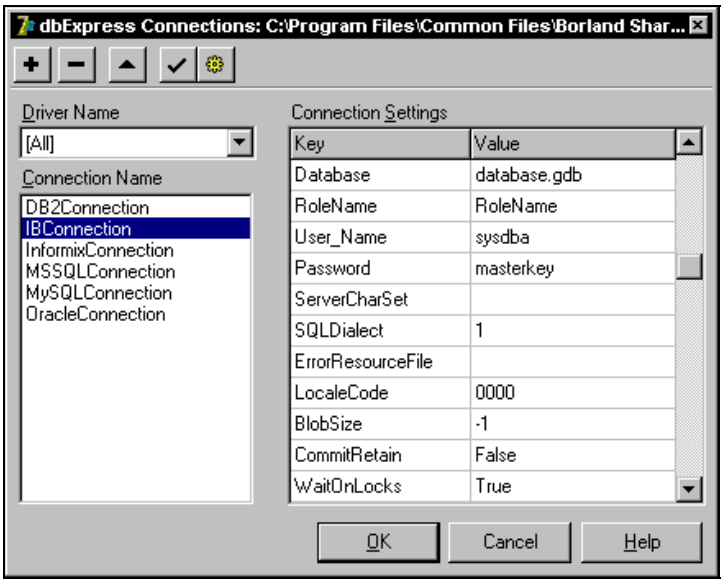


Рис. 9.2. Диалоговое окно Редактора соединений

Редактор соединений позволяет настроить существующее соединение dbExpress или создать новое соединение, а также проверить правильность настроек с помощью кнопки ☑ (**Test Connection**). Параметры соединения можно настраивать также с помощью Инспектора объектов (свойство `Params` типа `TStrings`).

Параметры соединения для основных серверов баз данных приведены в табл. 9.2.

Таблица 9.2. *Параметры соединений для серверов БД*

Сервер БД	Параметр	Значение
Все серверы	`BlobSize`	Ограничение на объем пакета для данных типа BLOB
	`DriverName`	Имя драйвера
	`ErrorResourceFile`	Файл сообщений об ошибках
	`LocaleCode`	Код локализации, определяющий влияние национальных символов на сортировку данных
	`User_Name`	Имя пользователя

Таблица 9.2 *(окончание)*

Сервер БД	Параметр	Значение
	Password	Пароль
	<имя сервера БД> TransIsolation	Уровень изоляции транзакций для сервера с заданным именем
InterBase	CommitRetain	Поведение курсора по завершению транзакции: True — обновление; False — удаление
	RoleName	Роль пользователя
	SQLDialect	Используемый диалект SQL
	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих их до полной строки
	WaitOnLocks	Разрешение на ожидание занятых ресурсов
	DataBase	Имя файла базы данных (с расширением gdb)
DB2	DataBase	Имя клиентского ядра
MySQL	DataBase	Имя базы данных
Microsoft SQL Server 2000, MySQL, Informix	HostName	Имя компьютера, на котором установлен сервер
Microsoft SQL Server 2000, Informix	DataBase	Псевдоним (алиас) имени базы данных
Microsoft SQL Server 2000, Oracle	OS Autenification	Использование учетной записи текущего пользователя операционной системы при доступе к ресурсам сервера
Informix, Oracle	Trim Char	Возможность удаления из строковых значений полей пробелов, дополняющих значение до полной строки
Oracle	AutoCommit	Флаг завершения транзакции. Устанавливается сервером
	BlockingMode	Режим завершения запроса. True — соединение дожидается окончания запроса; False — начинает выполнение следующего запроса
	Multiple Transaction	Возможность управления несколькими транзакциями в одном сеансе
	DataBase	Запись базы данных в файле TNSNames.ora

После описанной настройки параметров соединений для компонента SQLConnection с помощью Инспектора объектов можно выбрать соединение

(свойство `ConnectionName`) для нужного сервера баз данных. При этом автоматически устанавливаются значения связанных с ним свойств: `DriverName` (имя драйвера); `LibraryName` (имя динамически подключаемой библиотеки драйвера); `Params` (параметры соединения) и `VendorLib` (имя динамически подключаемой библиотеки клиентской части сервера).

Открытие соединения с сервером БД осуществляется заданием свойству `Connected` типа `boolean` значения `True`. Например, при выполнении приложения это можно сделать с помощью кода:

```
SQLConnection1.Connected := True;
```

Соответственно закрыть соединение с сервером можно путем задания этому свойству значения `False`. Названные операции могут быть выполнены также с помощью методов:

```
procedure Open;
procedure Close;
```

С помощью свойства `LoginPrompt` типа `boolean` выполняется задание необходимости (`True`) или ненужности (`False`) отображения окна авторизации для ввода имени пользователя (`User_Name`) и пароля (`Password`) при каждой попытке соединения с сервером. В последнем случае эти значения будут браться из файла dbxconnections.ini.

После открытия соединения все компоненты dbExpress, инкапсулирующие наборы данных и связанные с открытым компонентом `SQLConnection`, получают доступ к базе данных.

При открытии и закрытии соединения можно воспользоваться любым из доступных событий, например:

```
property AfterConnect: TNotifyEvent;
property AfterDisconnect: TNotifyEvent;
property BeforeConnect: TNotifyEvent;
property BeforeDisconnect: TNotifyEvent;
property OnLogin: TSQLConnectionLoginEvent;
```

Последнее событие наступает в случае, если свойству `LoginPrompt` установлено значение `True`. Если это свойство имеет значение `True`, и нет обработки события `OnLogin`, то пользователю будет предложен стандартный диалог авторизации.

Текущее состояние соединения указывает `ConnectionState` типа `TConnectionState`, тип которого описан так:

```
TConnectionState = (csStateClosed, csStateOpen, csStateConnecting,
  csStateExecuting, csStateFetching, csStateDisconnecting);
```

Указанные в определении типа значения соответствуют следующим состояниям соединения: закрыто, открыто, установление соединения, ожидание исполнения переданного SQL-запроса, получение данных с сервера, завершение соединения.

Компонент `SQLConnection` позволяет выполнять SQL-запросы с помощью следующих функций.

Функция `Execute(const SQL: string; Params: TParams; ResultSet: Pointer = Nil): Integer;` выполняет запрос, определенный значением константы `SQL` с параметрами `Params`, используемыми в SQL-запросе. Используемые в запросе параметры должны иметь тот же порядок следования, что и в `Params` и в инструкции SQL. Если SQL-запрос не содержит параметров, то свойству `Params` требуется задать значение `Nil`. Если при выполнении SQL-запроса возвращается результат, то в параметр `ResultSet` помещается указатель на объект типа `TCustomSQLDataSet`, содержащий результат.

При отсутствии в запросе параметров и возвращаемых записей рекомендуется использовать функцию `ExecuteDirect(const SQL: string): Integer;`, которая возвращает значение 0 при успешном завершении запроса или код ошибки в противном случае.

Подобно своим аналогам в BDE *(см. главу 7)* компонент `SQLConnection` позволяет выполнять запуск, фиксацию и откат *транзакций* соответственно с помощью методов: `StartTransaction`, `Commit` и `Rollback`.

Компоненты доступа к данным

Как отмечалось, ряд компонентов доступа к данным по технологии dbExpress (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) принадлежит к однонаправленным наборам данных, в которых отсутствует буферизация и при этом на них накладываются заметные ограничения.

В однонаправленных наборах данных используются однонаправленные курсоры. С их помощью допускается только получать данные, из методов навигации по набору данных поддерживаются лишь методы `First` и `Next`, подразумевающие последовательный перебор записей от начала к концу.

При использовании однонаправленных наборов данных отсутствует возможность прямого редактирования данных, т. к. для этого нужно размещение в буфере результатов редактирования. При этом соответствующее свойство названных компонентов (`CanModify`) всегда имеет значение `False`. Для однонаправленных наборов данных возможности редактирования данных все же доступны, например, путем задания оператора `UPDATE` языка SQL. Кроме того, для однонаправленных наборов данных по тем же причинам имеют место ограничения по выполнению фильтрации. Это ограничение также можно обойти путем указания параметров фильтрации в SQL-запросах.

Имеются также ограничения по отображению данных с помощью компонентов страницы **Data Controls**. В частности, нельзя использовать компоненты `DBGrid` и `DBCtrlGrid`, а также компоненты синхронного просмотра. Для компонента навигатора требуется отключить кнопки возврата на последнюю запись и перехода на последнюю запись. Остальные компоненты можно использовать как обычно.

Для компонента `SimpleDataSet` большинство из указанных ограничений не действует. Он использует внутренние компоненты типа `TSQLDataSet` и `TDataSetProvider` для получения и изменения данных.

Перед использованием любой из указанных компонентов доступа к данным следует подключить к серверу БД. Для этого достаточно его свойству `SQLConnection` в качестве значения установить имя компонента соединения `SQLConection`.

Компоненты доступа к данным можно использовать, поместив в форме, либо динамически — создав при выполнении приложения.

Универсальный доступ к данным

Для обеспечения универсального однонаправленного доступа к данным БД по технологии dbExpress служит компонент `SQLDataSet`. Он позволяет получать все записи из таблицы БД, производить выборку данных путем выполнения SQL-запросов или выполнять хранимые процедуры.

Для определения типа выполняемых действий (с запросами, таблицами или хранимыми процедурами) свойству `CommandType` рассматриваемого компонента нужно установить соответственно одно из трех возможных значений:

- `ctQuery` — в свойстве `CommandText` указывается SQL-запрос;
- `ctTable` — в свойстве `CommandText` указывается имя таблицы на сервере БД, при этом компонент автоматически генерирует SQL-запрос для получения всех записей для всех полей таблицы;
- `ctStoredProc` — в свойстве `CommandText` указывается имя хранимой процедуры.

В случае выбора значения `ctQuery` текст SQL-запроса можно ввести в поле свойства `CommandText` с помощью Инспектора объектов или редактора построения запроса `CommandTextEditor` (рис. 9.3). В списке **Tables** доступны имена таблиц в базе данных, а в списке **Fields** — имена полей выбранной таблицы. С помощью кнопок **Add Table to SQL** и **Add Field to SQL** можно добавить нужные таблицы и поля. При этом в поле **SQL** отображается автоматически сформированный SQL-запрос. После нажатия **OK** в окне редактора мы получаем нужное значение свойства `CommandText`.

При выполнении приложения аналогичные действия (как на рис. 9.3) для выполнения SQL-запроса можно задать так:

```
SQLDataSet1.CommandType := ctQuery;
SQLDataSet1.CommandText := 'select COUNTRY, CURRENCY from DEPARTMENT,
DEPT_NO, DEPARTMENT from COUNTRY ';
SQLDataSet1.ExecSQL;
```

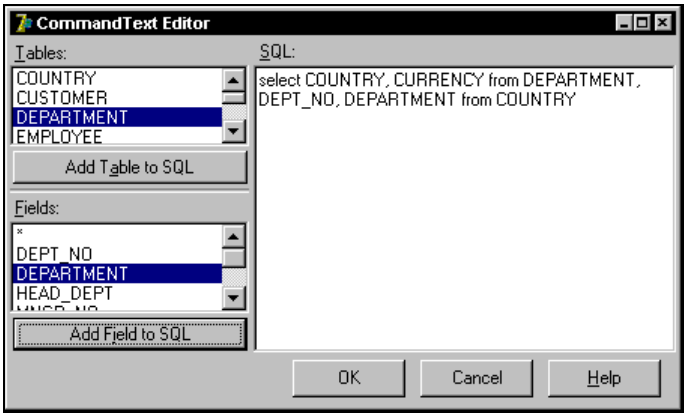


Рис. 9.3. Редактор построения SQL-запроса

При выборе значения `ctTable` в свойстве `CommandText` указывается просто имя таблицы, например, так:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'COUNTRY';
SQLDataSet1.ExecSQL;
```

Для открытия набора данных нужно установить значение `True` свойству `Active` или использовать метод `Open`. Если же SQL-запрос или хранимая процедура не возвращают набор данных, для их выполнения используется метод

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Параметр `ExecDirect` определяет необходимость подготовки параметров перед выполнением инструкции. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `False`.

При необходимости сортировки данных используют инструкцию `ORDER BY` в случае варианта `ctQuery`, либо устанавливают с помощью редактора SortFields Editor значение свойства `SortFieldNames` в случае варианта `ctTable`. Для варианта `ctStoredProc` порядок сортировки определяется в хранимой процедуре.

При необходимости в SQL-запросе или в хранимой процедуре можно использовать параметры, позволяющие настраивать запрос или процедуру, не изменяя их код *(см. главу 8)*. Параметры задаются с помощью свойства `Params`, которое является коллекцией объектов типа `TParam`.

Свойство `DataSource` типа `TDataSource` компонента `SQLDataSet` позволяет связать два набора данных по схеме "мастер-детальный" *(см. главу 2)*.

Метод `SetSchemaInfo` компонента `SQLDataSet` позволяет получить метаданные (список таблиц на сервере, список системных таблиц, информация о хранимых процедурах, информация о полях таблицы, параметры хранимой процедуры).

Рассмотрим пример формы (рис. 9.4) приложения БД, в котором с помощью компонентов `SQLDataSet` выполняется доступ к данным по технологии dbExpress. Пусть требуется обеспечить возможность просмотра и редактирования полей данных `'CUST_NO'`, `'CUSTOMER'` и `'PHONE_NO'` таблицы `CUSTOMER`, принадлежащей базе данных EMPLOYEE.GDB сервера InterBase.

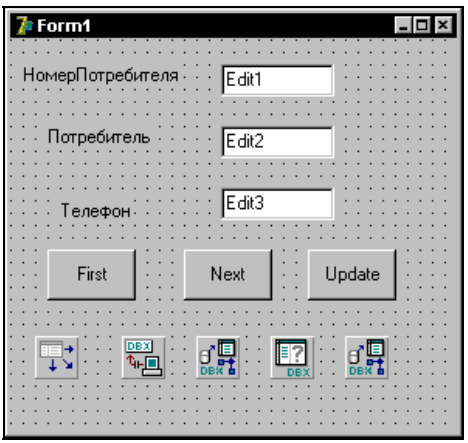


Рис. 9.4. Вид формы при разработке

Код модуля формы для решения поставленной задачи имеет вид:

```
unit UnitSQLDataSet;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, DBXpress, FMTBcd, DB, SqlExpr, StdCtrls, Mask, DBCtrls;

type
  TForm1 = class(TForm)
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
    DataSource1: TDataSource;
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
```

```
    SQLQuery1: TSQLQuery;
    SQLDataSet2: TSQLDataSet;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    procedure SQLDataSet1AfterScroll(DataSet: TDataSet);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.SQLDataSet1AfterScroll(DataSet: TDataSet);
begin
Edit1.Text:=SQLDataSet1.FieldByName('CUST_NO').AsString;
Edit2.Text:=SQLDataSet1.FieldByName('CUSTOMER').AsString;
Edit3.Text:=SQLDataSet1.FieldByName('PHONE_NO').AsString;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
SQLDataSet1.First;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
 SQLDataSet1.Next;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
with SQLDataSet2 do
```

```
try
CommandText := 'UPDATE CUSTOMER SET CUST_NO = :Cust_no, CUSTOMER =
:Customer, PHONE_NO = :Phone_no WHERE CUST_NO = :CUST_NO';
Params[0].AsString := Edit1.Text;
Params[1].AsString := Edit2.Text;
Params[2].AsString := Edit3.Text;
ExecSQL;
except
  MessageDlg('Customer info fixed error',mtError,[mbOK],0);
end;
end;
end.
```

Для просмотра и редактирования трех указанных полей таблицы `CUSTOMER` используются два компонента `SQLDataSet1` и `SQLDataSet2` соответственно. Оба компонента подсоединены к таблице сервера InterBase через компонент `SQLConnection1`. Первый компонент `SQLDataSet1` служит для просмотра записей таблицы `CUSTOMER`. Второй компонент `SQLDataSet2` служит для фиксации внесенных изменений (три компонента типа `TEdit`) в текущую запись в таблице на сервере БД.

Для заполнения компонентов типа `TEdit` при навигации по набору данных используется обработчик события `AfteScroll` для компонента `SQLDataSet1`. Навигация по набору данных выполняется с помощью обработчиков событий `OnClick` для кнопок с заголовками **First** и **Next**. Фиксация внесенных в просматриваемую запись изменений в таблице на сервере осуществляется с помощью SQL-запроса `UPDATE`, размещенного в обработчике события `OnClick` для кнопки с заголовком **Update**. В параметрах запроса передаются текущие значения полей из компонентов `Edit1`, `Edit2` и `Edit3` (рис. 9.5).

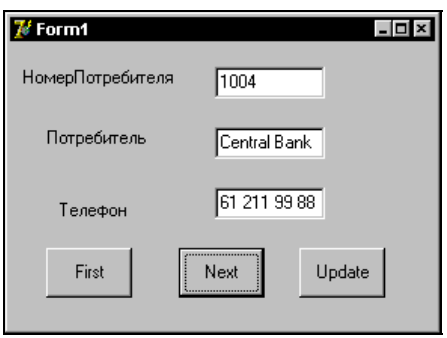


Рис. 9.5. Вид формы при выполнении приложения

Отметим, что необходимость использования компонента SQLDataSet2 для фиксации внесенных изменений в таблице на сервере вызвано тем, что компонент SQLDataSet1 является однонаправленным набором данных. Заметим также, что в нашем примере нельзя вместо компонентов Edit использовать компоненты DBEdit, т. к. здесь они не позволяют выполнять редактирование содержимого полей.

Просмотр таблиц

Для просмотра таблиц по технологии dbExpress служит компонент SQLTable. Компонент SQLTable генерирует SQL-запрос для получения всех строк и полей указанной таблицы.

Имя таблицы определяется с помощью свойства TableName, и при подключении компонента к соединению его можно выбрать в комбинированном списке в окне Инспектора объектов.

Для получения табличного набора данных компонент SQLTable с помощью метода

```
procedure PrepareStatement; override;
```

генерирует запрос на сервер БД.

Порядок следования данных в наборе определяется свойством IndexFieldNames типа String или IndexName типа String.

Список индексов таблицы можно получить с помощью метода GetIndexNames(List: TStrings) в качестве значения параметра List.

Связь между двумя наборами данных "главный-подчиненный" устанавливается с помощью свойств MasterFields и MasterSource. На этапе разработки приложения двойным щелчком на свойстве MasterFields в окне Инспектора объектов можно открыть диалоговое окно редактора связей для визуального построения отношения "мастер-детальный".

Компонент SQLTable является однонаправленным курсором, тем не менее, он допускает удаление записей из таблицы на сервер БД с помощью метода DeleteRecords.

Выполнение SQL-запроса

Для выполнения SQL-запроса на сервере БД служит компонент SQLQuery. Он позволяет представлять результаты выполнения запросов с помощью инструкции SELECT или осуществлять действия по изменению БД путем выполнения инструкций INSERT, DELETE, UPDATE, ALTER TABLE и т. п.

Для компонента SQLQuery текст SQL-запроса определяется как значение свойства SQL типа TStrings. На этапе разработки приложения текст SQL-запроса может быть набран непосредственно в окне редактора строк (или ре-

дактора кода), открываемого двойным щелчком на свойстве SQL в окне Инспектора объектов.

При выполнении приложения очистка содержимого свойства SQL компонента SQLQuery и его модификация могут быть выполнены, к примеру, с помощью следующего кода:

```
SQLQuery1.SQL.Clear;
SQLQuery1.SQL.Add('SELECT ' + Edit1.Text + ' FROM ' + Edit2.Text);
if Length(Edit3.Text) <> 0 then
  SQLQuery1.SQL.Add('ORDER BY ' + Edit3.Text)
```

Свойство Text типа String в качестве значения содержит строковое представление SQL-запроса в виде одной строки, а не нескольких, как это может быть в случае свойства SQL.

Свойство Active типа Boolean определяет, открыт набор данных или нет. Это свойство обычно используют, чтобы определить или установить заполнение набора данными.

Открыть набор данных SQL-запроса также можно с помощью метода Open, либо с помощью метода

```
function ExecSQL(ExecDirect: Boolean = False): Integer; override;
```

Здесь значение False параметра ExecDirect означает, что запрос не имеет настраиваемых параметров.

Свойство DataSource связывает набор данных SQL-запроса с другим (главным) набором данных. Этот набор данных используется для получения параметров запроса в случае, когда предусматривается параметризованный запрос, но приложение этими параметрами не обеспечивает.

Выполнение хранимых процедур

Для выполнения хранимых процедур, размещенных на сервере БД, служит компонент SQLStoredProc.

Имя хранимой процедуры задает свойство StoredProcName типа String. Для задания параметров хранимой процедуры предназначено свойство Params типа TParams. При обращении к параметрам хранимой процедуры целесообразно использовать метод ParamByName. Это обусловлено тем, что при работе с некоторыми серверами порядок следования параметров до и после выполнения процедуры может меняться.

Для подготовки хранимой процедуры к выполнению на сервере служит метод PrepareStatement(var RecordsAffected: Integer): TCustomSQLDataSet;. При его вызове сервером БД выделяются ресурсы и связываются их параметры. Поименованные параметры временно преобразуются к непоименованным параметрам, поскольку dbExpress поименованные параметры не поддерживает.

Если хранимая процедура не возвращает набор данных, то ее запускают с помощью метода

```
function ExecProc: Integer; virtual;
```

В противном случае используется метод `Open` либо свойству `Active` задают значение `True`.

Компонент редактирования набора данных

Для редактирования набора данных (получения данных, их кэширования и отправления измененных данных на сервер) предназначен компонент `SimpleDataSet`. Этот компонент использует двунаправленный курсор и позволяет редактировать данные, но в режиме редактирования. Тем самым он исправляет основные недостатки рассматриваемой технологии.

Для подготовки компонента `SimpleDataSet` к работе с данными нужно с помощью свойства `Connection` связать его с компонентом соединения `SQLConnection`. В качестве альтернативы можно с помощью подсвойства `ConnectionName` свойства `Connection` задать тип соединения непосредственно.

Произведенные над данными изменения размещаются в локальном кэше, в связи с этим для подтверждения изменений и отправки данных на сервер БД используют метод

```
function ApplyUpdates(MaxErrors: Integer); Integer; virtual;
```

Здесь параметр метода определяет число ошибок, допустимых при передаче данных.

Локальный кэш компонента после сохранения изменений на сервере можно очистить от данных с помощью метода

```
function Reconcile(const Results: OleVariant): Boolean;
```

Отмена локальных изменений данных может быть выполнена с помощью метода

```
procedure CancelUpdates;.
```

Пересылка данных между сервером и рассматриваемым компонентом осуществляется с помощью пакетов. Размер пакета (по числу записей) можно задать с помощью свойства `PacketRecords` типа `Integer`. По умолчанию устанавливается значение −1, которое означает, что один пакет должен содержать все записи набора данных.

Если значение свойства `PacketRecords` больше 0, то оно определяет число записей, которые можно получить в пакете от провайдера с помощью метода

```
function GetNextPacket: Integer;
```

Если значение свойства `PacketRecords` равно 0, то в пакете передаются только метаданные.

Свойство `DataSize` типа `Integer` устанавливает размер (в байтах) текущего пакета, доступного с помощью свойства `Data` типа `OleVariant`.

На общее число записей в источнике данных указывает свойство `RecordCount` типа `Integer`, номер текущей записи определяет свойство `RecNo` типа `Integer`.

Изменения в текущей записи можно отменить с помощью метода

```
procedure RevertRecord;
```

Отменить последнюю операцию по изменению клиентского источника данных можно с помощью метода

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Здесь значение параметра определяет, где будет установлен курсор после восстановления записи: `True` — на восстановленной записи, `False` — на текущей записи.

Обновить значение полей для текущей записи с сервера можно с помощью метода `RefreshRecord`.

Рассмотрим пример формы (рис. 9.5) приложения БД, в котором с помощью компонента `SimpleDataSet1` выполняется доступ к данным по технологии dbExpress. Пусть требуется обеспечить возможность просмотра и редактирования записей таблицы `DEPARTMENT`, принадлежащей базе данных EMPLOYEE.GDB сервера InterBase.

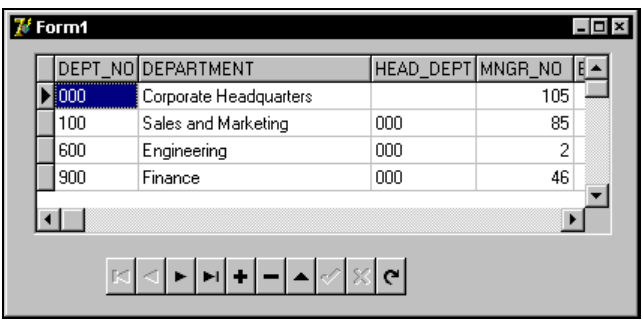


Рис. 9.6. Вид формы `UnitSimpleDataSet` при выполнении приложения

Код модуля формы для решения поставленной задачи имеет вид:

```
unit UnitSimpleDataSet;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,
  Dialogs, FMTBcd, DBXpress, DB, SqlExpr, ExtCtrls, DBCtrls, Grids,
  DBGrids, DBClient, SimpleDS;
```

```
type
  TForm1 = class(TForm)
    SimpleDataSet1: TSimpleDataSet;
    SQLConnection1: TSQLConnection;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    DBNavigator1: TDBNavigator;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure SimpleDataSet1AfterPost(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
SimpleDataSet1.Open;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 SimpleDataSet1.Active:=False;
end;

procedure TForm1.SimpleDataSet1AfterPost(DataSet: TDataSet);
begin
    SimpleDataSet1.ApplyUpdates(-1);
end;
end.
```

Компонент `SimpleDataSet1` работает в табличном режиме. При разработке формы подсвойству `CommandType` свойства `DataSet` задано значение `ctTable`, с помощью подсвойства `CommandText` указано имя таблицы `DEPARTMENT`.

Для отображения записей редактируемого набора данных и навигации по нему используются компоненты `DBGrid1` и `DBNavigator1` соответственно. Фиксация внесенных при редактировании изменений и отправка данных на сервер осуществляются в обработчике события `AfterPost`, возникающего при нажатии кнопки `Post` (утвердить результат изменения записи) компонента `DBNavigator1` и при переходе в компоненте `DBGrid1` на другую строку *(см. главу 6)*.

Отладка соединения с сервером

Для получения информации о местах ошибок в SQL-запросах при отладке соединения приложения с сервером БД служит компонент `SQLMonitor`. Он осуществляет перехват сообщений между соединением и сервером баз данных и помещает их в список строк, определяемый свойством `TraceList` типа `TStrings`.

Для начала работы с компонентом нужно установить связь с соединением через свойство `SQLConnection` и активизировать его, задав свойству `Active` значение `True`.

Перед каждой записью сообщений в список строк возникает событие `OnTrace` типа

```
TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc;
     var LogTrace: Boolean) of object;
```

а сразу после записи в список возникает событие `OnLogTrace` типа

```
TTraceLogEvent = procedure (Sender: TObject; CBInfo: pSQLTRACEDesc)
   of object;
```

Содержимое списка можно сохранить в файле на жестком диске с помощью метода `SaveToFile`.

При задании свойству `AutoSave` типа `Boolean` значения `True` данные о проходящих командах автоматически заносятся в текстовый файл с именем, заданным значением свойства `FileName` типа `String`.

Свойство `MaxTraceCount` типа `Integer` определяет максимальное число контролируемых команд, а также управляет процессом контроля. При значении −1 ограничения снимаются, а при значении 0 контроль не выполняется.

На число сохраненных в списке команд указывает свойство `TraceCount` типа `Integer`.

Рассмотрим следующий пример: пусть требуется выполнять оперативное отображение информации о последнем сообщении в окне многострочного редактора (компонент `memo1`).

Для решения названной задачи подходит следующий код:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject;
  CBInfo: pSQLTRACEDesc);
  var
   LogFileName: string;
```

```
begin
   with Sender as TSQLMonitor do
   begin
    memo1.Lines.Clear;
    memo1.Lines:=TraceList;
    TraceList.Clear; {очистка списка сообщений}
   end;

end;
```

Как видно из приведенного кода, мы использовали обработчик события `OnLogin`.

Замечание

Для автономного dbExpress-приложения нужны две динамически подключаемые библиотеки (DLL). Первая библиотека содержит драйвер dbExpress, например, для сервера БД InterBase это dbexpint.dll (см. табл. 9.1). Вторая библиотека есть midas.dll, используемая для поддержки компонента `SimpleDataSet`. Обе библиотеки могут быть откомпилированы вместе с проектом и включены в exe-файл распространяемого приложения.

При использовании компонента `SimpleDataSet` в приложении Delphi может возникать ошибка, сопровождаемая сообщением: "Error loading Midas.dll". Возможной причиной ее появления является неправильная регистрация названной библиотеки в среде операционной системы. Для выполнения регистрации библиотеки Midas.dll с целью устранения этой ошибки при работе под управлением Windows 2000 достаточно запустить на выполнение утилиту regsvr32.exe. Обычно она размещается в системном каталоге Windows, например, \winnt\system32.

Глава 10

Технология ADO

Общая характеристика

Технология Microsoft ActiveX Data Objects (ADO) представляет собой универсальный механизм доступа к различным источникам данных из приложений баз данных. Основу технологии ADO составляет использование набора интерфейсов общей модели объектов COM, описанных в спецификации OLE DB. Достоинством этой технологии является то, что базовый набор интерфейсов OLE DB имеется в каждой современной операционной системе Microsoft. Отсюда следует простота обеспечения доступа приложения к данным. При применении технологии ADO (рис. 10.1) приложение БД может использовать данные из электронных таблиц, таблиц локальных и серверных баз данных, XML-файлов и т. д.

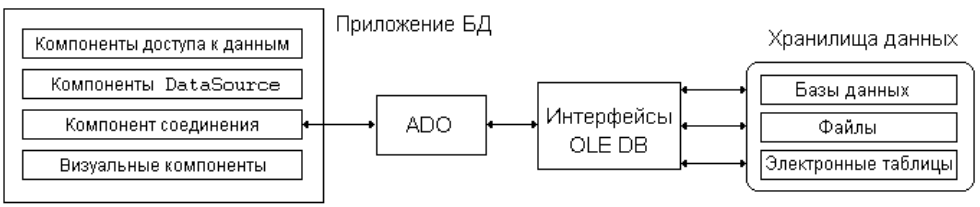


Рис. 10.1. Схема доступа к данным по технологии ADO

В соответствии с терминологией ADO, любой источник данных (базу данных, файл, электронную таблицу) называют *хранилищем данных*. Приложение взаимодействует с хранилищем данных с помощью *провайдера*. Для каждого типа хранилища данных используется свой провайдер ADO. Провайдер обеспечивает обращение к данным хранилища с запросами, интерпретацию возвращаемой служебной информации и результатов выполнения запросов для передачи их приложению.

Все объекты и интерфейсы ADO являются объектами и интерфейсами COM. Согласно спецификации OLE DB, в состав COM входит следующий набор объектов:

- *Command* (команда) — служит для обработки команд (обычно SQL-запросов);
- *Data Source* (источник *данных*) — используется для связи с провайдером данных;

- *Enumerator* (перечислитель) — служит для перечисления провайдеров ADO;
- *Error* (ошибка) — содержит информацию об исключениях;
- *Rowset* (набор рядов) — строки данных, являющиеся результатом выполнения команды;
- *Session* (сессия) — совокупность объектов, обращающихся к одному хранилищу данных;
- *Transaction* (транзакция) — управление транзакциями в OLE DB.

Мы привели состав объектов COM, чтобы получить общую картину о технологии ADO. В Delphi некоторые из этих объектов получили реализацию, и при необходимости мы будем рассматривать более подробно отдельные понятия.

В системе программирования Delphi компоненты, используемые для создания приложений по технологии ADO, расположены на странице **ADO** Палитры компонентов *(см. главу 2)*. Охарактеризуем кратко назначение этих компонентов:

- `ADOConnection` — ADO-соединение, используется для установки соединения с ADO-источником данных и обеспечивает поддержку транзакций;
- `ADOCommand` — ADO-команды, используется для выполнения SQL-команд доступа к ADO-источнику данных без возвращения результирующего набора данных;
- `ADODataSet` — набор данных ADO, обеспечивает доступ к одной или более таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource` аналогично тому как используется компонет `DataSet`. Может использоваться в компонентах `ADOTable`, `ADOQuery`, `ADOStoredProc`;
- `ADOTable` — таблица ADO, обеспечивает доступ к одной таблице ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOQuery` — запрос ADO, позволяет выполнять SQL-команды для получения информации из ADO-источника данных и позволяет другим компонентам управлять этими данными, связываясь с компонентом `ADOTable` через компонент `DataSource`;
- `ADOStoredProc` — хранимая процедура ADO, позволяет приложениям получать доступ к хранимым процедурам, используя интерфейс ADO;
- `RDSConnection` — RDS-соединение, служит для управления передачей объекта `Recordset` от одного процесса (компьютера) к другому. Компонент используется для создания серверных приложений.

Компонент `ADOConnection` может использоваться как посредник между данными и другими компонентами ADO, что позволяет более гибко управлять соединением.

Возможен вариант использования других компонентов ADO, соединяясь с источником данных напрямую, для чего компоненты ADO имеют свойство

ConnectionString, с помощью которого могут создавать свой собственный канал доступа к данным.

Установление соединения

При использовании компонентов доступа к данным по технологии ADO (ADOCommand, ADODataSet, ADOTable, ADOQuery и ADOStoredProc) установление соединения с хранилищем данных можно выполнить двумя путями: с помощью свойства ConnectionString или с помощью компонента ADOConnection. Имя последнего задается в свойстве Connection компонентов доступа к данным. При этом во втором случае для компонента ADOConnection предварительно также с помощью его свойства ConnectionString нужно установить соединение с хранилищем данных.

Рассмотрим технологию установления соединения с хранилищем данных с помощью свойства ConnectionString. Это свойство представляет собой строку с параметрами соединения, отделяемыми друг от друга точкой с запятой. Предварительно соответствующий компонент доступа к данным (например, ADODataSet) или компонент соединения (ADOConnection) должен быть помещен на форму приложения. Настройка параметров соединения осуществляется в диалоге (рис. 10.2), открываемом двойным щелчком в строке ConnectionString свойства соответствующего компонента доступа к данным в окне Инспектора объектов.

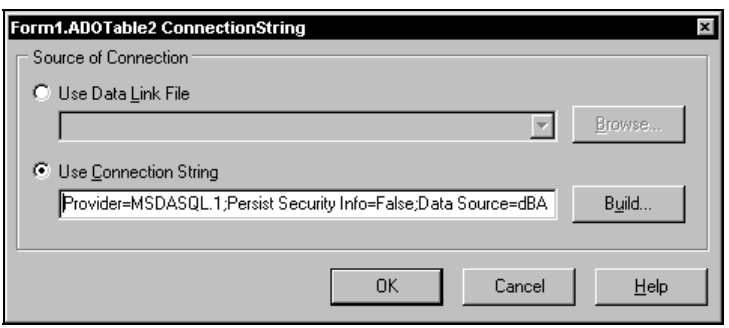


Рис. 10.2. Первое окно настройки строки соединения

При установке переключателя **Use Data Link File** можно выбрать из списка или найти (после нажатия кнопки **Browse**) ***файл связи с данными*** .udl. По умолчанию он расположен в папке c:\Program Files\Common Files\System\Ole DB\Data Links. По существу файлы связи с данными играют ту же роль, что и псевдонимы при использовании BDE. Они позволяют разработчику не связывать откомпилированные приложения с точным расположением хранилища данных. При перемещении хранилища данных в другое место достаточно исправить содержимое файла связи с данными.

При установке переключателя **Use Connection String** выполняются действия по ***созданию строки соединения***. Рассмотрим их более подробно. Для продолжения выбранного варианта диалога нужно нажать кнопку **Build**. В результате открывается диалоговое окно **Data Link Properties**, содержащее 4 вкладки. С помощью вкладки **Provider** (рис. 10.3) осуществляется выбор провайдера с учетом характера решаемой задачи. По умолчанию предлагается вариант **Microsoft OLE DB Provider for ODBC Drivers**.

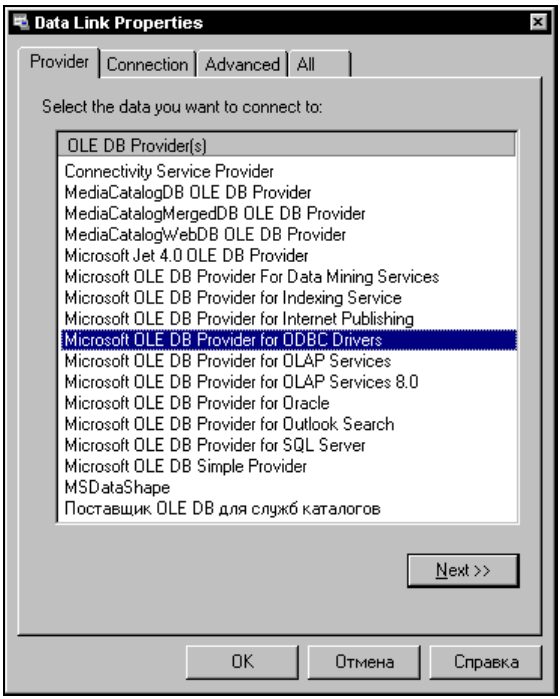


Рис. 10.3. Вкладка **Provider** окна настройки соединения

Отметим, что после установки Microsoft ActiveX Data Objects в операционной системе доступны стандартные провайдеры ADO, обеспечивающие следующее:

- **Microsoft Jet OLE DB Provider** — соединение с данными СУБД Microsoft Access;
- **Microsoft OLE DB Provider for Microsoft Indexing Service** — доступ для чтения к ресурсам Microsoft Indexing Service;
- **Microsoft OLE DB Provider for Microsoft Active Directory Service** — доступ к ресурсам службы каталогов Active Directory;
- **Microsoft OLE DB Provider for Internet Publishing** — доступ к ресурсам Microsoft FrontPage и Microsoft Internet Information Server;

- **Microsoft Data Shaping Service for OLE DB** — доступ к иерархическим наборам данных;
- **Microsoft OLE DB Simple Provider** — доступ к хранилищам данных, поддерживающим основные возможности OLE DB;
- **Microsoft OLE DB Provider for ODBC drivers** — доступ к данным для драйверов ODBC;
- **Microsoft OLE DB Provider for Oracle** — соединение с сервером Oracle;
- **Microsoft OLE DB Provider for SQL Server** — соединение с сервером Microsoft SQL Server.

При нажатии на кнопку **Next** (см. рис. 10.3) происходит переход на вкладку **Connection**, содержимое которой несколько изменяется в зависимости от выбора провайдера. К примеру, вид вкладки **Connection** диалогового окна **Data Link Properties** для случая выбора провайдера **Microsoft Jet 4.0 OLE DB Provider** приведен на рис. 10.4.

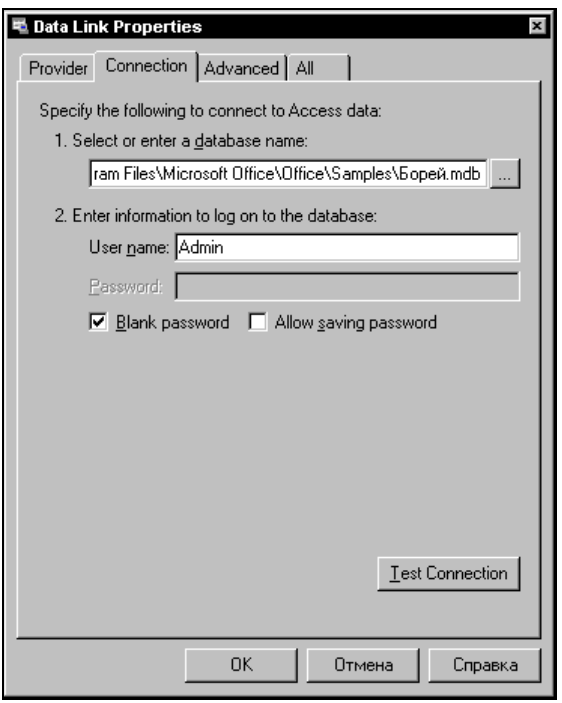


Рис. 10.4. Вкладка **Connection** окна настройки соединения

На вкладке **Connection** можно указать имя базы данных, имя пользователя и пароль (для защищенных БД). Кроме того, нажав кнопку **Test Connection**, можно проверить правильность функционирования соединения. Далее можно нажатием

кнопки **OK** установить строку соединения, либо перейти на остальные две вкладки.

На вкладке **Advanced** в поле **Network Settings** задается уровень защиты при сетевом доступе к базе данных. В поле **Connect timeout** задается предельное время ожидания соединения в секундах. В списке **Access permissions** для определения прав доступа задается перечень допустимых операций: *Read* — только чтение; *ReadWrite* — чтение и запись; *Share Deny None* — нет запрета на чтение и запись; *Share Deny Read* — запрещено открытие для чтения; *Share Deny Write* — запрещено открытие для записи; *Share Exclusive* — эксклюзивное (монопольное) использование; *Write* — только запись.

На вкладке **All** диалогового окна настройки можно просмотреть и отредактировать параметры соединения, заданные с помощью других вкладок.

В случае использования компонента `ADOConnection` для активизации соединения после настройки достаточно установить свойству `Connected` этого компонента значение `True` или при выполнении приложения вызвать метод `Open`.

В случае использования любого из компонентов доступа к данным (`ADODataSet`, `ADOTable`, `ADOQuery` и `ADOStoredProc`) для активизации соединения после настройки используют свойство `Active`.

Управление соединением и транзакциями

Как следует из изложенного, установление соединения с хранилищем данных может осуществляться компонентами доступа к данным ADO через их свойство `ConnectionString`.

Кроме того, компоненты доступа к данным ADO могут соединяться с хранилищем данных через свойство `Connection`, связывающее их с компонентом соединения `ADOConnection`. При этом последний компонент связывается с хранилищем данных через свое свойство `ConnectionString`.

Достоинство применения компонента `ADOConnection` для связи компонентов доступа к данным с хранилищами данных состоит в том, что он позволяет управлять параметрами соединения и транзакциями.

С помощью свойства `CoursorLocation` типа `TCursorLocation` можно задать библиотеку, используемую курсором при соединении с хранилищем данных. Используемое по умолчанию значение `clUseClient` означает, что все данные располагаются и обрабатываются на компьютере-клиенте. При этом достигается наибольшая гибкость: возможны операции, которые могут не поддерживаться сервером. Тем не менее операторы SQL все равно выполняются на сервере, а клиенту передаются результаты выполнения запроса.

Если свойству `CoursorLocation` задано значение `clUseServer`, то обработка данных ведется на сервере. Такое значение свойства целесообразно устанавливать в случае задания команд, возвращающих большой объем данных.

При использовании технологии ADO соединение может быть *синхронным* или *асинхронным*, что можно задать с помощью свойства `ConnectOptions` типа

`TConnectOption`, который описан так:

```
type TConnectOption = (coConnectUnspecified, coAsyncConnect);
```

Значение `coConnectUnspecified` задает синхронное соединение, которое всегда ожидает результат последнего запроса, значение `coAsyncConnect` задает асинхронное соединение, при котором новые запросы выполняются, не ожидая ответа от предыдущих запросов.

Перед установлением соединения возникает событие `OnWillConnect` типа `TWillConnectEvent`, который описан так:

```
TWillConnectEvent = procedure(Connection: TADOConnection; var
 ConnectionString, UserID, Password: WideString; var ConnectOptions:
 TConnectOption; var EventStatus: TEventStatus) of object;
```

Здесь: `Connection` указывает на соответствующий компонент соединения; `ConnectionString`, `Password`, `WideString` определяют строку соединения, имя и пароль пользователя соответственно; `ConnectOptions` указывает на синхронное или асинхронное соединение.

Параметр `EventStatus` указывает на успешность выполнения запроса на соединение. Тип этого параметра описан так:

```
type TEventStatus =
    (esOK,esErrorsOccured,esCantDeny,esCancel,esUnwantedEvent);
```

Здесь, например: `esOK` — соединение выполнено успешно; `esErrorsOccured` — ошибка при выполнении операции; `esCantDeny` — незаконченную операцию соединения нельзя отменить.

После установки соединения возникает событие `OnConnectComplete` типа `TConnectErrorEvent`, который описан так:

```
TConnectErrorEvent = procedure(Connection: TADOConnection; Error: Error;
  var EventStatus: TEventStatus) of object;
```

В случае возникновения ошибки в процессе соединения параметр `EventStatus` будет иметь значение `esErrorsOccured`, а параметр `Error` будет содержать объект ошибки ADO.

Замечание

Объекты ошибки ADO содержат информацию об ошибке, возникшей при выполнении операции над каким-либо объектом ADO. Разработчик может использовать методы объекта ошибки интерфейса `Error`, предоставляемого многими методами других объектов ADO. Укажем важные свойства объекта ошибки. Свойство `Description` возвращает описание ошибки, переданное из объекта. Свойство `SQLState` содержит текст команды, вызвавшей ошибку. Свойство `NativeError` возвращает код ошибки, переданный из объекта, в котором ошибка произошла.

При разрыве соединения возникает событие `OnDisconnect` типа `TDisconnectEvent`, который описан так:

```
TDisconnectEvent = procedure(Connection: TADOConnection; var EventStatus:
 TEventStatus) of object;
```

Управление транзакциями осуществляется с помощью методов и свойств компонента `ADOConnection`.

Для *запуска* транзакции используется функция `BeginTrans`, возвращающая целое значение — уровень вложенности новой транзакции. В случае успешного запуска транзакции свойство `InTransaction` принимает значение `True`, указывающее на то, что компонент соединения находится в транзакции.

Для *завершения* транзакции и сохранения внесенных изменений в хранилище данных используется метод `CommitTrans`. При успешном его выполнении свойство `InTransaction` принимает значение `False`.

Для *отката* транзакции служит метод `RollbackTrans`. При его успешном выполнении отменяются все изменения, внесенные в ходе транзакции, а свойство `InTransaction` принимает значение `True`.

Для управления запуском транзакций, оставшихся незавершенными, служит свойство `Attributes` типа `TXactAttributes`. Оно принимает два значения: `xaCommitRetaining` — незавершенная транзакция начинается при подтверждении предыдущей транзакции; `xaAbortRetaining` — незавершенная транзакция начинается при отмене предыдущей транзакции.

Компоненты доступа к данным

Перечень и назначение компонентов работы по технологии ADO мы приводили в начале главы. Большинство из них имеет аналоги компонентов в технологиях BDE и dbExpress (табл. 10.1). В этой таблице компоненты, используемые в технологии dbExpress, аналогами можно назвать достаточно условно, т. к. большинство из них поддерживают однонаправленные наборы данных.

Таблица 10.1. *Соответствие компонентов в технологиях ADO, dbExpress и BDE*

Компонент ADO	Компонент dbExpress	Компонент BDE
ADOTable	SQLTable	Table
ADOQuery	SQLQuery	Query
ADOStoredProc	SQLStoredProc	StoredProc
ADOConnection	SQLConnection	Database
ADODataSet	SQLDataSet, SQLSimpleDataSet	Table, Query, StoredProc
ADOCommand	–	–
RDSConnection	–	–

Стандартные компоненты доступа к данным в ADO наследуют механизм доступа от родительского класса `TCustomADODataSet`. Поэтому важнейшие свойства и методы этого класса во многом определяют поведение компонентов доступа к данным в ADO. Коротко охарактеризуем их.

К числу основных свойств названного класса можно отнести свойства, устанавливающие параметры обмена с хранилищем данных. Значения их нужно задать до открытия набора данных.

Тип блокировки записей в наборе данных определяет свойство `LockType` типа `TADOLockType`, который описан так:

```
type TADOLockType = (ltUnspecified, ltReadOnly, ltPessimistic, ltOptimistic,
  ltBatchOptimistic);
```

Здесь, например:

- `ltUnspecified` — блокировка задается источником данных;
- `ltReadOnly` — набор данных открывается в режиме только для чтения;
- `ltPessimistic` — блокировка на время редактирования до момента подтверждения.

Как отмечалось, местоположение курсора задает свойство `CursorLocation` (см. предыдущий подраздел). При работе с клиентским курсором важную роль играет следующее свойство.

Передаваемые серверу данные определяет свойство `MarshalOptions` типа `TMarshalOption` , который описан так:

```
type TMarshalOption = (moMarshalAll, moMarshalModifiedOnly);
```

Здесь:

- `moMarshalAll` — возврат всех записей локального набора данных серверу;
- `moMarshalModifiedOnly` — возврат только измененных записей.

Тип курсора определяет свойство `CursorType` типа `TCursorType`, который описан так:

```
type TCursorType = (ctUnspecified, ctOpenForwardOnly,
ctKeyset,ctDynamic,ctStatic);
```

Здесь, например:

- `ctUnspecified` — тип курсора не задан и определяется возможностями источника данных;
- `ctOpenForwardOnly` — однонаправленный курсор, используемый для одиночного прохода по всем записям;
- `ctKeyset` — двунаправленный курсор, не отображающий добавленные или удаленные другими пользователями записи;
- `ctStatic` — двунаправленный курсор, не учитывающий изменения записей другими пользователями.

Набор данных может быть открыт с помощью свойства `Active` или метода `Open`. На стороне клиента записи из набора данных хранятся в буфере, размер которого может быть получен с помощью свойства `CacheSize` типа `Integer`.

При необходимости с помощью свойства `BlockReadSize` типа `Integer` можно организовать передачу записей в виде блоков. Кроме того, с помощью свойства `MaxRecords` типа `Integer` можно ограничить размер набора данных. По умолчанию блочная пересылка не используется, а число записей в наборе данных не ограничено.

Текущее *состояние записи* набора данных определяет свойство `RecordStatus` типа `TrecordStatusSet`, который описан так:

```
type TRecordStatus = (rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
  rsInvalid, rsMultipleChanges, rsPendingChanges, rsCanceled,
  rsCantRelease, rsConcurrencyViolation, rsIntegrityViolation,
  rsMaxChangesExceeded, rsObjectOpen, rsOutOfMemory, rsPermissionDenied,
  rsSchemaViolation, rsDBDeleted);
```

Здесь, например:

- `rsOK` — запись успешно изменена;
- `rsNew` — новая запись вставлена;
- `rsModified` — запись изменена;
- `rsDeleted` — запись удалена;
- `rsUnmodified` — запись осталась без изменений.

Дадим сравнительно развернутую характеристику компонентам, используемым в технологии ADO для доступа к данным.

Доступ к таблицам

Для обеспечения доступа к таблицам хранилищ данных по технологии ADO служит компонент `ADOTable`. Для установления соединения с хранилищем данных этого компонента через провайдеры ADO служит свойство `ConnectionString` или `Connection`, как описано ранее. Для управления набором данных таблицы в приложение включают компонент источника данных `DataSource`. При этом свойству `DataSet` этого компонента в качестве значения задается имя компонента `ADOTable`. Для отображения данных таблицы к источнику данных подключаются различные компоненты отображения, к примеру, `DBGrid`.

После установления связи компонента `ADOTable` с хранилищем данных с помощью свойства `TableName` типа `WideString` задается имя таблицы. Не все провайдеры ADO допускают непосредственный доступ к таблицам, поэтому может потребоваться доступ с помощью SQL-запроса. Вариант доступа к данным таблицы определяет свойство `TableDirect` типа `Boolean`. Если оно имеет значение `False` (по умолчанию), то компонент `ADOTable` автоматически генерирует SQL-запрос для доступа к таблице, в противном случае выполняется непосредственный доступ к данным таблицы.

Рассматриваемый нами компонент по своим возможностям и технике работы с ним во многом схож с компонентом Table. Здесь также с помощью редактора полей можно задавать свойства отдельных полей. При этом имеется ограничение, состоящее в том, что в компонентах ADO нельзя работать со словарями. Поэтому свойства полей требуется задавать вручную. Кроме того, у драйверов ADO имеются ограничения по работе с отдельными типами полей, в частности, с графическими.

Программирование действий по работе с хранилищем данных с помощью рассматриваемого компонента используются аналогичные средства, как и в случае компонента Table. В частности, для навигации по табличному набору данных используются методы First, Last, Next и Prior. Для поиска записей используются методы Find, Seek и Locate.

Выполнение запросов

Для выполнения SQL-запросов при использовании технологии ADO служит компонент ADOQuery. По функциональным возможностям и технике применения этот компонент во многом подобен компоненту Query. Установка соединения с хранилищем данных, свойства и методы фильтрации и поиска аналогичны используемым для компонента ADOTable.

Текст запроса задается с помощью свойства SQL типа TStrings.

С помощью свойства Parameters типа TParameters определяются параметры запроса.

Открытие набора данных может быть выполнено с помощью свойства Active типа Boolean или с помощью метода Open. Если же запрос не должен возвращать данных, то для открытия набора данных можно вызвать метод ExecSQL.

Вызов хранимых процедур

Для вызова хранимых процедур по технологии ADO служит компонент ADOStoredProc. По своим возможностям и технике использования он подобен своему аналогу из BDE. Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента ADOTable. Для отображения данных (выходных параметров) к источнику данных также подключаются компоненты отображения.

Для задания имени хранимой процедуры служит свойство ProcedureName типа WideString.

Свойство Parameters типа TParameters служит для определения входных параметров процедуры. А именно, после задания имени хранимой процедуры в свойстве Parameters отображаются входные параметры процедуры. Их можно просмотреть с помощью Инспектора объектов.

Выходные параметры хранимой процедуры являются объектами полей рассматриваемого компонента ADOStoredProc. Чтобы просмотреть и изменить их значения, достаточно с помощью контекстного меню компонента ADOStoredProc

вызвать редактор полей Fields Editor. С помощью команд редактора можно легко изменить состав полей (выходные параметры), отображаемых в наборе данных компонента ADOStoredProc.

Компонент *ADODataSet*

Компонент ADODataSet служит для представления набора данных из хранилища данных ADO. По своим функциональным возможностям компонент в определенной мере аналогичен компонентам SQLDataSet и SQLSimpleDataSet, используемым в технологии dbExpress. Этот компонент позволяет получать данные из таблиц, SQL-запросов, хранимых процедур, файлов и т. д.

Тип команды, указываемой в свойстве CommandText, определяет свойство CommandType типа TCommandType, который определен так:

```
TCommandType = (cmdUnknown, cmdText, cmdTable, cmdStoredProc, cmdFile,
   cmdTableDirect)
```

Здесь:

- cmdUnknown — тип не известен;
- cmdText — текст SQL-оператора или вызова процедуры (значение по умолчанию);
- cmdTable — имя таблицы;
- cmdStoredProc — имя хранимой процедуры;
- cmdFile — имя сохраненного файла набора данных;
- cmdTableDirect — имя таблицы, все поля которой возвращаются.

Значение свойства CommandText типа WideString определяет команду, которая выполняется. В качестве команды может быть задана строка с SQL-оператором, имя таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства Parametrs. Выполнение команды, заданной с помощью свойства CommandText, происходит при открытии набора данных.

Установка соединения с хранилищем данных и управление набором данных аналогичны используемым для компонента ADOTable. Для отображения данных к источнику данных (компоненту DataSource) также подключаются компоненты отображения.

Команды ADO

Для выполнения команд предназначен компонент ADOCommand, являющийся реализацией объекта ADO Command в среде Delphi. Этот компонент служит для выполнения команд, не возвращающих результаты, например, SQL-операторов языка определения данных или хранимых процедур. Для выполнения хранимых процедур и SQL-запросов, возвращающих результаты в виде наборов данных, целесообразно применять компоненты доступа к данным ADODataSet, ADOQuery и ADOStoredProc.

Рассмотрим важнейшие свойства и методы компонента `ADOCommand`. Отметим, что большинство свойств компонента `ADOCommand` в Delphi эквивалентны свойствам объекта ADO `Command`.

Свойство `CommandObject` типа `_Command` предоставляет непосредственный доступ к базовому объекту команды ADO `Command`.

Тип команды, указываемой в свойстве `CommandText`, определяет свойство `CommandType` (см. описание компонента `ADODataSet`). Для свойства `CommandType` объекта `ADOCommand` значения `cmdFile` и `cmdTableDirect` являются недопустимыми.

Напомним, что значение свойства `CommandText` типа `WideString` определяет команду, которая выполняется с помощью метода `Execute`. В качестве команды может быть задана строка с SQL-оператором, имя таблицы или имя хранимой процедуры. При необходимости могут быть заданы параметры с помощью свойства `Parametrs`.

Текущее состояние компонента `ADOCommand` (объекта `Command`) определяет свойство `States` типа `ObjectStates`, который описан так:

```
TObjectState = (stClosed, stOpen, stConnecting, stExecuting, stFetching);
```

Здесь:

- `stClosed` — объект `Command` не активен и не соединен с хранилищем данных;
- `stOpen` — объект `Command` не активен, соединен с хранилищем данных;
- `stConnecting` — объект `Command` соединяется с хранилищем данных;
- `stExecuting` — объект `Command` выполняет команду;
- `stFetching` — объект `Command` получает данные от хранилища данных.

Условия выполнения команды ADO определяет свойство `ExecuteOptions` типа `TExecuteOptions`, который описан так:

```
TExecuteOption = (eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlocking,
  eoExecuteNoRecords);
```

Здесь:

- `eoAsyncExecute` — асинхронное выполнение команды;
- `eoAsyncFetch` — асинхронная передача данных;
- `eoAsyncFetchNonBlocking` — асинхронная передача данных без блокирования потока;
- `eoExecuteNoRecords` — выполнение команды ADO без возвращения данных.

Выполнение команды ADO осуществляется с помощью одной из трех перегружаемых версий метода `Execute` с различными наборами параметров:

- `function Execute: _Recordset; overload;` — такое объявление метода используется при выполнении команды без параметров;

- `function Execute(const Parameters: OleVariant): _Recordset; overload;` — в этом объявлении параметр `Parameters` определяет параметры команды;
- `function Execute(var RecordsAffected: Integer; const Parameters: OleVariant): _RecordSet; overload;` — здесь параметр `RecordsAffected` возвращает общее число обработанных запросом записей.

Пример приложения

Рассмотрим пример (рис. 10.5) приложения для работы с БД, использующего технологию ADO. Пусть назначением приложения является организация доступа к различным таблицам из базы данных, их редактирование и сохранение внесенных изменений на сервере.

Рис. 10.5. Вид формы при выполнении приложения

Основными компонентами приложения, используемыми для установления соединения с базой данных и отображения данных, являются следующие: `ADOConnection1`, `ADODataSet1`, `DataSource1` и `DBGrid1`. Кроме того, имеется ряд вспомогательных компонентов, служащих для управления работой приложения.

Соединение с хранилищем данных компонента `ADODataSet1` выполнено через его свойство `Connection`, указывающее на имя `ADOConnection1` компонента соединения. Для компонента `ADOConnection1` с помощью его свойства `ConnectionString` установлено соединение с хранилищем данных. При этом использован файл связи с данными. Для наглядности в процедуре обработки события создания формы приведен оператор задания нужного значения свойству `ConnectionString` компонента `ADOConnection1`.

Для компонента DataSource1 его свойству DataSet установлено значение ADODataSet1. Для компонента DBGrid1, используемого для отображения набора данных, его свойству DataSource установлено значение DataSource1.

Код модуля формы для решения поставленной задачи имеет вид:

```
unit ADODataSet;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, DB, ADODB, Grids, DBGrids, ExtCtrls;

type
  TForm1 = class(TForm)
    ADODataSet1: TADODataSet;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    ADOConnection1: TADOConnection;
    Panel1: TPanel;
    UpdateButton: TButton;
    GetTableButton: TButton;
    Button1: TButton;
    Edit1: TEdit;
    procedure Form1Create(Sender: TObject);
    procedure Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure UpdateButtonClick(Sender: TObject);
    procedure GetTableButtonClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    private
public
    procedure UpdateData;
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
```

```
const
  BaseFileName = 'EMPLOYEE.ADTG';

procedure TForm1.UpdateData;
begin
  { Обновление данных на сервере }
  ADODataSet1.UpdateBatch;
end;

procedure TForm1.Form1Create(Sender: TObject);
begin
  ADOConnection1.ConnectionString := 'FILE NAME=' + DataLinkDir +
'\DBDEMOS.UDL';
  ADODataSet1.Open;
end;

procedure TForm1.UpdateButtonClick(Sender: TObject);
begin
  UpdateData;
end;

procedure TForm1.Form1CloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if ADODataSet1.Active then
  try
    { Обновление данных на сервере при закрытии формы }
    UpdateData
  except
    on E: Exception do
    begin
      Application.HandleException(Self);
      CanClose := MessageDlg('Данные не сохранены/обновлены, выйти?',
        mtConfirmation, mbYesNoCancel, 0) = mrYes;
    end;
  end;
end;

procedure TForm1.GetTableButtonClick(Sender: TObject);
begin
  { Повторное открытие табличного набора данных 'Country'}
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
  ADODataSet1.CommandText := 'Country';
```

```
  ADODataSet1.Open;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ADODataSet1.Close;
  ADODataSet1.CommandType := cmdTable;
  ADODataSet1.CommandText := Edit1.Text;
  try
      ADODataSet1.Open;
  except
    on E: Exception do
    begin
      MessageDlg('Исправьте имя таблицы', mtError,[mbOK],0);
      ADODataSet1.Close;
    end;
end;
end;
end.
```

Первоначально при запуске приложения автоматически открывается таблица с именем `'Country'`. При необходимости открыть другую таблицу для просмотра и редактирования ее имя следует ввести в однострочном редакторе (компонент `Edit1`) и затем нажать кнопку с заголовком **Открыть таблицу с именем**.

Обновление внесенных в открытую таблицу изменений происходит при нажатии на кнопку с заголовком **Сохранить на сервере** или автоматически при закрытии формы приложения.

Глава 11

Создание и просмотр отчетов с помощью Rave Reports

Отчет представляет собой печатный документ, содержащий данные, аналогичные получаемым в результате выполнения запроса к БД или из некоторого другого источника — электронной таблицы, сообщения электронной почты, текстового документа и др. Можно выделить следующие виды отчетов:

- простой отчет;
- отчет с группированием данных;
- отчет для таблиц, связанных отношением "главный-подчиненный";
- составной отчет, объединяющий несколько разных отчетов.

Характеристика генератора отчетов

В Delphi 7 для создания отчетов предназначен генератор отчетов Rave Reports 5.0 фирмы Nevrona, компоненты которого, предназначенные для управления отчетами, размещены на странице **Rave** Палитры компонентов. Назначение компонентов указано в *главе 2*. Генератор отчетов имеет развитые средства визуального проектирования отчетов.

Из состава Палитры компонентов изъята страница **QReport**, содержащая набор компонентов, предназначенных для создания отчетов. Как отмечалось, для обеспечения обратной совместимости компоненты страницы **QReport** размещены в пакете dclqrt70.bpl в папке \Delphi 7\Bin и могут быть установлены в Палитру компонентов.

Генератор отчетов Rave Reports позволяет создавать отчеты для обычных приложений и для приложений баз данных. При этом обеспечивается возможность создания отчетов для приложений баз данных, использующих различные механизмы доступа к данным BDE, ADO и dbExpress.

Компоненты, расположенные на странице **Rave** Палитры компонентов, предназначены для управления отчетами в приложении. Важнейшими из них являются компоненты `RvProject` и `RvSystem`. Так, компонент `RvProject` типа `TRvProject` задает представление проекта отчета в приложении Delphi, а ком-

понент `RvSystem` служит для управления отчетом и, к примеру, позволяет задать печать, предварительный просмотр или передачу отчета в файл.

Отчет размещается в файле с расширением rav, который представляет проект создаваемого отчета и содержит информацию об отчете, параметры оформления страниц отчета и др. В состав проекта отчета может входить один или несколько отчетов, в каждом из которых может быть произвольное число страниц и глобальных страниц. На страницах отчета и на глобальных страницах могут размещаться графические и текстовые объекты, связанные с источниками данных приложения Delphi.

Визуальный конструктор отчетов Rave Reports 5.0 повышает удобство и упрощает непосредственную разработку отчетов. Вызвать визуальный конструктор отчетов можно тремя путями: с помощью команды **Tools\Rave Designer** меню Delphi, двойным щелчком мыши на компоненте `RvProject` или с помощью команды **Rave Visual Designer** контекстного меню компонента `RvProject`.

Кроме того, в состав генератора отчетов входит ядро, которое обеспечивает управление отчетом, предварительный просмотр и отправку на печать. Код ядра при компиляции помещается в приложение, тем самым обеспечивается автономность последнего.

Визуальное конструирование отчетов

Визуальный конструктор отчетов Rave Reports 5.0 служит для повышения удобства и упрощения непосредственной разработки отчетов. С его помощью можно быстро создать новый отчет, глобальную страницу, страницу отчета, настроить параметры отчета, выполнить подключение источника данных и др.

Интерфейс визуального конструктора

Вызов визуального конструктора отчетов можно выполнить с помощью команды **Tools\Rave Designer** меню Delphi, двойным щелчком мыши на компоненте `RvProject` или с помощью команды **Rave Visual Designer** контекстного меню компонента `RvProject`. Возможный вид окна Rave Reports 5.0 приведен на рис. 11.1.

Визуальный конструктор отчетов имеет практически стандартный интерфейс приложений Windows. В верхней части окна визуального конструктора расположено меню. Под ним находится панель инструментов, кнопки которой дублируют основные команды визуального конструктора. В правой верхней части конструктора размещена многостраничная панель инструментов, на вкладках которой содержатся компоненты, используемые при создании отчетов, а также элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета в окне.

Инспектор компонентов отчета, расположенный в левой части окна визуального конструктора, позволяет выполнить настройку свойств компонентов отчета, которые выбираются в дереве проекта отчета. В нижней части Инспектора компонентов отчета может отображаться подсказка для выбранного свойства. Управление

отображением этой подсказки и подсветкой измененных свойств компонентов выполняется с помощью контекстного меню Инспектора компонентов отчета.

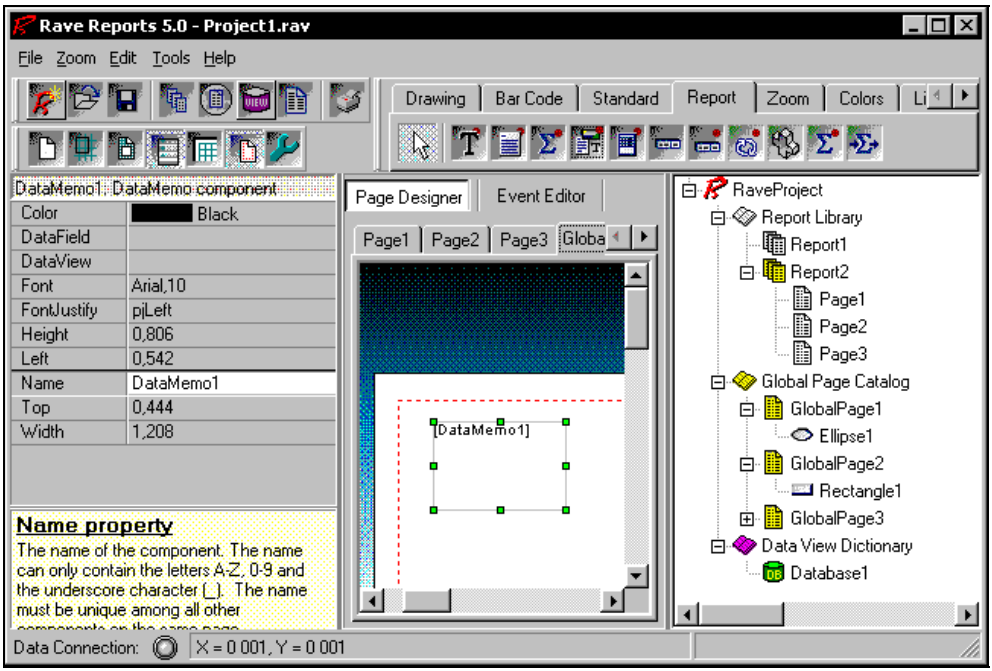


Рис. 11.1. Вид окна Rave Reports 5.0

В центральной части окна визуального конструктора расположен блокнот с двумя вкладками: **Page Designer** (Конструктор страниц) и **Event Editor** (Редактор событий). С помощью вкладки **Page Designer** можно добавлять, удалять и настраивать компоненты на отдельных страницах отчета. На этой вкладке имеется еще один блокнот, каждая из вкладок которого содержит одну страницу отчета. С помощью вкладки **Event Editor** создаются обработчики событий для отчетов, страниц, элементов оформления и компонентов отчета.

Дерево проекта отчета, размещенное в правой части окна визуального конструктора, отображает состав компонентов открытого проекта отчета. С помощью двойного щелчка удобно выбирать компоненты проекта отчета для отображения в центральной части окна и настройки свойств в Инспекторе компонентов.

Состав проекта отчетов

В дереве проекта отчетов RaveProject (см. рис. 11.1) имеются следующие три составляющие:

- **Report Library** (Библиотека отчетов проекта) — содержит все отчеты проекта;

- **Global Page Catalog** (Каталог глобальных страниц) — содержит перечень страниц, являющихся общими для всего проекта;
- **Data View Dictionary** (Словарь просмотров данных) — содержит объекты соединения с данными из внешних источников.

Рассмотрим более подробно каждую из названных составляющих проекта отчетов.

Библиотека отчетов проекта содержит все отчеты, входящие в состав проекта. Отдельный отчет в составе проекта включает произвольное число страниц, содержащих компоненты с данными, текстом и графикой, а также компоненты, повышающие наглядность оформления отчета.

При работе с проектом отчетов выделяют текущий отчет. При открытии проекта отчетов текущим становится первый отчет в составе проекта отчетов. При необходимости двойным щелчком мыши в дереве проекта отчетов можно установить текущим произвольный отчет. При выходе из среды визуального конструктора текущий отчет в составе проекта сохраняется и может использоваться при работе с отчетом с помощью компонента `RvProject` в приложении Delphi.

К проекту отчета можно добавлять новые отчеты с помощью команды **File\New Report** визуального конструктора. К текущему отчету можно добавлять страницы с помощью команды **File\New Report Page**. Для удаления отчета, страницы отчета или глобальной страницы достаточно выделить отчет или страницу отчета в дереве проекта отчетов и нажать клавишу <Delete>.

В отчет может входить произвольное число страниц. Перед предварительным просмотром и печатью отчета можно выбрать произвольное подмножество страниц отчета и добавить нужные страницы из каталога глобальных страниц. В отчете для управления составом и очередностью печати страниц предназначено свойство `PageList`. Если значение этого свойства не определено, то печатаются все страницы отчета в порядке их следования в дереве проектов отчета. Для задания значения свойства `PageList` используется редактор списка страниц Page List Editor (рис. 11.2), позволяющий сформировать список страниц для печати.

В проекте отчетов RaveDemo.rav, расположенном в папке \Borland\Delphi7\Rave\Demos, имеется библиотека, содержащая множество различных отчетов. Например, в ее составе имеются такие отчеты, как MultiPageReport (многостраничный отчет), BarCodes (штрихкоды), MasterDetailReport (отчет "мастер-детальный"). При необходимости полезно использовать отчеты из этой библиотеки в качестве шаблона при создании аналогичных отчетов.

Каталог глобальных страниц содержит страницы, которые являются доступными в любом отчете из состава библиотеки отчетов открытого проекта. Это позволяет получить стандартное оформление для ряда страниц отчета, таких как титульный лист, рамка для дипломных проектов и отчетов о НИР и т. п.

Напомним, что имеющуюся в каталоге глобальную страницу можно добавить в список страниц отчета для печати с помощью редактора списка страниц (см. рис. 11.2).

Для открытого проекта отчетов в состав каталога пустую глобальную страницу можно добавить, выполнив команду **File\New Global Page**. После этого глобальную страницу можно редактировать и добавлять в состав страниц любого из отчетов.

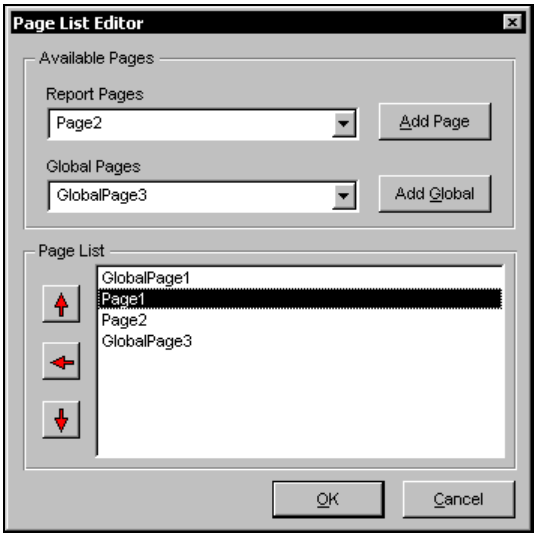


Рис. 11.2. Окно редактора Page List Editor

Словарь просмотров данных содержит объекты доступа к данным из внешних источников. Создание новых объектов выполняется с после задания команды **File\New Data Object**. При этом открывается диалоговое окно **Data Connections** (рис. 11.3), в котором для выбора предлагаются следующие типы объектов:

- **Data Lookup Security Controller** (Контроллер безопасности поиска данных) — обеспечивает аутентификацию пользователей по имени и паролю;
- **Database Connection** (Соединение с базой данных) — устанавливает соединение с внешним источником данных для требуемой технологии доступа (ADO, BDE, dbExpress);

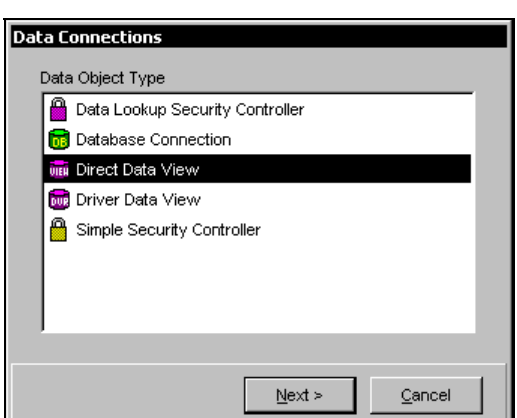


Рис. 11.3. Диалоговое окно **Data Connections**

- **Direct Data View** (Прямой просмотр данных) — создает просмотр данных для активного соединения с источником данных;
- **Driver Data View** (Просмотр данных с помощью драйвера) — создает просмотр данных на основе уже имеющегося в словаре соединения;
- **Simple Security Controller** (Простой контроллер безопасности) — задает список пользователей для возможной организации доступа в отчетах.

Объекты в составе словаря просмотров данных доступны для всех отчетов, входящих в состав проекта отчетов. Названные типы объектов используются преимущественно при создании отчетов для приложений баз данных.

Редактор событий

В проекте любому отчету, странице, элементу оформления или компоненту отчета можно назначить один или несколько обработчиков событий. Сделать это можно с помощью редактора событий, расположенного на вкладке **Event Editor** (рис. 11.4) блокнота в центральной части окна визуального конструктора.

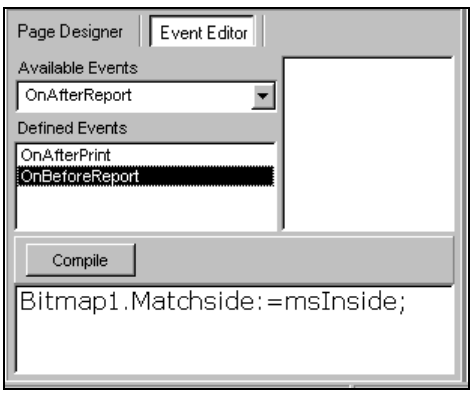


Рис. 11.4. Вкладка **Event Editor**

Создание обработчика событий выполняется следующим образом. В дереве проекта отчетов выбирается текущим отчет, страница или компонент, для которого требуется создать обработчик. Далее в списке **Available Events** выбирается одно из доступных для обработки событий. Это событие оказывается в списке **Defined Events** (Определенные события) и автоматически создается обработчик этого события. Теперь для этого события следует задать собственно код обработки события в нижней части окна ректора событий и нажать кнопку **Compile**. При этом происходит компиляция кода обработчика с выдачей сообщения о завершении компиляции или с сообщением об ошибке. При необходимости можно выбрать другое событие в списке **Available Events** и назначить ему свой обработчик.

При написании кода обработки событий используется синтасис, близкий к синтаксису языка Object Pascal, допускается применение некоторых методов Delphi.

К примеру, на рис. 11.4 приведен код обработчика события `OnBeforeReport`, с помощью которого свойству `Matchside` компонента `Bitmap1` присваивается значение `msInside`.

Компоненты, представленные на многостраничной панели инструментов

Укажем назначение вкладок многостраничной панели инструментов, расположенной в верхней правой части окна визуального конструктора отчетов. Компоненты, используемые при создании отчетов, расположены на следующих вкладках:

- **Drawing** — графические компоненты, повышающие наглядность оформления отчета;
- **Bar Code** — компоненты, задающие различные варианты штрихкода;
- **Standard** — компоненты, задающие отображение текста и графики;
- **Report** — компоненты, предназначенные для отображения данных из внешних источников.

Элементы, управляющие параметрами объектов отчета (цвет линий, шрифт, выравнивание) и масштабом отображения отчета, расположены на следующих вкладках:

- **Zoom** — компоненты, управляющие масштабом отображения текущей страницы;
- **Colors** — компоненты, задающие цвета для графических компонентов и страниц;
- **Lines** — компоненты, задающие стиль и толщину линий графических компонентов;
- **Fills** — компоненты, задающие заливку графических компонентов;
- **Fonts** — компоненты, задающие параметры шрифта для текста;
- **Alignment** — компоненты, задающие выравнивание графических компонентов на странице.

Добавление компонентов в отчет выполняется традиционным для систем визуального программирования способом: щелчком мыши требуемый компонент выбирается на палитре и повторным щелчком мыши размещается в нужном месте страницы отчета. Дадим краткую характеристику компонентам, расположенным на различных страницах панели инструментов.

Страница **Drawing** многостраничной панели инструментов содержит компоненты, обеспечивающие создание графических объектов.

Компоненты `Line`, `Hline` и `Vline` служат для рисования универсальной, горизонтальной и вертикальной линий соответственно. Все они имеют набор свойств, определяющих длину и местоположение на странице отрезка линии. Для компонента `Line` отрезок можно развернуть произвольным образом визуально (с помощью мыши). Для двух других компонентов поворот отрезка также возможен, но не визуально, а только путем изменения свойств компонентов.

Компоненты `Rectangle` и `Square` служат для рисования прямоугольника и квадрата, а компоненты `Ellipse` и `Circle` — для рисования эллипса и круга соответственно. Эти четыре компонента также имеют набор свойств, определяющих их размеры и местоположение на странице.

Страница **Bar Code** многостраничной панели инструментов содержит компоненты, задающие штрихкоды, которые реализуют различные стандарты.

Компонент `PostNetBarCode` соответствует коду PostNet почтовой службы США, задающему код адреса.

Компонент `I2of5BarCode` соответствует перемежающемуся коду I2of5 и служит для задания числовых последовательностей, в которых цифры поочередно кодируются штрихами и пробелами.

Компонент `Code39BarCode` соответствует коду Code39 и служит для кодирования цифр, заглавных латинских букв и ряда символов с помощью штрихов и пробелов.

Компонент `Code128BarCode` соответствует коду Code128 и служит для представления первых 128 символов кода ASCII.

Компонент `UPCBarCode` соответствует коду UPC (Universal Product Code) и служит для маркировки продуктов, может включать до 12 цифр.

Компонент `EANBarCode` соответствует коду EAN (European Article Numbering system), аналогичен коду UPC, может включать до 13 цифр.

Для всех компонентов кодируемое значение задается свойством `text`. Оно может быть также загружено из базы данных с помощью свойств `FileLink` и `DataLink`.

С помощью свойства `BarCodeRotation` штрихкод можно поворачивать с дискретностью 90°.

Страница **Standard** многостраничной панели инструментов содержит компоненты, задающие отображение текста и графики.

Компонент `text` служит для отображения однострочного текста. Сам отображаемый текст задается с путем редактирования в Инспекторе компонентов отчета значения свойства `text` этого компонента. Непосредственно на странице отчета отображаемый текст редактировать нельзя.

Компонент `memo` служит для отображения многострочного текста. Отображаемый компонентом текст определяется свойством `text`, которое в Инспекторе компонентов отчета визуального конструктора связано с многострочным текстовым редактором Memo Editor.

Остальные свойства компонентов `text` и `memo` позволяют настроить цвет, шрифт, выравнивание местоположения текста и др.

Компонент `section` (секция) является невизуальным и предназначен для объединения в одну группу нескольких компонентов, к примеру, строк текста и изображений внутри заголовка отчета. Все размещенные внутри секции компонен-

ты можно по отдельности перемещать в пределах секции, а при перемещении секции по странице вместе с секцией перемещаются и они. Выравнивание компонентов внутри секции осуществляется относительно ее границ.

Компонент `Bitmap` служит для отображения растровых изображений, хранящихся в файлах формата BMP. Отображаемый компонентом рисунок определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора связано с диалогом загрузки файла изображения.

Компонент `MetaFile` служит для отображения изображений, хранящихся в файлах форматов EMF и WMF. Отображаемое компонентом изображение определяется с помощью свойства `Image`, которое в Инспекторе компонентов отчета визуального конструктора также связано с диалогом загрузки файла изображения. При этом файлы формата WMF таким способом загрузить нельзя, для этой цели нужно использовать свойство `FileLink` или `DataField`.

Изображения, отображаемые с помощью компонентов `Bitmap` и `MetaFile`, можно масштабировать. При этом учитывается свойство `MatchSide`. При значениях этого свойства `msHeight` или `msWidth` (по умолчанию) высота или ширина изображения соответственно устанавливаются равными высоте (свойство `Height`) или ширине (свойство `Width`) исходной области, выделенной под изображение, а ширина или высота изображения соответственно устанавливаются с сохранением пропорций изображения. Если свойство `MatchSide` имеет значение `msBoth`, то ширина и высота изображения устанавливаются равными ширине и высоте выделенной области, при этом пропорции изображения могут быть нарушены. При значении `msInside` этого свойства изображение помещается внутри выделенной области с сохранением пропорций.

Компонент `FontMaster` является невизуальным и предназначен для задания единых параметров шрифта для группы компонентов отчета. С его свойством `Font` в Инспекторе компонентов отчета связано диалоговое окно определения параметров шрифта. Эти параметры шрифта служат для определения параметров шрифта всех компонентов отчета, в свойстве `FontMirror` которых указан данный компонент `FontMaster`.

Компонент `PageNumInit` является невизуальным и предназначен для задания нумерации страниц, начиная с той, где он расположен. Начальный номер при этом определяется свойством `InitValue`. При необходимости его значение можно загрузить из базы данных с помощью свойств `InitDataView` и `InitDataField`.

Компоненты отображения данных

Страница **Report** многостраничной панели инструментов визуального конструктора отчетов содержит компоненты, служащие для представления данных. Их можно разделить на компоненты отображения данных из связанных с ними полей просмотра данных и компоненты управления структурой размещения данных (структурные компоненты), на которые помещаются

ных (структурные компоненты), на которые помещаются компоненты отображения данных.

К компонентам *отображения данных* относятся следующие.

- Компонент `DataText` служит для представления строковых или числовых значений полей связанного с ним просмотра данных.
- Компонент `DataMemo` обеспечивает представление данных формата Memo или BLOB.
- Компонент `CalcText` служит для представления результатов вычисления агрегатной функции по значениям связанного поля. Выполняемая функция выбирается из списка возможных значений свойства `CalcType`.
- Компонент `DataMirrorSection` является невизуальным и подобно своему потомку `Section` служит для объединения других компонентов в одну группу.

У всех указанных компонентов представления данных свойство `DataView` задает связь с просмотром данных, а свойство `DataField` задает связь с полем просмотра, данные из которого отображаются компонентом.

К *структурным компонентам* относятся следующие.

- Компонент `Region` служит для выделения области (части страницы отчета), на которой размещаются другие компоненты отображения данных. Более того, компоненты `Band` и `DataBand` размещаются только в области, определяемой компонентом `Region`. Свойство `Columns` этого компонента определяет число колонок при печати соответствующей части отчета.
- Компонент `Band` задает полосу отчета, на которой располагаются другие компоненты отчета — отображения текста и графики, штрихкоды, отображения данных. Эта полоса служит для оформления заголовков, сносок, верхних и нижних колонтитулов и других фрагментов отчета, не изменяющиеся при печати просмотра данных.
- Компонент `DataBand` задает полосу отчета, представляющую модель строки просмотра данных. На ней размещают компоненты отображения данных. При печати для каждой строки просмотра данных выводится новый экземпляр полосы со всеми расположенными на ней компонентами отчета. Тем самым в отчете построчно отображается весь просмотр данных.

У компонентов `Band` и `DataBand` имеется свойство, которое определяет стиль (назначение и поведение) полосы отчета. С этим свойством в Инспекторе компонентов отчета связан редактор стиля полосы **Band Style Editor**. В его окне (рис. 11.5) отображается взаимосвязь полос (компонентов `Band` и `DataBand`) в общей области (определяемой компонентом `Region`) и настраивается стиль текущей полосы.

В левой части окна редактора **Band Style Editor** отображается список полос области со взаимосвязями (реляционные отношения, группировка и вложенность и т. д.). Текущая полоса выделяется полужирным начертанием с подчеркиванием.

Полосы типа `Band` и `DataBand` верхнего уровня (т. е. не являющиеся подчиненными другим полосам типа `DataBand`) помечаются слева цветным ромбиком. При этом подчиненные полосы отчета типов `Band` и `DataBand` определяются с помощью их свойства `ControllerBand`, в качестве значения которого указывается имя полосы типа `DataBand`, которой они подчинены. Подчиненные полосы помечаются слева треугольником, острый угол которого направлен в сторону главной полосы, а цвет совпадает с цветом ромбика главной полосы.

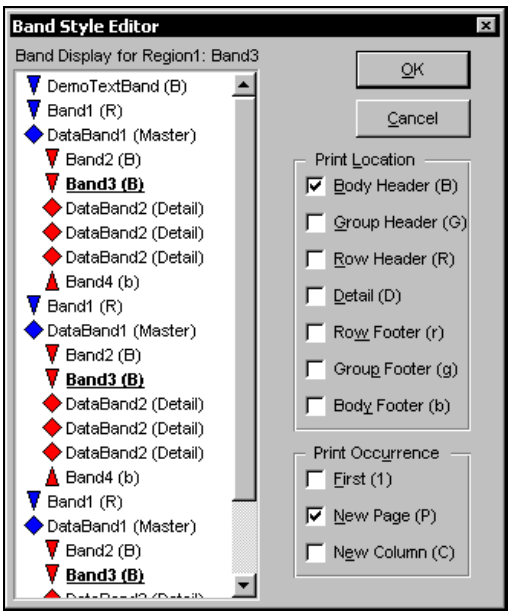


Рис. 11.5. Диалоговое окно **Band Style Editor**

В правой части окна редактора **Band Style Editor** с помощью группы флажков **Print Location** задается назначение текущей полосы отчета: **Body Header (B)** — заголовок в начале отчета; **Group Header (G)** — заголовок в начале группы, определенной в просмотре с помощью выражения `GROUP BY`; **Row Header (R)** — заголовок в начале каждой записи; **Detail (D)** — заголовок в начале подчиненного набора записей в отношении "один-ко-многим"; **Row Footer (r)** — окончание записи; **Group Footer (g)** — окончание группы записей; **Body Footer (b)** — окончание отчета. С помощью группы флажков **Print Occurrence** задается местоположение полосы в отчете: **First (1)** — на первой странице; **New Page (P)** — на новой странице, **New Column (C)** — в начале каждой колонки отчета.

Компоненты `DataCycle`, `CalcOp`, `CalcTotal` и `CalcController` являются невизуальными и предоставляют ряд дополнительных возможностей по вычислению данных на основе просмотров и отображению их в отчетах.

Компоненты управления отчетом

Компоненты, предназначенные для управления отчетами, находятся на странице **Rave** Палитры компонентов главного окна Delphi. С помощью этих компонентов можно выполнить все действия по настройке параметров, установлению соединения с источником данных, предварительному просмотру и печати имеющегося отчета с использованием генератора отчетов Rave Reports. Другое дело, что ряд этих действий, например предварительный просмотр и печать отчета, удобнее выполнить с использованием визуального конструктора отчетов Rave Reports.

Ниже кратко освещены важнейшие компоненты управления отчетом, без которых практически невозможны предварительный просмотр и печать отчетов вне среды визуального конструктора отчетов.

Компонент-проект отчета

При подготовке и использовании отчета обязательным является использование компонента-проекта отчета `RvProject`. С помощью свойства `ProjectFile` типа `String` этого компонента в окне Инспектора компонентов отчета или программно устанавливается связь с проектом отчета (файлом проекта отчета с расширением rav). Например:

```
RvProject1.ProjectFile:='C:\Program
   Files\Borland\Delphi7\Rave5\ProjectSTR.rav';
```

Для обеспечения связи компонента `RvProject` с компонентом `RvSystem`, предназначенным для управления отчетом, служит свойство `Engine` типа `TRpComponent`. Естественно, перед заданием ссылки на компонент `RvSystem` последний должен быть предварительно помещен в форму приложения. Этот компонент может отсутствовать в форме приложения отчета, в этом случае при запуске отчета открывается диалоговое окно **Output Options**, в котором выбираются вариант дальнейших действий с отчетом (печать, просмотр или помещение в файл) и, при необходимости, формат файла.

В проекте отчета может быть несколько отчетов. Для их указания в компоненте `RvProject` предназначены свойства `ReportName`, `ReportFullName` и `ReportDesc` (соответственно задающие имя, полное имя и описание отчета), имеющие тип `String` и определяющие параметры текущего отчета. Для смены в проекте текущего отчета можно использовать функцию `SelectReport(ReportName: String; FullName: boolean): boolean;`. Если параметр `FullName` имеет значение `True`, то параметр `ReportName` задает полное имя отчета, в противном случае — имя отчета.

Для получения списка имен отчетов, входящих в состав проекта, служит процедура `GetReportList(ReportList: TString; FullName: boolean);`. При ее вызове имена отчетов помещаются в список строк `ReportList`. При значении `True` параметра `FullName` в названный список помещаются полные имена отчетов, в противном случае — имена отчетов.

Для выполнения текущего отчета предназначены процедуры `Execute;` и `ExecuteReport(ReportName: String);`. Во втором случае имя отчета, задаваемое параметром процедуры, должно совпадать с именем отчета, определяемого свойством `ReportName` рассматриваемого нами компонента.

Компонент управления отчетом

Для управления отчетом служит компонент `RvSystem`, который обеспечивает выбор варианта действий с отчетом (печать, просмотр или помещение в файл) и при необходимости выбор формата файла. Как отмечалось, связь этого компонента с проектом отчета реализуется с помощью ссылки из компонента `RvProject` через свойство `Engine`.

Компонент `RvSystem` в своем составе инкапсулирует объекты, которые обеспечивают вывод отчета в один из трех системных приемников: `SystemPrinter` (печать отчета); `SystemPreview` (предварительный просмотр) и `SystemFiler` (вывод в файл).

Перед направлением отчета одному из системных приемников компонент `RvSystem` открывает диалоговое окно **Output Options**, в котором с помощью группы зависимых переключателей **Printer**, **Preview** и **File** осуществляется выбор системного приемника.

Компоненты установления соединения

Для установления соединения с источниками данных служат следующие компоненты странице **Rave** Палитры компонентов:

- `RvCustomConnection` — соединение с источниками данных, не относящихся к базам данных (текстовые файлы, электронные таблицы и др.);
- `RvDataSetConnection` — соединение с наборами данных приложения баз данных;
- `RvTableConnection` — соединение с компонентом типа `TTable`, устанавливаемое при использовании механизма BDE;
- `RvQueryConnection` — соединение с компонентом типа `TQuery`, устанавливаемое при использовании механизма BDE.

При помещении в форму проекта приложения указанные компоненты оказываются доступными для выбора при создании просмотров **Direct Data View**. При этом предварительно компонент должен быть связан с соответствующим набором данных.

Для важнейшего из указанных компонента `RvDataSetConnection` свойство `DataSet` указывает на имя набора данных. Имя компонента `RvDataSetConnection`, задаваемое с помощью его свойства `Name`, используется в отчете для указания имени соединения в просмотрах данных отчета. Напомним, что используемое в просмотре имя соединения можно определить с помо-

щью библиотеки просмотров **Data View Dictionary** как значение свойства `ConnectionName` для интересующего нас компонента `DataView`.

Схема управления отчетом и подсоединения данных

Мы рассмотрели назначение основных компонентов, служащих для управления отчетом и подсоединения данных к отчету. Отметим, что соединение с источниками данных в отчете можно выполнить двумя способами: с помощью драйвера Rave Reports, а также с помощью компонентов приложения доступа к данным и компонентов соединения. Для наглядности на рис. 11.6 приведена обобщенная схема управления отчетом и подсоединения данных к нему.

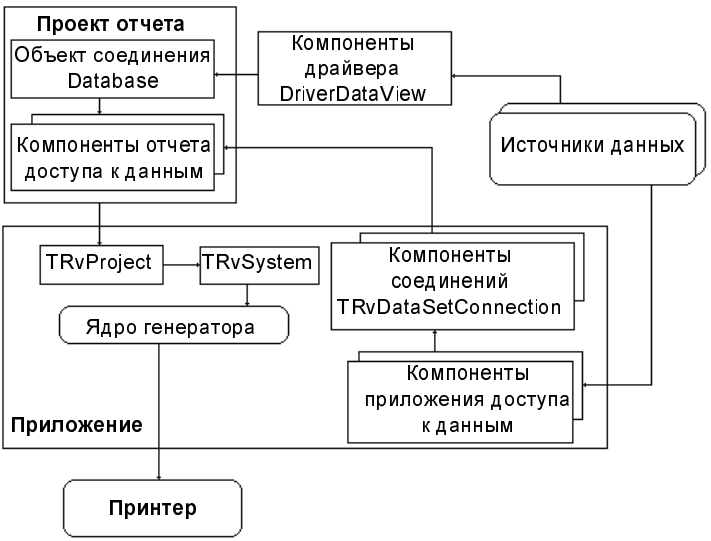


Рис. 11.6. Схема управления отчетом и подсоединения данных

Для организации соединения с помощью драйвера Rave Reports прежде всего командой **File\New Data Object** визуального конструктора отчетов создается объект соединения — компонент `Database`. При этом в открывшемся диалоговом окне **Data Connections** (см. рис. 11.3) выбирается вариант **Database Connection** и в очередном диалоговом окне нужно выбрать тип соединения (ADO, BDE или DBX). Каждому типу соединения соответствует свой драйвер (файл с расширением rdv). Далее в диалоговом окне **Database Connection Parameters** выполняется настройка параметров соединения, состав которых соответствует выбранному типу, и создается объект-соединение (с именем по умолчанию `DadabaseN`).

Теперь для связи компонентов данных отчета с данными соединения требуется создать просмотр данных, выполняющий формирование соответствующего на-

бора данных. Для этого нужно командой **File\New Data Object** визуального конструктора открыть диалоговое окно **Data Connections** (см. рис. 11.3) и выбрать вариант **Driver Data View**. В очередном диалоговом окне требуется выбрать одно из имеющихся соединений проекта. При этом открывается диалоговое окно **Query Advanced Designer**, в котором на вкладке **Layout** можно указать таблицы из выбранного нами соединения и задать связи между полями таблиц. С помощью вкладки **Sorting & Grouping** окна (рис. 11.7) можно выбрать поля для сортировки и группирования. Кнопка **Editor** открывает окно редактора с текстом сформированного запроса SQL, где его можно откорректировать вручную.

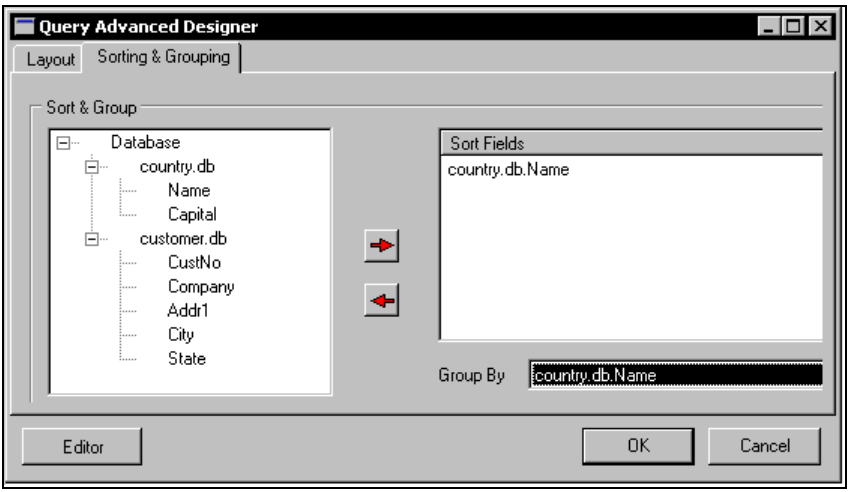


Рис. 11.7. Диалоговое окно **Query Advanced Designer**

После завершения настройки запроса SQL созданный нами объект-просмотр (компонент `DriverDataView`) появляется в словаре **Data View Dictionary** (с именем по умолчанию `DriverDataViewN`). Строка подготовленного таким образом запроса SQL является значением свойства `Query` объекта-просмотра. С его помощью можно повторно вызвать окно редактора Query Advanced Designer.

Теперь созданный нами объект-просмотр данных можно подключать к компонентам доступа к данным в отчете, таким как `DataText`, `DataMemo` и `DataBand`, с помощью их свойства `DataView`. Для таких компонентов, как `DataText` и `DataMemo` с помощью свойства `DataField` выбирается имя поля из таблицы объекта-просмотра.

Организация соединения с помощью компонентов приложения доступа к данным и компонентов соединения выполняется следующим образом. Для каждого соединения в приложении предварительно создается компонент набора данных, например, `Table` или `Query`, который соединяется с источником данных с помощью его свойств `DatabaseName` и `TableName`. Далее с таким компонентом набора данных устанавливается соединение с помощью соответствующего ком-

понента: `RvCustomConnection`, `RvDataSetConnection`, `RvTableConnection`, `RvQueryConnection` страницы **Rave** Палитры компонентов.

Напомним, что для компонента `RvDataSetConnection` свойство `DataSet` указывает на имя набора данных. Имя компонента `RvDataSetConnection`, задаваемое с помощью его свойства `Name`, используется в отчете для указания имени соединения в просмотрах данных отчета. Как отмечалось, используемое в просмотре имя соединения можно определить с помощью библиотеки просмотров **Data View Dictionary** как значение свойства `ConnectionName` для компонента `DataView`.

Теперь в среде визуального конструктора отчетов требуется создать объект прямого просмотра, с помощью команды **File\New Data Object** открыв диалоговое окно **Data Connections** (см. рис. 11.3) и выбрав вариант **Direct Data View**. Затем в очередном диалоговом окне **Data Connections** нужно выбрать интересующее нас соединение.

Как и при предыдущем способе организации соединения, созданный таким образом объект просмотра данных — компонент `DataView` — можно подключать к компонентам доступа к данным в отчете, например таким как `DataText`, `DataMemo` и `DataBand`.

Примеры создания и просмотра отчетов

Создание отчета включает три основных этапа:

- создание заготовки отчета в среде визуального конструктора;
- настройка соединения и подключение просмотра данных в случае создания отчета для приложения баз данных;
- помещение в форму приложения и настройка компонентов `RvProject` и `RvSystem`, которые осуществляют связь приложения с заготовкой отчета и управление предварительным просмотром и печатью отчета.

Заготовку собственно отчета в среде визуального конструктора можно выполнить так:

- путем настройки одного из уже имеющихся отчетов, например, в проекте RaveDemo.rav (находится в папке Delphi7/Rave5/Demos);
- путем конструирования отчета для обычного приложения или для приложения баз данных в среде визуального конструктора;
- для приложения баз данных можно создать простой табличный отчет или отчет для отношений типа "главный-подчиненный" с помощью Мастера отчетов.

В приведенном списке вариантов создания заготовки отчета второй вариант представляется наиболее трудоемким и требует хороших навыков работы с компонентами визуального конструктора.

Кроме перечисленных выше действий, при создании отчета можно также выполнять преобразование файлов отчета в различные форматы и др.

Предварительный просмотр отчета

Чтобы создать отчет путем настройки одного из уже имеющихся в проекте RaveDemo.rav (находится в папке Delphi7/Rave5/Demos) отчетов, нужно выполнить предварительный просмотр представляющего интерес отчета, а затем уже заняться его настройкой.

Рассмотрим, как выполнить предварительный просмотр отчета `TwoDetails` из проекта RaveDemo.rav. Для отчетов приложений баз данных, использующих просмотры данных, к которым относится отчет `TwoDetails`, перед просмотром и выполнением нужно настроить соединения. Вид отчета `TwoDetails` в окне визуального конструктора на этапе проектирования и соответствующее ему дерево отчета приведены на рис. 11.8.

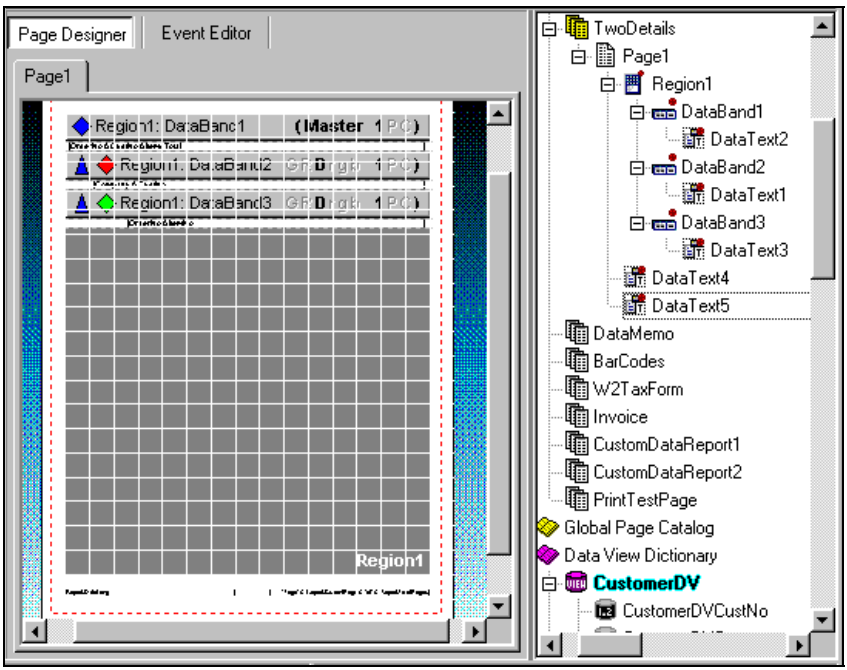


Рис. 11.8. Вид отчета `TwoDetails` в окне визуального конструктора

Чтобы определить имена просмотров данных, используемых компонентами отображения данных отчета (`DataText1`, `DataText2` и `DataText3`), поочередно выделим эти компоненты в окне проекта отчета и с помощью Инспектора компонентов отчета выясним значение свойства `DataView`. В нашем случае это просмотры с именами `CustomerDV`, `OrdersDV` и `ItemsDV`.

Далее нам нужно определить используемые этими просмотрами имена соединений. Для этого в словаре просмотров данных **Data View Dictionary** также пооче-

редно выделим просмотры с полученными именами и по значению их свойств `ConnectionName` выясним имена соединений. В нашем случае это значения `CustomerCXN`, `OrdersCXN` и `ItemsCXN` соответственно.

Все это означает, что в приложении баз данных имеются три набора данных, с которыми мы должны установить связь с помощью трех компонентов `RvDataSetConnection` (размещенных на странице **Rave** Палитры компонентов) с именами `CustomerCXN`, `OrdersCXN` и `ItemsCXN`. Для связи компонентов `RvDataSetConnection` с наборами данных из базы данных используется свойство `DataSet` этих компонентов. В качестве наборов данных в нашем примере используются таблицы из базы данных `DBDEMOS` с именами `orders.db`, `customer.db` и `items.db`.

После настройки соединения отчета с базой данных приложения можно выполнить предварительный просмотр отчета `TwoDetails` в среде визуального конструктора, задав для этого команду **File\Execute Report**. Выберем в открывшемся диалоговом окне **Output Options** вариант **Preview** и нажмем кнопку **OK**. Вид отчета `TwoDetails` в окне предварительного просмотра приведен на рис. 11.9.

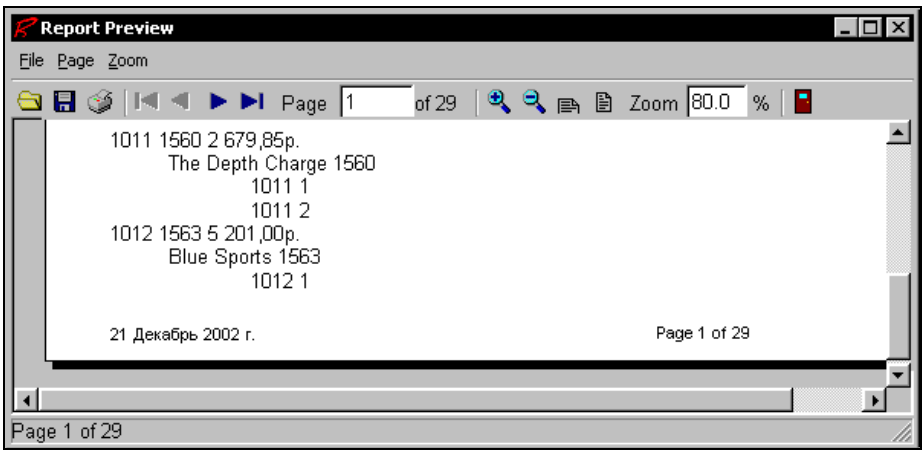


Рис. 11.9. Вид отчета `TwoDetails` при предварительном просмотре

Из вида отчета на рис. 11.9 и из значений свойства `DataField` компонентов отображения данных `DataText1`, `DataText2` и `DataText3` видно, что в отчете компонент `DataText1` отображает данные из трех полей `OrderNO`, `CustomNO` и `ItemsTotal` таблицы `orders.db`; компонент `DataText2` отображает данные из полей `Company` и `CustNO` таблицы `customer.db`; компонент `DataText3` отображает данные из полей `OrderNO` и `ItemNO` таблицы `items.db`. Кроме того, компоненты `DataText4` и `DataText5` соответственно отображают номер текущей страницы из общего числа страниц и дату просмотра отчета.

Полученная нами при предварительном просмотре информация, а также данные о составе компонентов и их свойствах позволяют при необходимости настроить отчет соответствующим образом. Например, можно изменить таблицы базы данных приложения, можно изменить имена отображаемых полей и т. п.

Простой отчет приложения базы данных

Простой отчет представляет собой отчет на основе данных из одного набора данных и содержит сведения, которые выводятся в табличном виде без какой-либо дополнительной обработки данных (например, группирования). Размещение и вид печатаемых в отчете данных аналогичны размещению и виду данных, отображаемых в сетке `DBGrid`. Отличием является то, что данные отчета не размещаются в форме, а представлены в виде бумажного документа, и их нельзя редактировать.

Рассмотрим последовательность шагов, которую требуется выполнить при добавлении простого отчета к приложению базы данных.

1. Откроем приложение базы данных Delphi, для которого требуется добавить простой отчет.
2. В форму приложения базы данных со страницы **Rave** Палитры поместим компонент `RvDataSetConnection` и с помощью Инспектора объектов присвоим свойству `DataSet` этого компонента значение, указывающее на имя набора данных, используемого в приложении.
3. С помощью визуального конструктора отчетов подготовим отчет и создадим файл проекта отчета. Для этого:
   - с помощью команды **Tools\Rave Designer** меню Delphi запустим визуальный конструктор Rave Reports 5.0;
   - выбором команды **File\New Data Object** откроем диалоговое окно **Data Connections** (см. рис. 11.3) и выберем в нем вариант **Direct Data View**;
   - в очередном диалоговом окне в списке **Active Data Connections** выберем вариант **RVDataSetConnection1** и нажмем кнопку **Finish**;
   - в дереве проекта в правой части окна визуального конструктора отчетов раскроем узел **Data View Dictionary** и в нем раскроем вновь созданный узел **DataView1**;
   - выбрав команду **Toolls\Report Wizards\Simple Table**, запустим Мастер создания простых таблиц в отчете, выберем вариант **DataView1** и нажмем кнопку **Next**;
   - на последующих шагах работы с Мастером выберем поля таблицы для отображения в отчете, при необходимости изменим очередность следования полей, установим параметры полей страницы, текст заголовков и шрифтов, используемых в отчете (рис. 11.10);

- на заключительном этапе работы с Мастером нажатием кнопки **Generate** запустим процесс генерации отчета;
- для просмотра сгенерированного отчета выберем команду **File\Execute Report**, в открывшемся диалоговом окне **Output Options** в поле **Report Destination** выберем переключатель **Preview** и нажмем **OK**. Вид полученного нами отчета при просмотре приведен на рис. 11.11;

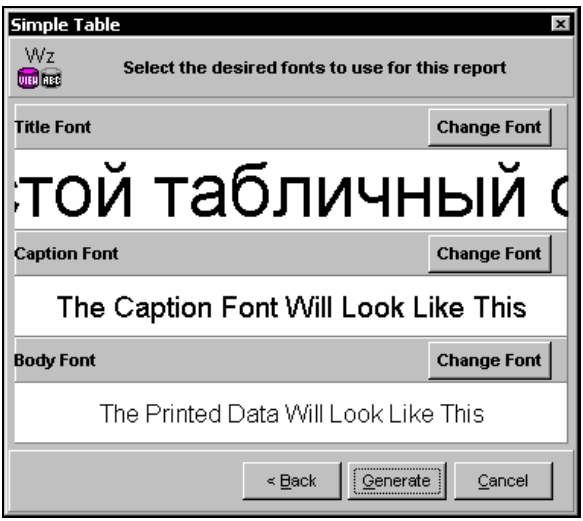


Рис. 11.10. Диалоговое окно настройки параметров шрифта

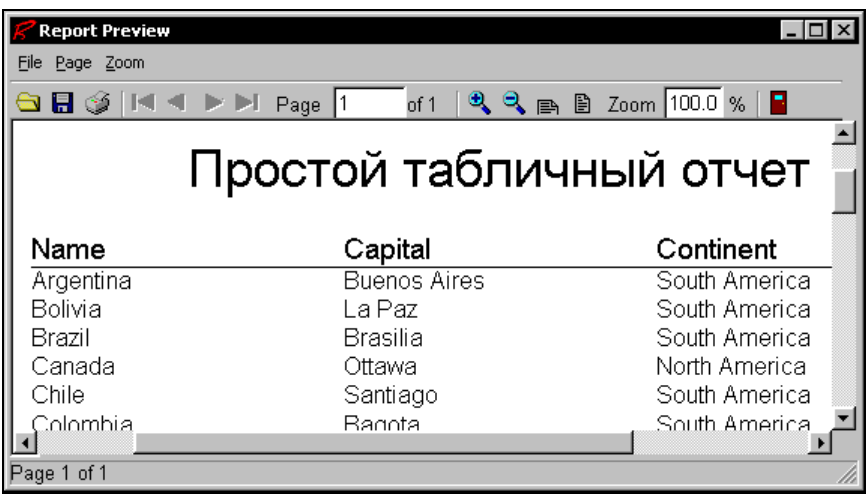


Рис. 11.11. Вид простого табличного отчета при просмотре

- при необходимости выполним настройку параметров (шрифта, цвета и др.) отдельных составляющих созданного отчета;
- сохраним созданный нами проект отчета в файле с произвольным именем, например STRep, и расширением rav с помощью команды **File\Save As**;
- свернем или закроем диалоговое окно работы с визуальным конструктором отчета и вернемся к работе с приложением в Delphi.

4. В форму приложения базы данных со страницы **Rave** Палитры поместим компонент проекта отчета `RvProject` и установим в значение `C:\Program Files\Borland\Delphi7\Rave5\STRep.RAV` (спецификация созданного нами файла проекта) свойство `ProjectFile` этого компонента с помощью Инспектора объектов.
5. В форму приложения базы данных со страницы **Standard** Палитры компонентов поместим кнопку (компонент `Button`) и в качестве обработчика события `OnClick` нажатия этой кнопки зададим вызов метода `ExecuteReport`, обеспечивающего выполнение отчета с заданным именем (в нашем примере `Report5`) из состава проекта отчета (компонента `RvProject1`), например так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
RvProject1.Open;
try
  RvProject1.ExecuteReport('Report5');
finally
  RvProject1.Close;
end;
end;
```

6. С помощью клавиши <F9> запустим приложение на выполнение, и в форме щелкнем мышью на кнопке, за которой закреплен созданный нами обработчик события. В открывшемся диалогоовм окне выберем вариант печати отчета и нажмем кнопку **OK**.

В результате выполненных действий будет выдан на печать простой табличный отчет, содержащий данные из таблицы приложения базы данных, вид которого в окне просмотра приведен на рис. 11.11.

Глава 12

Инструменты

Инструменты — это программы, предназначенные для обслуживания БД, а также для выполнения вспомогательных действий при разработке приложений, например, для создания таблиц и отладки SQL-запросов. Совместно с Delphi поставляется большое число инструментов, применимых для работы как с локальными, так и с удаленными БД. В этой главе рассматриваются инструменты для работы с локальными базами данных. Инструментальные программы, предназначенные для работы с удаленными БД, представлены в *главе 15*.

Программа BDE Administrator

Программа BDE Administrator представляет собой инструмент администрирования процессора баз данных BDE (далее Администратор BDE, или Администратор). Для вызова Администратора BDE запускается файл bdeadmin.exe, находящийся в одном каталоге с процессором баз данных. Внесенные изменения сохраняются по окончании работы с Администратором в файле конфигурации idapi32.cfg. Программу также можно вызвать через главное меню Windows выбором пункта **Program\Borland Delphi 7\BDE Administrator**.

Администратор BDE позволяет настраивать параметры БД и операционной системы. Основные параметры:

- параметры псевдонима БД:
  - название;
  - тип;
  - путь;
- параметры драйвера:
  - тип;
  - язык;
- системные установки:
  - установки по умолчанию;
  - форматы даты, времени и числовые форматы.

Для настройки требуемого параметра в левой части окна Администратора BDE нужно выбрать нужный объект, после чего в правой части окна открывается доступ к списку параметров этого объекта. При редактировании выбранного в панели инструментов объекта становятся доступными кнопки отмены/подтверждения сделанных изменений (с красной/синей стрелкой). Отменить или подтвердить изменения также можно командами **Cancel** и **Apply** главного или контекстного меню. Слева от объектов, имеющих неутвержденные изменения, отображается зеленый треугольник.

Добавить новый объект можно, выбрав в окне Администратора пункт меню **Object\New**. Удаление выделенного объекта выполняется командой **Object\Delete** главного или контекстного меню или нажатием кнопки **Delete** панели инструментов (с синим косым крестом).

Текущее состояние объекта, выбранного в левой части окна, отображает значок, который появляется слева от имени объекта. Варианты значков и их связь с состоянием объекта перечислены ниже.

- Зеленый треугольник — объект находится в режиме редактирования (имеет неутвержденные изменения).
- Зеленый треугольник с красными лучами — вновь созданный и еще не сохраненный в конфигурации объект, для которого выполняется редактирование.
- Красный треугольник — объект находится в режиме редактирования, некоторые изменения являются некорректными и не могут быть сохранены.
- Красный треугольник с красными лучами — созданный и еще не сохраненный в конфигурации объект находится в режиме редактирования, некоторые данные являются некорректными и не могут быть сохранены.
- Ярко-зеленый квадрат — объект открыт.

Из приложения можно управлять настройками BDE с помощью соответствующих методов компонента `Session`, который рассматривается в *главе 13*.

Работа с псевдонимами

Псевдоним (alias) указывает местонахождение файлов БД и представляет собой специальное имя для обозначения каталога. Использование псевдонимов существенно облегчает перенос файлов БД в другие каталоги и на другие компьютеры. При этом не требуется изменять приложение, которое осуществляет доступ к таблицам БД. Если в приложении расположение таблиц указано с помощью псевдонима, то после перемещения БД для обеспечения работоспособности приложения достаточно изменить значение пути, заданное в псевдониме. Если же в приложении путь к БД указан в явном виде, т. е. без псевдонима, то после перемещения БД нужно изменять само приложение — вносить изменения в исходный код и заново его транслировать.

Для создания псевдонима базы данных перед вызовом пункта меню **Object\New** Администратора BDE нужно выбрать вкладку **Database** в левой части окна, в противном случае команда **New** меню будет недоступна. После задания этой

команды появляется диалоговое окно **New Database Alias** (Новый псевдоним БД), в котором нужно выбрать тип драйвера. Для локальных таблиц Paradox и dBase выбирается тип `STANDARD`, для других таблиц указывается соответствующий тип, например, для удаленного сервера InterBase — тип `INTRBASE`.

После нажатия кнопки **OK** создается псевдоним и его данные отображаются в окне Администратора BDE (рис. 12.1). Новый псевдоним автоматически получает имя `STANDARD1` и параметры по умолчанию. Можно переименовать псевдоним, выполнив команду **Rename** контекстного меню псевдонима или меню **Object** главного меню Администратора BDE.

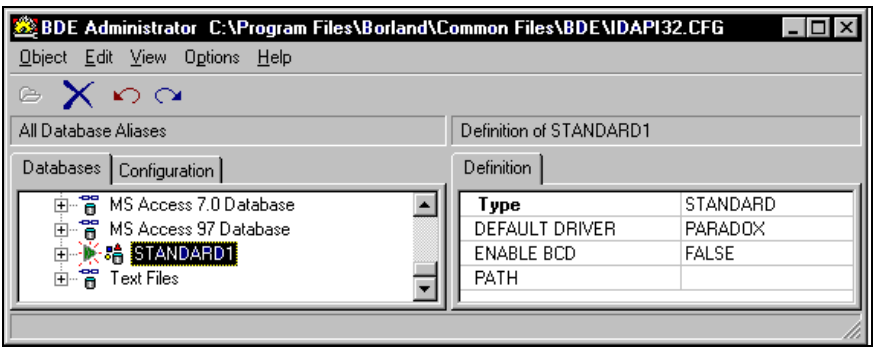


Рис. 12.1. Установка параметров псевдонима

Псевдоним для работы с локальными БД имеет три параметра.

- `DEFAULT DRIVER` — указывает формат таблиц БД (по умолчанию имеет значение `Paradox`). Кроме того, можно установить значения `dBase` или `ASCIIDRV` для текстовых файлов, разбитых на колонки.
- `ENABLE BCD` — указывает на необходимость перевода чисел в формат BCD, что позволяет более точно выполнять вычисления, но уменьшает скорость их выполнения. По умолчанию имеет значение `False` и, соответственно, формат BCD не используется.
- `PATH` — указывает расположение (каталог) БД. После создания псевдонима путь не определен, и разработчик должен установить его самостоятельно.

Отметим, что псевдонимы для удаленных БД имеют большее число параметров, например, у псевдонима типа `INTRBASE` пятнадцать параметров.

При необходимости можно изменить параметры псевдонима, например, имя и путь. Для параметров `DEFAULT DRIVER` и `ENABLE BCD` значение выбирается в раскрывающемся списке. Значение параметра `PATH` можно ввести вручную или с помощью выбора нужного каталога в окне **Select Directory**, которое появляется при двойном щелчке в поле значения параметра. После выбора диска, каталога и нажатия кнопки **OK** соответствующий путь автоматически присваивается параметру `PATH` в качестве значения.

Смена пути выполняется при перемещении БД в другой каталог.

Ненужный псевдоним можно удалить, выполнив команду **Delete** контекстного меню псевдонима или команду **Object\Delete** главного меню Администратора.

Для каждого псевдонима указывается соответствующий драйвер, для локальных таблиц — обычно `dBase` или `Paradox`, при этом параметры драйвера устанавливаются по умолчанию.

Для псевдонима типа `INTRBASE`, предназначенного для доступа к удаленной БД InterBase, в меню **Object** программы появляется команда **Diagnostics**. Эта команда открывает окно **Communication Diagnostic Tool** (Диагностирование соединения) проверки соединения с удаленной БД (рис. 12.2).

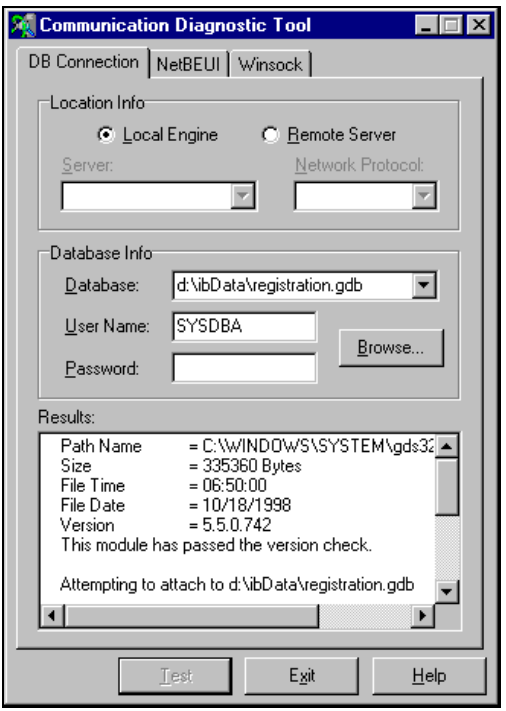


Рис. 12.2. Диалоговое окно проверки соединения с удаленной БД

После указания параметров соединения (расположения БД, имени и пароля пользователя) и нажатия кнопки **Test** выполняется соединение с БД, результаты которого выводятся в поле **Results**. Более подробно вопросы, касающиеся соединения с удаленной БД, рассматриваются в *главе 13*.

Для работы с псевдонимами можно использовать и другие программы, например, Database Desktop.

Параметры драйвера

Для доступа к параметрам драйвера в левой части окна Администратора нужно выбрать вкладку **Configuration** и требуемый драйвер. Драйверы процессора баз данных делятся на "родные" (**Native**) для него драйверы и драйверы ODBC. При выборе соответствующего драйвера в правой части окна появляется список параметров драйвера, которые можно просматривать или изменять. В качестве примера на рис. 12.3 показаны параметры локального драйвера базы данных Paradox.

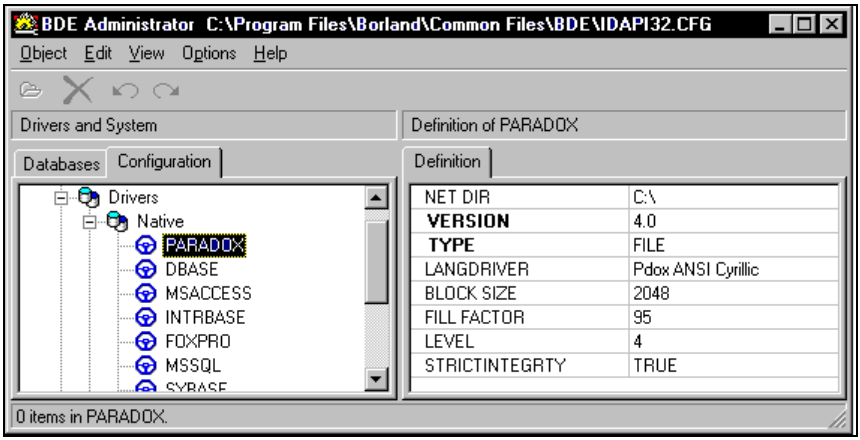


Рис. 12.3. Установка параметров драйвера базы данных Paradox

Наибольший интерес представляют параметры `TYPE` и `LANGDRIVER`.

Параметр `TYPE` указывает тип драйвера и принимает следующие значения:

- `FILE` — для локальных БД;
- `SERVER` — для удаленных БД.

Параметр `LANGDRIVER` определяет драйвер языка (языковой драйвер), используемый для кодировки символов. В нашей стране для этого параметра рекомендуется использовать значения `dBASE RUS cp866` и `Pdox ANSI Cyrillic` соответственно для драйверов dBase и Paradox. Это обеспечивает корректное отображение символов кириллицы в приложениях, их правильную сортировку и преобразование, например, при использовании функций `AnsiUpperCase` и `AnsiLowerCase`.

Замечание

Параметры драйвера действуют только на те приложения, которые осуществляют доступ к БД через процессор баз данных.

Как отмечалось, у драйвера базы данных InterBase гораздо больше параметров (рис. 12.4), чем у драйвера Paradox.

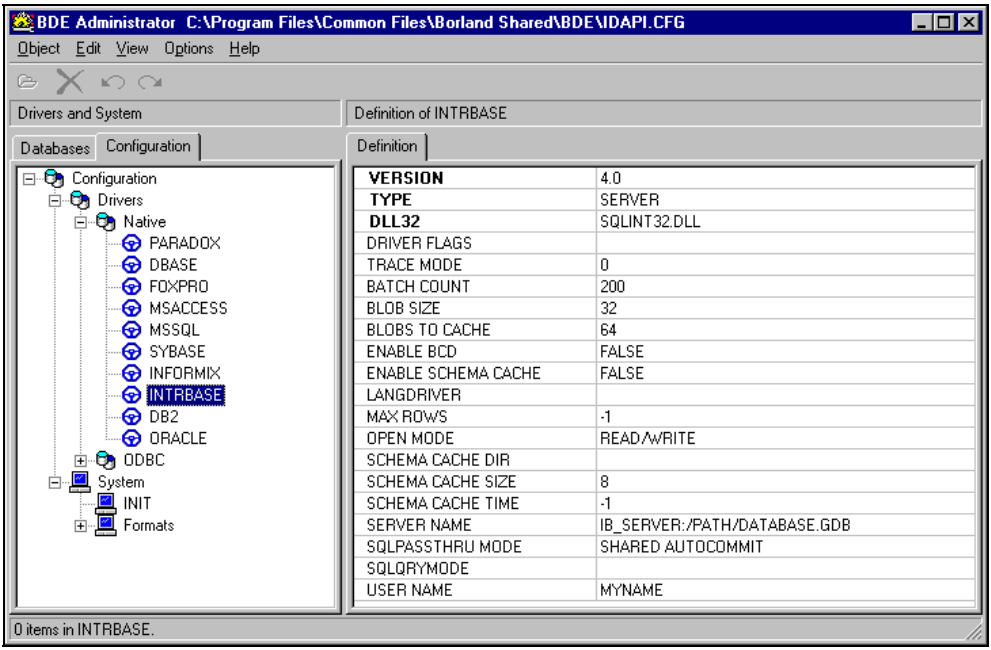


Рис. 12.4. Установка параметров драйвера InterBase

Наибольший интерес представляют параметры, перечисенные ниже.

- `DLL32` — указывает драйвер SQL-Links, используемый для доступа к БД (sqlint32.dll).
- `ENABLE BCD` — указывает на необходимость перевода чисел в формат BCD, что позволяет более точно выполнять вычисления, но уменьшает скорость их выполнения; по умолчанию имеет значение `False`, и формат BCD не используется.
- `LANGDRIVER` — определяет драйвер языка, используемый для кодировки символов; выбор языкового драйвера `Pdox ANSI Cyrillic` обеспечивает корректную работу с символами русского алфавита.
- `MAXROWS` — определяет максимальное число записей, которые могут быть считаны из удаленной БД при одном запросе к ней. Этот параметр используется для блокировки попыток считывания большого объема информации при ошибочном или неправильно сформированном SQL-запросе к удаленной БД. Установка небольшого значения параметра `MAXROWS` часто приводит к ошибкам, связанным с тем, что возвращаемое на основании запроса число записей может легко превысить установленное ограничение. По умолчанию параметр имеет значение `-1`, что соответствует отсутствию ограничений на число возвращаемых записей.

- `OPEN MODE` — определяет режим доступа к данным. По умолчанию имеет значение `READ/WRITE`, что обеспечивает как чтение, так и изменение записей. При установке значения `READ ONLY` разрешается только чтение записей.
- `SERVER NAME` — указывает имя удаленной БД. Формат задания этого имени зависит от сетевого протокола. Например, для протокола TCP/IP имя БД состоит из имени сервера, пути к БД и собственно имени БД.
- `SQLPASSTHRU MODE` — определяет способ взаимодействия процессора баз данных BDE с сервером на уровне транзакций. По умолчанию имеет значение `SHARED AUTOCOMMIT`, при котором в случае, если приложение явно не управляет транзакциями, сервер выполняет это управление автоматически.
- `SQLQRYMODE` — задает режим выполнения запросов. Параметр имеет следующие возможные значения:
  - `LOCAL` — запрос выполняется локально (на компьютере пользовательского приложения);
  - `SERVER` — запрос выполняется на сервере; если сервер не может выполнить запрос, то данный запрос не выполняется;
  - если не выбрано ни одно из этих значений (по умолчанию), запрос посылается серверу; если сервер не может выполнить запрос, то он выполняется локально.
- `USER NAME` — задает начальное имя пользователя, которое при соединении с БД содержится в соответствующем поле. По умолчанию имеет значение `MYNAME`.

Для доступа к удаленным БД с помощью BDE используются драйверы SQL-Links:

- Microsoft Access — idda3532.dll, iddao32.dll;
- Microsoft SQL Server — sqlmss32.dll;
- SyBase — sqlssc32.dll, sqlsyb32.dll;
- Informix — sqlinf9.dll, sqlinf32.dll;
- InterBase — sqlint32.dll;
- DB2 — sqldb2v5.dll, sqldb232.dll;
- Oracle — sqlora32.dll, sqlora8.dll.

Эти драйверы поставляются совместно с процессором баз данных BDE и находятся в его каталоге.

Если для БД нет драйвера SQL Links, то для него можно использовать драйвер ODBC. Отметим, что ODBC (Open DataBase Connectivity — совместимость открытых баз данных) представляет собой интерфейс прикладного программирования (API) в виде библиотеки функций, вызываемых из различных программных сред и позволяющих приложениям унифицированно обращаться на языке SQL к базам данных различных форматов. В рамках концепции ODBC есть стандарт для драйверов. Разработчики многих БД для доступа к ним предостав-

ляют драйверы, соответствующие этому стандарту, обеспечивая тем самым совместимость различных типов баз.

Настройка драйверов ODBC выполняется с помощью программы-администратора ODBC, окно **ODBC Data Source Administrator** которого открывается через элемент **ODBC Data Source (32-bit)** Панели управления Windows. Администратор ODBC можно вызвать также из Администратора BDE командой **ODBC Administrator** контекстного меню драйвера.

Отметим, что для работы с БД могут существовать драйверы как SQL-Links, так и ODBC. В этом случае предпочтительнее использовать драйверы SQL-Links, т. к. они работают быстрее.

Системные установки

Системные установки делятся на два вида: установки по умолчанию и форматы.

Параметры системных *установок по умолчанию* становятся доступными при выборе в окне Администратора BDE объекта **Configuration\System\INIT**. Основными параметрами этих установок являются следующие:

- `DATA REPOSITORY` (активный словарь данных);
- `DEFAULT DRIVER` (драйвер, устанавливаемый для локальных БД), по умолчанию драйвер Paradox; устанавливается как драйвер по умолчанию для каждой новой БД;
- `LANGDRIVER` (драйвер языка); рекомендуется установить этот параметр в значение `dBASE RUS cp866` (для драйвера dBase) или `Pdox ANSI Cyrillic` (для драйвера Paradox);
- `LOCAL SHARE` (флаг разделяемого доступа к локальным БД между приложениями); по умолчанию имеет значение `False`, и разделяемый доступ к локальным данным запрещен;
- `SQLQRYMODE` (режим выполнения запросов).

Системные установки форматов позволяют задавать следующие параметры форматов даты, времени и чисел:

- параметры даты:
  - `SEPARATOR` (разделитель дня, месяца и года); по умолчанию применяется символ из системных установок Windows;
  - `MODE` (порядок расположения дня (Д), месяца (М) и года (Г) в дате); принимает следующие значения: `0` (МДГ), `1` (ДМГ), `2` (ГМД); по умолчанию используется порядок расположения, заданный в системных установках Windows;
  - `FOURDIGITYEAR` (число цифр, отображаемых для года), принимает значения: `False` (две цифры) — по умолчанию, `True` (четыре цифры);
  - `YEARBIASED` (способ преобразования значения года, заданного двумя цифрами), принимает значения: `True` (к двузначному числу прибавляется число `2000`) — по умолчанию, `False` (значение года не изменяется);

- `LEADINGZEROM` (незначащий ноль для месяца, обозначенного одной цифрой), принимает значения: `True` (добавляется незначащий ноль) — по умолчанию, `False` (ноль не добавляется);
- `LEADINGZEROD` (незначащий ноль для дня, обозначенного одной цифрой), принимает значения: `True` (добавляется незначащий ноль) — по умолчанию, `False` (ноль не добавляется);

❒ параметры времени:

- `TWELVEHOUR` (формат времени), принимает значения: `True` (время отображается в 24-часовом формате) — по умолчанию, `False` (время отображается в 12-часовом формате);
- `AMSTRING` (символы для 12-часового формата, обозначающие время до полудня), по умолчанию используются символы `AM`;
- `PMSTRING` (символы для 12-часового формата, обозначающие время после полудня), по умолчанию используются символы `PM`;
- `SECONDS` (отображение секунд), принимает значения: `True` (секунды отображаются) — по умолчанию, `False` (секунды не отображаются);
- `MILLISECONDS` (отображение миллисекунд), принимает значения: `True` (миллисекунды отображаются), `False` (миллисекунды не отображаются) — по умолчанию;

❒ числовые параметры:

- `DECIMALSEPARATOR` (разделитель целой и дробной части числа); по умолчанию используется символ из системных установок Windows;
- `THOUSENDSEPARATOR` (разделитель тысяч в целой части числа); по умолчанию используется символ из системных установок Windows;
- `DECIMALDIGIT` (число разрядов в дробной части числа) — по умолчанию имеет значение `2`;
- `LEADINGZERON` (незначащий ноль для чисел, по модулю меньших единицы), принимает значения: `True` (ноль отображается), `False` (ноль не отображается) — по умолчанию.

Системными установками Windows можно управлять через Панель управления.

Использование конфигурационных файлов

Как отмечалось выше, по окончании работы с Администратором BDE и сохранения внесенных изменений параметры запоминаются в файле конфигурации idapi32.cfg. Этот конфигурационный файл используется процессором баз данных автоматически (по умолчанию).

Кроме того, имеется возможность сохранить выполненные настройки в собственном конфигурационном файле. Для этого нужно выполнить команду главного меню Администратора BDE **Object\Save As Configuration**. В появившемся окне указываются каталог и имя конфигурационного файла (.cfg). После нажатия

кнопки **Save** указанный файл сохраняется на диске и доступен для последующего использования.

Для загрузки (открытия) собственного конфигурационного файла нужно выполнить команду **Object\Open Configuration**. После выбора конфигурационного файла и нажатия кнопки **Open** содержащиеся в нем установки вступают в действие.

Можно объединить несколько конфигурационных файлов. Для этого нужно выбрать команду **Object\Merge Configuration**. После выбора файла и нажатия кнопки **Open** данные открытого и выбранного конфигурационных файлов объединяются и соответствующие им установки вступают в действие.

Этой возможностью удобно пользоваться при наличии нескольких приложений, обращающихся к процессору баз данных. Процессор баз данных можно включить только в дистрибутив приложения, устанавливаемого первым. При установке последующих приложений кроме файлов приложения желательно иметь и конфигурационные файлы этих приложений. После объединения данных файлов с конфигурационным файлом idapi32.cfg необходимые настройки, например, псевдонимы приложений, становятся доступными процессору баз данных.

Отметим, что при настройке компьютера пользователя вместо объединения конфигураций можно ввести необходимые данные вручную. Однако это более трудоемкий вариант, он рекомендуется только когда таких данных немного.

Программа Database Desktop

Программа Database Desktop предназначена для создания и редактирования таблиц, визуальных запросов и SQL-запросов, а также для выполнения действий с псевдонимами БД. Эту программу можно вызвать из среды Delphi командой **Tools\Database Desktop** или запуском файла dbd32.exe, находящегося в одном каталоге с файлами программы Database Desktop (по умолчанию это каталог \Database Desktop). Программу также можно вызвать через главное меню Windows, выбрав команду **Program\Borland Delphi 7\Database Desktop**.

Создание и изменение структуры таблиц уже были рассмотрены в *главе 4* при описании технологии создания приложения БД. Здесь мы познакомимся с применением программы Database Desktop для выполнения остальных действий.

Если при манипулировании таблицами и запросами не указывается расположение БД, то по умолчанию они считаются расположенными в рабочем каталоге программы Database Desktop. Для изменения этого каталога нужно выполнить команду **File\Working Ditectory** меню программы, в результате чего появится окно установки каталога **Set Working Directory** (рис. 12.5).

Задать рабочий каталог можно двумя способами:

- непосредственно ввести путь в поле **Working Directory** или нажатием кнопки **Browse** открыть окно просмотра каталогов **Directory Browser**, в котором выбрать нужные диск (псевдоним) и каталог;
- выбрать в списке псевдонимов **Aliases** ранее определенный псевдоним.

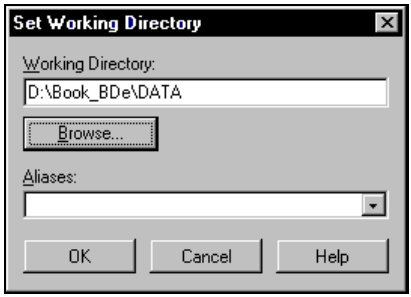


Рис. 12.5. Установка рабочего каталога

При одновременном использовании обоих способов рабочий каталог выбирается по полю **Working Directory**. После задания рабочего каталога и нажатия кнопки **OK** выбранные установки вступают в действие.

Сразу после инсталляции программа Database Desktop неправильно отображает символы русского алфавита, поэтому нужно изменить используемый ею шрифт. Для этого командой **Edit\Preferences** нужно открыть окно настройки **Preferences**, в поле **Default system font** которого выводится название используемого по умолчанию шрифта. Нажатие кнопки **Change** открывает окно выбора шрифта. В этом окне следует выбрать новый шрифт, содержащий символы русского алфавита, например, Arial или Times New Roman.

Редактирование записей таблиц

Чтобы открыть таблицу, нужно выбрать команду **File\Open\Table** или нажать на панели инструментов кнопку с изображением таблицы. Затем в открывшемся окне **Open Table** нужно выбрать имя главного файла таблицы. После нажатия кнопки **Open** в поле программы Database Desktop открывается окно с записями таблицы.

По умолчанию для таблицы включен режим просмотра данных **View Data**, и ее записи доступны только для чтения. Переключение в режим редактирования **Edit Data** выполняется одноименной командой меню **Table**. После этого в таблице можно добавлять и удалять записи, а также изменять значения полей.

Добавление записи выполняется нажатием клавиши <Insert>, новая запись появляется над текущей записью, в любом поле которой установлен курсор ввода. Удаляется текущая запись комбинацией клавиш <Ctrl>+<Delete>.

Для редактирования значения поля нужно установить в него курсор ввода (мышью или клавишами управления курсором) и нажать любую алфавитно-цифровую клавишу, при этом старое значение поля удаляется. Фиксация нового значения выполняется нажатием клавиши <Enter> или переводом курсора ввода в другое поле или другую запись. Перейти к редактированию поля можно также двойным щелчком мыши, при этом старое значение поля не удаляется. Возврат в режим просмотра таблицы выполняется командой **Table\View Data**.

Режим просмотра и редактирования включается с помощью кнопок панели инструментов, на которых изображены буквы **ab** и карандаш на фоне таблицы соответственно. При включении режима кнопка остается нажатой. Переключение между режимами выполняется с помощью клавиши <F9>.

Работа с псевдонимами

Для работы с псевдонимами БД используется программный инструмент **Alias Manager** (Менеджер псевдонимов, рис. 12.6), вызываемый командой **Tools\Alias Manager** меню программы Database Desktop.

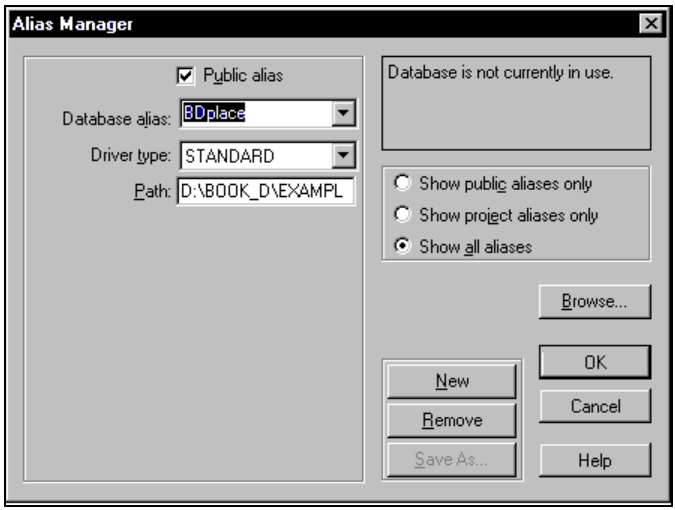


Рис. 12.6. Диалоговое окно Менеджера псевдонимов

С помощью Менеджера псевдонимов можно создавать (кнопка **New**) и удалять (кнопка **Remove**) псевдонимы. Имя псевдонима вводится (для нового) или выбирается (для существующего) в списке **Database alias**. Кроме того, можно изменять параметры псевдонимов: тип драйвера (список **Driver type**) и путь (поле **Path**). Путь можно ввести вручную или выбрать в окне просмотра каталогов, открываемом нажатием кнопки **Browse**.

Нажатие кнопки **Save As** открывает окно сохранения конфигурационного файла (idapi.cfg) процессора баз данных. При необходимости можно запомнить конфигурацию под другим именем.

Работа с SQL-запросами

Работа с SQL-запросами включает:

- ❒ создание запроса;
- ❒ редактирование запроса;
- ❒ выполнение запроса.

Для создания SQL-запроса нужно, выполнив команду **New\SQL File** программы Database Desktop, открыть окно редактора **SQL Editor** (рис. 12.7) и набрать в нем текст SQL-запроса.

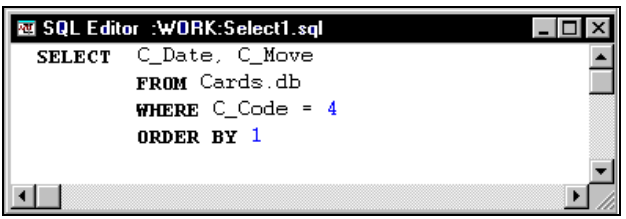


Рис. 12.7. Окно редактора SQL-запроса

Как и любой другой документ, запрос можно сохранять (**Save**) на диске в виде текстового файла, сохранять под другим именем (**Save As**), а также редактировать (в окне **SQL Editor**). Можно также открыть ранее сохраненный запрос (**Open**). Файл запроса имеет расширение sql. (В скобках указаны соответствующие команды пункта **File** программы Database Desktop.)

Для выполнения SQL-запроса нужно выбрать команду **SQL\un SQL** меню программы Database Desktop или нажать кнопку панели инструментов. Перед выполнением запроса производится проверка правильности его синтаксиса.

Отметим, что для работы с локальными таблицами в программе Database Desktop реализована версия SQL-92, несколько отличающаяся от стандарта. В частности, в ней больше, чем в стандарте, типов полей и упрощены конструкции отдельных операторов.

При выполнении правильно оформленного запроса появляется окно с записями, отобранными в результате обработки запроса (рис. 12.8).

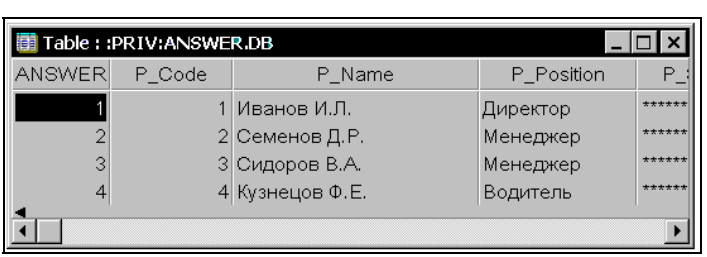


Рис. 12.8. Результат выполнения SQL-запроса

В среде программы Database Desktop удобно отлаживать SQL-запросы, которые в последующем можно присвоить в качестве значения свойству `SQL` компонента `Query`.

Визуальное конструирование запросов

Для удобного конструирования запросов можно использовать визуальный конструктор, вызываемый командой **New\QBE Query** программы Database Desktop. При этом открывается окно **Select File**, где нужно указать имя главного файла таблицы, на основании данных которой строится запрос. После выбора файла и нажатия кнопки **Open** открывается окно визуального конструктора (рис. 12.9).

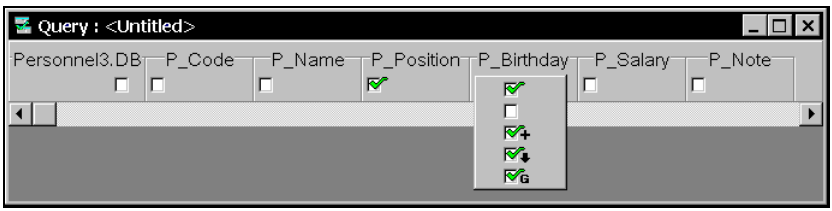


Рис. 12.9. Диалоговое окно визуального конструктора запросов SQL

В этом окне показаны имена таблицы и всех ее полей. Под именем каждого поля находятся флажок и текстовое поле (слева от флажка для имени таблицы и справа — для имени поля).

С помощью визуального конструктора можно:

- создавать и изменять запрос по образцу;
- выполнять запрос по образцу;
- сохранять запрос по образцу как SQL-запрос.

Бóльшая часть работы с запросом по образцу проходит в окне визуального конструктора, где для каждого поля таблицы задаются условия отбора и сортировки. Для этого щелчком правой кнопки мыши в поле имени таблицы (слева от флажка) вызывается контекстное меню, в котором выбирается вид запроса:

- пустая строка (отбор и редактирование записей таблицы);
- `INSERT` (вставка записей в таблицу);
- `DELETE` (удаление записей из таблицы);
- `SET` (сравнение записей в таблицах).

Для выполнения запроса нужно выполнить команду **Query\Run Query** программы Database Desktop или нажать кнопку панели инструментов. Перед выполнением запроса производится проверка его правильности.

Для получения текста запроса, соответствующего визуальному запросу, нужно выполнить команду **Query\Show SQL** или нажать на панели инструментов кнопку с буквами **SQL**. При отсутствии ошибок в конструкции запроса автоматически открывается окно **SQL Editor** (Редактор SQL-запроса, см. рис. 12.7), в котором содержится текст запроса на языке SQL. При наличии ошибок окно Редактора SQL-запроса не появляется, а выдается сообщение об ошибке.

Запрос можно сохранять (**Save**) на диске под своим именем или под другим (**Save As**). Можно также открыть ранее сохраненный запрос (**Open**). Файл запроса имеет расширение qbe. (В скобках указаны команды меню **File** программы Database Desktop.)

Отбор записей из таблицы

При отборе записей из таблицы щелчком правой кнопки мыши на флажках полей вызывается контекстное меню (см. рис. 12.9), в котором можно выбрать метку для флажка:

- пусто (поле не включается в результат запроса);
- галочка со знаком + (поле включается в результат запроса);
- галочка (поле включается в результат запроса с сортировкой записей по возрастанию значений этого поля);
- галочка с черной стрелкой (поле включается в результат запроса с сортировкой записей по убыванию значений этого поля);
- галочка с буквой **G** (поле включается в результат запроса и используется для группирования записей).

Кроме того, справа от флажка для каждого поля таблицы можно вручную ввести условие отбора записей. Поле, для которого допускается ввод условия запроса, обозначается текстовым курсором. Этот курсор отображается черным прямоугольником и переключается мышью или клавишами управления курсором. При вводе (редактировании) условия отбора черный прямоугольник сменяется мигающей вертикальной линией.

При выполнении запроса, показанного на рис. 12.10, из таблицы `Personnel` отбираются записи, для которых значение оклада (поле `P_Salary`) больше или равно `3500`. Кроме поля оклада, в результат запроса включаются поля имени (`P_Name`) и должности (`P_Position`).

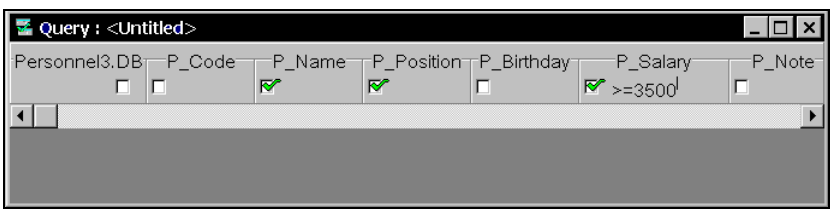


Рис. 12.10. Отбор записей по величине оклада

В приведенном примере поле `P_Salary` используется для отбора записей и одновременно включается в результат запроса. В общем случае включать поле, для которого задано условие отбора записей, в результат запроса не обязательно. Таким образом, записи можно отбирать на основании одних полей, а в результат запроса включать другие.

При выполнении запроса, показанного на рис. 12.11, в набор данных, как и в предыдущем примере, отбираются записи, для которых значение поля `P_Salary` больше или равно `3500`. Но в результат запроса включается только поле `Name`.

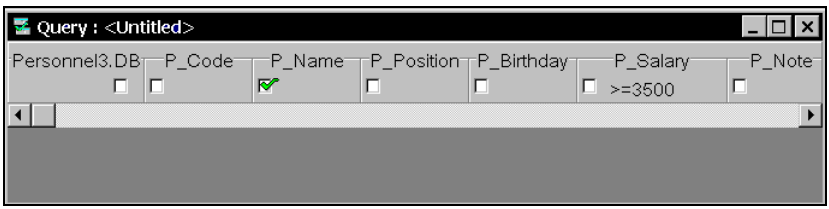


Рис. 12.11. Отбор записей по полю, не входящему в результат запроса

Условие отбора может быть более сложным, чем просто сравнение, и содержать операции логического умножения (`AND`) и сложения (`OR`).

Для объединения отдельных условий с помощью операции логического умножения нужно перечислить эти условия в одной строке через запятую. Например, при выполнении запроса, показанного на рис. 12.12, будут отобраны записи, для которых значение поля `P_Salary` находится в интервале `3500 .. 5000`.

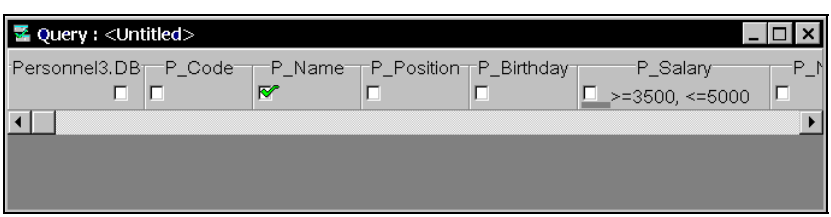


Рис. 12.12. Использование в условии отбора операции логического умножения

Для соединения отдельных условий операцией логического сложения нужно перечислить эти условия в разных строках. Так, при выполнении запроса, показанного на рис. 12.13, в набор попадут записи, для которых поле `P_Position` содержит значения `Водитель` или `Секретарь`. В результат запроса не включаются поля кода и примечания.

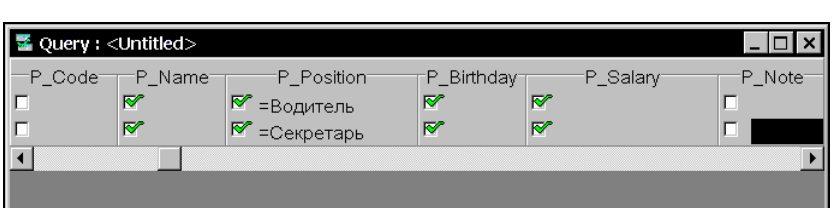


Рис. 12.13. Использование в условии отбора операции логического сложения

Добавление к запросу новой строки выполняется переводом текстового курсора вниз при нахождении его на последней строке (с помощью клавиш управления курсором). При этом курсор может находиться в любом поле запроса.

Замечание

Отметки для каждого поля во всех строках запроса должны быть одинаковыми. Например, если в приведенном на рис. 12.13 запросе во второй строке убрать отметку для поля `P_Birthday`, то при попытке выполнить запрос будет выдано сообщение об ошибке.

Удаление строки запроса выполняется одновременным нажатием клавиш <Ctrl> и <Delete>. Предварительно текстовый курсор должен быть установлен в любом поле удаляемой строки.

При отборе записей по нескольким полям условие отбора задается отдельно для каждого поля. Условия связываются операцией логического умножения — таким образом, в результат запроса попадают записи, для полей которых выполняются все условия.

Например, при выполнении запроса, показанного на рис. 12.14, будут отобраны записи, для которых значение поля `P_Position` равно `Водитель`, а значение поля `P_Salary` больше или равно `3000`. В результат запроса попадают поля `P_Name`, `P_Position` и `P_Salary`.

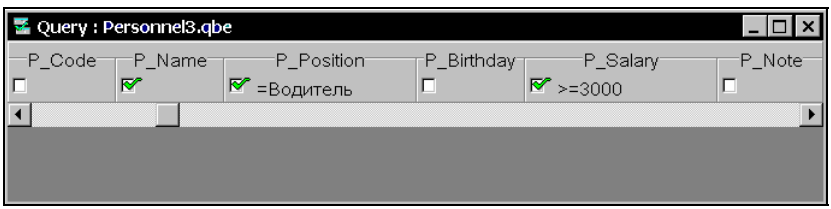


Рис. 12.14. Отбор записей по нескольким полям

Редактирование записей

При редактировании записей необходимо определить:

- условия отбора записей;
- новые значения изменяемых полей.

Условия отбора записей формируются, как рассмотрено в предыдущем разделе. Для изменяемых полей следует задать новые значения, перед которыми указывается ключевое слово `CHANGETO`.

Замечание

При вставке и удалении записей флажки всех полей должны быть пустыми.

При выполнении запроса, показанного на рис. 12.15, для всех записей, имеющих значение `Секретарь` поля `P_Position`, значение поля `P_Salary` будет установлено в `4700`.

Рис. 12.15. Редактирование записей

Аналогично, в одном запросе можно отобрать записи на основании более сложного условия, а также одновременно изменить значения нескольких полей.

Вставка и удаление записей

При вставке и удалении записей справа от флажков для полей таблицы указываются значения соответствующих полей. При вставке записей нужно с помощью контекстного меню в поле имени таблицы выбрать в качестве типа запроса `INSERT`.

При выполнении запроса, показанного на рис. 12.16, в таблицу `Personnel3` добавляется новая запись, имеющая следующие значения полей: `P_Name` — `Семенова Е.Н.`, `P_Position` — `Менеджер`, `P_Birthday` — `12.03.67`, `P_Salary` — `4000`.

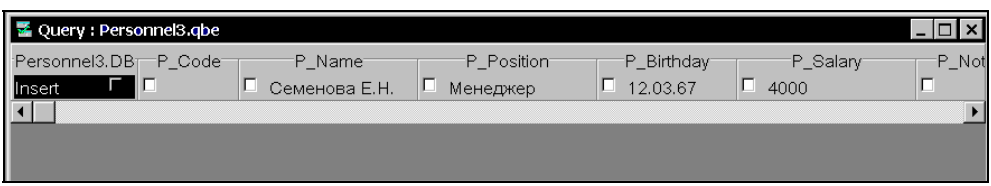


Рис. 12.16. Вставка записи

Замечание

При вставке и удалении записей флажки всех полей должны быть пустыми.

Значения отдельных полей можно не задавать, однако если поле требует обязательного ввода, то при выполнении запроса возникнет ошибка.

В результате отработки одного запроса в таблицу можно добавить сразу несколько записей. При этом значения полей каждой новой записи вводятся отдельной строкой запроса. Например, при выполнении запроса, показанного на рис. 12.17, в таблицу `Personnel3` добавляются две записи, значения полей первой: `P_Name` — `Семенова Е.Н.`, `P_Position` — `Менеджер`, `P_Birthday` — `12.03.67`,

P_Salary — 4000, а второй: P_Name — Петров В.Э., P_Salary — 4500, P_Position — Менеджер. Поля P_Code обеих записей являются автоинкрементными и заполняются автоматически. Поле P_Birthday второй записи и поля P_Note обеих записей остаются пустыми.

Рис. 12.17. Одновременная вставка двух записей

Замечание

При вставке нескольких записей вид запроса INSERT в поле имени таблицы нужно указывать для каждой записи.

При удалении записей с помощью контекстного меню в поле имени таблицы следует выбрать вид запроса DELETE. Справа от флажков полей указываются значения полей удаляемых записей.

Например, запрос, показанный на рис. 12.18, удаляет запись, имеющую значения полей: P_Name — Семенова Е.Н., P_Position — Менеджер, P_BirthDay — 12.03.67, P_Salary — 4000.

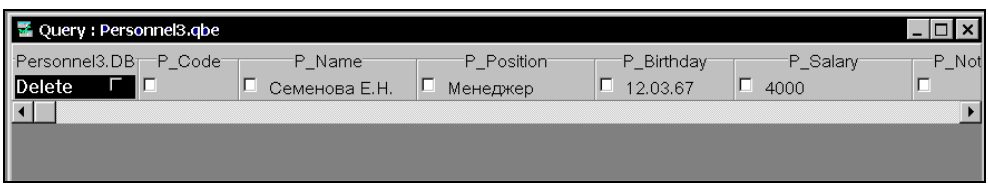


Рис. 12.18. Удаление записи

При выполнении запроса на удаление записей возможны следующие ситуации:

- в таблице нет записи с указанными значениями полей — удаление записи не происходит;
- в таблице есть несколько записей с указанными значениями полей — удаляются все такие записи.

Если значение поля не задано, то это поле не влияет на отбор записей для удаления. В запросе, показанном на рис. 12.19, такими полями являются, например, P_Name и P_Birthday. В результате его выполнения из таблицы Personnel3 удаляются все записи, соответствующие менеджерам с окладом 4000.

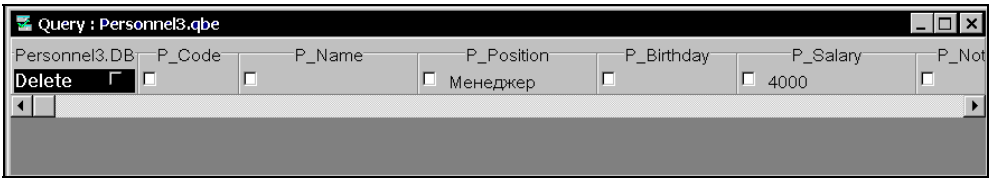


Рис. 12.19. Удаление записи

Можно задать более сложные критерии отбора записей для удаления, например, определив вторую строку условия (рис. 12.20). При выполнении этого запроса из таблицы `Personnel3` удалятся все записи, соответствующие менеджерам и водителям с окладом 4000.

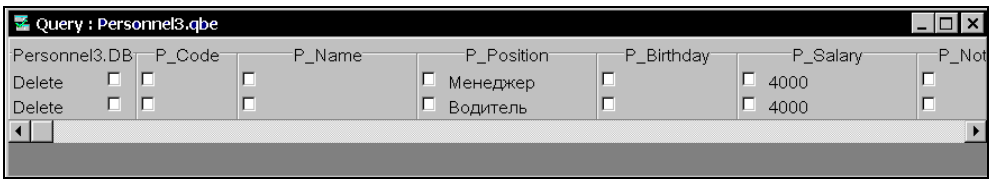


Рис. 12.20. Удаление записей

В качестве результата запроса, выполняющего добавление или удаление записей, возвращается совокупность добавленных или удаленных записей, соответствующих заданным в запросе условиям.

Связывание таблиц

Можно построить запрос по образцу не только для одиночных таблиц, как это рассмотрено выше, но и для связанных таблиц. Для этого требуется:

- добавить к запросу новую таблицу;
- связать между собой две таблицы.

Добавление таблицы выполняется нажатием кнопки **Add Table** панели инструментов (на кнопке изображена таблица и знак +) и выбором в открывшемся окне **Select File** главного файла таблицы.

Для связывания таблицы нужно нажать кнопку **Join Tables** панели инструментов (на кнопке изображены две таблицы), при этом кнопка останется в нажатом состоянии, а к указателю мыши добавляется изображение двух таблиц. Затем щелчком на полях связи обеих таблиц между ними устанавливается соединение, получающее имя `join1`. Имя соединения отображается в полях связи. Так же можно связать между собой три и более таблиц.

Для связанных таблиц операции отбора, редактирования, вставки и удаления записей выполняются так же, как и для одиночных таблиц. Рассмотрим в качестве примера технику отбора записей. Запрос, показанный на рис. 12.21, выво-

дит из подчиненной таблицы `Cards` записи с движением товара, который указан в главной таблице `Store` склада. Отбор записей осуществляется по полю `S_Name` названия товара, в результирующий набор данных включаются поля `C_Move` количества поступившего или убывшего товара и `C_Date` даты выполнения операции. При выполнении запроса будут выведены записи, соответствующие покупке и продаже моркови.

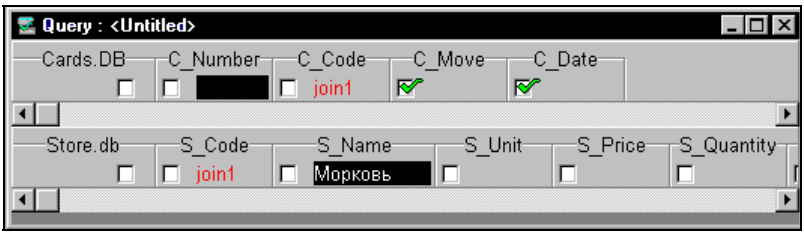


Рис. 12.21. Связывание таблиц в программе Database Desktop

Если требуется, например, вывести записи за определенный период, то для поля `C_Date` дополнительно указываются нижнее и верхнее значения даты.

Для удаления таблицы из запроса нужно нажать кнопку **Remove Tables** панели инструментов (с изображением таблицы и знака –), в открывшемся одноименном окне выбрать имя таблицы и нажать кнопку **OK**.

Замечание

После удаления одной из таблиц в поле связи другой таблицы остается имя связи.

Для удаления связи нужно поочередно установить курсор в поля связи обеих таблиц, где отображается имя связи, например, `join1`, и нажать клавишу <Delete>.

Программа SQL Builder

Программа SQL Builder предназначена для упрощения создания SQL-запросов. С помощью этой программы разработчику достаточно удобно конструировать запросы, которые сохраняются в виде текстового файла с расширением sql.

Программа SQL Builder (рис. 12.22) вызывается выбором команды **SQL Builder** контекстного меню компонента `Query`. Отметим, что во время работы программы SQL Builder нельзя перейти в другие окна Delphi, такой переход возможен только по завершении работы с ней.

В верхней части окна программы SQL Builder размещаются таблицы, выбранные для построения запроса, а в нижней части — многостраничный блокнот.

Для выполнения запроса нужно вызвать команду **Query\Run Query** программы SQL Builder или нажать кнопку панели инструментов. Перед выполнением

запроса производится проверка его правильности. На рис. 12.23 показано окно с результатами выполнения запроса.

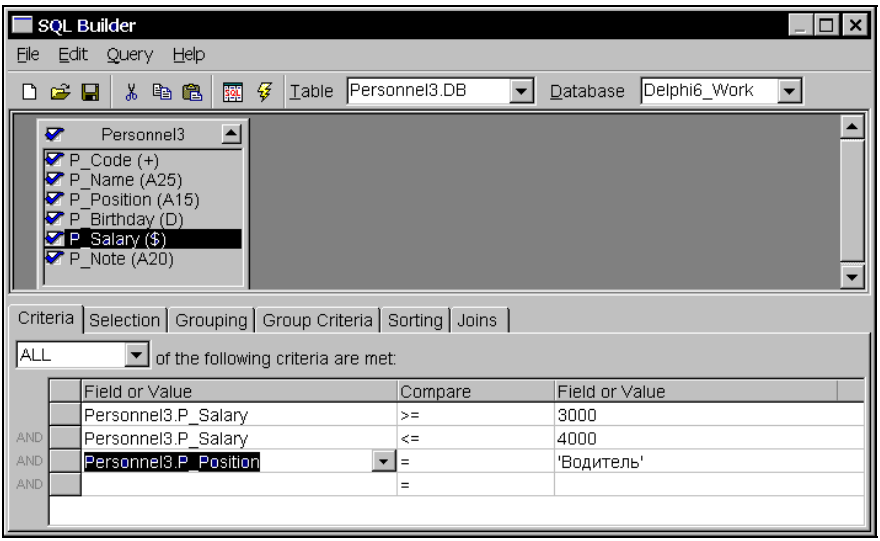


Рис. 12.22. Диалоговое окно программы SQL Builder

Рис. 12.23. Диалоговое окно с результатами выполнения запроса

Для получения текста, соответствующего визуальному запросу, нужно выполнить команду **Query\Show SQL** или нажать на панели инструментов кнопку с буквами **SQL**. При отсутствии ошибок в конструкции запроса автоматически открывается окно **SQL Query Text Entry** (рис. 12.24), в котором содержится текст запроса на языке SQL. В этом окне можно выполнить такие действия, как сохранение, загрузка или выполнение запроса.

При наличии ошибок названное окно не появляется, а выдается сообщение об ошибке.

Сохранение запроса в виде текстового файла с расширением sql выполняется командой **File\Export to file**. Открытие запроса осуществляется командой **File\Import from file**.

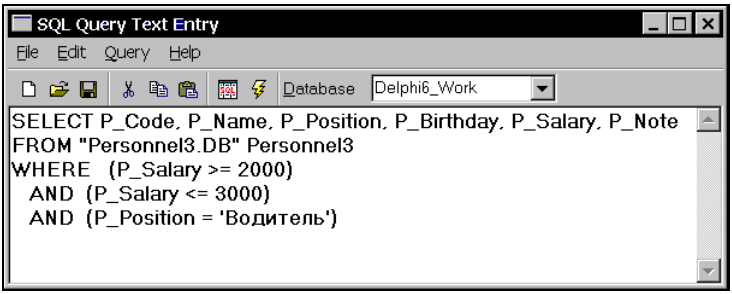


Рис. 12.24. Окно **SQL Query Text Entry**

Замечание

Если сохраненный запрос содержит ошибки, то при попытке его открыть выдается соответствующее сообщение, и запрос не открывается. Поэтому перед сохранением запроса в файле .sql следует проверять его правильность, например, путем его выполнения.

Сохранить и открыть запрос можно также с помощью кнопок панели инструментов, на которых изображены дискета и открытая папка соответственно. Отметим, что всплывающие подсказки этих кнопок не совпадают с названиями соответствующих команд меню. Для кнопки сохранения — это **Save the current query** (Сохранить текущий запрос), а для кнопки открытия — **Open an existing SQL Query** (Открыть существующий SQL-запрос).

Чтобы добавить в окно программы новую таблицу, нужно в поле списка **Database** указать расположение базы данных, выбрав псевдоним в раскрывающемся списке или введя путь к каталогу с файлами БД вручную. После этого в списке **Table** выбирается нужная таблица, которая автоматически добавляется в верхнюю часть окна. Добавляемые таблицы могут располагаться и в разных каталогах.

Таблица представляется изображением, похожим на комбинированный список. В верхней части этого списка находятся псевдоним (Alias) таблицы, флажок и стрелка. Стрелка позволяет свернуть или развернуть список, а флажок служит для управления включением всех полей таблицы в результат запроса или выключением из него.

Замечание

Псевдоним таблицы представляет собой имя, используемое в тексте SQL-запроса для замены имени таблицы, и не имеет ничего общего с псевдонимом, указывающим расположение БД и конкретной таблицы.

Для смены псевдонима таблицы нужно вызвать команду **Edit Table Alias** контекстного меню этой таблицы. При этом в псевдониме отображается текстовый курсор, что показывает его готовность к изменению. После нажатия клавиши <Enter> новый псевдоним вступает в действие.

Остальную часть списка занимают строки с информацией о полях таблицы — имя поля, его тип и размер. Размер указывается только для тех полей, для которых он не устанавливается автоматически на основании типа поля, например, для поля строкового типа. Слева от названия поля находится флажок, управляющий включением этого поля в результат запроса. Если флажок установлен (стоит галочка), то поле включается в результат запроса.

Удаление таблицы из окна программы выполняется командой **Remove Table** контекстного меню этой таблицы. Можно также выделить таблицу, щелкнув на ее изображении, при этом вокруг ее псевдонима отобразится пунктирный прямоугольник. После выделения таблица удаляется нажатием клавиши <Delete>.

Программа SQL Builder позволяет достаточно просто и удобно выполнить связывание (соединение) таблиц. Для этого нужно выбрать мышью поле в одной таблице и перетащить его в используемое для связи поле другой таблицы. После отпускания кнопки мыши между таблицами устанавливается связь, что отображается линией, соединяющей эти таблицы (рис. 12.25). В обеих таблицах поля, используемые для связи, отображаются в списке полей первыми (верхними) и выделяются жирным шрифтом. В отличие от ряда других программ, например, Microsoft Access, эта линия соединяет названия таблиц, а не используемые для связи поля обеих таблиц.

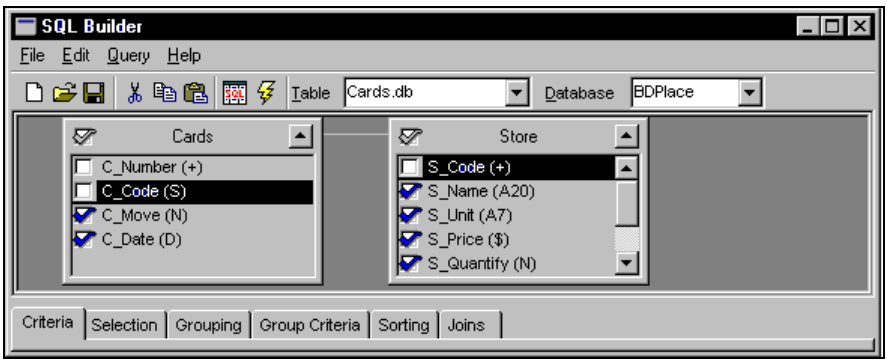


Рис. 12.25. Связывание таблиц

Напомним, что в главной таблице поле связи должно быть ключевым, а в подчиненной таблице — индексным. Поля связи должны иметь одинаковые или совместимые типы, в противном случае выдается сообщение об ошибке несоответствия типов.

Для удаления связи нужно щелкнуть правой кнопкой мыши на линии, обозначающей связь между таблицами, вызвав контекстное меню, и выполнить его единственную команду **Delete Join**. После подтверждения этой операции связь удаляется.

В нижней части окна программы SQL Builder находится блокнот, который автоматически появляется при добавлении к окну программы первой таблицы.

С помощью блокнота можно включать в запрос условия отбора записей или указывать направления их сортировки.

Страница **Criteria** позволяет задать условия отбора записей. Каждое условие вводится в отдельной строке и состоит из двух имен полей или значений, разделенных операцией сравнения. Имена полей выбираются в списке или перетаскиваются мышью из соответствующей таблицы в верхней части окна. Операция сравнения выбирается в списке. Если условие введено с ошибкой, то выдается соответствующее сообщение, а само условие в запрос не включается и отображается красным цветом.

Удаление строки с условием выполняется командой **Delete Row** контекстного меню удаляемой строки. Отметим, что аналогично можно удалить строку и из списка других страниц блокнота.

Список **of the following criteria are met** (из перечисленных условий выполнены) позволяет задать логические операции, которые связывают строки с отдельными условиями. В левой части строк с условиями отображаются названия логических операций. По умолчанию список содержит значение **ALL** (ВСЕ), что соответствует операции логического умножения (см. рис. 12.22) и означает, что все перечисленные условия выполнены. При этом в левой части строк с условиями, кроме первой, отображается операция `AND`. Кроме значения **ALL**, в списке можно также выбрать:

- **ANY** (операция логического сложения `OR`);
- **NONE** (инверсия значения **ANY**);
- **NOT ALL** (инверсия значения **ALL**).

Страница **Selection** позволяет изменить названия заголовков столбцов, используемых для отображения значений полей в результирующем наборе, а также включить в запрос статистические функции по полям.

В правой части списка указываются поля, включенные в результат запроса — им соответствуют установленные флажки в изображениях таблиц. В левой части списка для каждого поля помещается название соответствующего столбца, которое по умолчанию совпадает с именем поля (без указания имени таблицы).

В запросе, показанном на рис. 12.26, английские названия столбцов трех первых полей заменены на русские.

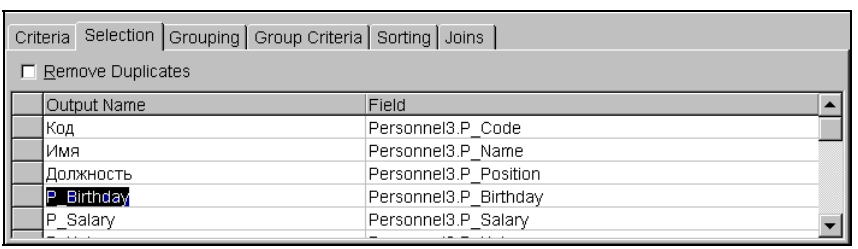


Рис. 12.26. Задание названий для столбцов результирующего набора данных

Результирующий набор данных (с русскими названиями столбцов) можно видеть на рис. 12.27.

Рис. 12.27. Вид результирующего набора данных с измененными названиями столбцов

Добавление к запросу статистической функции выполняется с помощью команды **Summary** (Итог) контекстного меню страницы. В результате появляются списки, в которых можно выбрать функцию (например, `SUM`) и поле.

Флажок **Remove Duplicates** (Исключить повторы) позволяет убрать из результирующего набора одинаковые записи. По умолчанию он снят, и в результат попадают все записи (в том числе и совпадающие), которые отвечают заданным условиям отбора.

Страница **Grouping** позволяет сгруппировать записи по полям, при этом список **Output Fields** содержит все поля, которые можно использовать для группирования записей. В список **Grouped On** отбираются имена полей, по которым выполняется группирование (рис. 12.28). Перемещение полей первого списка во второй и удаление полей из второго списка выполняются с помощью кнопок **Add** и **Remove**, которые становятся активными при выборе поля (полей) в соответствующем списке. Переместить поле между списками можно также двойным щелчком на его имени.

Доступными являются поля, флажок которых в изображении таблицы установлен. Имя поля состоит из псевдонима (имени) таблицы и названия столбца для этого поля, заданного на странице **Selection**.

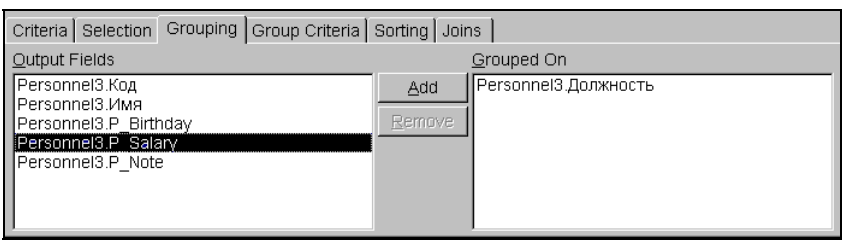


Рис. 12.28. Задание полей для группирования записей

Страница **Group Criteria** содержит условия, используемые для группирования записей. Условия группирования вводятся аналогично условиям отбора записей, задаваемым на странице **Criteria**.

Страница **Sorting** определяет порядок сортировки записей. Список **Sorted By** содержит имена полей, по которым выполняется сортировка (рис. 12.29). Значение **Sorted Order (**Порядок следования полей) определяет последовательность сортировки: сначала записи упорядочиваются по значениям поля, указанного в списке первым (верхним), затем записи, имеющие одинаковое значение первого поля, упорядочиваются по второму полю и т. д. Для изменения порядка расположения полей, по которым выполняется сортировка, предназначены кнопки в виде черных треугольников, расположенные над именами полей. Кнопка с треугольником, направленным вниз, перемещает выделенное поле (поля) на одну строку вниз, кнопка с треугольником, направленным вверх, — на одну строку вверх.

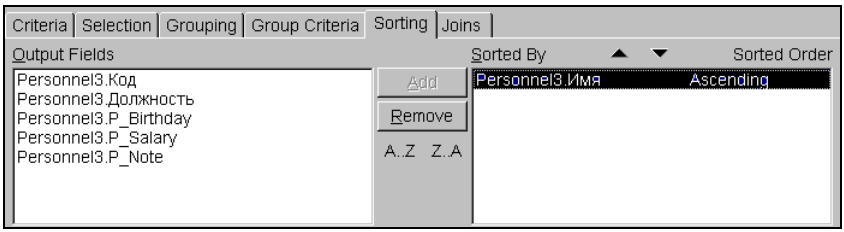


Рис. 12.29. Задание полей для сортировки записей

По умолчанию сортировка выполняется в порядке возрастания значения поля, на что указывает признак **Ascending** справа от имени поля. Для задания обратного порядка сортировки по полю нужно его выделить и нажать кнопку **Z..A**, при этом вместо **Ascending** устанавливается признак **Descending**. Нажатие кнопки **A..Z** возвращает прежний порядок сортировки.

Список **Output Fields** содержит поля, которые можно использовать для сортировки записей. Обозначение полей и способы их перемещения между списками не отличаются от обозначения полей и способов их перемещения для страницы **Grouping**.

Страница **Joins** позволяет устанавливать связи (соединения) между таблицами (рис. 12.30).

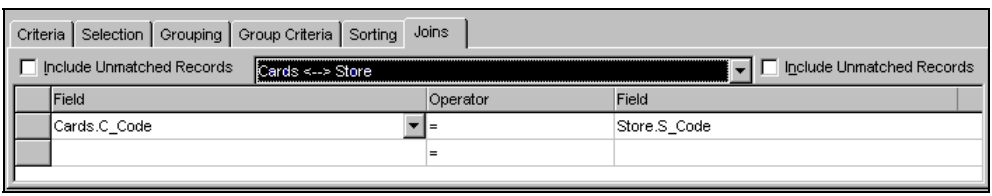


Рис. 12.30. Задание связи между таблицами

Действия, выполняемые на этой странице, частично дублируют описанные ранее действия по связыванию таблиц и разрыве связи между ними. Кроме того, здесь можно задать дополнительные условия для связи и выбрать оператор для сравнения значений полей связываемых таблиц, например, >, < или = (по умолчанию).

Программа SQL Explorer

Программа SQL Explorer представляет собой аналог Проводника Windows, с помощью которого можно просматривать и редактировать базу данных. В зависимости от версии эта программа имеет различные возможности и даже разные названия, например, SQL Explorer, Explorer или Database Explorer. Для запуска SQL Explorer нужно выполнить команду **Database\Explorer** меню Delphi или запустить файл dbexplor.exe, находящийся в каталоге BIN главного каталога Delphi. Программу также можно вызвать через главное меню Windows выбором пункта **Programs\Borland Delphi 7\SQL Explorer**.

Интерфейс программы SQL Explorer практически не отличается от интерфейса программы BDE Administrator. В левой части окна (рис. 12.31) в древовидной структуре выбирается объект, после чего свойства этого объекта становятся доступными для просмотра и редактирования в правой части окна.

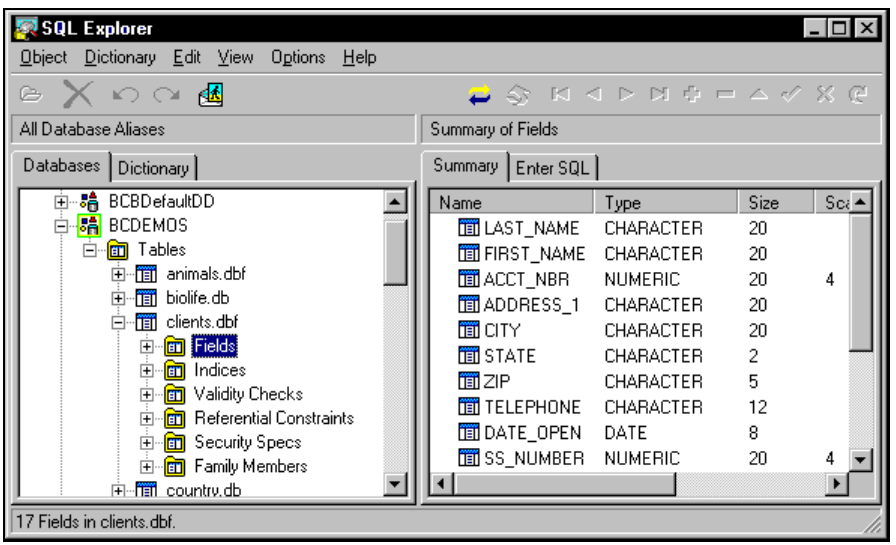


Рис. 12.31. Диалоговое окно программы SQL Explorer

SQL Explorer позволяет:

- выполнять действия с псевдонимами;
- просматривать структуру БД и таблиц;

- просматривать и редактировать записи таблиц;
- редактировать и выполнять SQL-запросы;
- выполнять операции со словарями данных.

Операции с псевдонимами не отличаются от аналогичных операций в программе BDE Administrator.

Для просмотра структуры БД (таблицы) в левой панели окна выбирается нужная БД (таблица) — на рис. 12.31 это локальная таблица clients.dbf, расположение которой указывает псевдоним `BCDEMOS`. Для этой таблицы можно просмотреть ее объекты:

- поля (**Fields**);
- индексы (**Indices**);
- ограничения на значения полей (**Validity Checks**);
- ограничения ссылочной целостности (**Referential Constraints**);
- члены семейства (**Family Members**)
- права доступа (**Security Spects**).

Список доступных объектов базы данных зависит от особенностей ее или таблицы. Например, для удаленной БД InterBase в состав ее структуры включаются триггеры и генераторы.

Если для БД или таблицы установлен режим ограничения доступа, то при попытке открыть ее (получить к ней доступ) запрашиваются имя пользователя и его пароль.

Для просмотра и редактирования содержимого таблицы нужно выделить ее название и в правой панели программы SQL Explorer выбрать вкладку **Data**. Записи таблицы отображаются в сетке, не отличающейся от компонента `DBGrid`. Для навигации по записям и их редактирования можно использовать навигационный интерфейс, расположенный в панели инструментов и аналогичный компоненту `DBNavigator`.

Для просмотра содержимого данных формата BLOB (например, рисунков) следует нажать кнопку **Explorer BLOB**, которая расположена в панели инструментов. Объект BLOB выводится в отдельном окне.

Редактирование и запуск SQL-запросов осуществляется в текстовом редакторе, расположенном на вкладке **Enter SQL**. Выполнение запроса происходит при нажатии кнопки, размещенной справа от текстового редактора. Результаты работы запроса выводятся в окне, расположенном под окном текстового редактора. Переключение между запросами выполняется при нажатии кнопок с изображениями синих треугольников: нажатие верхней кнопки вызывает переход к предыдущему запросу, а нижней — к следующему.

Для операций со словарями данных нужно выбрать вкладку **Dictionary** (Словарь) в левой части окна программы SQL Explorer (см. рис. 12.31). При этом структу-

ра выбранного словаря данных отобразится в виде иерархического дерева. Со словарем можно выполнить такие операции, как:

- выбор нового словаря (**Select**);
- включение словаря в Хранилище объектов (**Register**);
- исключение словаря из Хранилища объектов (**Unregister**);
- создание нового словаря (**New**);
- удаление выбранного словаря (**Delete**);
- вставка в словарь данных из БД (**Import from Dadabase**).

В скобках указаны команды меню вкладки **Dictionary**.

Словарь данных включает в себя описание структур БД (объект **Database**) и атрибуты (объект **Attribute Sets**).

Программа Data Pump

С помощью этой программы можно выполнить перенос данных между таблицами БД. Под *переносом* понимается не перемещение, а копирование данных из таблиц исходной БД (источника) в таблицы другой БД (приемника). В результате переноса в базе-приемнике автоматически создаются таблицы с именами копируемых таблиц источника, и в них дублируется информация этих таблиц.

Для вызова программы запускается файл datapump.exe, находящийся в одном каталоге с процессором баз данных. Программу также можно вызвать через главное меню Windows командой **Programs\Borland Delphi 7\Datapump**.

Работа с программой напоминает работу с программой-мастером (Wizard) и осуществляется по шагам. В процессе выполнения действий открываются окна в следующей последовательности:

- **Select Source Alias** (Выбор псевдонима источника);
- **Select Target Alias** (Выбор псевдонима приемника);
- **Select Tables to Move** (Выбор таблиц для перемещения);
- **Inspect or Modify Items** (Проверка или изменение элементов).

Кроме того, на четвертом шаге при изменении (просмотре) структуры таблиц открываются дополнительные окна.

В каждом окне выбирается или вводится соответствующая информация. Переход к следующему или возврат к предыдущему шагам (окнам) выполняется нажатием кнопок **Next** и **Back** соответственно. Нажатие кнопки **Exit** прекращает процесс подготовки переноса данных и завершает работу программы.

Выбор источника заключается в задании псевдонима БД или в непосредственном указании диска и каталога ее размещения. Для базы-приемника также необходимо задать псевдоним. Размещение источника и приемника не могут совпадать, даже если изменить имена создаваемых таблиц так, чтобы они не совпадали с именами исходных таблиц.

Выбор таблиц заключается в указании имен главных файлов таблиц, для которых будет выполнен перенос данных. Одной операцией переноса можно выполнить копирование данных сразу для нескольких таблиц БД.

На последнем шаге выполняется управление полями выбранных таблиц. В окне, показанном на рис. 12.32, можно просмотреть или изменить структуру перемещаемых таблиц.

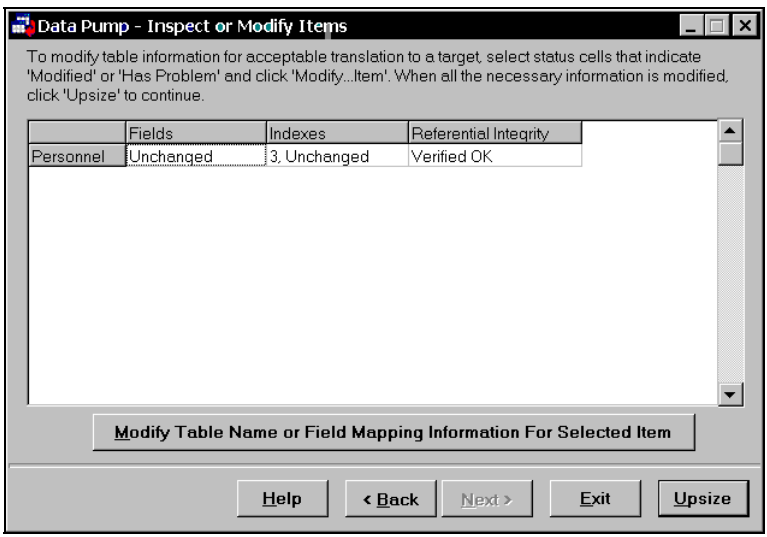


Рис. 12.32. Управление полями таблиц при переносе БД

Изменение структуры относится не к исходным таблицам, а к вновь создаваемым таблицам базы-приемника. Для новых таблиц можно изменить следующие элементы структуры:

- **Fields** — описания полей и имя таблицы;
- **Indexes** — индексы;
- **Referential Integrity** — ограничения ссылочной целостности.

Для изменения элемента таблицы нужно его выделить, тогда вокруг него отобразится пунктирный прямоугольник (на рис. 12.32 выбран элемент **Fields** таблицы `Album`, что отображается едва заметным пунктирным выделением ячейки с текстом `Unchanged`). Нажатие кнопки **Modify Table Name or Field Mapping Information For Selected Item** (Изменить имя таблицы или информацию о полях для выбранного элемента) открывает окно изменения полей **Modify Fields** (рис. 12.33).

Список **Source Field Names** содержит имена всех полей исходной таблицы, при этом в полях ввода **Table Name** выводятся имена исходной (столбец **Source**) и создаваемой (столбец **Target**) таблиц соответственно. По умолчанию эти имена

совпадают, но имя таблицы-приемника можно изменить (поле ввода **Table Name** для таблицы-источника заблокировано).

Поля ввода (ввод заблокирован), расположенные в столбце **Source**, отображают характеристики поля таблицы, выбранного в списке полей источника. Поля ввода и списки, расположенные в столбце **Target**, содержат те же характеристики для создаваемой таблицы, там же расположено поле ввода **Field Name**, указывающее имя поля таблицы-приемника. По умолчанию описания полей таблицы (имена и характеристики) приемника совпадают с описаниями полей таблицы источника, однако при необходимости их можно изменять. Поле таблицы имеет следующие характеристики:

- **Field Type** (тип);
- **Is Required** (требование обязательного ввода значения);
- **Min Value** (минимальное значение);
- **Max Value** (максимальное значение);
- **Default** (значение по умолчанию).

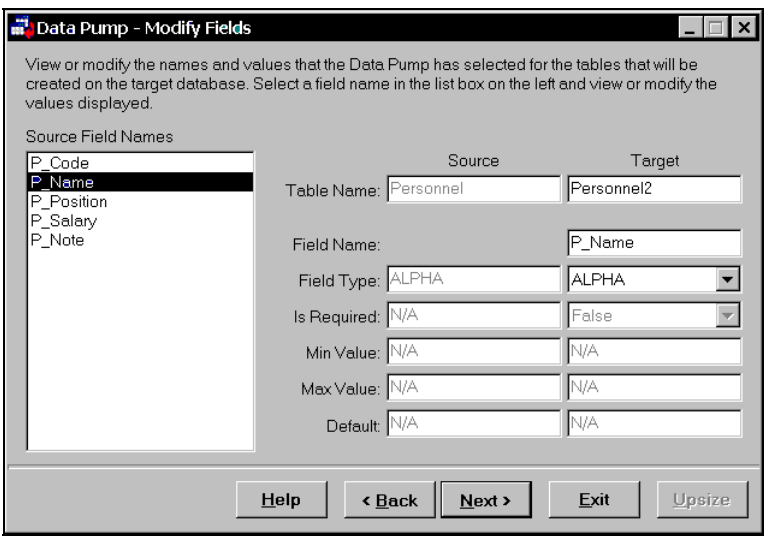


Рис. 12.33. Изменение полей и имени таблицы

Нажатие кнопки **Next** (**Back**) подтверждает (отменяет) сделанные изменения и открывает следующее (предыдущее) окно.

Изменить индексы можно только для таблиц, у которых они есть, в этом случае в столбце **Indexes** таблицы выводится число индексов. Так, на рис. 12.32 исходная таблица `Personnel` имеет 3 индекса, включая индекс, автоматически созданный по ключу. При нажатии кнопки модификации индексов вызывается окно **Modify an Index on Table** (рис. 12.34).

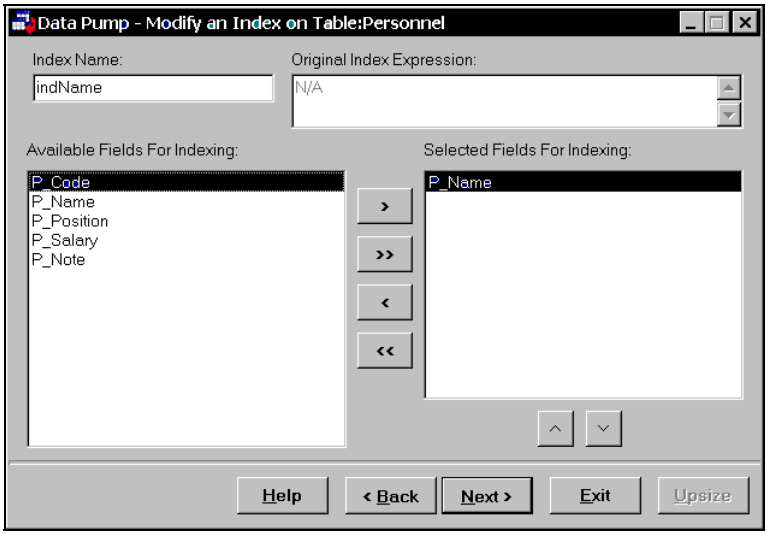


Рис. 12.34. Окно изменения индексов

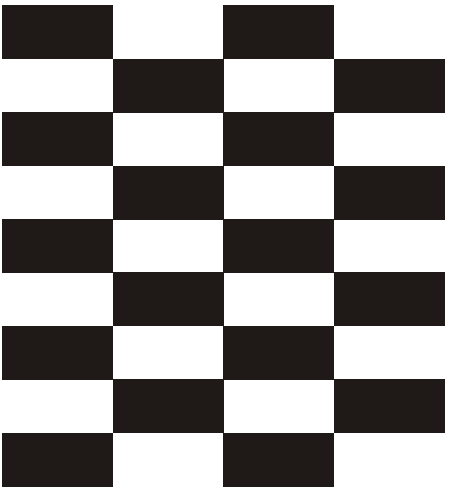
В этом окне можно просмотреть определение индексов и при необходимости изменить их, задав для индекса новые:

- ❒ имя;
- ❒ состав полей;
- ❒ порядок полей.

Напомним, что ключ таблицы Paradox не имеет имени.

Нажатие кнопки **Next** (**Back**) отображает в окне определений следующий (предыдущий) индекс таблицы. После вывода определения последнего индекса осуществляется возврат в окно **Inspect or Modify Items**.

Копирование данных запускается нажатием кнопки **Upsize** (Перенос). В процессе копирования отображается индикатор, показывающий ход переноса данных, а по окончании копирования выводится отчет о его результатах. Если в базе-приемнике уже существует таблица с заданным именем, то выдается запрос на подтверждение операции.



ЧАСТЬ III

Удаленные базы данных

Глава 13

Введение в работу с удаленными базами данных

Ранее были рассмотрены локальные базы данных, когда и БД, и взаимодействующее с ней приложение располагаются на одном компьютере. В данной главе мы рассмотрим некоторые особенности работы с удаленными БД, используемыми в сети, где приложение и БД располагаются на разных компьютерах.

Основные понятия

В принципе локальную БД тоже можно использовать для коллективного доступа, т. е. в сетевом варианте. В этом случае файлы базы данных и приложение для работы с ней располагаются на сервере сети. Пользователь запускает со своего компьютера находящееся на сервере приложение, при этом у него запускается копия приложения. Можно установить приложение и непосредственно на компьютере пользователя, в этом случае приложению должно быть известно местонахождение общей БД, заданное, например, через псевдоним. Подобный сетевой вариант использования локальной БД соответствует архитектуре "файл-сервер".

Достоинствами сетевой архитектуры "файл-сервер" являются простота разработки и эксплуатации БД и приложения. Разработчик фактически создает локальную БД и приложение, которые затем просто используются в сетевом варианте. При этом не требуется дополнительное программное обеспечение для организации работы с БД.

Однако архитектуре "файл-сервер" свойственны и существенные недостатки.

- Для работы с данными используется навигационный способ доступа, при этом по сети циркулируют большие объемы данных. В результате сеть оказывается перегруженной, что является причиной ее низкого быстродействия и плохой производительности при работе с БД.
- Требуется синхронизация работы отдельных пользователей, связанная с блокировкой в таблицах тех записей, которые редактирует другой пользователь.
- Приложения не только обрабатывают данные, но и управляют самой базой данных. В связи с тем, что управление БД осуществляется с разных компьютеров, затрудняются управление доступом, соблюдение конфиденциальности и поддержание целостности БД.

Из-за этих недостатков архитектура "файл-сервер", как правило, используется в небольших сетях. Для сетей с большим количеством пользователей предпочтительным вариантом (а порой и единственным возможным) является архитектура "клиент-сервер".

Архитектура "клиент-сервер"

В сетевой архитектуре "клиент-сервер" БД размещается на компьютере-сервере сети (сервере или удаленном сервере) и называется также удаленной БД. Приложение, осуществляющее работу с этой БД, находится на компьютере пользователя. Приложение пользователя является *клиентом*, его также называют *приложением-клиентом.*

Клиент и сервер взаимодействуют следующим образом. Клиент формирует и отсылает запрос (SQL-запрос) серверу, на котором размещена БД. Сервер выполняет запрос и выдает клиенту в качестве результатов требуемые данные.

Таким образом, в архитектуре "клиент-сервер" клиент посылает запрос и получает только те данные, которые ему действительно нужны. Вся обработка запроса выполняется на удаленном сервере. К достоинствам такой архитектуры относятся следующие факторы.

- Для работы с данными используется реляционный способ доступа, что снижает нагрузку на сеть.
- Приложения не управляют напрямую базой, управлением занимается только сервер. В связи с этим можно обеспечить высокую степень защиты данных.
- В приложении отсутствует код, связанный с управлением БД, поэтому приложения упрощаются.

Отметим, что сервером называют не только компьютер, но и специальную программу, которая управляет БД. Так как в основе организации обмена данными между клиентом и сервером лежит язык SQL, такую программу еще называют *SQL-сервером*, а БД — *базой данных SQL.* В широком смысле слова под сервером понимают компьютер, программу и саму базу данных. SQL-серверами являются промышленные СУБД, такие как InterBase, Oracle, Informix, SyBase, DB2, Microsoft SQL Server и др. Каждый из серверов имеет свои преимущества и особенности, связанные, например, со структурой БД и реализацией языка SQL, которые необходимо учитывать при разработке приложения. Далее мы будем понимать под *сервером* программу (т. е. SQL-сервер), а установленную на компьютере-сервере базу данных будем называть *удаленной БД.*

При работе в архитектуре "клиент-сервер" приложение должно:

- устанавливать соединение с сервером и завершать его;
- формировать и отсылать запрос серверу, получая от него результаты выполнения запроса;
- обрабатывать полученные данные.

При этом обработка данных не имеет принципиальных отличий по сравнению с обработкой данных в локальных БД.

Сервер и удаленная БД

Удаленная БД, как и локальная, представляет собой совокупность взаимосвязанных таблиц. Однако данные этих таблиц, как правило, содержатся в одном общем файле. Как и в случае с локальной БД, для таблиц удаленной БД могут устанавливаться связи (отношения), ограничения ссылочной целостности, ограничения на значения столбцов и т. д.

Замечание

Для удаленных БД поле называется столбцом.

Для управления БД сервер использует:

- триггеры;
- генераторы;
- хранимые процедуры;
- функции, определяемые пользователем;
- механизм транзакций;
- механизм кэшированных изменений;
- механизм событий.

Многие из перечисленных элементов обеспечиваются возможностями языка SQL-сервера, в котором, по сравнению с локальной версией, имеются существенные особенности, рассматриваемые ниже.

Средства работы с удаленными БД

Система Delphi обеспечивает разработку приложений для различных серверов, предоставляя для этого соответствующие средства. Отметим, что многие описанные ранее принципы разработки приложений и средства для работы с локальными БД относятся и к работе с удаленными БД. В частности, для разработки приложений используются такие компоненты, как источник данных `DataSource`, наборы данных `Table`, `ADOTable`, `SQLTable`, `IBTable`, `Query`, `ADOQuery` и `SQLQuery`, сетка `DBGrid` и др.

Замечание

Для реализации реляционного способа доступа к удаленной БД с помощью BDE необходимо использовать только средства языка SQL. Поэтому в качестве компонентов должны выбираться такие как `Query`, `StoredProc` или `UpdateSQL`. Кроме того, для набора данных нельзя использовать методы, характерные для навигационного способа доступа, например, `Next` и `Previous` для перемещения текущего указателя или `Edit`, `Insert`, `Append` или `Delete` для изменения записей.

Напомним, что если при выполнении модифицирующего БД запроса с помощью компонента `Query` не нужен результирующий набор данных, то этот запрос предпочтительнее выполнять с помощью метода `ExecSQL`. Например:

```
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('DELETE FROM Personnel WHERE Position = 'Менеджер');
Query1.ExecSQL;
```

Для работы с таблицами и запросами по-прежнему можно использовать такие программы, как Database Desktop и SQL Explorer.

Средства Delphi, предназначенные для работы с удаленными БД, можно разделить на два вида:

- инструменты;
- компоненты.

К *инструментам* относятся специальные программы и пакеты, обеспечивающие обслуживание БД вне разрабатываемых приложений. Среди них:

- InterBase Server Manager — программа управления запуском сервера InterBase;
- IBConsole — консоль сервера InterBase;
- SQL Monitor — программа отслеживания порядка выполнения SQL-запросов к удаленным БД.

Компоненты предназначены для создания приложений, выполняющих операции с удаленной БД. Перечислим наиболее важные из них:

- `Database` (соединение с БД);
- `Session` (текущий сеанс работы с БД);
- `StoredProc` (вызов хранимой процедуры);
- `UpdateSQL` (модификация набора данных, основанного на SQL-запросе);
- `DCOMConnection` (DCOM-соединение);
- компоненты страниц **ADO**, **dbExpress** и **InterBase** Палитры компонентов.

Отметим, что многие из названных компонентов, например, `Database`, `UpdateSQL` и `Session`, используются также при работе с локальными БД. Так, компонент `Database` позволяет реализовать механизм транзакций, как было показано в *главе 7*, посвященной навигационному способу доступа к данным с помощью механизма BDE. Однако наиболее часто эти компоненты применяются именно при работе с удаленными базами.

Часть компонентов, например, клиентский набор данных `ClientDataSet` и соединение с сервером `DCOMConnection`, предназначена для работы в трехуровневой (трехзвенной) архитектуре "клиент-сервер" ("тонкий" клиент) и используется для построения сервера приложений.

В основе операций, выполняемых с удаленными БД как с помощью инструментов, так и программно, лежит язык SQL. Например, при создании таблицы с

помощью программы IBConsole (в ее окне **Interactive SQL**) необходимо набрать и выполнить SQL-запрос (инструкцию) `CREATE TABLE`. Если создание таблицы с помощью механизма BDE осуществляется из приложения пользователя, то для этой цели используется набор данных `Query`, который выполняет такой же запрос. Основное различие заключается в том, каким образом выполняется SQL-запрос к удаленной БД.

Итак, для удаленных БД разница между средствами, используемыми в приложении, и инструментами намного меньше, чем для локальных баз данных. Поэтому далее для удаленных БД использование инструментов (на примере программы IBConsole) и создание приложений рассматриваются параллельно.

Сервер InterBase

Все серверы имеют похожие принципы организации данных и управления ими. В качестве примера рассмотрим работу с сервером InterBase 6.x, который является "родным" для Delphi. Совместно с Delphi поставляются две части сервера InterBase 6.x: серверная и клиентская. Несмотря на то что сервер InterBase поставляется совместно с Delphi, устанавливается он отдельно: после установки Delphi выдается запрос на установку сервера InterBase. Установка происходит в автоматическом режиме, основные файлы сервера копируются в подкаталог INTERBASE, находящийся в каталоге BORLAND. Отметим, что бесплатная пробная (trial) версия сервера доступна по адресу **www.borland.com/interbase**.

Серверная часть InterBase является локальной версией сервера InterBase и используется для отладки приложений, предназначенных для работы с удаленными БД, позволяя на одном компьютере проверить их в сетевом варианте. После отладки на локальном компьютере приложение можно перенести на сетевые компьютеры без изменений, для чего нужно:

- скопировать БД на сервер;
- установить для приложения новые параметры соединения с удаленной БД.

Скопировать БД можно с помощью программ типа Проводник Windows.

Клиентская часть нужна для обеспечения доступа приложения к удаленной БД.

При разработке БД и приложений с использованием локальной версии сервера InterBase нужно иметь в виду, что она имеет ряд ограничений и может не поддерживать, например, механизм событий сервера или определяемые пользователем функции. Полнофункциональная версия сервера InterBase приобретается и устанавливается отдельно от Delphi.

Как упоминалось, в основе работы с удаленной БД лежат возможности языка SQL, обеспечивающие соответствующие операции. Назначение и возможности языка SQL для удаленных БД в принципе совпадают с назначением и возможностями этого языка для локальных БД. Ниже мы рассмотрим особенности языка SQL для удаленных БД.

При рассмотрении инструкций языка будем опускать несущественные операнды и элементы. При описании формата инструкций языка SQL используются следующие правила:

- символы < и > обозначают отдельные элементы формата инструкций, например, имена таблиц и столбцов, и при записи инструкций SQL не указываются;
- в квадратные скобки заключаются необязательные элементы конструкций языка;
- элементы списка, из которого при программировании можно выбрать любой из этих элементов, разделяются знаком |, а сам список заключается в фигурные скобки.

Для наглядности зарезервированные слова языка SQL будем писать заглавными (прописными) буквами, а имена — строчными (маленькими). Регистр букв не влияет на интерпретацию инструкций языка.

Бизнес-правила

Как отмечалось, бизнес-правила представляют собой механизмы управления БД и предназначены для поддержания БД в целостном состоянии. Кроме того, они нужны для реализации ограничений БД, а также для выполнения ряда других действий, например, накапливания статистики работы с БД.

Бизнес-правила можно реализовывать на физическом и программном уровнях. В первом случае эти правила (например, ограничение ссылочной целостности для связанных таблиц) задаются при создании таблиц и входят в структуру БД. Для этого в синтаксис инструкции `CREATE TABLE` включаются соответствующие операнды, например, `DEFAULT` (значение по умолчанию). В дальнейшей работе нельзя нарушить или обойти ограничение, заданное на *физическом* уровне.

На программном уровне бизнес-правила можно реализовать в сервере и в приложении. Причем эти бизнес-правила не должны быть определены на физическом уровне. Для реализации бизнес-правил в сервере обычно используются *триггеры*. Достоинствами такого подхода является то, что вычислительная нагрузка по управлению БД целиком ложится на сервер, что снижает нагрузку на приложение и сеть, а также то, что действие ограничений распространяется на все приложения, осуществляющие доступ к БД. Однако одновременно снижается гибкость управления БД. Кроме того, нужно учитывать, что средства отладки триггеров и хранимых процедур сервера развиты недостаточно хорошо.

Для программирования бизнес-правил в приложении используются *компоненты* и их средства. Достоинство такого подхода заключается в легкости изменения бизнес-правил и возможности определить правила "своего" приложения. Недостатком является снижение безопасности БД, связанное с тем, что каждое приложение может устанавливать свои правила управления БД. В *главе 7*, посвященной навигационному способу доступа, мы рассмотрели программирование бизнес-правил в приложении на примере каскадного удаления записей в связанных локальных таблицах.

Организация данных

Информация всей БД сервера InterBase хранится в одном файле с расширением gdb. Размер этого файла может составлять единицы и даже десятки гигабайт. Отметим, что аналогичный размер БД имеет СУБД Microsoft SQL Server, в то время как для более мощных СУБД Oracle и SyBase размер БД достигает десятков и сотен гигабайт.

В отличие от локальной БД, структуру которой составляли таблицы (отдельные или связанные), удаленная БД имеет более сложную структуру, которая включает в свой состав следующие элементы:

- таблицы;
- индексы;
- ограничения;
- домены;
- просмотры;
- генераторы;
- триггеры;
- функции пользователя;
- хранимые процедуры;
- исключения;
- BLOB-фильтры;
- привилегии.

Элементы структуры удаленной БД также называют *метаданными*. Слово "мета" имеет смысл "над", т. е. метаданные представляют собой данные, которые описывают структуру БД.

Для InterBase максимальное число таблиц в БД равно 65 536, а максимальное число столбцов в таблице — 1000. Отметим, что таблицы InterBase имеют меньшее число допустимых типов столбцов (полей), чем таблицы локальных БД Paradox. Типы столбцов базы InterBase приведены в табл. 13.1.

Таблица 13.1. *Типы столбцов таблиц InterBase*

Тип	Описание
`SMALLINT`	Целое число. Диапазон –32 768 .. 32 767
`INTEGER`	Целое число. Диапазон –2 147 483 648 .. 2 147 483 647
`FLOAT`	Число с плавающей точкой. Диапазон по модулю 3.4×10^{-38} .. 3.4×10^{38}. Точность 7 цифр мантиссы
`DOUBLE PRECISION`	Число с плавающей точкой. Диапазон по модулю 1.7×10^{-308} .. 1.7×10^{308}. Точность 15 цифр мантиссы
`CHARACTER(N)`	Строка длиной `N` символов (не более 32 767)
`VARCHAR(N)` или `CHARACTER(N) VARYING`	Строка символов длиной до `N` символов (не более 32 767)
`DATE`	Дата. Диапазон 01.01.0100 .. 11.12.5941
`BLOB`	Двоичные данные любого типа. Размер не ограничивается

В таблицах InterBase отсутствуют такие типы, как логический и автоинкрементный. Логический тип заменяется типом `CHAR(1)`, а вместо автоинкрементного типа для обеспечения уникальных значений используются генераторы и триггеры.

Запуск сервера

Для запуска сервера предназначена программа InterBase Server Manager (рис. 13.1), вызываемая одноименной командой главного меню Windows или через Панель управления. Функции управления сервером InterBase, например, управление подключением к серверу пользователей и просмотр информации о БД, в последних версиях Delphi реализует программа IBConsole.

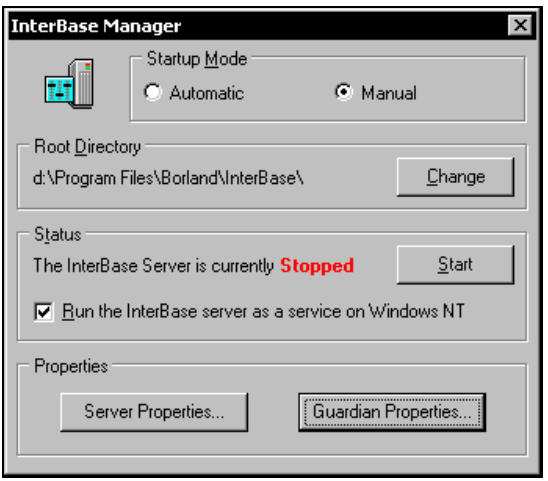


Рис. 13.1. Диалоговое окно программы InterBase Server Manager

Состояние сервера выводится в панели **Status**: запущенному состоянию сервера соответствует надпись **Running**, остановленному — **Stopped**.

Сервер InterBase может запускаться автоматически или в ручном режиме, чем управляют переключатели группы **Start Mode** (Режим запуска). Если выбран переключатель **Automatic**, то сервер будет автоматически вызываться при каждом запуске (перезапуске) Windows. Если же выбран ручной запуск (**Manual**), то сервер запускается нажатием кнопки **Start**. После запуска сервера кнопка **Start** изменяет свое название на **Stop**, и ее повторное нажатие приводит к остановке сервера.

В Windows NT сервер можно запустить как службу (service). Установленный флажок **Run the InterBase server as a service on Windows NT** указывает, что сервер InterBase запускается как служба Windows NT.

Панель **Root Directory** показывает главный каталог, в котором установлен сервер InterBase и который можно изменить, нажав кнопку **Change** и выбрав нужный каталог. Однако на практике изменять главный каталог не требуется.

Завершить работу сервера также можно, открыв щелчком мыши на значке контекстное меню и выбрав команду **Shutdown**.

При запущенном сервере с помощью кнопок **Server Properties** (Свойства сервера) и **Guardian Properties** (Свойства безопасности) открываются диалоговые окна настройки соответствующих свойств. Обычно в них уже заданы нужные значения, изменять которые нет необходимости.

При запуске сервера в качестве службы Windows NT управлять его параметрами, а также остановить сервер можно в окне **Services**.

Особенности приложения

Есть основные принципы разработки и использования приложений, выполняющих операции с удаленными БД. Эти принципы являются общими для различных систем, например, таких как InterBase, Microsoft SQL Server или Oracle. Поэтому примеры программ (приложений) для работы с БД InterBase, которые приводятся ниже, годятся и для других серверов. При разработке приложений использованы компоненты, стандартные для всех БД.

Наряду с этим в Delphi имеется ряд компонентов, предназначенных только для работы с сервером InterBase. Эти компоненты расположены на странице **InterBase** Палитры компонентов. Многие из них являются аналогами соответствующих компонентов страницы **BDE** и отличаются тем, что адаптированы специально под InterBase.

Особенность использования данных компонентов заключается в том, что для них доступ к БД осуществляется напрямую через процессор баз данных BDE. При этом нет необходимости в соответствующем драйвере SQL-Links.

Компоненты страницы **InterBase** принципиально не отличаются от соответствующих компонентов других страниц, поэтому здесь они описаны кратко.

Соединение с базой данных

Для выполнения любых операций с БД с ней необходимо установить соединение, т. е. *открыть* БД. По завершении работы соединение нужно разорвать, или завершить (*закрыть* БД). Для соединения с БД программы типа IBConsole имеют соответствующие средства, вызываемые с помощью команд меню. При создании приложения разработчик должен организовывать соединение самостоятельно, для чего Delphi предоставляет соответствующие компоненты, в первую очередь компонент `Database`.

Рассмотрим, как установить соединение с удаленной БД с помощью программы IBConsole и компонентов Delphi, поддерживающих механизм доступа BDE.

Соединение с базой из программы IBConsole

Перед установлением соединения с БД нужно с помощью команд **Server\Register** и **Server\Login** выполнить регистрацию сервера InetrBase, а также задать имя и пароль пользователя.

Замечание

Заметим, что первоначально на сервере InetrBase для пользователей есть стандартное имя SYSDBA и пароль "masterkey", которые могут изменяться администратором.

Соединение с БД выполняется командой **Database\Connect**. После открытия БД доступна для работы, например, для изменений структуры путем добавления и удаления таблиц или для редактирования данных.

Для отключения от БД следует выполнить команду **Database\Disconnect**, при этом запрашивается подтверждение на выполнение этой операции. При утвердительном ответе соединение с БД разрывается. При необходимости также запрашивается подтверждение текущей незавершенной транзакции.

На программном уровне соединение с БД выполняет инструкция `CONNECT`, имеющая следующий формат:

```
CONNECT DATABASE "<Имя файла БД>"
USER "<Имя пользователя>" PASSWORD "<Пароль пользователя>"
```

Отключение от БД выполняет инструкция `EXIT`.

Компонент *Database*

Компонент `Database` служит для соединения с БД. Если программист при разработке приложения не разместил в форме компонент `Database`, то для организации соединения с БД в процессе выполнения приложения будет создаваться динамический (временный) объект типа `TDatabase`. Динамическое создание этого объекта выгодно тем, что не требует от программиста каких-либо действий. Однако при использовании динамического объекта типа `TDatabase` он автоматически создается для каждого соединения с БД, что при одновременной работе с несколькими БД требует бóльших ресурсов Windows, чем в случае применения компонента `Database`.

Использование компонента `Database` позволяет:

- настраивать параметры соединения с БД;
- явно управлять транзакциями при работе с БД.

Компонент `Database` можно использовать для локальных и для удаленных БД. Например, действия, связанные с настройкой псевдонимов BDE, одинаковы для обоих типов БД.

Свойство `SessionName` типа `String` указывает компонент сеанса `Session`, с которым связан компонент `Database`. Если значение этого свойства не задано

(пустое значение или значение `Default`), то для сеанса создается динамический объект типа `TSession` (объект по умолчанию).

Свойство `AliasName` типа `String` указывает псевдоним БД. На этапе разработки приложения псевдоним выбирается из списка в окне Инспектора объектов.

Расположение БД может быть определено также с помощью свойства `DriverName` типа `String`, задающего драйвер БД. Следует иметь в виду, что значения этого свойства и свойства `AliasName` являются взаимоисключающими: при установке одного из них второе автоматически сбрасывается. Если расположение БД задано значением свойства `DriverName`, то остальные параметры соединения должны быть заданы через свойство `Params`.

Свойство `DatabaseName` типа `String` задает имя БД, используемое в приложении для соединения с БД. Отметим, что это имя не совпадает ни с псевдонимом, ни с собственно именем БД (именем главного ее файла). Имя БД, заданное свойством `DatabaseName`, действует только в приложении и только для организации подключения к БД, указанной ее псевдонимом.

Впоследствии набор данных (обычно `Query`) связывается с БД через свойство `DatabaseName`, значение которого должно совпадать со значением одноименного свойства компонента `Database`, используемого для соединения. На этапе разработки приложения имя БД для набора данных выбирается в окне Инспектора объектов в списке, содержащем его наряду с псевдонимами BDE.

Свойство `Params` типа `TStrings` определяет параметры соединения, для изменения которых при разработке приложения используется редактор списка значений **Value List editor**, вызываемый двойным щелчком в области значения Инспектора объектов. С его помощью можно (неясно, почему) установить значение одного параметра: в поле **Key** указывается имя параметра, а в поле **Value** указывается значение параметра.

Нажатие кнопки **Code Editor** преобразует список к текстовому виду и отображает его на отдельной вкладке в окне редактора кода (рис. 13.2). В окне редактора кода можно задавать и редактировать произвольное число параметров. Более того, теперь при двойном щелчке в области значения свойства `Params` Инспектора объектов всегда происходит вызов редактора кода, а не редактора Value List editor.

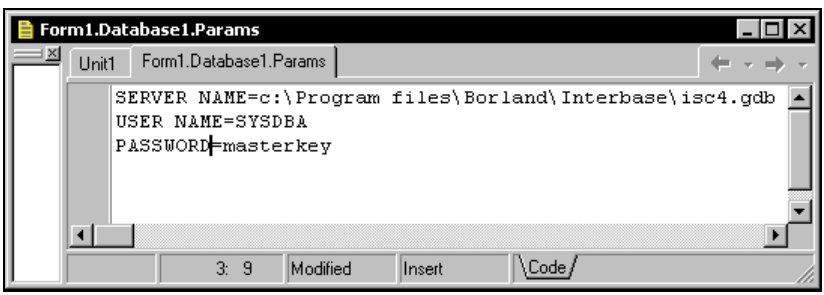


Рис. 13.2. Параметры соединения с БД в окне редактора кода

В приведенном на рис. 13.2 примере указаны имя сервера, имя пользователя и его пароль. Отметим, что имя сервера лучше задавать через псевдоним, что облегчает перенос БД на другое место.

При выполнении приложения доступ к отдельному параметру осуществляется по его индексу в массиве (списке) параметров `Params`. При этом необходимо учитывать, что в значение параметра также входит его название. Рассмотрим следующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Edit1.text := Database1.Params[1];
end;
```

Здесь при нажатии кнопки в поле редактирования (компонент `Edit`) выводится второй параметр соединения.

В частности, для параметров, приведенных на рис. 13.2, будет получена следующая строка:

```
USER NAME = SYSDBA
```

Кроме параметров, заданных в свойстве `Params` компонента `Database`, соединение с БД также определяется:

- установками системных параметров BDE;
- параметрами драйвера, используемого для доступа к БД;
- параметрами псевдонима БД.

Если в списке `Params` какой-либо параметр, например, `LANGDRIVER`, не задан, он выбирается из перечисленных выше установок. Поэтому с помощью свойства `Params` нужно устанавливать (переустанавливать) только те параметры, значения которых не устраивают разработчика.

Свойство `LoginPrompt` типа `Boolean` управляет режимом отображения окна ввода имени пользователя и пароля **Database Login** (рис. 13.3). По умолчанию свойство имеет значение `True`, и окно появляется при первом соединении с БД. Если установить свойство `LoginPrompt` в значение `False`, то запрос на идентификацию пользователя не выдается, и его имя и пароль должны быть указаны в параметрах соединения (свойство `Params`).

Свойство `KeepConnection` типа `Boolean` определяет, сохранять ли соединение с БД, если с ней не связан ни один набор данных. По умолчанию свойство имеет значение `True`, и однократно установленное соединение сохраняется в течение всей работы с БД. Использование постоянного соединения с БД экономит время, затрачиваемое на доступ к данным, но требует определенных системных затрат на его поддержание. Постоянное соединение разрывается при окончании работы приложения или при программной установке свойства `Connected` в значение `False`.

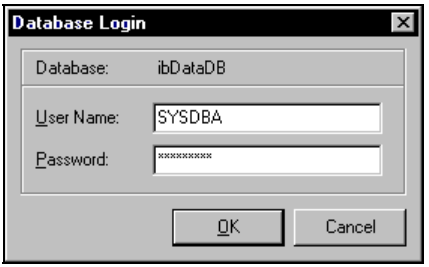


Рис. 13.3. Окно ввода имени пользователя и пароля

На этапе разработки управлять значениями названных свойств можно в окне Редактора параметров соединения (рис. 13.4). Оно вызывается двойным щелчком на компоненте `Database` или вызовом команды **Database editor** контекстного меню компонента.

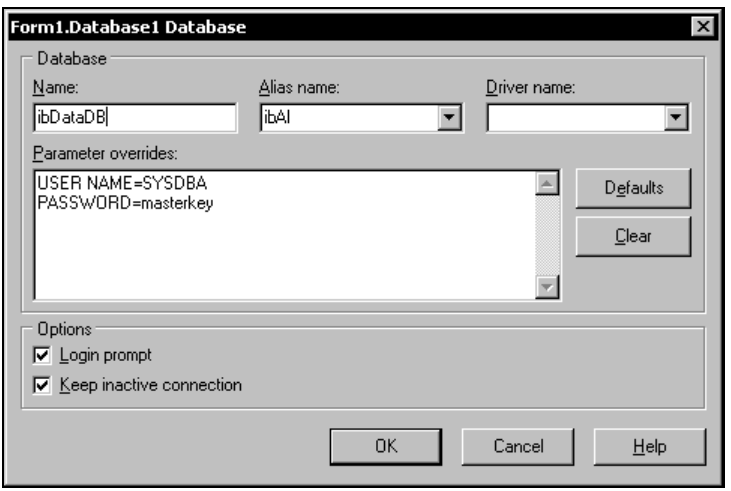


Рис. 13.4. Редактор параметров соединения с БД

Элементы управления этого окна отображают и позволяют изменять значения соответствующих свойств компонента `Database`:

- поле редактирования **Name** — свойство `DatabaseName`;
- комбинированный список **Alias name** — свойство `AliasName`;
- комбинированный список **Driver name** — свойство `DriverName`;
- многострочное поле редактирования **Parameter overrides** (Новые значения параметров) — свойство `Params`;
- флажок **Login prompt** (Входное сообщение) — свойство `LoginPrompt`;

- флажок **Keep inactive connection** (Сохранять неактивное соединение) — свойство `KeepConnection`.

Отметим, что на рис. 13.3 и 13.4 в качестве имени базы данных выводится именно значение свойства `DatabaseName`, а не имя (значение свойства `Name`) компонента `Database`.

Свойство `Connected` типа `Boolean` определяет, установлено ли соединение с БД. Соединение с БД можно установить при разработке приложения, задав для этого свойства значение `True` через Инспектор объектов.

При выполнении приложения можно также установить или разорвать соединение с БД, используя методы `Open` и `Close` соответственно.

Свойства `DataSetCount` и `DataSets` позволяют получить доступ к наборам данных, связанным в настоящий момент времени с БД. Свойство `DataSetCount` типа `Integer` определяет число таких наборов, а свойство `DataSets [Index: Integer]` типа `TDBDataSet` представляет собой коллекцию (массив) этих наборов данных. Доступ к отдельному набору данных можно получить по его номеру в массиве (нумерация начинается с нуля, т. е. последний набор данных имеет номер `DataSetCount - 1`).

При выполнении операций с удаленными БД автоматически запускается механизм транзакций, в данном случае называемый *неявным*. Кроме того, программист может управлять транзакциями и явно, но при этом нужно учитывать, что явные транзакции несколько замедляют выполнение операций с БД. Механизм транзакций реализуется с помощью ряда свойств и методов компонента `Database`.

Методы `StartTransaction` (инициирует начало транзакции), `Commit` (подтверждает текущую транзакцию и `Rollback` (отменяющий ее) были рассмотрены в *главе 7*, посвященной навигационному способу доступа к данным с помощью механизма BDE.

Свойство `InTransaction` типа `Boolean` позволяет определить, существует ли активная транзакция для текущего соединения с БД. Если имеется незавершенная транзакция, то свойство имеет значение `True`, и значение `False` — в противном случае. Это свойство можно использовать, например, при подтверждении транзакции. Если активной транзакции нет, то вызов метода `Commit` приводит к генерации исключения. Например:

```
if Database1.InTransaction then Database1.Commit;
```

Если имеется активная транзакция, то она подтверждается, а все сделанные в ее рамках изменения БД вступают в действие.

Свойство `TransIsolation` типа `TTransIsolation` управляет *уровнем изоляции* транзакций. Уровень изоляции определяет, каким образом транзакция взаимодействует с другими транзакциями. Ниже приведен список возможных значений свойства `TransIsolation`.

- `tiDirtyRead` — разрешается чтение изменений в записях БД, сделанных в рамках других транзакций (другими пользователями). В связи с тем, что на

момент чтения эти изменения являются неподтвержденными и впоследствии могут быть отменены, этот уровень изоляции обеспечивает минимальную изоляцию (защиту) от других транзакций.

- `tiReadCommitted` — разрешается чтение только подтвержденных изменений в записях БД (по умолчанию). Если изменения еще не подтверждены, то читается предыдущая версия записи.
- `tiRepeatableRead` — считывание сразу всей БД, после чего изменения в БД, сделанные в рамках других транзакций, становятся невидимыми. Этот уровень обеспечивает максимальную изоляцию.

Для локальных БД dBase и Paradox допустимым является только значение `tiDirtyRead`. Для сервера InterBase значение `tiDirtyRead` интерпретируется как `tiReadCommitted`.

Компонент *Session*

Компонент `Session` представляет собой текущий сеанс работы с БД и наряду с компонентом `Database` используется для управления соединением с БД. Однако если компонент `Database` позволяет задавать параметры одного соединения с БД или параметры соединения с одной БД, то компонент `Session` управляет всеми соединениями со всеми БД. Если программист при разработке приложения не разместил в форме компонент `Session`, то будет создаваться динамический (временный) объект типа `TSession` (`TSessionList`).

Наряду с управлением соединениями с БД, использование компонента `Session` также позволяет:

- управлять установками BDE;
- управлять паролями БД.

Свойство `SessionName` типа `String` задает имя сеанса. Это имя устанавливается программистом, и его назначение не совпадает с назначением свойства `Name`, которое определяет имя самого компонента `Session`. Однако в частном случае эти имена могут совпадать.

Свойства сеанса, заданного компонентом `Session`, действуют на все связанные с ним компоненты `Database`. Напомним, что компонент `Database` связывается с сеансом через свое свойство `SessionName`, значение которого при разработке приложения выбирается в списке Инспектора объектов.

С помощью свойств `DatabaseCount` и `Databases` можно получить доступ к активным БД, с которыми в настоящее время установлено соединение. Свойство `DatabaseCount` типа `Integer` содержит число таких БД, а свойство `Databases[Index: Integer]` типа `TDatabase` представляет собой коллекцию (массив) объектов `Database`, используемых для соединения с этими БД. Доступ к отдельной БД можно получить по ее номеру в массиве (нумерация начинается с нуля, поэтому последняя БД имеет номер `DatabaseCount - 1`).

Свойство `KeepConnections` типа `Boolean` определяет, сохранять ли соединение с БД, если с ней не связан ни один набор данных. Это свойство аналогично свойству `KeepConnection` соединения с отдельной БД, заданного компонентом `Database`. Разорвать соединение с БД можно, вызвав метод `DropConnections`.

Свойства `NetFileDir` и `PrivateDir` типа `String` позволяют задать каталоги для хранения управляющего сетевого файла с расширением net (для таблиц Paradox) и рабочего каталога для хранения временных файлов сеанса соответственно.

Свойство `Active` типа `Boolean` управляет активностью сеанса. Установка этого свойства в значение `True` активизирует сеанс, а значения `False` — деактивизирует его. При выполнении приложения управлять активностью сеанса можно также с помощью методов `Open` и `Close` соответственно.

Непосредственно перед активизацией сеанса генерируется событие `OnStartup` типа `TNotifyEvent`.

Для управления установками BDE компонент `Session` имеет средства, позволяющие читать и изменять параметры драйверов и псевдонимов БД. Рассмотрим наиболее важные из них.

Методы `GetAliasNames` и `GetAliasParams` обеспечивают получение информации о псевдонимах BDE. Процедура `GetAliasNames(List: TStrings)` позволяет получить список псевдонимов BDE. Список заносится в объект, заданный параметром `List` типа `TStrings`, которым может быть, например, список строк компонента `ListBox`. Приведем соответствующий пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Session1.GetAliasNames(ListBox1.Items);
end;
```

Здесь нажатие кнопки `Button1` приводит к загрузке в компонент `ListBox1` списка псевдонимов BDE. Предыдущее содержимое компонента `ListBox1` теряется.

Процедура `GetAliasParams (const AliasName: String; List: TStrings)` возвращает параметры конкретного псевдонима, имя которого задано параметром процедуры `AliasName`. Если имя псевдонима указано неправильно, то генерируется исключение. Например:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Session1.GetAliasParams(Edit1.Text, ListBox1.Items);
end;
```

При нажатии кнопки `Button2` в компонент `ListBox1` заносятся параметры псевдонима, введенного в компонент `Edit1`. Предыдущее содержимое компонента `ListBox1` теряется.

Для добавления и удаления псевдонимов предназначены методы `AddAlias` и `DeleteAlias` соответственно. Процедура `AddAlias(const Name, Driver: String; List: TStrings)` добавляет новый псевдоним с именем `Name`. Параметр `Driver` определяет драйвер нового псевдонима, а параметр `List` содержит список параметров псевдонима. Пример создания псевдонима для локальной БД:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
Session1.AddAlias(Edit1.Text, 'STANDARD', Memo1.Lines);
end;
```

В процедуре создается новый псевдоним, имя которого набрано в компоненте `Edit1`. Псевдоним предназначен для работы с локальными БД dBase или Paradox, на что указывает тип `STANDARD` псевдонима. Напомним, что локальные БД этих типов работают под управлением драйвера `STANDARD`. Параметры псевдонима должны быть указаны в компоненте `Memo1`. Необходимо указать хотя бы расположение БД, например, введя в поле редактирования следующий текст:

```
PATH = d:\data
```

Имя нового псевдонима должно быть уникальным: если указанный псевдоним уже существует, то генерируется исключение.

В качестве примера рассмотрим следующую процедуру:

```
procedure TForm1.Button3Click(Sender: TObject);
var Params: TStringList;
begin
Params := TStringList.Create;
try
Params.Add('SERVER NAME=d:\ibData\registration.gdb');
Params.Add('USER NAME=SYSDBA');
Params.Add('PASSWORD=masterkey');
Session1.AddAlias('ibData', 'INTRBASE', Params);
finally
  Params.Free;
end;
end;
```

В ней создается псевдоним `ibData` для работы с удаленной БД InterBase. Для БД заданы три параметра: путь, имя и пароль пользователя. Для определения параметров в приложении динамически создается список `Params` типа `TStringList`. Заполнение и удаление списка также выполнено динамически. При заполнении списка и создании псевдонима выполняется обработка исклю-

чений. При их возникновении в любом случае будет удален динамически созданный список `Params`.

Процедура `DeleteAlias(const Name: String)` удаляет псевдоним, указанный параметром `Name`. Например:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
Session1.DeleteAlias(Edit1.Text);
end;
```

Здесь удаляется псевдоним, имя которого указано в компоненте `Edit1`. Если указанный псевдоним отсутствует, то процедура `DeleteAlias` не выполняет никаких действий.

Методы `GetDriverNames` и `GetDriverParams` обеспечивают получение информации о драйверах BDE и их параметрах соответственно, а методы `AddDriver` и `DeleteDriver` — добавление и удаление драйверов.

Соединение с базой данных из приложения

Для соединения с БД из приложения разработчик должен создать соответствующую форму. В целом создание формы приложения для работы с удаленной БД не отличается от процесса создания формы для работы с локальной БД. Визуальные и невизуальные компоненты размещаются в форме и связываются между собой обычным способом.

Пусть в форме размещены сетка `DBGrid1`, источник данных `DataSource1` и набор данных `Query1`. (Напомним, что в качестве набора данных следует выбирать компонент `Query` и ему подобные, а не компонент `Table`.)

Отличие формы для работы с удаленной БД заключается в том, что дополнительно организуется соединение с БД. С этой целью необходимо:

1. Создать псевдоним БД.
2. Разместить в форме компонент `Database`, установить нужные значения его свойств и связать с набором данных.

Псевдоним для работы с удаленной БД удобно создать с помощью рассмотренной ранее программы BDE Administrator. Для сервера InterBase псевдоним имеет тип `INTRBASE`. Для псевдонима нужно задать следующие параметры:

- имя пользователя `USER NAME` (`SYSDBA`);
- расположение БД `SERVER NAME` (для локального сервера InterBase это путь к БД, для удаленного сервера этот путь прописан в файле hosts, размещенном в каталоге WINDOWS);
- языковой драйвер `LANGDRIVER` (`Pdox ANSI Cyrillic`).

Для примера создадим псевдоним `ibData` для доступа к БД registration.gdb, размещенной в каталоге D:\ibData. Параметры псевдонима показаны на рис. 13.5.

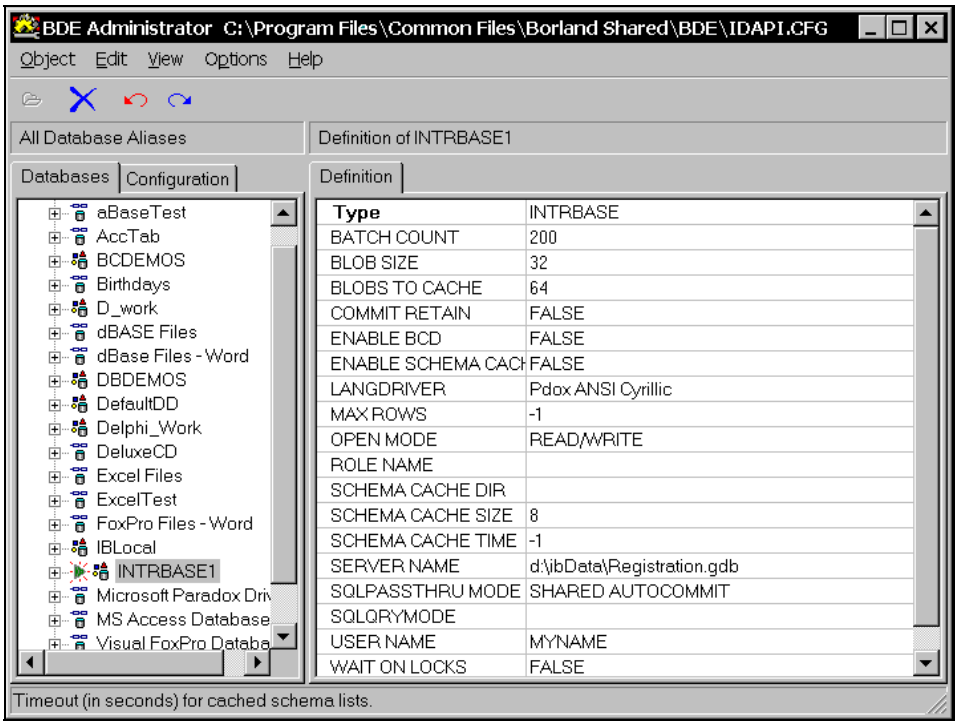


Рис. 13.5. Параметры псевдонима БД

Собственно соединение с БД выполняется с помощью компонента `Database`, рассмотренного в предыдущем разделе. Разместим в форме компонент `Database1`. Для соединения с БД, обозначенной псевдонимом `ibData`, свойства компонентов `Database1` и `Query1` нужно установить в значения, показанные в табл. 13.2.

Таблица 13.2. *Свойства компонентов, используемых для соединения с БД*

Компонент	Свойство	Значение	Примечание
`Database1`	`AliasName`	`ibData`	Выбирается в списке
	`DatabaseName`	`ibDataBD`	Вводится разработчиком
`Query1`	`DatabaseName`	`ibDataBD`	Выбирается в списке

Соединение с БД устанавливается следующими способами:

- через открытие набора данных `Query1`;
- путем установки свойства `Connected` компонента `Database1` в значение `True`.

Оба способа по результатам являются эквивалентными и взаимозависимыми. Так, при открытии набора данных, например, установкой в значение `True` его свойства `Active`, свойство `Connected` компонента `Database` автоматически принимает это же значение.

Соединение с БД можно устанавливать как при разработке, так и при выполнении приложения.

Глава 14

Работа с удаленными базами данных

В данной главе мы рассмотрим основные операции, осуществляемые при работе с удаленной БД: создание БД и управление ее структурой, использование событий сервера и механизма транзакций, управление привилегиями и манипулирование данными.

Создание базы данных

В отличие от локальной БД, являющейся скорее логическим понятием, поскольку ее таблицы находятся в разных файлах и, возможно, в разных каталогах, удаленная БД представляет собой физический объект.

Для создания удаленной БД InterBase удобно использовать программу IBConsole, на примере работы с которой мы и изучим операции с БД. Процесс создания БД начинается с вызова команды **Database\Create Database**, открывающей окно **Create Database** (рис. 14.1).

Надпись **Server** отображает название сервера InterBase, в нашем случае — это локальный сервер (**Local Server**).

Для новой БД необходимо указать ее псевдоним, файлы и параметры. Псевдоним, задаваемый в поле редактирования **Alias**, предназначен для идентификации БД при работе внутри сервера InterBase и не связан с псевдонимами BDE. В списке **File(s)** перечисляются файлы создаваемой БД и их размеры (в страницах), достаточно указать один файл. Для файла необходимо задать его точное расположение, для локального сервера — это каталог.

Среди параметров наибольший интерес представляет установка набора, используемого по умолчанию для кодировки символов. Кодировка важна при выполнении таких операций, как сортировка строк. В Windows для символов, включающих русские буквы, применяется вариант Windows 1251 кода ANSI (**WIN1251**).

После нажатия кнопки **OK** создается БД с указанными параметрами. Новая база первоначально является пустой и не содержит ни таблиц, ни данных. Ее файл,

тем не менее, имеет размер около 600 Кбайт, т. к. включает в себя разнообразную служебную информацию.

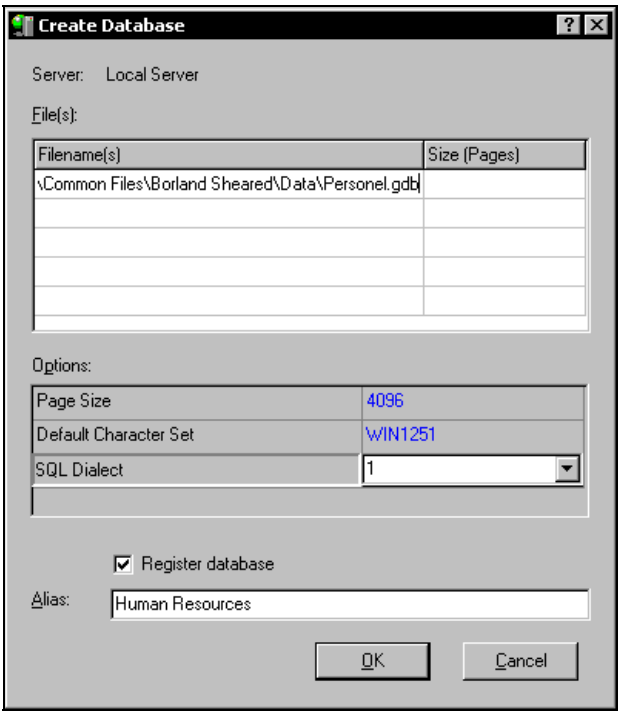


Рис. 14.1. Окно создания удаленной БД

С вновь созданной БД автоматически устанавливается соединение.

Для удаления базы данных следует выполнить команду **Database\Drop Database**, при этом появляется окно с предупреждением. После подтверждения операции происходит удаление, которое заключается в удалении с диска файлов .gdb, содержащих всю информацию базы данных.

Замечание

Восстановить удаленную БД невозможно.

Удалить БД имеет право только ее создатель или системный администратор, имеющий имя SYSDBA.

Создать и удалить БД можно также, подготовив и выполнив соответствующие запросы. Запрос можно выполнить как в среде программы IBConsole, так и из приложения с помощью компонента `Query`.

Инструкция SQL создания БД имеет следующий формат:

```
CREATE DATABASE "<Имя файла БД>"
        [USER "Имя пользователя" [PASSWORD "Пароль пользователя"]]
        [PAGE_SIZE [=] <Целое число>]
        [LENGTH [=] <Целое число> [PAGE[S]]]
        [DEFAULT CHARACTER SET <Набор символов>]
[<Вторичный файл>];
<Вторичный файл> = FILE "<Имя файла БД>" [<Файловая информация>]
        [<Вторичный файл>]
<Файловая информация> = LENGTH [=]<Целое число> [PAGE[S]] |
      STARTING [AT [PAGE]] <Целое число> [<Файловая информация>]
```

Имя файла БД указывает спецификацию (имя и путь) физического файла, в котором будет храниться информация создаваемой БД. Имя файла является единственным обязательным параметром, который должен быть задан для новой БД. Остальные параметры являются факультативными.

Если данные хранятся более, чем в одном физическом файле, то говорят о *многофайловой БД*. Такая структура может быть использована при большом объеме данных, когда файл базы (с расширением gdb) достигает значительных размеров. При этом первый файл БД называется *первичным*, а все последующие — *вторичными*. Файлы могут отличаться именами, расширениями и расположением. Вторичные файлы можно размещать в других каталогах и на других дисках сервера. Размеры (длины) файлов измеряются в страницах.

Для файлов БД задаются следующие параметры:

- `PAGE_SIZE` — размер страницы в байтах, допустимые значения: 1024 (по умолчанию), 2048, 4096 и 8192; параметр задается для первичного файла и действует на все файлы БД;
- `LENGTH` — длина файла в страницах (по умолчанию 75);
- `STARTING` — страница, начиная с которой располагается файл (этот параметр фактически является альтернативой параметру `LENGTH`; если для предшествующего вторичного файла не указана длина, то должна быть указана начальная страница).

При задании размера страницы следует учитывать, что за одно обращение к БД считывается одна страница. Если при работе с БД считывается большой объем информации (много записей), то малый размер страницы увеличит число операций чтения. Кроме того, может увеличиться глубина индекса, что также нежелательно. В то же время большой размер страницы при малом объеме информации приводит к чтению лишних данных. В обоих случаях увеличивается время доступа к данным.

Например, инструкция

```
CREATE DATABASE "D:\IBDATA\personnel.gdb";
```

создает однофайловую БД с именем personnel.gdb, расположенную в каталоге D:\IBDATA.

В свою очередь, инструкция

```
CREATE DATABASE "D:\STORE\Store.gdb"
      FILE " D:\STORE\Store.gd2" STARTING AT PAGE 501 LENGTH = 300
      FILE " D:\STORE\Store.gd3";
```

приводит к созданию многофайловой БД, при этом ее данные размещаются в трех файлах с одинаковым именем Store. Файлы располагаются в общем каталоге D:\STORE и отличаются расширениями. Длина первичного файла (.gdb) составляет 500 страниц и задается косвенным путем — через указание номера начальной страницы следующего (вторичного) файла .gd2, равного 501. Длина первого вторичного файла задана равной 300 страницам, а длина второго не ограничивается.

Отметим, что при задании параметров файлов некоторые элементы являются необязательными — можно опускать знак равенства (при указании длины) и слова `AT PAGE` (при указании начальной страницы).

Параметры `USER` и `PASSWORD` определяют имя и пароль пользователя, создавшего БД. Эти параметры важны при последующем удалении БД, поскольку, как отмечалось, удалить БД может только системный администратор или создатель этой базы данных. Например, с помощью инструкции

```
CREATE DATABASE "D:\IBDATA\test.gdb" USER "User1" PASSWORD "Secret";
```

пользователь с именем `User1` и паролем `Secret` создает однофайловую базу данных с названием `test.gdb`.

Операнд `DEFAULT CHARACTER SET` позволяет определить набор, используемый для кодировки символов. Как отмечалось, для нашей страны рекомендуется задавать набор `WIN1251`. Например:

```
CREATE DATABASE "D:\IBDATA\test.gdb" DEFAULT CHARACTER SET WIN1251;
```

Здесь для БД `test.gdb` по умолчанию устанавливается набор символов `WIN1251`. Отметим, что набор символов можно также отдельно определить для каждого столбца таблицы с помощью аналогичного параметра. Однако удобнее задать набор символов сразу для всей БД, в этом случае для ее таблиц по умолчанию будет использована указанная кодировка.

Инструкция удаления БД имеет следующий вид:

```
DROP DATABASE;
```

После ее выполнения удаляется текущая БД, с которой установлено соединение.

Управление структурой таблиц

Структуру таблицы определяют следующие элементы:

- описания столбцов;
- ограничения столбцов;
- описания ключей;
- описания индексов;
- ограничения таблицы.

Описания и ограничения столбцов объединяют также под общим термином "описание столбцов". В InterBase ключи могут быть первичными и уникальными. Ограничения таблицы включают в себя ограничения ссылочной целостности и ограничения на значения столбцов.

Отметим, что ограничения таблицы, описания ключей и индексов относятся также к структуре базы данных, поэтому их можно просмотреть при выводе метаданных БД, например, с помощью программы IBConsole.

При управлении структурой таблицы удаленной БД состав и использование инструкций определения данных в языке SQL в принципе не отличаются от ранее рассмотренной версии этого языка для локальных БД. Однако в дополнение к локальной версии для столбцов таблицы удаленной БД можно задавать различные ограничения, например, значение по умолчанию или диапазон возможных значений.

Создание таблицы заключается в задании имени и структуры таблицы и выполняется с помощью инструкции `CREATE TABLE`:

```
CREATE TABLE <Имя таблицы> [EXTERNAL [FILE] "<Имя файла>"]
      (<Имя столбца1> <Описание столбца1> [<Ограничения столбца1>,]
      ...
       [<Имя столбцаN> <Описание столбцаN> [<Ограничения столбцаN>],]
       [<Ограничение1 таблицы>,]
      ...
       [<ОграничениеN таблицы>,]
       [<Описание ключа1>,]
      ...
       [<Описание ключаN>,]
       [<Описание индекса1>,]
      ...
       [<Описание индексаN>]);
```

Обязательно должны быть заданы имя таблицы и как минимум один столбец. Часть элементов структуры БД — ограничения столбца и описание ключей — можно специфицировать либо на уровне столбца, либо на уровне таблицы. В первом случае текст описания указывается в описании соответствующего столбца, во втором случае — отдельно, после описания всех столбцов.

По умолчанию таблица добавляется к текущей БД. Если таблицу нужно разместить не в файле БД, а в другом файле, то его имя указывается в операнде `EXTERNAL`.

Удаление таблицы выполняется так же, как в случае локальной БД — инструкцией `DROP TABLE`, имеющей формат:

```
DROP TABLE <Имя таблицы>;
```

Замечание

Удалить таблицу, для которой есть подчиненные таблицы, невозможно. Для удаления главной таблицы нужно предварительно удалить ограничения ссылочной целостности, связывающие эти таблицы.

Структуру таблицы можно изменять путем удаления или добавления столбцов, ключей, индексов и ограничений.

Замечание

Для изменения структуры таблиц нужно иметь соответствующие права (полномочия) доступа к БД и ее таблицам. Кроме того, изменяемые элементы структуры не должны в этот момент использоваться другими пользователями. Например, нельзя удалить индекс, который в данный момент применяется для поиска записей.

Как и для локальных БД, изменение таблицы выполняется инструкцией `ALTER TABLE`, которая позволяет добавлять и удалять отдельные столбцы, а также ограничения. В отличие от инструкции `ALTER TABLE`, другие инструкции типа `ALTER xxx` (например, `ALTER TRIGGER`) удаляют предыдущее описание указанного объекта и заменяют его на новое.

Удаляя и добавляя столбцы, можно требуемым образом изменить состав таблицы. Однако следует иметь в виду, что после удаления столбца теряется вся его информация. Поэтому в случае, когда необходимо только немного изменить описание столбца, рекомендуется поступать так:

1. Создать новый столбец с требуемым описанием.
2. Скопировать данные из старого столбца, описание которого изменяется, в новый столбец.
3. Удалить старый столбец.

Рассмотрим пример, иллюстрирующий приведенную последовательность действий:

```
CREATE TABLE List
      (Name VARCHAR(15));
      ...
ALTER TABLE List
      ADD Name2 VARCHAR(20);
UPDATE List
      SET Name2 = Name;
ALTER TABLE List
      DROP Name;
```

Для ведения списка создана таблица `List`, имеющая столбец `Name` строкового типа длиной не более 15 символов. В процессе работы с таблицей возникла необходимость увеличить длину столбца до 20 символов, что и было сделано выполнением последовательности трех инструкций. Столбец получил новое имя `Name2`.

Замечание

При уменьшении длины столбца возможна потеря части информации из-за усечения значений, расположенных в конце более длинного столбца.

В рассмотренном способе изменения описания столбца имя нового столбца отличается от старого, что может оказаться нежелательным. В этом случае следует повторить указанные действия еще раз, присвоив новому столбцу прежнее имя.

Описание столбца

Описание столбца имеет формат:

```
<Описание столбца> =
      {<Имя столбца> | COMPUTED [BY] (<Выражение>) | <Домен>}
```

Столбец можно определить следующими тремя способами.

- Задать тип столбца, например, `DATE` или `INTEGER`, при этом создается обычный столбец указанного типа. Типы столбцов InterBase описаны в данной главе ниже.
- Создать вычисляемый столбец, задав в операнде `COMPUTED` выражение.
- Создать столбец на основе домена.

Вычисляемый столбец содержит значения, которые рассчитываются на основе заданного выражения. Такой столбец удобно использовать, когда его значения зависят от каких-либо других значений, в том числе от значений других столбцов записи, и вид зависимости заранее известен. Тип вычисляемого столбца автоматически определяется типом результата выражения. Например, если результат выражения имеет целочисленный тип, то вычисляемый столбец также будет целочисленным.

Домен представляет собой предварительное описание столбца, которое затем можно использовать для описания столбцов. Создание домена также рассматривается в данной главе немного позже.

Пример:

```
CREATE TABLE List
      (Name      VARCHAR(20),
      Price      FLOAT,
      Number     INTEGER,
      PriceAll   COMPUTED BY (Price * Number));
```

Здесь создается таблица `List`, в которой ведется учет продажи товаров. Столбец `Name`, предназначенный для названия товара, имеет строковый тип (длина не более 20 символов). Столбец `Price` содержит цену единицы товара и имеет вещественый тип, а целочисленный столбец `Number` указывает количество товара. Столбец `PriceAll` содержит общую стоимость всего товара, которая рассчитывается как произведение цены товара на его количество. Исходя из типа операндов, тип этого вычисляемого столбца автоматически будет определен как `FLOAT`.

Ограничения столбца

Ограничения являются необязательными элементами, однако их использование позволяет автоматизировать процесс ввода значений, предотвращать ошибки ввода и управлять порядком сортировки строк.

Ограничения столбца имеют следующий формат:

```
[DEFAULT {<Значение> | NULL | USER}]
[NOT NULL]
[COLLATE <Порядок сортировки>]
[CHECK <Условия>]
```

Операнд `DEFAULT` определяет для столбца значение по умолчанию, которое автоматически заносится в столбец при добавлении к таблице новой записи.

В качестве значения по умолчанию можно указать:

- константу (литерал) — в столбец заносится указанное значение (должно иметь тип, совместимый с типом столбца);
- `NULL` — в столбец заносится нулевое значение;
- `USER` — в столбец заносится имя текущего пользователя (для столбцов строковых типов).

Операнд `NOT NULL` указывает, что столбец не может быть пустым и в обязательном порядке должен содержать значение допустимого типа и диапазона.

Замечание

При программировании инструкций нельзя допускать взаимоисключающие конструкции, например, одновременно указывать операнды `DEFAULT NULL` и `NOT NULL`.

Операнд `COLLATE` определяет порядок сортировки строковых значений.

Операнд `CHECK` позволяет установить для столбца разнообразные условия, управляющие его значениями. При редактировании записей столбцу разрешается присваивать только те значения, которые удовлетворяют заданным условиям. Эти условия могут быть как простым сравнением, так и сложной комбинацией нескольких критериев.

Операнд `CHECK` имеет следующий формат:

```
CHECK
   {<Выражение> <Операция сравнения > {<Выражение> | (<Отбор1>)}
   | <Выражение> [NOT] <Мин. значение> AND <Макс. значение>
   | <Выражение> [NOT] LIKE <Выражение1> [ESCAPE <Выражение2>]
   | <Выражение> [NOT] IN
    (<Выражение1> [, <Выражение2>] ... [, <ВыражениеN>] | (<ОтборМ>))
   | <Выражение> IS [NOT] NULL
   | <Выражение> [NOT] {<Операция сравнения>|ALL|SOME|ANY} (<ОтборМ>)
   | EXISTS (<ОтборМ>)
   | SINGULAR (<ОтборМ>)
   | <Выражение> [NOT] CONTAINING <Выражение1>
   | <Выражение> [NOT] STARTING [WITH] <Выражение1>}
```

Условия, ограничивающие возможные значения столбца, по сути и синтаксису идентичны условиям, задаваемым при отборе записей в инструкции `SELECT`, и

здесь подробно не рассматриваются. Более того, при задании условий в операнде CHECK можно использовать инструкцию SELECT. Как и в инструкции SELECT, условия можно объединять (комбинировать) с помощью логических операций NOT, OR и AND.

Выражение должно иметь тип, совместимый с типом столбца, например, числовой или строковый. В выражении используются операнды, знаки операций (операторы), круглые скобки и функции. Операндами могут быть значения и имена столбцов. Состав функций (SUM, MAX, MIN, UPPER и др.) практически не отличается от состава функций, включенных в локальную версию языка SQL. Отметим функцию GEN_ID, которая возвращает уникальное значение генератора и служит для установки значений ключа. В частном случае выражение представляет собой имя столбца или значение.

Элементы, обозначенные как Отбор, представляют собой совокупность значений, отобранных с помощью инструкции SELECT. Условие отбора записей, заданное в этой инструкции, для элемента Отбор1 должно возвращать одно значение или ни одного, а для элемента ОтборМ — несколько значений, в частном случае ни одного. Инструкция SELECT составляется по обычным правилам и служит для отбора записей из различных таблиц.

Чтобы задать диапазон ограничений для значений столбца, используется операция сравнения или конструкция вида

```
BETWEEN <Минимальное значение > AND <Максимальное значение >
```

Например:

```
CREATE TABLE Test
      (Name       VARCHAR(20) NOT NULL,
      Price       FLOAT CHECK (Price > 0),
      Number      INTEGER CHECK (Number BETWEEN 1 AND 1000));
```

Для таблицы Test значение столбца Name не может быть пустым, значение столбца Price должно быть положительным, а значение столбца Number — находиться в диапазоне 1 .. 1000.

С помощью условия можно обеспечить для столбца установку одного из значений, содержащихся в определенном наборе (списке). Набор значений бывает фиксированным (статическим) и задается непосредственно в конструкции IN или формируется динамически с помощью инструкции SELECT. В последнем случае состав набора зависит от результата работы инструкции SELECT, которая выполняется каждый раз при модификации записей таблицы.

Например, инструкция

```
CREATE TABLE Test2
      (Name       VARCHAR(20),
      Position    VARCHAR(20)
            CHECK (Post IN ("Директор", "Менеджер", "Бухгалтер"),
      Attrib      VARCHAR(20)
            CHECK (Attrib IN (SELECT Attrib From Positions)));
```

для полей таблицы Test2 устанавливает список возможных значений: значение поля Position должно входить в фиксированный набор из трех строковых констант. Недостатком подобного ограничения является то, что оно действует на уровне таблицы и в приложении не может быть изменено или отменено. Поэтому фиксированный набор обычно задают для столбцов, допустимые значения которых известны заранее и изменяются редко. В остальных случаях удобнее определять возможные значения динамически, как это сделано для поля Attrib: набор допустимых величин для этого поля образуется совокупностью всех значений одноименного поля таблицы Positions.

При необходимости можно сформировать набор значений на основании записей, удовлетворяющих определенным условиям. Например, если требуются атрибуты должностей, которым соответствует оклад, превышающий 2500 рублей, то описание поля Attrib будет иметь следующий вид:

```
Attrib VARCHAR(20)
CHECK (Attrib IN (SELECT Attrib From Positions WHERE Salary > 2500)));
```

Ограничение, заданное операндом CHECK, указывается для конкретного столбца и располагается в строке его описания до начала описания следующего столбца. (Это остается верным и в случае, когда описание столбца занимает несколько строк.) Такое же ограничение можно установить и на уровне таблицы, разместив его после описания всех столбцов. Единственным отличием является то, что для ограничения, задаваемого на уровне столбца, можно указать имя. Операнд CHECK, используемый на уровне таблицы, имеет следующий формат:

```
[CONSTRAINT <Имя ограничения >]
CHECK <Условия>
```

Ограничения на значения столбцов, как и другие ограничения, именуются. Имя может быть задано при определении ограничения в операнде CONSTRAINT, но это не является обязательным: если для ограничения столбца имя не задано, то это ограничение получает имя по умолчанию.

В качестве примера рассмотрим создание таблицы с ограничениями:

```
CREATE TABLE Test5
      (Name        VARCHAR(20),
      OrderDate    DATE,
      PerformDate  DATE,
      CHECK (PerformDate >= OrderDate));
```

В таблице Test5 хранится информация о заказе товара. Столбец OrderDate содержит дату заказа, а столбец PerformDate — дату выполнения заказа. Так как заказ не может быть выполнен раньше, чем он сделан, для столбца PerformDate на уровне таблицы установлено соответствующее ограничение. Имя для ограничения не задано. Это ограничение можно указать и при описании столбца PerformDate.

Добавить ограничение к уже существующей таблице можно с помощью инструкции вида

```
ALTER TABLE <Имя таблицы>
      ADD CHECK ([CONSTRAINT <Имя ограничения >] <Условия>);
```

Для удаления ограничения предназначается инструкция

```
ALTER TABLE <Имя таблицы> DROP <Имя ограничения>;
```

Можно произвести удаление и с помощью программ типа SQL Explorer.

Описание ключей

В таблицах InterBase ключи разделяются на первичные и уникальные (для главных таблиц) и внешние (для подчиненных таблиц). Первичный ключ по построению и использованию не отличается от ключа (главного ключа) рассмотренных ранее таблиц Paradox. Напомним, что ключ предназначен для обеспечения уникальности записей и ссылочной целостности БД.

Для описания первичного ключа используется операнд `PRIMARY KEY` формата:

```
PRIMARY KEY (<Список столбцов ключа>)
```

Имена столбцов, образующие ключ, перечисляются в списке через запятую. Например, инструкция

```
CREATE TABLE Personnel2
       (Code       INTEGER NOT NULL,
       Name        VARCHAR(30),
       PRIMARY KEY (Code));
```

создает таблицу `Personnel2`, для которой по столбцу `Code` строится первичный ключ. Отметим, что для ключевого столбца задан описатель `NOT NULL`.

Замечание

В связи с тем что в таблицах InterBase отсутствует такой тип, как автоинкрементный, в обязанности разработчика входит гарантия уникальности значений ключевых столбцов. Уникальность ключа обеспечивается на программном уровне через реализацию в приложении соответствующих алгоритмов выбора. Кроме того, можно использовать генераторы, которые будут рассмотрены ниже.

Построить первичный ключ можно также несколько иным способом, указав описатель `PRIMARY KEY` при определении соответствующего столбца:

```
CREATE TABLE Personnel2
      (Code     INTEGER NOT NULL PRIMARY KEY,
      Name      VARCHAR(30));
```

Как и в предыдущем примере, здесь создается таблица `Personnel2` с первичным ключом по столбцу `Code`. Оба приведенных способа определения ключа являются эквивалентными.

Таблица может иметь только один первичный ключ, однако кроме него можно определить еще несколько уникальных ключей. Для описания уникального ключа используется операнд UNIQUE следующего формата:

```
UNIQUE (<Список столбцов ключа>)
```

Так, инструкция:

```
CREATE TABLE Position
      (Code       INTEGER NOT NULL,
      Position    VARCHAR(20) NOT NULL,
      PRIMARY KEY (Code),
      UNIQUE (Position));
```

создает таблицу Position, для которой строятся два ключа — первичный ключ по столбцу Code и уникальный ключ по столбцу Position. Отметим, что таблица и один из ее столбцов могут иметь одинаковые имена (в данном случае это имя Position).

По своему назначению и использованию уникальный ключ не отличается от первичного и служит для обеспечения единственности значений ключевого столбца (столбцов) и ссылочной целостности.

Описание и использование внешнего ключа рассмотрено в следующем разделе.

Определение ограничений ссылочной целостности

Напомним, что действие ограничений ссылочной целостности заключается в следующем: если для записи главной таблицы имеются записи в подчиненной таблице (таблицах), то эту запись нельзя удалить, а также изменить значения столбцов, образующих ключ.

Определение ограничения ссылочной целостности имеет следующий формат:

```
[CONSTRAINT <Имя ограничения >]
FOREIGN KEY (<Список столбцов ключа>)
REFERENCES <Имя главной таблицы> [<Список столбцов ключа главной таблицы >]
```

При задании ограничений ссылочной целостности ключу (первичному или уникальному) главной таблицы ставится в соответствие внешний ключ подчиненной таблицы. Для описания внешнего ключа используется операнд FOREIGN KEY. Список этого операнда задает столбцы, по которым строится внешний ключ. Если внешний ключ используется для задания ограничений ссылочной целостности (для чего он собственно и предназначен), то его список столбцов должен соответствовать списку столбцов ключа главной таблицы.

В операнде REFERENCES указывается главная таблица. Описания столбцов ключа (первичного или уникального) главной таблицы и внешнего ключа подчиненной таблицы должны полностью совпадать.

Ограничения ссылочной целостности, как и другие ограничения, именуются. Имя может быть задано при определении ограничения в операнде `CONSTRAINT`, что, вообще говоря, не обязательно: если для ограничения ссылочной целостности не задано имя, то оно получает имя по умолчанию.

Пример задания ограничений ссылочной целостности:

```
CREATE TABLE Store
      (S_Code   INTEGER NOT NULL PRIMARY KEY,
      S_Name    VARCHAR(20) NOT NULL,
      S_Price   FLOAT,
      S_Number  Float);

CREATE TABLE Cards
      (C_Code     INTEGER NOT NULL PRIMARY KEY,
      C_Code2     INTEGER NOT NULL,
      C_Move      VARCHAR(20) NOT NULL,
      C_Date      DATE,
      CONSTRAINT  rStoreCards
FOREIGN KEY (C_Code2) REFERENCES Store);
```

Создаются две таблицы `Store` и `Cards`, в каждой из которых определен первичный ключ. С таблицей `Store` связывается таблица `Cards`, для чего в ней по полю `C_Code2` определен внешний ключ и задано ограничение ссылочной целостности с именем `rStoreCards`.

Замечание

При использовании программы IBConsole приведенные инструкции нужно вводить и выполнять поочередно или использовать файл сценария.

Аналогично можно определить для таблицы несколько различных ограничений ссылочной целостности.

Рассмотренные ограничения ссылочной целостности для связанных таблиц задаются на физическом уровне — уровне таблиц. Вместо этого на программном уровне можно использовать триггеры и реализовать с их помощью, например, каскадное удаление записей. В таком случае ограничения ссылочной целостности не определяются, и их требуется удалить, если они были ранее созданы.

Удалить ограничение ссылочной целостности можно следующей инструкцией:

```
ALTER TABLE <Имя таблицы>
      DROP <Имя ограничения ссылочной целостности>;
```

или из программы типа SQL Explorer.

Использование индексов

Создание и удаление индексов выполняется инструкциями CREATE INDEX и DROP INDEX соответственно. Инструкция создания индекса CREATE INDEX имеет формат:

```
CREATE [UNIQUE] {ASCENDING] [DESCENDING] INDEX
      <Имя индекса> ON <Имя таблицы > (<Имя столбца>, ..., [<Имя столбца>]);
```

В отличие от версии языка SQL для локальных БД, для индекса таблицы InterBase дополнительно можно задать несколько описателей:

- UNIQUE — индекс (как и ключ) требует уникальности значений столбца (столбцов), по которому он построен;
- ASCENDING — индексные столбцы сортируются в порядке возрастания значений (по умолчанию);
- DESCENDING — индексные столбцы сортируются в порядке убывания значений.

Так, инструкция

```
CREATE DESCENDING INDEX
       indNamePosition ON Personnel (Name, Position)
```

создает для таблицы Personnel индекс indNamePosition, построенный по полям Name и Position. Для индекса устанавливается сортировка в порядке убывания значений.

В остальном операции с индексами для таблиц InterBase и соответствующие средства языка SQL не отличаются от ранее рассмотренных для локальных БД.

Для каждого ключа таблицы автоматически строится соответствующий индекс, и ему присваивается имя по умолчанию. В отличие от обычного индекса, индекс, построенный по ключу, нельзя удалить инструкцией DROP INDEX: для этого нужно выполнить реструктуризацию таблицы инструкцией ALTER TABLE.

Описание индекса indNamePosition таблицы Personnel, созданного в предыдущем примере, имеет следующий вид:

```
INDNAMEPOSITION INDEX ON PERSONNEL (NAME, POSITION)
```

В свою очередь, строки

```
RDB$FOREIGN4 INDEX ON CARDS(C_CODE)
RDB$PRIMARY3 UNIQUE INDEX ON STORE(S_CODE)
```

описывают индексы для связанных таблиц, построенные на основании первичного ключа таблицы STORE и внешнего ключа таблицы CARDS. По умолчанию индексы получили имена RDB$FOREIGN4 и RDB$PRIMARY3 соответственно.

Приведенные описания получены с помощью программы IBConsole. Отметим, что просмотреть описания индексов можно и с помощью других программ, например, SQL Explorer, а также программно — с помощью инструкции

```
SHOW INDEX [<Имя таблицы>];
```

Эта инструкция выводит для указанной таблицы описания всех индексов, в том числе построенных по ключам. Если имя таблицы отсутствует, то выводятся описания всех индексов БД.

Как отмечалось, индекс обеспечивает быстрый доступ к данным, хранящимся в столбце (столбцах), для которого создан этот индекс. Это обеспечивается за счет использования индексно-последовательного метода доступа к данным.

При оценке эффективности (полезности) индекса применяется понятие глубины индекса. *Глубина индекса* представляет собой значение, показывающее максимальное число операций, которые необходимо выполнить при доступе к записи с использованием этого индекса. Допустимая глубина индекса не должна превышать двух операций, в противном случае говорят о разбалансировке индекса. *Разбалансировка индекса* может наступить при добавлении и удалении большого количества записей в БД. Отметим, что на глубину индекса также влияет размер страницы БД, который задается при ее создании. С увеличением размера страницы глубина индекса может уменьшиться.

Для уменьшения глубины индекса и устранения его разбалансировки нужно перестроить индекс путем выполнения следующей последовательности инструкций:

```
ALTER INDEX <Имя индекса> INACTIVE;
ALTER INDEX <Имя индекса> ACTIVE;
```

Первая инструкция деактивизирует (отключает) указанный индекс, а вторая — вновь активизирует (включает) его. Перестроить индекс можно и более кардинальным способом, удалив его и создав заново. Отметим, что вновь созданный индекс является активным.

Использование доменов

Домен представляет собой именованное описание столбца. После определения домена его имя можно использовать при описании других столбцов. Домен удобен в случаях, когда несколько столбцов, в том числе принадлежащих различным таблицам, имеют одинаковое описание. Если провести сравнение с языком Object Pascal, то аналогом домена является тип данных. Таким образом, домен можно рассматривать как *шаблон* описания столбца.

Перед использованием домена его нужно создать с помощью инструкции

```
CREATE DOMAIN <Имя домена>
      [AS] <Описание домена>
```

Имя созданного домена затем можно включать в описания столбцов:

```
<Имя столбца> <Имя домена>
```

Описание домена не отличается от ранее рассмотренного описания столбца и может включать различные ограничения, применимые к столбцу. Однако в отличие от ограничений для столбца, имя домена нельзя использовать в выражениях для ограничений: вместо имени домена в них используется слово `VALUE`.

Пример использования доменов:

```
CREATE DOMAIN D_Position AS VARCHAR(20) NOT NULL;

CREATE DOMAIN D_Price AS FLOAT CHECK(VALUE > 0);

CREATE TABLE Position
      (Code        INTEGER NOT NULL PRIMARY KEY,
      Position     D_Position,
      Note         VARCHAR(50));

CREATE TABLE Personnel
      (Code       INTEGER NOT NULL PRIMARY KEY,
      Name        VARCHAR(30),
      Position    D_Position,
      Salary      FLOAT,
      BirthDay    DATE);
```

Домен `D_Position` представляет собой описание строкового столбца длиной не более 20 символов, который не может быть пустым. Этот домен использован при описании столбцов `Position` таблиц `Position` и `Personnel`. Второй домен `D_Price` представляет собой числовой столбец, значение которого должно быть положительным. Это ограничение задано конструкцией `CHECK(VALUE > 0)`. Для приводимых таблиц домен `D_Price` не нужен, а создан с целью последующего использования.

Использование просмотров

Просмотр (view) является логической (виртуальной) таблицей, записи в которую отобраны с помощью инструкции `SELECT`. Преимущество просмотра заключается в том, что можно один раз отобрать записи и использовать их в дальнейшем без повторного выполнения инструкции `SELECT`. Это выгодно при частом выполнении одинаковых запросов, особенно тех, в которых реализованы сложные условия отбора. Иногда просмотр также называют *представлением*.

Просмотр создается следующей инструкцией:

```
CREATE VIEW <Имя просмотра> [<Список столбцов>]
      AS <Инструкцию SELECT> [WITH CHECK OPTION];
```

Имя просмотра после его создания можно использовать как имя физической таблицы. Состав столбцов просмотра определяет список, который следует за именем просмотра. Если список столбцов не задан, то в просмотр отбираются все столбцы таблиц, указанных в инструкции `SELECT`. Инструкция `SELECT` составляется обычным образом.

Просмотр может быть редактируемым (модифицируемым) или нередактируемым. Для того чтобы записи просмотра можно было изменять, удалять и добавлять, должны выполняться как минимум следующие условия:

- в просмотр отобраны записи только из одной таблицы;
- в просмотр включены все столбцы таблицы, имеющие ограничение `NOT NULL`;
- в инструкции `SELECT` не были использованы статистические функции, хранимые процедуры, пользовательские функции, описатель `DISTINCT`, операнд `HAVING`, а также соединение таблиц.

Операнд `WITH CHECK OPTION` для редактируемого просмотра запрещает добавление записей, не удовлетворяющих условиям отбора, заданным в инструкции `SELECT`.

Для иллюстрации использования просмотра рассмотрим следующий пример:

```
CREATE VIEW vStore AS
      SELECT S_Name, C_Number
            FROM Store
            WHERE S_Number >= 0.5

SELECT * FROM vStore
```

Создается просмотр, состоящий из столбцов `S_Name` (название) и `C_Number` (количество товара) таблицы `Store`. В просмотр отбираются записи, для которых количество товара равно или превышает `0.5`. Далее в инструкции `SELECT` просмотр `vStore` используется как обычная таблица.

Удалить просмотр можно следующей инструкцией:

```
DROP VIEW <Имя просмотра>;
```

Например:

```
DROP VIEW vStore;
```

Хранимые процедуры

Хранимая процедура представляет собой подпрограмму, расположенную на сервере и вызываемую из приложения клиента. Использование этих объектов увеличивает скорость доступа к БД по следующим причинам:

- вместо текста запроса, который может быть достаточно длинным, серверу передается по сети относительно короткое обращение к хранимой процедуре;
- хранимая процедура, в отличие от запроса, не требует предварительной синтаксической проверки.

Еще одним преимуществом при обращении к хранимым процедурам является то, что будучи общими для всех приложений-клиентов, они реализуют единые для них правила работы с БД.

Для выполнения процедуры, хранимой на сервере, в приложении используется компонент StoredProc.

Создание и удаление хранимых процедур

Хранимая процедура создается инструкцией

```
CREATE PROCEDURE
      <Имя процедуры> [(<Список входных параметров>)]
      [RETURNS (<Список выходных параметров>)]
      AS <Тело процедуры>
```

После имени процедуры идет необязательный список входных параметров, с помощью которых из приложения в процедуру передаются исходные данные. В свою очередь, список выходных параметров, с помощью которых в приложения передаются результаты выполнения процедуры, указывается после описателя RETURNS. Для каждого параметра указываются его имя и тип (через пробел), разделителем параметров в списке служит запятая.

```
<Список параметров> =
       [<Имя параметра1> <Тип параметра1>,]
       ...
       [<Имя параметраN> <Тип параметраN>]
```

При задании имен процедур и параметров (как и других объектов) для удобства целесообразно использовать мнемонические правила. Например, имя процедуры можно начинать с префикса p (от Procedure), имя входного параметра — с ip (от Input Parameter), а имя выходного параметра — с op (от Output Parameter).

При использовании параметра в выражениях тела процедуры перед его именем нужно указывать двоеточие (:).

Тело процедуры состоит из двух частей — описательной и исполнительной — и имеет следующий формат:

```
[<Объявление переменных>]
BEGIN
      <Инструкция>
      ...
      [<Инструкция>]
END
```

В описательной части объявляются переменные, используемые внутри процедуры. Эти переменные являются локальными и по окончании работы процедуры теряют свои значения. В исполнительной части располагаются инструкции, выполняющие необходимые действия. Как и в языке Pascal, инструкции располагаются между ключевыми словами BEGIN и END. В теле процедуры должна содержаться как минимум одна инструкция.

Созданную хранимую процедуру можно удалить или изменить. Удаляется процедура инструкцией

```
DROP PROCEDURE <Имя процедуры>
```

Изменение процедуры выполняется инструкцией `ALTER PROCEDURE`, имеющей такой же формат, как и инструкция `CREATE PROCEDURE`. После выполнения инструкции `ALTER PROCEDURE` предыдущее описание хранимой процедуры с указанным именем заменяется на новое описание.

Язык хранимых процедур

Для написания хранимых процедур и триггеров используется язык хранимых процедур. Язык хранимых процедур сервера InterBase представляет собой процедурный алгоритмический язык, а его синтаксис имеет много общего с синтаксисом языка Pascal, который составляет основу Delphi. Язык хранимых процедур включает в свой состав инструкции, позволяющие управлять ходом вычислительного процесса (условную инструкцию и инструкцию цикла). Кроме того, язык хранимых процедур обладает рядом возможностей языка SQL.

Рассмотрим кратко основные составляющие языка хранимых процедур.

В текст процедуры (триггера) можно вставлять комментарии. Для ограничения комментариев используются символы `/*` и `*/`.

В язык хранимых процедур включены инструкции:

- объявления переменных;
- присваивания;
- условные;
- составные;
- цикла;
- выбора записи;
- выбора нескольких записей (набора данных);
- возврата значений;
- выхода из процедуры;
- вызова процедуры;
- посылки сообщения.

Инструкции должны заканчиваться точкой с запятой (кроме составной инструкции).

Инструкция объявления переменных имеет следующий формат:

```
DECLARE VARIABLE <Имя переменной> <Тип переменной>;
```

Допустимыми типами для переменных являются типы столбцов InterBase, например, `VARCHAR` или `DATE`.

Переменные являются локальными и действуют с момента входа в процедуру до момента выхода из нее. Например, в командах

```
DECLARE VARIABLE Number INTEGER;
DECLARE VARIABLE NewString VARCHAR(10);
DECLARE VARIABLE WorkDate DATE;
```

объявляются три переменных `Number`, `NewString` и `WorkDate` целого типа, типа "строка" и "дата" соответственно.

Инструкция присваивания предназначена для установки значения переменной и имеет вид

```
<Имя переменной> = <Выражение>;
```

При выполнении этой инструкции вычисляется значение выражения и присваивается переменной, имя которой задано в левой части инструкции. Можно указывать имена локальных переменных и выходных параметров, перед именем выходного параметра двоеточие не ставится. Переменная и выражение должны иметь совместимые типы, в противном случае при выполнении процедуры возникает ошибка. В качестве операндов выражения можно использовать значения, имена переменных и параметров, встроенные и пользовательские функции, а также вызовы генераторов.

Так, в примере

```
DECLARE VARIABLE n INTEGER;
DECLARE VARIABLE s VARCHAR(30);
...
n = 17;
s = UPPER(s);
```

целочисленной переменной `n` присваивается значение `17`, а символы строки `s` переводятся в верхний регистр.

Замечание

Если тип выражения не соответствует типу переменной, то ошибка возникает при выполнении хранимой процедуры, а не при ее создании, что нужно учитывать при отладке хранимых процедур.

Условная инструкция в общем случае позволяет организовать ветвление вычислительного процесса и имеет следующий формат:

```
IF (<Условие>) THEN <Инструкция1> [ELSE <Инструкция2>];
```

`Условие` представляет собой выражение логического типа. Условная инструкция выполняется так: если условие истинно (имеет значение `True`), то выполняется `Инструкция1`, в противном случае — `Инструкция2`. Обе указанные инструкции могут быть составными.

Условная инструкция может быть записана в сокращенной форме, когда слово `ELSE` и инструкция после него отсутствуют. В этом случае при невыполнении условия управление сразу передается на инструкцию, следующую за условной.

Составная инструкция представляет собой группу из произвольного числа любых инструкций, ограниченную операторными скобками BEGIN и END. Составная инструкция имеет следующий формат:

```
BEGIN <Инструкция1> ... <ИнструкцияN> END
```

Независимо от числа входящих в нее инструкций, составная инструкция воспринимается как единое целое и может располагаться в любом месте, где допускается наличие инструкции. Наиболее часто составная инструкция используется в условных инструкциях и инструкциях цикла. В конце составной инструкции точка с запятой не ставится. Составная инструкция объединяет также все инструкции тела процедуры или триггера.

Пример использования условной и составной инструкций:

```
IF (x < 0)
      THEN BEGIN n = 0; s = ""; END
      ELSE n = 10;
```

Если значение числовой переменной x меньше нуля, то значения переменных n и s сбрасываются. Соответствующие инструкции присваивания объединены в составную инструкцию. Если условие не выполняется, то числовой переменной n присваивается новое значение, а значение строковой переменной s остается без изменений.

Инструкция цикла (повтора) обеспечивает повтор группы инструкций при соблюдении некоторых условий. Инструкция цикла имеет следующий формат:

```
WHILE (<Условие>) DO <Инструкция>;
```

Инструкция, расположенная после слова DO, будет выполняться до тех пор, пока соблюдается указанное условие. Если в цикле требуется выполнить несколько инструкций, то их нужно объединить в составную инструкцию. Например:

```
S = 0;
n = 1;
WHILE (n <= 10) DO
      BEGIN
            s = s + n;
            n = n + 1;
      END
```

Подсчитывается сумма чисел от 1 до 10. Перебор чисел и накапливание суммы выполняется в цикле.

Отметим, что на практике для подсчета суммы чисел удобнее использовать не хранимую процедуру сервера, а процедуру, расположенную в модуле Delphi. Этот и некоторые следующие примеры приводятся в упрощенном виде с целью показать отдельные инструкции языка хранимых процедур.

Инструкция выбора записи представляет собой инструкцию SELECT, которая возвращает одну строку, поэтому ее называют также инструкцией выбора записи. Значения столбцов возвращаемой строки присваиваются указанным переменным

или параметрам. При своем выполнении инструкция выбора записи (как и инструкция SELECT) отбирает несколько строк, но возвращает только одну.

Формат инструкции выбора записи, по сравнению с форматом инструкции SELECT языка SQL, дополнен операндом:

```
INTO :<Имя1>, ... [:<ИмяN>]
```

Имя после двоеточия, указывает переменную или выходной параметр, которым должны быть присвоены значения столбцов записи, полученной в результате выполнения инструкции SELECT. Эта инструкция часто используется совместно со статистическими функциями, такими как SUM (сумма) или AVG (среднее значение).

Рассмотрим использование инструкции SELECT в хранимой процедуре на примере:

```
CREATE PROCEDURE pSalary
      RETURNS (opSum FLOAT, opAvg FLOAT)
      AS
      BEGIN
            SELECT SUM(Salary), AVG(Salary)
            FROM Personnel
            INTO :opSum, :opAvg;
      END
```

Создается хранимая процедура pSalary, в которой для сотрудников организации (таблица Personnel) подсчитываются общая сумма зарплаты и средняя зарплата. Полученные значения присваиваются выходным параметрам opSum и opAvg, которые возвращают полученные значения в вызывающую программу. Входных параметров у процедуры нет.

Инструкция выбора нескольких записей также представляет собой инструкцию SELECT, способную возвращать несколько записей (в частности, ни одной). Инструкция выбора нескольких записей имеет формат:

```
FOR <Инструкция выбора записи> DO <Инструкция>;
```

Инструкция выбора нескольких записей может задавать не только отбор, но и обработку записей. После отбора записей согласно инструкции SELECT для каждой из них выполняется инструкция, указанная после слова DO.

Пример хранимой процедуры с инструкцией выбора:

```
CREATE PROCEDURE pCountCode
      RETURNS (opSum INTEGER)
      AS
      DECLARE VARIABLE n INTEGER;
      DECLARE VARIABLE x INTEGER;
      BEGIN
            x = 0;
            FOR SELECT Code FROM List
                  INTO n
```

```
            DO x = x + n;
        opSum = x;
    END
```

Создается хранимая процедура `pCountCode`, которая вычисляет сумму значений целочисленного столбца `Code` таблицы `List`. Инструкция `SELECT` обеспечивает отбор записей таблицы, а инструкция присваивания увеличивает значение суммы `x` на значение столбца `Code` очередной записи. После перебора всех записей результат возвращается через выходной параметр `opSum`.

Часто после слова `DO` указывается инструкция `SUSPEND` возврата значений, которая передает в вызывающее приложение или хранимую процедуру значения выходных параметров и имеет вид:

```
SUSPEND;
```

Обычно инструкция `SUSPEND` используется в инструкции `SELECT` выбора нескольких записей хранимой процедуры. Совместное использование этих инструкций фактически обеспечивает возврат в приложение совокупности записей, представляющих собой набор данных. Поэтому инструкцию выбора нескольких записей называют также *инструкцией, возвращающей набор данных.*

Рассмотрим пример:

```
CREATE PROCEDURE pSalary2 (ipSalaryMin FLOAT, ipSalaryMax FLOAT)
    RETURNS (opName VARCHAR(20), opSalary FLOAT)
    AS
    BEGIN
        FOR SELECT Name, Salary
        FROM Personnel
        WHERE Salary >= :ipSalaryMin
        AND Salary <= :ipSalaryMax
            INTO :opName, :opSalary
            DO SUSPEND;
    END
```

Создается хранимая процедура `pSalary2`, которая возвращает набор данных. Совокупность записей этого набора образуют записи таблицы `pSalary2`, соответствующие сотрудникам организации, у которых оклад принадлежит заданному диапазону. Границы диапазона определяют входные параметры `ipSalaryMin` и `ipSalaryMax`. Набор данных содержит столбцы `Name` и `Salary`.

Замечание

Хранимую процедуру, обеспечивающую возврат набора данных, можно вызывать с помощью средств, позволяющих получить одну строку результатов. При этом будет обрабатываться только одна (первая) запись получаемого набора. Такими средствами являются инструкция `EXECUTE PROCEDURE` вызова процедуры и компонент `StoredProc`, позволяющий обратиться к процедуре, хранимой на сервере. Обычно их используют только при отладке хранимых процедур, возвращающих набор данных.

Инструкция выхода из процедуры `EXIT` прекращает выполнение хранимой процедуры и осуществляет передачу управления в вызывающую программу или процедуру. Если инструкция выхода отсутствует, то завершение процедуры и передача управления в вызывающую программу происходят после выполнения последней инструкции процедуры. Например, инструкция

```
CREATE PROCEDURE pDivide (ipA FLOAT, ipB FLOAT)
      RETURNS (opRes FLOAT)
      AS
      BEGIN
            IF (pB = 0) THEN BEGIN pRes = 0; EXIT; END
            opRes =: ipA / ipB;
      END
```

создает хранимую процедуру `pDivide`, выполняющую деление двух чисел. Если делитель равен нулю, то результат обнуляется, и выполнение процедуры заканчивается.

Инструкция `EXECUTE` предназначена для вызова из процедуры другой хранимой процедуры и имеет формат:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
       [RETURNING_VALUES (<Список выходных параметров>)]
```

Эта инструкция вызывает хранимую процедуру с указанным именем, входными и выходными параметрами. В частном случае у вызываемой процедуры параметры могут отсутствовать. Так, инструкция:

```
CREATE PROCEDURE pTest
      RETURNS (opResult FLOAT)
      AS
      BEGIN
            EXECUTE PROCEDURE pDivide (100, 20)
                  RETURNING_VALUES (opResult);
      END
```

создает хранимую процедуру `pTest`, которая в свою очередь вызывает ранее созданную процедуру `pDivide` деления двух чисел. В качестве входных параметров для деления передаются числа 100 и 20. Результат, возвращаемый процедурой `pDivide`, присваивается выходному параметру `opResult` процедуры `pTest`.

Инструкция `POST_EVENT` предназначена для отправки сообщения приложениям, связанным с сервером. Сообщение посылается всем приложениям при возбуждении в сервере определенного события. Инструкция отправки сообщения имеет следующий формат:

```
POST_EVENT "<имя события>";
```

Например, инструкцией

```
POST_EVENT "evInsertStore";
```

приложениям, связанным с сервером, посылается сообщение `evInsertStore`.

Отметим, что для получения соответствующего сообщения приложение должно предварительно зарегистрироваться на сервере. Для регистрации приложения и обеспечения посылки им сообщений можно использовать компонент `IBEvents` со страницы **InterBase** Палитры компонентов.

Виды хранимых процедур

По числу строк, возвращаемых в качестве результата, можно выделить следующие виды хранимых процедур:

- возвращающие одну строку;
- возвращающие несколько строк.

Процедуры, возвращающие одну строку, практически не отличаются от процедур языка Pascal и обеспечивают возврат значений выходных параметров. Такие хранимые процедуры также называют *процедурами действия*.

Процедуры, возвращающие несколько строк, передают набор данных, записями в котором являются строки результатов. Такие хранимые процедуры также называют *процедурами выбора*. В их теле размещаются совместно используемые инструкции выбора нескольких записей и возврата значений — `FOR SELECT ... DO ... SUSPEND`, которые и обеспечивают отбор требуемых записей и построчную передачу значений их столбцов в точку вызова.

Вызов хранимой процедуры представляет собой обращение к процедуре с передачей исходных данных для обработки и последующим получением результатов. Исходные данные передаются через входные параметры, а результаты возвращаются через выходные параметры. В зависимости от вида хранимой процедуры различаются способы ее вызова.

Вызов хранимой процедуры выбора

Хранимая процедура выбора возвращает набор данных, состоящий в общем случае из нескольких записей. Поэтому при ее вызове нужно обеспечить возможность приема совокупности записей. В связи с этим хранимая процедура выбора, как правило, вызывается с помощью инструкции выбора `SELECT`, внутри которой размещается вызов процедуры. Например:

```
SELECT * FROM pSalary2 (2000, 3000)
```

Здесь вызывается хранимая процедура `pSalary2`, возвращающая фамилии и оклады сотрудников, значение оклада которых принадлежит заданному диапазону. Результат выполнения приведенной инструкции `SELECT` имеет следующий вид:

```
OPNAME           OPSALARY
============= =============
Иванов П.Л.      2000
Петров О.Д.      2700
Кузнецов В.Ф.    2400
Васина Ю.Б.      2200
```

Замечание

Хранимую процедуру выбора в программе IBConsole также можно вызвать инструкцией `EXECUTE PROCEDURE`, например, так:

```
EXECUTE PROCEDURE pSalary2 (2000, 3000)
```

В этом случае будет получена только первая запись набора данных. Напомним, что на практике такая инструкция вызова процедуры обычно используется только при отладке хранимых процедур выбора.

В приложении инструкция `SELECT`, вызывающая хранимую процедуру выбора, выполняется с помощью набора данных `Query`, для чего используется его свойство `SQL`.

Рассмотрим пример:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Query1.Close;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM pSalary2 (2000, 3000)');
Query1.Open;
end;
```

При нажатии кнопки `Button1` свойству `SQL` компонента `Query1` присваивается код вызова хранимой процедуры `pSalary2`, затем запрос выполняется. В результате набор данных `Query1` содержит записи, удовлетворяющие заданному в процедуре условию отбора. Отметим, что столбцы набора данных образуют выходные параметры процедуры, в данном случае ими являются столбцы `opName` и `opSalary`.

Замечание

Для соединения с удаленной БД в форме, кроме набора данных `Query`, следует разместить компонент `Database`. (Вопросы соединения с удаленной БД из приложения были рассмотрены в *главе 13*.)

В приведенном примере SQL-запрос является статическим, т. к. входные параметры, ограничивающие диапазон оклада, передаются процедуре через явно указанные значения `2000` и `3000`. Чтобы менять границы диапазона, можно считывать их значения, например, из компонентов `Edit` (полей редактирования). Например:

```
Query1.SQL.Add('SELECT * FROM pSalary2 (' + Edit1.Text + ',' +
       Edit2.Text + ')');
```

Значения нижней и верхней границ диапазона для оклада вводятся в поля редактирования `Edit1` и `Edit2` соответственно.

Значения входных параметров процедуры также можно передавать через параметры компонента `Query1` (свойство `Params`). При этом запрос компонента `Query1` будет иметь вид:

```
'SELECT * FROM pSalary2 (:pMin, :pMax)'
```

Здесь границы диапазона для оклада задаются двумя параметрами: `pMin` — для нижней, а `pMax` — для верхней границы. Эти параметры являются входными и предназначены для передачи вещественных значений. Поэтому для них нужно задать следующие значения свойств:

- `DataType` (тип данных) — `ftFloat`;
- `ParamType` (тип параметра) — `ptInput`.

Управление свойствами параметров осуществляется с помощью Инспектора объектов и Редактора параметров SQL-запроса (рис. 14.2), который вызывается щелчком в области значения свойства `Params` в окне Инспектора объектов. При необходимости для параметров можно задать значения (свойство `Value`).

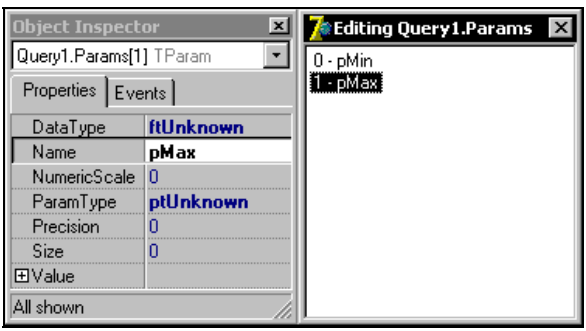


Рис. 14.2. Редактирование параметров SQL-запроса

При выполнении приложения значения параметров можно установить так:

```
Query1.ParamByName('pMin').Value := 2000;
Query1.ParamByName('pMax').Value := 3000;
```

или

```
Query1.ParamByName('pMin').Value := StrToFloat(Edit1.Text);
Query1.ParamByName('pMax').Value := StrToFloat(Edit2.Text);
```

Более подробно использование параметров SQL-запроса было рассмотрено в *главе 8*, посвященной реляционному способу доступа.

Вызов хранимой процедуры действия

Хранимая процедура действия вызывается инструкцией следующего формата:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
```

Инструкция вызывает указанную процедуру, передавая ей значения входных параметров. Данная инструкция используется в программе IBConsole, при этом результаты работы процедуры (значения выходных параметров) выводятся в окне результатов выполнения запроса.

Например:

```
EXECUTE PROCEDURE pSalary
```

Ранее была создана хранимая процедура `pSalary`, в которой для сотрудников организации подсчитывались общая сумма зарплаты и средняя зарплата. Процедура не имеет входных параметров. В качестве выходных параметров процедуры заданы параметры `opSum` и `opAvg` для суммы и среднего значения соответственно. При выполнении инструкции `EXECUTE PROCEDURE` в среде программы IBConsole результат будет иметь следующий вид:

```
      OPSUM        OPAVG
=============== ================
      10300        2600
```

В программе IBConsole с помощью инструкции `EXECUTE PROCEDURE` также можно отлаживать хранимые процедуры выбора. Однако в этом случае в качестве их результатов возвращается только первая строка (первая запись набора данных).

Для вызова хранимой процедуры действия из приложения предназначен компонент `StoredProc`. Рассмотрим свойства и особенности использования этого компонента.

Свойство `DatabaseName` типа `String` указывает на компонент `Database`, используемый для соединения с БД. Соединение с удаленной БД компонента `StoredProc` не отличается от соединения с ней компонента `Query`, рассмотреного в *главе 13*.

Свойство `StoredProcName` типа `String` определяет вызываемую хранимую процедуру. Имя процедуры выбирается через список Инспектора объектов. Если при попытке выбрать процедуру соединение с БД еще не установлено (свойство `Connected` компонента `Database` имеет значение `False`), то выдается запрос на его установление. После ввода имени и пароля пользователя происходит соединение с БД, а свойство `Connected` компонента `Database` устанавливается в значение `True`.

Когда имя хранимой процедуры задано, становятся доступными ее входные и выходные параметры, определяемые значением свойства `Params` типа `TParams`. Это свойство определяет коллекцию (массив) параметров компонента `StoredProc`, работа с которыми аналогична работе с параметрами SQL-запроса компонента `Query`.

Свойство `ParamBindMode` типа `TParamBindMode` определяет, каким образом устанавливается соответствие между параметрами компонента `StoredProc` и параметрами процедуры. Оно может принимать следующие значения:

- `pbByName` (соответствие устанавливается по именам) — по умолчанию; имена параметров компонента `StoredProc` и соответствующих параметров процедуры должны совпадать;
- `pbByNumber` (соответствие устанавливается по номерам); первый параметр компонента `StoredProc` соотносится с первым параметром процедуры, второй — со вторым и т. д.

Выполнение выбранной хранимой процедуры осуществляется последовательным вызовом методов `Prepare` и `ExecProc`. Метод `Prepare` подготавливает хранимую процедуру к выполнению путем связывания параметров компонента `StoredProc` и параметров процедуры. Метод `ExecProc` выполняет процедуру.

При необходимости перед выполнением процедуры можно установить или изменить значения ее входных параметров аналогично тому, как это делается для параметров SQL-запроса. Результаты работы процедуры возвращаются в выходных параметрах. Для доступа к параметрам при выполнении приложения удобно использовать метод `ParamByName`.

Рассмотрим пример, иллюстрирующий вызов хранимой процедуры действия:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
StoredProc1.StoredProcName := 'pSalary';
StoredProc1.Prepare;
StoredProc1.ExecProc;
Edit1.Text := StoredProc1.ParamByName('opSum').Value;
Edit2.Text := StoredProc1.ParamByName('opAvg').Value;
end;
```

Как и в предыдущем примере, здесь вызывается процедура действия `pSalary`. Ее результаты — значения выходных параметров `opSum` и `opAvg` — выводятся в компонентах `Edit1` и `Edit2` соответственно. Для вызова процедуры используется компонент `StoredProc1`.

Использование триггеров

Триггер представляет собой процедуру, которая находится на сервере БД и вызывается автоматически при модификации записей БД, т. е. при изменении столбцов или при их удалении и добавлении. В отличие от хранимых процедур, триггеры нельзя вызывать из приложения клиента, а также передавать им параметры и получать от них результаты.

Триггер по своей сути похож на обработчики событий `BeforeEdit`, `AfterEdit`, `BeforeInsert`, `AfterInsert`, `BeforeDelete` и `AfterDelete`, связанных с модификацией таблиц. Триггер может вызываться при редактировании, добавлении или удалении записей до и/или после этих событий.

Замечание

Изменения, внесенные триггером в транзакции, которая оказалась отмененной, также отменяются.

Триггеры обычно используются для программной реализации так называемых бизнес-правил. С их помощью удобно реализовывать различные ограничения, например, ограничения на значения столбцов или ограничения ссылочной це-

лостности, а также выполнять такие действия, как накапливание статистики работы с БД или резервное копирование записей.

Замечание

Устанавливаемые триггером бизнес-правила не должны противоречить аналогичным правилам, заданным для БД на физическом уровне.

Создание и изменение триггера

Создание триггера выполняется инструкцией `CREATE TRIGGER`, имеющей формат:

```
CREATE TRIGGER <Имя триггера> FOR <Имя таблицы>
      [ACTIVE | INACTIVE]
      {BEFORE | AFTER}
      {UPDATE | INSERT | DELETE}
       [POSITION <Число>]
            AS <Тело триггера>
```

Описатели `ACTIVE` и `INACTIVE` определяют активность триггера сразу после его создания. По умолчанию действует `ACTIVE`, и созданный триггер активен, т. е. при наступлении соответствующего события будет выполняться. Если триггер неактивен, то при наступлении соответствующего события он не вызывается. Ранее созданный триггер можно активизировать или, наоборот, деактивизировать.

Описатели `BEFORE` и `AFTER` задают момент начала выполнения триггера до или после наступления соответствующего события, связанного с изменением записей.

Описатели `UPDATE`, `INSERT` и `DELETE` определяют, при наступлении какого события вызывается триггер — при редактировании, добавлении или удалении записей соответственно.

Для одного события можно создать несколько триггеров, каждый из которых будет автоматически выполнен (если находится в активном состоянии). При наличии нескольких триггеров порядок их вызова (выполнения) определяет число, указанное в операнде `POSITION`. Триггеры выполняются в порядке возрастания этих чисел.

Созданный триггер можно удалить или изменить. Удаляется триггер инструкцией

```
DROP TRIGGER <Имя триггера>
```

Изменение триггера выполняется инструкцией `ALTER TRIGGER`, формат которой не отличается от формата инструкции создания триггера. После выполнения инструкции `ALTER TRIGGER` предыдущее описание триггера с указанным именем заменяется на новое.

Программирование триггера аналогично программированию хранимой процедуры, для чего используется специальный язык, позволяющий также создавать хранимые процедуры.

Отметим, что для доступа к значениям столбца используются конструкции формата:

```
OLD.<Имя столбца>
NEW.<Имя столбца>
```

Первая из них позволяет обратиться к старому (до внесения изменений), а вторая — к новому (после внесения изменений) значениям столбца.

Примеры использования триггера

Рассмотрим использование триггера для реализации ограничений ссылочной целостности и для занесения в ключевые столбцы уникальных значений.

В связи с тем, что InterBase не поддерживает автоинкрементные поля, при создании ключевого столбца, требующего уникальности значений, рекомендуется поступать так:

1. При создании таблицы задать ключевой столбец целочисленного типа.
2. Создать генератор, который при обращении к нему возвращает уникальное целочисленное значение.
3. Создать триггер, который при добавлении к таблице новой записи обращается к генератору и заносит возвращаемое им значение в ключевое поле.

Следующий пример иллюстрирует описанную последовательность действий:

```
/* Создание таблицы */
CREATE TABLE Store
       (S_Code  INTEGER NOT NULL,
      ...
      PRIMARY KEY (S_Code));

/* Создание генератора */
CREATE GENERATOR GenStore;
SET GENERATOR GenStore TO 1;

/* Создание триггера */
CREATE TRIGGER CodeStore FOR Store
ACTIVE
BEFORE INSERT
      AS
      BEGIN
            NEW.S_Code = GEN_ID(GenStore, 1);
      END
```

После добавления к таблице `Store` новой записи ключевому столбцу `S_Code` этой записи автоматически присваивается уникальное значение. Это обеспечивается обращением `GEN_ID` к генератору `GenStore`, который создается отдельно от триггера (работа с генераторами рассмотрена ниже).

Напомним, что ограничения ссылочной целостности для связанных таблиц включают в себя:

- каскадное удаление записей;
- запрет на редактирование ключевых столбцов.

Рассмотрим, как можно реализовать каскадное удаление записей с участием триггера.

Пусть имеются две таблицы: главная `Store` и подчиненная `Cards`, связанные по полям кода `S_Code` и `C_Code2` соответственно:

```
CREATE TABLE Store
       (S_Code      INTEGER NOT NULL,
      ...
      PRIMARY KEY (S_Code));

CREATE TABLE Cards
       (C_Code      INTEGER NOT NULL,
      C_Code2       INTEGER NOT NULL,
      ...
      PRIMARY KEY (C_Code));
```

Напомним, что каскадное удаление записей для связанных таблиц заключается в том, что если из главной таблицы удаляется запись, то и в подчиненной таблице должны быть удалены все соответствующие ей записи.

В нашем случае это выполняется следующим образом:

```
CREATE TRIGGER DeleteStore FOR Store
ACTIVE
AFTER DELETE
      AS
      BEGIN
            DELETE FROM Cards WHERE Store.S_Code = Cards.C_Code2;
      END
```

После удаления записи в таблице `Store` (склад) будут автоматически удалены все соответствующие записи в таблице `Cards` (движение товара).

Замечание

Для таблиц не должны действовать ограничения ссылочной целостности, заданные на физическом уровне, например, так:

```
CONSTRAINT rStoreCards

FOREIGN KEY (C_Code2) REFERENCES Store
```

В противном случае при попытке удалить запись из главной таблицы генерируется исключение. Если есть такие ограничения, то их можно удалить, например, с помощью программы SQL Explorer.

Аналогичным способом можно реализовать и обновление столбцов связи (ключевых столбцов) связанных таблиц, заключающееся в том, что при изменении значения столбца связи главной таблицы соответственно изменяются значения столбца связи всех связанных записей подчиненной таблицы. Например:

```
CREATE TRIGGER ChangeStore FOR Store
ACTIVE
BEFORE UPDATE
      AS
      BEGIN
            IF(OLD.S_Code <> NEW.S_Code)
                  THEN UPDATE Cards
                        SET C_Code2 = NEW.S_Code
                        WHERE C_Code2 = OLD.S_Code;
      END
```

При изменении столбца `S_Code`, используемого для связи главной таблицы `Store` с подчиненной таблицей `Cards`, автоматически изменяются значения столбца связи `C_Code2` соответствующих записей подчиненной таблицы.

Замечание

Чтобы столбец связи главной таблицы можно было редактировать, по нему не должен быть создан ключ. Если ключ создан, то его следует удалить, например, с помощью программы SQL Explorer. Более кардинальным решением проблемы является создание таблицы заново с помощью инструкции `ALTER TABLE`.

Поскольку использование триггеров не допускает существования ограничений на физическом уровне (при определении структуры таблиц), необходимо обеспечить также занесение значения в столбец связи подчиненной таблицы при добавлении в нее новой записи.

Создание генераторов

Напомним, что, в отличие от базы данных Paradox, для таблиц InterBase отсутствует автоинкрементный тип, обеспечивающий автоматическую установку уникальных значений. Поэтому для обеспечения уникальности значений ключевых столбцов совместно с триггерами используются генераторы. *Генератор* возвращает уникальное целочисленное значение.

С помощью языка SQL-сервера можно создать генератор и установить для него начальное значение. Генератор создается следующией инструкцией:

```
CREATE GENERATOR <Имя генератора>;
```

Начальное значение задается инструкцией:

```
SET GENERATOR <Имя генератора> TO <Начальное значение>;
```

Начальное значение представляет собой целое число, начиная с которого формируется числовой ряд.

Обращение к созданному генератору выполняется с помощью функции

```
GEN_ID (<Имя генератора>, <Шаг>);
```

Эта функция возвращает значение, увеличенное на целочисленный шаг относительно предыдущего значения генератора.

Замечание

После определения начального значения генератора изменять его нельзя. Также нельзя изменять шаг. Невыполнение этого правила приведет к тому, что генератор может повторно возвратить уже имеющееся значение со всеми вытекающими последствиями.

Пример создания генератора:

```
CREATE GENERATOR GenStore;
SET GENERATOR GenStore TO 1;
```

Здесь создается генератор `GenStore`, имеющий начальное значение `1`. Пример обращения к этому генератору приведен в предыдущем разделе — `GEN_ID(GenStore, 1)`.

Использование функций, определяемых пользователем

Функция, определяемая пользователем, представляет собой обычную функцию, написанную на алгоритмическом языке, например, Pascal. Созданная функция оформляется в виде динамической библиотеки DLL, откуда может быть вызвана обычным способом. В общем случае библиотека содержит несколько функций. Для обеспечения вызова функции системе Windows должен быть известен путь к соответствующей библиотеке.

Достоинства применения функций, определяемых пользователем, состоят в следующем:

- расширяется состав функций языка SQL;
- появляется возможность использования функций другими приложениями.

Отметим, что состав встроенных функций языка SQL относительно небольшой, и в нем отсутствуют, например, функции обработки дат, времени и чисел, а для строк реализованы только несколько наиболее употребительных операций. При необходимости пользователь может сам создать и в дальнейшем использовать такие часто применяемые функции, как вычисление модуля или квадратного корня.

Порядок действий при работе с пользовательской функцией следующий:

1. Создать функцию и включить ее в соответствующую библиотеку; при необходимости создается также и библиотека (модуль DLL).
2. Объявить функцию в сервере.
3. Вызвать функцию в инструкции SQL.

Рассмотрим в качестве примера использование функции, выполняющей извлечение квадратного корня из числа. Назовем функцию `Sqrt2`, а библиотеку `Ibdll`.

В среде Delphi код библиотеки имеет следующий вид:

```
LIBRARY Ibdll;

USES
      SysUtils;
      Classes;

FUNCTION Sqrt2 (x: REAL): REAL; CDECL; EXPORT;
BEGIN
IF x < 0 THEN Sqrt2 := 0 ELSE Sqrt2 := SQRT(x);
END;

EXPORTS
      Sqrt2;

BEGIN
END.
```

По сравнению с обычным для модулей Delphi описанием функции в ее заголовке указаны слова `CDECL` и `EXPORT`, определяющие соглашения для параметров и назначение функции. Имя экспортируемой функции указывается в разделе `EXPORTS`.

Полученный в результате трансляции модуль библиотеки Ibdll.dll нужно скопировать в каталог, где этот модуль будет доступен системе Windows, например, в каталог C:\WINNT\SYSTEM32.

Объявление пользовательской функции в сервере выполняется с помощью следующей инструкции:

```
DECLARE EXTERNAL FUNCTION <Имя функции>
      [<Тип данных1>, ... <Тип данныхN>]
      RETURNS {<Тип данных> [BY VALUE] | CSTRING (<Целое число>)}
      ENTRY_POINT "<Зарегистрированное имя функции>"
      MODULE_NAME "<Имя библиотеки>";
```

`Имя функции` указывает имя, с помощью которого ее можно вызывать из сервера. В общем случае это имя не совпадает с именем, под которым функция регистрируется в библиотеке при ее создании (операнд `ENTRY_POINT`). За именем следует список типов входных параметров функции. Отметим, что строковые значения должны передаваться по ссылке (указателю). Поэтому для строковых параметров следует указывать тип `CSTRING`, который соответствует типу `PChar` языка Pascal. Описание типа `CSTRING` имеет формат:

```
CSTRING (<Число символов строки>)
```

Тип возвращаемого функцией результата указывается после слова RETURNS. По умолчанию результат возвращается по ссылке, при необходимости обеспечить его возврат по значению указывается описатель BY VALUE. Имя используемой библиотеки определяет операнд MODULE_NAME, в котором задается имя модуля библиотеки DLL (без расширения).

Регистрацию рассмотренной функции выполняет инструкция

```
DECLARE EXTERNAL FUNCTION Sqrt2 FLOAT
      RETURNS FLOAT BY VALUE
      ENTRY_POINT "Sqrt2"
      MODULE_NAME "IBDLL";
```

Отмену объявления функции в сервере (удаление) выполняет инструкция:

```
DROP EXTERNAL FUNCTION <Имя функции>;
```

После объявления пользовательской функции ее можно вызывать в инструкциях языка SQL наряду с его встроенными функциями, такими как SUM, AVG или CAST. Например, вызов функции Sqrt2 в инструкции отбора записей будет иметь следующий вид:

```
SELECT * FROM Graphics
      WHERE CoordY <= Sqrt2(CoordX)
```

Приведенная инструкция отбирает из таблицы Graphics записи, для которых значение столбца CoordY не превосходит квадратного корня из значения столбца CoordX.

Отметим, что локальная версия сервера InterBase не поддерживает работу с функциями, определяемыми пользователем.

Реализация механизма транзакций

В *главе 7*, посвященной навигационному способу доступа с помощью BDE, механизм транзакций был рассмотрен на примере локальных БД. Напомним, что механизм транзакций используется для поддержания целостности БД: транзакция переводит БД из одного целостного состояния в другое. Чтобы транзакция была успешной, должны выполниться все операции, входящие в ее состав. В случае возникновения ошибки хотя бы в одной из операций вся транзакция считается неуспешной, и результаты всех операций отменяются. Транзакция может быть явной и неявной.

Неявная транзакция запускается и завершается автоматически, *явной* транзакцией управляет программист. С явным управлением транзакциями для локальных БД с помощью методов и свойств компонента Database мы познакомились в *главе 7*. Рассмотренные средства компонента Database можно использовать и для удаленных баз данных. Кроме того, для удаленных БД имеются дополнительные возможности по управлению транзакциями.

При модификации удаленной БД также осуществляются неявные и явные транзакции. Для удаленных БД, кроме поддержания целостности данных, использование механизма транзакций позволяет определить порядок взаимодействия запросов. Для соперничающих запросов устанавливается режим одновременного доступа к одним и тем же данным.

Для модификации данных может использоваться SQL-запрос, выполняемый с помощью метода `ExecSQL` компонента `Query`. Такой запрос называют PassThrough SQL, его выполнение приводит к запуску неявной транзакции. Способ взаимодействия с сервером на уровне такой транзакции определяет параметр `SQLPASSTHRU MODE` псевдонима БД или драйвера (в нашем случае типа InterBase), который может принимать следующие значения:

- `SHARED AUTOCOMMIT` — инструкциями модификации БД, например, `UPDATE` или `INSERT`, автоматически запускается неявная транзакция (по умолчанию); после внесения изменений эта транзакция автоматически подтверждается; разные транзакции могут использовать общее соединение с БД;
- `SHARED NO AUTOCOMMIT` — инструкциями модификации БД также автоматически запускается неявная транзакция, но ее автоматического подтверждения не происходит, и нужно самостоятельно выполнять инструкцию `COMMIT`; разные транзакции могут использовать общее соединение с БД;
- `NOT SHARED` — транзакции должны использовать различные соединения с БД и подтверждаться выполнением инструкции `COMMIT`.

Возможность явного управления транзакциями предоставляет язык SQL сервера, в составе которого есть следующие инструкции:

- `SET TRANSACTION` (начать транзакцию);
- `COMMIT` (подтвердить транзакцию);
- `ROLLBACK` (откатить транзакцию).

Инструкция запуска явной транзакции имеет формат:

```
SET TRANSACTION
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL]
      {SNAPSHOT [TABLE STABILITY] | READ COMMITED}]
[RESERVING <Список таблиц>
      [FOR [{SHARED | PROTECTED}] [{READ | WRITE}]];
```

Все операнды этой инструкции являются необязательными и позволяют управлять перечисленными ниже режимами транзакции.

- Режим доступа к данным:
  - `READ WRITE` (разрешены чтение и модификация записей) — по умолчанию;
  - `READ ONLY` (разрешено только чтение записей).

- Поведение в случае конфликта транзакций при обновлении записей:
  - `WAIT` (ожидание завершения другой транзакции) — по умолчанию;
  - `NO WAIT` (прекращение данной транзакции).
- Уровень изоляции от других транзакций (операнд `ISOLATION LEVEL`):
  - `SNAPSHOT` (чтение данных в состоянии на момент начала транзакции) — по умолчанию; изменения, сделанные другими транзакциями, в данной транзакции не видны;
  - `SNAPSHOT TABLE STABILITY` (предоставление транзакции исключительного доступа к таблицам); другие транзакции могут читать записи из таблиц;
  - `READ COMMITED` (чтение только подтвержденных изменений в записях); если изменения еще не подтверждены, то читается предыдущая версия записи.
- Блокирование таблиц, указанных в списке операнда `RESERVING`, для других транзакций:
  - `PROTECTED READ` (разрешено только чтение записей);
  - `PROTECTED WRITE` (для транзакций с уровнем изоляции `SNAPSHOT` или `READ COMMITTED` разрешено только чтение записей);
  - `SHARED READ` (разрешены чтение и модификация записей);
  - `SHARED WRITE` (разрешено чтение записей; для транзакций с уровнем изоляции `SNAPSHOT` или `READ COMMITTED` разрешена модификация записей).

Действие инструкций `COMMIT` и `ROLLBACK` по подтверждению и откату транзакции аналогично действию одноименных методов компонента `Database`.

Использование механизма кэшированных изменений

Механизм кэшированных изменений заключается в том, что на компьютере клиента в кэше (буфере) создается локальная копия данных, и все изменения в данных выполняются в этой копии. Для хранения локальной копии используется специальный буфер (кэш). Сделанные изменения можно подтвердить, перенеся их в основную БД, хранящуюся на сервере, или отказаться от них. Этот механизм напоминает механизм транзакций, но, в отличие от него, снижает нагрузку на сеть, т. к. все изменения передаются в основную БД одним пакетом. Следует иметь в виду, что для всех записей локальной копии отсутствуют блокировки на изменение их значений. Блокировки могут быть установлены другими приложениями для основной БД, находящейся на сервере.

Механизм кэшированных изменений реализуется в приложении, для чего компоненты, в первую очередь `Database`, `Table` и `Query` (используемые при доступе с помощью BDE), имеют соответствующие средства. Кроме того, механизм кэ-

шированных изменений поддерживается предназначенным для этого компонентом `UpdateSQL`.

Основные достоинства рассматриваемого механизма проявляются для удаленных БД, но его можно использовать и при работе с локальными БД.

Использование компонентов *Database*, *Table* и *Query*

Включением для набора данных режима кэшированных изменений управляет свойство `CachedUpdates` типа `Boolean`: значение `True` этого свойства включает (активизирует) режим, значение `False` — выключает его (по умолчанию). Для рассматриваемых в этой главе удаленных БД и механизма доступа BDE, как правило, используется набор данных `Query`, в то время как для локальных БД нет принципиальной разницы между использованием наборов данных `Table` и `Query`.

После включения режима кэшированных изменений данные автоматически копируются в буфер и дальнейшая работа осуществляется именно с этими данными. Остальные пользователи (наборы данных) не видят проделанных изменений до тех пор, пока они не перенесены в основную БД. Для того чтобы измененные данные из буфера были переписаны в основную БД, нужно подтвердить изменения. Подтверждение кэшированных изменений выполняется в два этапа:

- запись кэшированных изменений в основную БД;
- подтверждение или отмена сделанных изменений.

Приведенный порядок действий напоминает порядок действий при управлении транзакцией. Для их реализации нужно вызвать соответствующие методы набора данных или компонента `Database`, с которым связан набор.

Метод `ApplyUpdates` набора данных записывает в БД изменения в кэшированных данных. При этом результаты выполнения всех операций редактирования, вставки или удаления записей пересылаются одним пакетом, что может значительно снизить нагрузку на сеть. Вызов метода `ApplyUpdates` производится на первом этапе. При записи данных в основную БД возможен вариант, когда сделанные изменения вызовут исключение, например, связанное с нарушениями ограничений ссылочной целостности или блокировкой записей другими пользователями. Поэтому необходимо перехватывать и обрабатывать возможное исключение. Для анализа результата выполнения первого этапа удобно использовать конструкцию `TRY ... EXCEPT` локальной обработки исключений.

На втором этапе в зависимости от результата первого этапа сделанные изменения утверждаются или отменяются. Метод `CommitUpdates` набора данных подтверждает изменения, а метод `CancelUpdates` отменяет их, возвращая БД в исходное состояние.

Методы, реализующие оба этапа подтверждения кэшированных изменений, рекомендуется выполнять в рамках транзакции, чтобы обеспечить возможность восстановления данных при возникновении ошибок.

Так, в процедуре

```
PROCEDURE TForm1.Button1Click(Sender: TObject);
BEGIN
Database1.StartTransaction;
TRY
Query1.ApplyUpdates;
Database1.Commit;
Query1.CommitUpdates;
EXCEPT
MessageDlg('Изменения не сохранены!', mtError, [mbOK], 0);
Query1.CancelUpdates;
END;
END;
```

выполняется подтверждение кэшированных изменений, сделанных в наборе данных `Query1`. Перед выполнением этой операции запускается явная транзакция, которая при отсутствии ошибок также явно подтверждается. При возникновении ошибки, связанной с выполнением любого из методов, необходимых для подтверждения изменений, эти изменения отменяются, о чем пользователю выдается сообщение. Проверку успешности операций подтверждения осуществляет локальный обработчик исключений.

Метод `ApplyUpdates (const DataSets: array of TDBDataSet)` компонента `Database` используется для подтверждения изменений одновременно в нескольких наборах данных, заданных в параметре `DataSets`. В отличие от одноименного метода набора данных, метод `ApplyUpdates` компонента `Database` при записи изменений в БД автоматически запускает транзакцию и при ее успешном завершении также автоматически подтверждает сделанные изменения. Таким образом, при отсутствии ошибок этот метод реализует оба этапа выполнения транзакции. В случае возникновения ошибки необходимо самостоятельно вызывать метод `CancelUpdates` набора данных, отменяющий изменения.

Замечание

Если набор данных не находится в режиме кэшированных изменений и его свойство `CachedUpdates` имеет значение `False`, то вызов методов `ApplyUpdates`, `CommitUpdates` или `CancelUpdates` приводит к исключению.

Пример подтверждения кэшированных изменений в рамках неявной транзакции:

```
PROCEDURE TForm1.Button2Click(Sender: TObject);
BEGIN
TRY
Database1.ApplyUpdates([Query1]);
EXCEPT
MessageDlg('Изменения не сохранены!', mtError, [mbOK], 0);
```

```
Query1.CancelUpdates;
END;
END;
```

Как и в предыдущем примере, выполняется подтверждение кэшированных изменений, сделанных в наборе данных `Query1`. При возникновении ошибок подтверждения сделанные изменения отменяются.

В приведенных примерах кэшированные изменения или целиком подтверждались, или целиком отменялись. Кроме этого, при возникновении ошибок подтверждения изменений возможен анализ отдельных записей и их столбцов (полей), в результате чего выявляются приведшие к ошибке записи и корректируются неправильные данные. Такую возможность предоставляет событие `OnUpdateError` типа `TUpdateErrorEvent`, в обработчике которого следует выполнить действия по исправлению ситуации. Тип события описан следующим образом:

```
type TUpdateErrorEvent =
   procedure (DataSet: TDataSet; E: EDatabaseError; UpdateKind:
         TUpdateKind; var UpdateAction: TUpdateAction) of object;
```

Список параметров, передаваемых обработчику, приведен ниже:

- `DataSet` — набор данных, при сохранении изменений которого произошла ошибка;
- `UpdateKind` — вид изменений записи:
  - `ukModify` (модификация);
  - `ukInsert` (вставка);
  - `ukDelete` (удаление);
- `UpdateAction` — вид операции по исправлению ошибочной записи; программист устанавливает значение этого параметра в зависимости от требуемых действий:
  - `uaAbort` (отмена изменения записи без выдачи сообщения);
  - `uaApplied` (удаление записи из кэша);
  - `uaFail` (отмена изменения записи с выдачей сообщения);
  - `uaRetry` (повтор операции сохранения);
  - `uaSkip` (пропуск записи);
- `E` — объект-исключение, содержащий описание возникшей ошибки; в первую очередь используются свойства `ErrorCode` и `Message` этого объекта, которые определяют код ошибки и текстовое сообщение соответственно.

Например, в процедуре:

```
PROCEDURE TForm1.Query1UpdateError (DataSet: TDataSet;
      E: EDatabaseError; UpdateKind: TUpdateKind;
      VAR UpdateAction: TUpdateAction);
```

```
BEGIN
UpdateAction := uaFail;
END;
```

при подтверждении кэшированных изменений набора данных `Query1` отменяются изменения, сделанные во всех записях, которые приводят к ошибке.

Более сложно организовать исправление ошибочных записей. Доступ к значениям столбцов (полей) набора данных имеет особенность, связанную с тем, что существуют две версии записи: старая и неподтвержденная новая. Для доступа к старому и новому значениям столбца текущей записи предназначены соответственно свойства `NewValue` и `OldValue`.

Рассмотрим следующий пример:

```
PROCEDURE TForm1.Query1UpdateError (DataSet: TDataSet;
      E: EDatabaseError; UpdateKind: TUpdateKind;
      VAR UpdateAction: TUpdateAction);
BEGIN
Query1.FieldByName('Code').NewValue :=
      Query1.FieldByName('Code').OldValue;
UpdateAction := uaRetry;
END;
```

При редактировании записей набора данных вносились изменения в столбец `Code`. При возникновении ошибки подтверждения изменений в столбце текущей записи восстанавливается его предыдущее значение. После этого операция сохранения изменений подтверждается.

Замечание

Если ошибочная ситуация в обработчике события `OnUpdateError` не устранена, то при повторных попытках сохранения записи происходит зацикливание программы.

В приведенном примере параметр `DataSet` обработчика не используется, а имя `Query1` набора данных указано явно. Если обработчик является общим (разделяемым) для нескольких наборов данных, то вместо имени конкретного набора данных можно указать конструкцию вида

```
(DataSet as TDataSet)
```

или

```
TDataSet(DataSet)
```

Обе конструкции приводят тип параметра `DataSet` к типу набора данных `TDataSet`, первая — неявно, вторая — явно. Если тип набора данных известен более точно, например, `TQuery`, то его можно указать вместо типа `TDataSet`.

Использование компонента *UpdateSQL*

Компонент `UpdateSQL` предназначен для подтверждения кэшированных изменений в наборе данных `Query`. Главная задача этого компонента — обеспечить модификацию наборов данных, доступных только для чтения.

Набор данных `Query` связывается с компонентом `UpdateSQL` через свое свойство `UpdateObject` типа `TDataSetUpdateObject`, в качестве значения которого выбирается имя компонента `UpdateSQL`. Кроме того, для набора данных `Query` нужно установить в значение `True` свойства `CachedUpdates` и `RequestLive`.

Роль компонента `UpdateSQL` заключается в выполнении SQL-запросов, обновляющих необходимые записи связанного с ним набора данных `Query`. При этом данный компонент позволяет одновременно хранить и выполнять три разных запроса:

- `UPDATE` (редактирование записей);
- `INSERT` (вставка записей);
- `DELETE` (удаление записей).

Указанные запросы содержатся в свойствах `ModifySQL`, `InsertSQL` и `DeleteSQL` типа `TStrings` соответственно. Таким образом, компонент `UpdateSQL` частично совмещает функциональность сразу трех компонентов `Query`.

Ввести тексты соответствующих запросов компонента `UpdateSQL` можно вручную аналогично тому, как это выполняется для свойства `SQL` компонента `Query`. Однако на этапе разработки приложения более удобно использовать для этих целей Редактор свойств компонента `UpdateSQL`, окно которого показано на рис. 14.3. Редактор свойств вызывается командой **UpdateSQL Editor** контекстного меню компонента или двойным щелчком на компоненте `UpdateSQL`. В названии окна выводятся имена компонента `UpdateSQL` и соответствующего набора данных `Query`.

Замечание

Если для соответствующего набора данных `Query` не задан SQL-запрос, то элементы редактора свойств будут пустыми. Если SQL-запрос задан, то состояние набора данных (свойство `Active`) при выполнении этого запроса не имеет значения.

Редактор содержит две вкладки, первоначально выбрана вкладка параметров **Options**. В списке **Table Name** выбирается таблица, для записей которой включен механизм кэшированных изменений. Список содержит имена таблиц, указанных в запросе набора данных `Query`.

После выбора таблицы в списках ключевых (**Key Fields**) и обновляемых полей (**Update Fields**) отображается перечень ее столбцов. В первом списке нужно выделить столбцы, входящие в состав индекса и используемые при доступе к данным, а во втором — столбцы, значения которых будут обновляться.

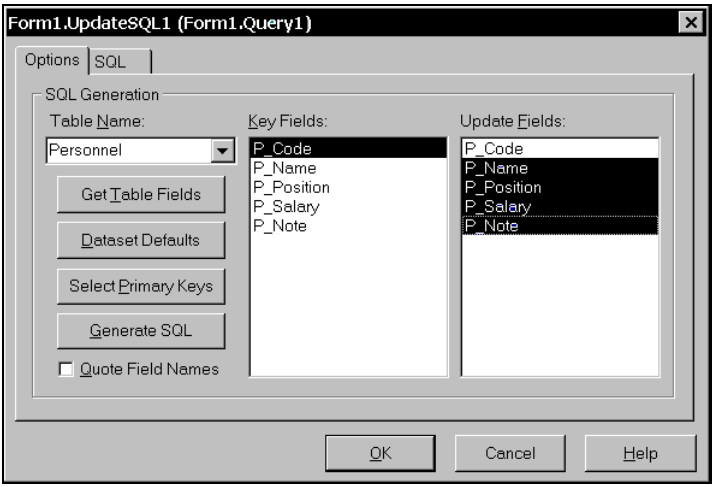


Рис. 14.3. Редактирование свойств компонента `UpdateSQL`

Замечание

При работе с локальными таблицами Paradox нельзя выбирать поля автоинкрементного типа. На рис. 14.3 таким полем является ключевое поле `P_Code`. Нажатие кнопки **Select Primary Keys** выделяет ключевые поля таблицы.

Нажатие кнопки **Generate SQL** инициирует автоматическую генерацию SQL-запросов для выполнения обновления записей. Тексты запросов можно просмотреть и при необходимости изменить на вкладке **SQL** (рис. 14.4), переход на которую осуществляется после генерации запросов.

Группа переключателей **Statement Type** определяет тип запроса, текст которого содержится в поле редактирования **SQL Text**:

- **Modify** — запрос редактирования `UPDATE`;
- **Insert** — запрос вставки `INSERT`;
- **Delete** — запрос удаления `DELETE`.

Запросы компонента `UpdateSQL` можно выполнить либо через вызов методов компонента `Database`, либо через метод самого компонента `UpdateSQL`.

При подтверждении кэшированных изменений методом `ApplyUpdates` компонента `Database` для указанных наборов данных `Query` автоматически отрабатываются запросы соответствующего компонента `UpdateSQL`. Эти запросы и выполняют сохранение кэшированных изменений "своего" набора. Например:

```
Database1.ApplyUpdates([Query1]);
```

Напомним, что набор данных `Query` должен быть связан с компонентами `Database` и `UpdateSQL` через свои свойства `DatabaseName` и `UpdateObject` соответственно.

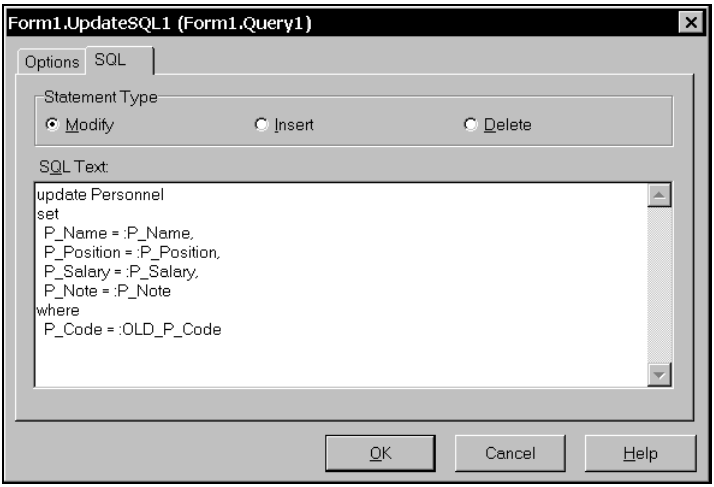


Рис. 14.4. SQL-запросы компонента `UpdateSQL`

Подтверждение кэшированных изменений также можно выполнить последовательным вызовом методов `SetParams` и `ExecSQL`. Метод `SetParams (UpdateKind: TUpdateKind)` устанавливает для изменяемой записи значения параметров соответствующего SQL-запроса, а метод `ExecSQL (UpdateKind: TUpdateKind)` выполняет указанный запрос. Параметр `UpdateKind`, одинаковый для обеих процедур, определяет вид изменения записи:

- `ukModify` (редактирование);
- `ukInsert` (вставка);
- `ukDelete` (удаление).

Так как эти методы работают с текущей записью, они должны вызываться для каждой измененной записи набора данных. Поэтому вызов методов подтверждения выполняют в обработчике `OnUpdateRecord` типа `TUpdateRecordEvent`. Тип события описан следующим образом:

```
type TUpdateRecordEvent =
      procedure(DataSet: TDataSet; UpdateKind: TUpdateKind;
          var UpdateAction: TUpdateAction) of object;
```

Параметры этого события отличаются от параметров рассмотренного ранее события `OnUpdateError` отсутствием параметра `E`, содержащего объект-исключение.

Пример использования компонента `UpdateSQL`:

```
PROCEDURE TForm1.Query1UpdateRecord(DataSet: TDataSet;
UpdateKind: TUpdateKind; VAR UpdateAction: TUpdateAction);
BEGIN
UpdateSQL1.SetParams(UpdateKind);
```

```
UpdateSQL1.ExecSQL(UpdateKind);
UpdateAction := uaApplied;
END;
```

Параметр `UpdateAction` устанавливается в значение `uaApplied`, что соответствует подтверждению изменений в записях.

Последовательный вызов методов `SetParams` и `ExecSQL` можно заменить вызовом метода `Apply (UpdateKind: TUpdateKind)`, объединяющего их функциональность.

В данном разделе мы рассмотрели использование для одного набора данных `Query` одного компонента `UpdateSQL`, связь с которым устанавливается через свойство `UpdateObject` набора данных. Если с одним набором данных нужно связать несколько компонентов `UpdateSQL`, то следует обратиться к их свойствам `DataSet`, которые должны указывать на набор данных `Query`.

Использование механизма событий сервера

Механизм событий (сообщений) может использоваться сервером для информирования клиентских приложений о том, что произошло определенное событие, например, сбой в системе. При возникновении события всем активным приложениям посылается соответствующее сообщение. Для этого требуется:

- создать на сервере триггер или хранимую процедуру;
- зарегистрировать сообщение в приложении.

В триггере или хранимой процедуре отправку сообщения выполняет инструкция `POST_EVENT` отправки сообщения, имеющая формат:

```
POST_EVENT "<имя события>";
```

Рассмотрим пример:

```
CREATE TRIGGER InsertStore FOR Store
ACTIVE
AFTER INSERT
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END

CREATE TRIGGER DeleteStore FOR Store
ACTIVE
AFTER DELETE
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END
```

```
CREATE TRIGGER UpdateStore FOR Store
ACTIVE
AFTER UPDATE
      AS
      BEGIN
            POST_EVENT "evChangeStore";
      END
```

При модификации (вставке, удалении или редактировании) записей таблицы приложениям посылается сообщение `evChangeStore`. Для этого созданы триггеры, автоматически реагирующие на соответствующие события.

Если для отправки сообщения используется не триггер, а хранимая процедура, которая, в отличие от триггера, не выполняется автоматически, то ее можно вызывать как из сервера, например из триггера или другой процедуры, так и из приложения. В последнем случае можно организовать обмен сообщениями через сервер, в том числе для задач, которые не работают с удаленной БД сервера. В этом случае сервер (сервер БД) используется приложениями как своеобразный диспетчер.

Для регистрации сообщений в приложении предназначен компонент `IBEvents`, свойство `Database` которого указывает на компонент `Database`, используемый для соединения с БД.

Замечание

В предыдущей версии Delphi в качестве основного средства регистрации сообщений в приложении применялся компонент `IBEventAlerter`, располагавшийся на изъятой теперь странице **Samples** Палитры компонентов.

Свойство `Events` типа `TStrings` определяет список событий, которые может обрабатывать приложение. При разработке приложения перечень событий редактируется в окне **EventAlerter Events** (рис. 14.5).

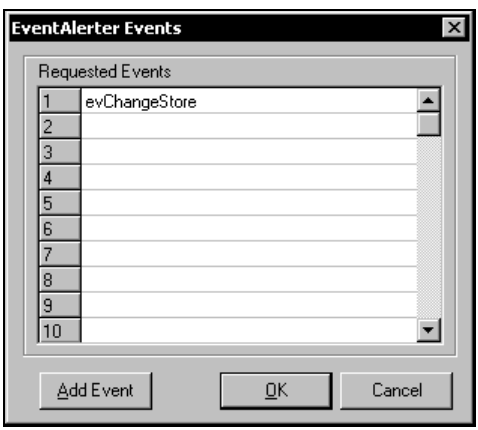


Рис. 14.5. Список событий компонента `IBEvents`

На этапе выполнения приложения списком сообщений можно управлять динамически, используя возможности класса `TSrtings`.

Например, команда

```
IBEvents1.Events.Add('evChangeStore');
```

добавляет событие `evChangeStore` к списку событий компонента `IBEventAlerter1`.

После определения списка интересующих приложение событий их следует зарегистрировать в сервере. Регистрация выполняется вызовом метода `RegisterEvents`, в свою очередь метод `UnRegisterEvents` отменяет регистрацию событий. Например:

```
IBEventAlerter1.RegisterEvents;
```

Для задания реакции на поступающие от сервера события используется обработчик события `OnEventAlert` типа `TEventAlert`, описанного как

```
TYPE TEventAlert = PROCEDURE (Sender: TObject; EventName: String;
         EventCount: Integer; VAR CancelAlerts: Boolean);
```

Кроме параметра `Sender`, указывающего объект-источник события, обработчику передаются также следующие параметры:

- `EventName` — имя полученного события, например, `evChangeStore`;
- `EventCount` — число событий, произошедших в сервере с момента передачи предыдущего события;
- `CancelAlerts` — режим приема событий: значение `True` (по умолчанию) означает, что сервер по-прежнему должен присылать сообщения о таком событии, значение `False` соответствует прекращению отправки таких событий.

Обработка поступившего события реализуется программистом в зависимости от характера события и назначения приложения.

Рассмотрим следующий пример:

```
VAR CountChange: Integer = 0;
...
// Регистрация события
PROCEDURE TForm1.FormCreate(Sender: TObject);
BEGIN
      IBEvents1.Events.Clear;
      IBEvents1.Events.Add('evChangeStore');
      IBEvents1.RegisterEvents;
END;
// Обработка события
PROCEDURE TForm1.IBEvents1EventAlert(Sender: TObject;
      EventName: String; EventCount: Integer;
      VAR CancelAlerts: Boolean);
BEGIN
IF EventName = 'evChangeStore' THEN BEGIN
```

```
      CountChange:= CountChange + EventCount;
      IF CountChange >= 30 THEN BEGIN
              Query1.Close;
              Query1.Open;
              CountChange := 0;
              END;
       END;
END;
```

При создании формы `Form1` приложения в список заносится событие `evChangeStore` и выполняется регистрация этого события в сервере. Обработка события заключается в следующем. В глобальной переменной `CountChange` подсчитывается число изменений таблицы `Store`. В случае превышения этим числом критического значения `30` обновляется набор данных `Query1`, который обеспечивает отбор записей из названной таблицы.

Управление привилегиями

Привилегии представляют собой права доступа к БД. Управление привилегиями заключается в их установке и удалении. После создания объекта БД (например, таблицы) доступ к ней разрешен только создателю и системному администратору, имеющему имя `SYSDBA`. Для доступа к БД остальных пользователей им нужно назначить соответствующие привилегии. Сразу после появления нового пользователя, созданного например, с помощью программы InterBase Manager Server, этот пользователь имеет минимальные права доступа: ему разрешено только войти в БД (соединиться с ней), указав свое имя и пароль, однако ни один объект этой БД ему не доступен. Чтобы обеспечить возможность активной работы с БД, нужно определить (переопределить) привилегии.

Установку привилегий выполняет инструкция

```
GRANT
<Список описателей вида доступа>
ON [TABLE] {<Имя таблицы> | <Имя просмотра>}
       TO {<Пользователь> | <Список пользователей>} |
       EXECUTE ON PROCEDURE <Имя процедуры>
            TO {<Пользователь> | <Список пользователей>};

<Пользователь> =
      PROCEDURE <Имя процедуры>
      | TRIGGER <Имя триггера>
      | VIEW    <Имя просмотра>
      | [USER]  <Имя пользователя>
      | PUBLIC
```

```
<Список пользователей> =
       [USER]   <Имя пользователя1>
       ...
       [,[USER] <ИмяпользователяN>]
       [WITH GRANT OPTION]
```

Привилегии позволяют разграничить доступ к таблицам и просмотрам со стороны пользователей. При этом под "пользователем" понимается любой объект, обращающийся к данным. Кроме собственно пользователя (приложения), такими объектами могут быть таблицы, просмотры, хранимые процедуры и триггеры. Если привилегия предоставляется одновременно нескольким пользователям, то их имена перечисляются через запятую. Описатель `PUBLIC` означает, что привилегия устанавливается для всех пользователей. Описатель `WITH GRANT OPTION` означает, что пользователь может устанавливать предоставленнные ему привилегии другим пользователям.

В качестве описателей, определяющих вид доступа, указываются следующие:

- `ALL` (все права доступа);
- `SELECT` (только чтение);
- `DELETE` (удаление);
- `INSERT` (вставка);
- `UPDATE` (модификация).

После описателя `UPDATE` в круглых скобках можно указать список редактируемых столбцов таблицы или просмотра. Если требуется определить несколько прав, то соответствующие описатели перечисляются через запятую.

Для установки привилегий, кроме вида доступа, нужно указать еще имя таблицы или просмотра, а также имя пользователя. Остальная информация является необязательной.

Пример установки привилегий:

```
GRANT ALL ON Personnel TO Chief;
GRANT SELECT ON Personnel TO Manager;
GRANT SELECT, UPDATE (Name, BirthDay) ON Personnel TO TopManager;
```

Права доступа устанавливаются к таблице `Personnel`. Пользователь с именем `Chief` получает полные права доступа к таблице. Пользователь с именем `Manager` может только читать данные, а пользователь `TopManager`, кроме того, имеет право редактировать столбцы `Name` и `BirthDay`.

Еще один пример установки прав доступа:

```
GRANT ALL ON Store TO Chief, Manager, TopManager;
GRANT SELECT ON List TO PUBLIC;
```

Здесь полные права доступа к таблице `Store` получают три пользователя: `Chief`, `Manager` и `TopManager`. Читать записи из таблицы `List` разрешается всем пользователям.

Удаление привилегий заключается в отмене заданного ранее права доступа. Отмену привилегии выполняет инструкция `REVOKE`, формат которой аналогичен

формату инструкции GRANT установки привилегии. Отличие заключается в том, что инструкция дополнительно имеет необязательный описатель GRANT OPTION FOR, который удаляет не саму привилегию, а право выдачи ее другим пользователям. Отметим, что удалить привилегию может только тот, кто ее установил.

Так, после выполнения команды

```
REVOKE ALL ON Personnel TO Chief;
```

пользователь с именем Chief лишается права доступа к таблице Personnel.

Манипулирование данными

Инструкции манипулирования данными предназначены для отбора и изменения данных. Отбор записей выполняется с помощью инструкции SELECT. Изменение данных заключается в редактировании, удалении и вставке записей, для чего используются инструкции UPDATE, INSERT и DELETE соответственно. Применительно к локальным БД указанные инструкции рассмотрены в *главе 8,* посвященной реляционному способу доступа с помощью механизма BDE. В этом разделе мы познакомимся с особенностями отбора записей в удаленных БД с помощью инструкции SELECT, которая имеет следующий формат:

```
SELECT [DISTINCT | ALL] {* | <Список выражений>}
       FROM <Список таблиц>
       [WHERE <Условия выбора>]
       [ORDER BY <Список столбцов для сортировки>]
       [GROUP BY  <Список столбцов для группирования>]
       [HAVING <Условия группирования>]
       [UNION <Вложенная инструкция SELECT>]
       [PLAN <План выполнения запроса>]
```

По сравнению с локальной версией дополнительными здесь являются операнд PLAN, задающий план выполнения запроса, и описатель ALL. Описатель ALL по своему действию противоположен описателю DISTINCT, обеспечивающему отбор уникальных записей (т. е. записей, значения столбцов которых не повторяются). Применение описателя ALL позволяет отобрать записи с повторяющимися значениями. Явно использовать этот описатель нет необходимости, т. к. он действует по умолчанию.

Кроме того, по сравнению с SQL-запросом для локальных БД условия отбора записей могут более сложными.

Формат условий следующий:

```
<Условие отбора>] =
      {<Выражение> <Операция сравнения> {<Выражение1> | (<Отбор1>)} |
      <Выражение> [NOT] <Мин. значение> AND <Макс. значение> |
      <Выражение> [NOT] LIKE < Выражение1> [ESCAPE < Выражение2>] |
      <Выражение> [NOT] IN
       (<Выражение1> [, <Выражение2>]...[, <ВыражениеN>] | (<ОтборM>)) |
```

```
<Выражение> IS [NOT] NULL |
<Выражение> [NOT] {<Операция сравнения> | ALL | SOME | ANY} (<ОтборМ>) |
EXISTS (<ОтборМ>) |
SINGULAR (<ОтборМ>) |
<Выражение> [NOT] CONTAINING <Выражение1> |
<Выражение> [NOT] STARTING [WITH] <Выражение1>
```

Элементы, обозначенные как `Отбор`, представляют собой совокупность значений, отобранных с помощью инструкции `SELECT`, т. е. вложенную инструкцию `SELECT`. Заданное в этой инструкции условие отбора записей для элемента `Отбор1` возвращает одно значение или ни одного, а для элемента `ОтборМ` — несколько значений, в частности, также ни одного.

Условие отбора может состоять из нескольких условий, связанных логическими операциями `NOT`, `OR` и `AND`.

Рассмотрим операнды, которые не встречались в локальной версии SQL.

Операнд `<Выражение> [NOT] CONTAINING <Выражение1>` позволяет отобрать записи на основании частичного совпадения значений. С его помощью выбираются записи, для которых значение `<Выражение>` включает в себя значение `<Выражение1>`. Обычно этот операнд применяется для строк. Например, с помощью инструкции

```
SELECT * FROM Store
       WHERE S_Name CONTAINING "то"
```

для таблицы `Store` склада выводятся все столбцы с данными о товарах, названия которых содержат символы `то`. Указанные символы могут входить в название товара с любой позиции, не обязательно с первой. Поэтому будут отобраны, например, `томаты` и `картофель`. Выражения заданы простейшими способами — указанием имени столбца и строкового значения. В общем случае выражения могут быть более сложными и состоять из нескольких операндов, знаков операций и круглых скобок.

Операнд `<Выражение> [NOT] STARTING [WITH] <Выражение1>` отличается от операнда `CONTAINING` тем, что требует вхождения, начиная с начала выражения.

Напомним, что отбор записей с частичным совпадением значений также можно выполнить, используя операнд `LIKE`, рассмотренный в *главе 8*.

Операнды `EXISTS` и `SINGULAR` позволяют выполнить проверку числа записей, возвращаемых указанным в них подзапросом. Операнд `EXISTS (<ОтборМ>)` имеет значение `True` в случае, если при отборе записей возвращено непустое множество значений. Например, инструкцией

```
SELECT S_Name, S_Number
FROM Store
        WHERE EXISTS
             (SELECT C_Move
             FROM Cards
                WHERE
                C_Date BETWEEN "1.10.2002" AND "31.10.2002")
```

из таблицы `Store` склада отбираются названия и количества товаров, для которых было движение (приход или расход) за октябрь 2002 года. Движение товара анализируется с помощью запроса к таблице `Cards`.

Операнд `SINGULAR (<ОтборМ>)` имеет значение `True` в случае, если при отборе записей возвращено только одно значение. Например, так

```
SELECT Name
FROM Personnel
      WHERE SINGULAR
            (SELECT Position
            FROM Position
                  WHERE Position.Code = Personnel.Code)
```

из таблицы `Personnel` отбираются фамилии сотрудников организации, которые занимают только одну должность.

Операнд `PLAN` позволяет задать план выполнения запроса:

```
<План выполнения запроса> =
       [{JOIN | [SORT] MERGE}]
       (<Таблица>) | <План выполнения запроса>,
      ...
       [(<Таблица>) | <План выполнения запроса>])

<Таблица> = {<Имя таблицы> | <Псевдоним таблицы>}
      {NATURAL | INDEX (<Список индексов>) | ORDER <Список индексов>}
```

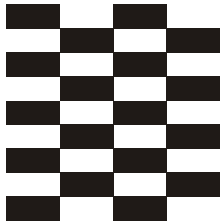
Задание плана выполнения запроса позволяет управлять методами доступа к данным и может увеличить скорость выполнения запроса. Для каждой таблицы, которая задается именем или псевдонимом, указанным после слова `FROM` инструкции `SELECT`, с помощью операндов можно определить:

- `NATURAL` — использование метода последовательного доступа к данным; если при поиске данных отсутствуют соответствующие индексы, то используется последовательный метод доступа;
- `INDEX` — использование для доступа к данным индексно-последовательного метода на основе указанных индексов;
- `ORDER` — сортировку таблицы по указанным индексам.

Описатель `JOIN` указывает, что план относится к связанным таблицам. Если для связанных таблиц существуют соответствующие индексы, то с их помощью будет реализован индексно-последовательный метод доступа. Если таких индексов нет, то для связанных таблиц следует указать описатель `MERGE` или `SORT MERGE`.

Инструкции `UPDATE`, `INSERT` и `DELETE` языка SQL сервера InterBase не имеют существенных отличий от своих аналогов, реализованных для локальной версии.

Глава 15



Технология InterBase Express

Общая характеристика

Технология InterBase Express ориентирована строго на работу с сервером InterBase версии не ниже 5.5. Отсюда следуют основные достоинства и недостатки этой технологии.

Преимущества технологии InterBase Express заключаются в том, все необходимые функции обеспечиваются путем прямого применения функций API сервера InterBase. В результате не нужно использовать BDE, повышается скорость работы компонентов доступа к данным.

Недостатком технологии InterBase Express является невозможность использовать серверы баз данных, отличные от сервера InterBase SQL Server.

Компоненты, предназначенные для работы по технологии InterBase Express, расположены на странице **InterBase** Палитры компонентов *(см. главу 2)*. Охарактеризуем кратко назначение основных из этих компонентов:

- `IBTable` — для получения данных из таблицы или представления базы данных. Полученный с помощью этого компонента набор данных является редактируемым. Является аналогом компонента `Table` для BDE, совместим с визуальными компонентами;
- `IBQuery` — для получения данных с помощью SQL-запроса. Является аналогом компонента `Query` для BDE, совместим с визуальными компонентами;
- `IBStoredProc` — для вызова хранимых процедур и получения набора данных на основе результатов выполнения процедуры. Соответствующий набор данных является нередактируемым. Совместим с визуальными компонентами;
- `IBDatabase` — для установления соединения с базой данных;
- `IBTransaction` — для управления транзакцией;
- `IBUpdateSQL` — для создания модифицируемых наборов данных, основанных на SQL-запросах. Является аналогом компонента `UpdateSQL` для BDE. Используется совместно с компонентом `IBQuery`;
- `IBDataSet` — для получения и редактирования данных. Совместим со всеми визуальными компонентами. Обеспечивает эффективный доступ к данным для просмотра и редактирования;

- `IBSQL` — для быстрого выполнения SQL-запроса с минимальными накладными расходами. Не имеет локального буфера данных, не совместим с визуальными компонентами;
- `IBDatabaseInfo` — для получения системной информации о свойствах базы данных, соединения и сервера;
- `IBSQLMonitor` — для перехвата и отслеживания SQL-запросов, которые выполняют приложения по технологии InterBase Express;
- `IBEvents` — для обработки событий сервера InterBase;
- `IBExtract` — для получения метаданных от сервера InterBase;
- `IBClientDataSet` — для получения данных и применения обновлений. Использует внутренние компоненты `TIBDataSet` и `TDataSetProvider`.

Компоненты `IBTable`, `IBQuery`, `IBStoredProc` и `IBUpdateSQL` во многом похожи на свои аналоги, рассмотренные при описании механизма BDE. Поэтому мы опишем свойства и методы, раскрывающие отличительные особенности их использования в технологии InterBase Express.

Установление соединения с сервером

За установление соединения с сервером БД отвечает компонент `IBDataBase`, являющийся аналогом компонента `DataBase` в технологии BDE и компонента `ADOConnection` в технологии ADO.

Имя базы данных, с которой устанавливается соединение, определяет свойство `DatabaseName` компонента `IBDataBase`. В случае локального набора данных InterBase это может быть имя файла базы данных с расширением gdb. Нажатие кнопки с многоточием в строке этого свойства в окне Инспектора объектов приводит к вызову стандартного диалога открытия файлов, с помощью которого можно выбрать нужный файл. В случае установления соединения с базой данных InterBase на удаленном сервере по протоколу TCP/IP значение свойства задается в виде:

```
<имя сервера>:<имя файла>
```

Параметры соединения с сервером можно установить также в диалоговом окне **Database Component Editor** (рис. 15.1), открываемом двойным щелчком на рассматриваемом компоненте, либо выбором пункта **Database Editor** его контекстного меню. В поле **Database** требуется задать спецификацию файла БД, а в поля **User Name** и **Password** — ввести имя пользователя и пароль.

Заданные таким образом параметры соединения автоматически отображаются в поле **Settings** рассматриваемого диалогового окна, они же являются значением свойства `Params` типа `TStrings` компонента `IBDataBase`.
С помощью флажка **Login Prompt** в окне **Database Component Editor** (см. рис. 15.1) задается необходимость появления окна запроса для указания имени пользователя и пароля при запуске приложения. При отказе от необходимости указания имени пользователя и пароля следует сбросить этот флажок.

Для проверки правильности соединения нужно нажать кнопку **Test**. По результатам тестирования выдается соответствующее сообщение.

Открыть соединение с базой данных можно в Инспекторе объектов при разработке приложения или при выполнении приложения, задав свойству `Connected` типа `Boolean` значение `True`. При выполнении приложения для открытия и закрытия соединения с базой данных можно воспользоваться методами `Open` и `Close` соответственно.

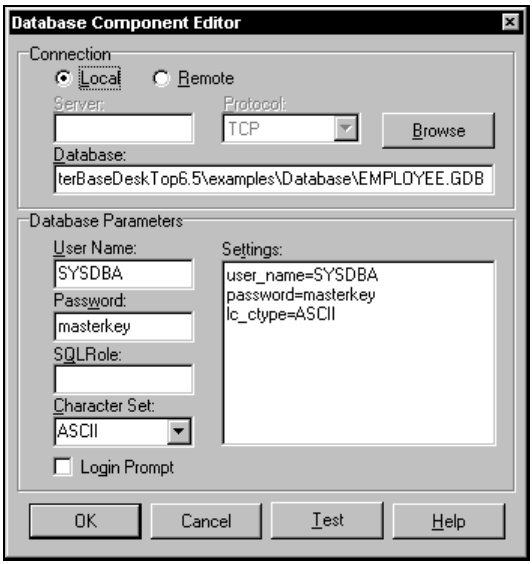


Рис. 15.1. Диалоговое окно **Database Component Editor**

Состояние соединения с сервером БД в ходе выполнения приложения можно проверить с помощью метода `CheckActive` или `CheckInactive` (противоположный вариант предыдущего метода).

Для получения списка имен таблиц в БД служит метод `GetTableNames(List: TStrings; SystemTables: Boolean = False);`. При задании параметру `SystemTables` значения `False` в этот список не попадут имена системных таблиц.

Для получения списка имен полей требуемой таблицы БД служит метод `GetFieldNames(const TableName: string; List: TStrings);`. Имя таблицы определяет параметр `TableName`.

Управление транзакциями

Для управления транзакциями при работе с сервером InterBase служит компонент `IBTransaction`. В разрабатываемом приложении БД может быть несколько

компонентов IBDataBase. Для компонента IBTransaction свойства Databases [Index: Integer] типа TIBDatabase и DatabaseCount типа Integer содержат соответственно список и общее число связанных с ним компонентов IBDataBase. Свойство DefaultDatabase типа TIBDatabase определяет, какой из компонентов IBDataBase используется по умолчанию при выполнении транзакции.

В свою очередь, в приложении может быть несколько компонентов управления транзакциями. Для компонента IBDataBase транзакцию по умолчанию можно задать через его свойство DefaultTransaction типа TIBTransaction. Его свойства Transactions [Index: Integer] типа TIBTransaction и TransactionCount типа Integer содержат список и общее число связанных с ним транзакций соответственно.

При выполнении приложения для транзакции можно установить и разорвать связь с соединением с помощью методов AddDatabase и RemoveDatabase соответственно.

Для запуска, фиксации и отката транзакции соответственно служат методы: StartTransaction, Commit и Rollback. Начиная с версии 6 сервер InterBase поддерживает два новых метода CommitRetaining и RollbackRetaining. Они оставляют транзакцию открытой после фиксации и после отката текущей транзакции соответственно.

Перед запуском транзакции целесообразно определить значение свойства InTransaction типа Boolean. Значение True означает, что не закончена предыдущая транзакция, и запуск новой транзакции без завершения предыдущей путем фиксации или отката приведет к исключению.

Допустимое время ожидания отклика на запущенную транзакцию задает свойство IdleTimer типа Integer. Если за указанный промежуток времени транзакция не будет завершена, то будет выполнено действие по умолчанию. Его определяет свойство DefaultAction типа TTransactionAction, который описан так:

```
type TTransactionAction = (taRollback, taCommit, taRollbackRetaining,
  taCommitRetaining);
```

Здесь:

- taRollback — откат транзакции;
- taCommit — фиксация транзакции;
- taRollbackRetainong — откат транзакции с сохранением ее контекста (начиная с 6 версии InterBase);
- taCommitRetaining — фиксация транзакции с сохранением ее контекста (начиная с 6 версии InterBase).

При управлении транзакциями целесообразно установить один из четырех возможных вариантов так называемого *уровня изоляции транзакции*. Удобно сделать это можно с помощью редактора Transaction Editor (рис. 15.2). Вызов его осуществляется с помощью одноименной команды контекстного меню компонента IBTransaction.

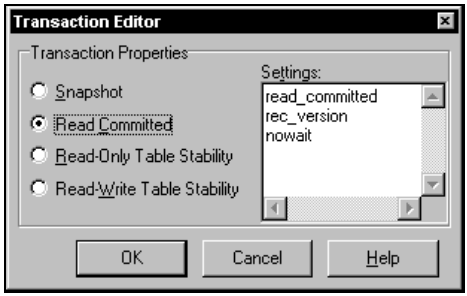


Рис. 15.2. Окно **Transaction Editor**

Уровень изоляции транзакции определяет, какие изменения, сделанные в других транзакциях, может видеть настраиваемая транзакция. Из числа возможных вариантов в окне редактора Transaction Editor обычно рекомендуется установка переключателя **Read Committed**. При этом запросы в одной транзакции могут просматривать изменения, внесенные и подтвержденные в контексте конкурирующих транзакций. Этот вариант уровня изоляции задается константой `read_committed` (возможность читать подтвержденные записи других транзакций). Такой уровень изоляции транзакций чаще всего используется для получения последнего состояния базы данных. Имеются две разновидности этого уровня изоляции: `rec_version` и `no_rec_version`. Вариант с параметром `rec_version` используется по умолчанию. Это означает, что при считывании записи считывается ее последняя подтвержденная версия.

Вариант **Snapshot** задается параметром `concurrency` (транзакция производит снимок маски транзакции в момент запуска). Изменения, производимые конкурирующими транзакциями, ей не видны. Ей доступны только свои изменения. Обычно такой вариант применяется для продолжительных запросов или для блокирования записей, чтобы предотвратить их изменение другими транзакциями.

Варианты **Read-Only Table Stability** и **Read-Write Table Stability** задаются параметром `consistency` (аналогичен параметру `concurrency`, дополнительно блокирует запись в таблицу). Указанные варианты различаются режимами доступа, которые соответственно задаются параметрами `read` (разрешает только операции чтения) и `write` (разрешает операции записи). Использование такого уровня изоляции поддерживает последовательные (сериализуемые) обновления таблицы. Такой уровень обычно используют для коротких обновляющих транзакций.

Компоненты доступа к данным

Стандартные компоненты (`IBTable`, `IBQuery`, `IBUpdateSQL` и `IBStoredProc`) доступа к данным по технологии InterBase Express наследуют механизм от родительского класса `TIBCustomDataSet`. Поэтому прежде всего представляет интерес рассмотрение основных его свойств и методов.

Стандартные компоненты доступа к данным подключаются к БД через компонент соединения `IBDataBase` с помощью своего свойства `Database` типа `TIBDatabase`.

Тип записей набора данных, для которых применима операция кэширования, определяет свойство `UpdateRecordTypes` типа `TIBUpdateRecordTypes`, который описан так:

```
type  TIBUpdateRecordTypes = set of (cusModified, cusInserted,
cusDeleted,
   cusUnmodified, cusUninserted);
```

Здесь:

- `cusModified` — измененные записи;
- `cusInserted` — добавленные записи;
- `cusDeleted` — удаленные записи;
- `cusUnmodified` — неизмененные записи;
- `cusUninserted` — недобавленные записи.

Свойство `ForcedRefresh` типа `Boolean` определяет, будет ли выполняться обновление набора данных при каждом сохранении внесенных изменений. Для ускорения работы приложения рекомендуется задать этому свойству значение `False` (действует по умолчанию). Чтобы обновление набора данных происходило как можно чаще, следует этому свойству задать значение `True`.

Генераторы для автоинкрементных полей

У ряда компонентов доступа к данным, реализующим запросы (например, `IBQuery` и `IBDataSet`), свойство `GeneratorField` типа `TIBGeneratorField` позволяет присваивать значения первичным ключам набора данных с помощью генератора, назначаемого с использованием редактора названного свойства.

С помощью этого свойства можно обеспечить автоматическое задание значения одного из автоинкрементных полей, указав при этом, какое поле должно получать значение и как вычисляться (шаг изменения и событие: вставка записи, сохранение записи, по команде сервера). Для задания рассматриваемого свойства используется редактор (рис. 15.3), вызываемый двойным щелчком мыши в строке свойства в окне Инспектора объектов.

В окне редактора следует выбрать название генератора, имя автоинкрементного поля в таблице, для которого будут генерироваться значения, и шаг увеличения счетчика (обычно 1). Кроме того, нужно выбрать вариант события, при наступлении которого будет происходить генерация значения поля: **On New Record** (при вставке новой записи), **On Post** (при сохранении записи) и **On Server** (по команде сервера с помощью триггера).

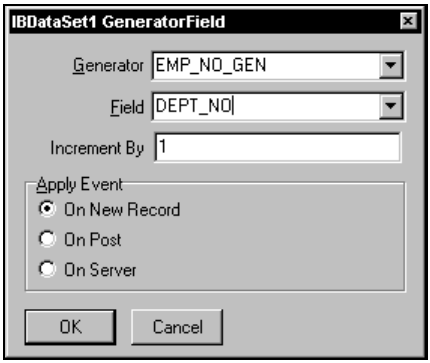


Рис. 15.3. Окно редактора свойства `GeneratorField`

Доступ к таблицам

Для доступа к таблицам по технологии InterBase Express служит компонент `IBTable`. Рассмотрим кратко отличительные свойства этого компонента.

Размер буфера по числу записей определяет свойство `BufferChunks` типа `Integer`.

Тип отношений, доступных в дополнение к пользовательским таблицам при выборе в раскрывающемся списке с помощью свойства `TableName`, определяет свойство `TableTypes` типа `TIBTableTypes`. Оно может принимать следующие значения:

- `ttSystem` — доступны системные таблицы и представления;
- `ttView` — доступны пользовательские представления.

Существует ли в БД таблица, указанная с помощью свойства `TableName`, определяет свойство `Exists` типа `Boolean`. Это свойство рекомендуется проверять перед открытием таблицы при выполнении приложения.

При использовании компонента `IBTable` указывается только имя таблицы с помощью свойства `TableName`. При этом автоматически формируется запрос на выборку всей таблицы со всеми полями. Если в таблице имеется очень много записей, то после выполнения такого запроса на сервер может лечь чрезмерно большая нагрузка. К примеру, при выполнении операций поиска нужной записи. Поэтому в крупных программных приложениях БД использование компонента `IBTable` не рекомендуется.

Выполнение запросов

Для выполнения SQL-запросов по технологии InterBase Express служит компонент `IBQuery`. Этот компонент является аналогом компонента `Query` в технологии BDE. Рассмотрим кратко отличительные свойства компонента `IBQuery`.

Установить значение свойства `SQL` типа `TStrings`, определяющего текст исполняемого SQL-запроса, можно с помощью редактора CommandText Editor, работа с которым описана в *главе 9*.

Параметры исполняемого SQL-запроса хранятся в свойстве `Params` типа `TParams`, а общее число параметров запроса возвращает свойство `ParamCount` типа `Word`.

Специалисты рекомендуют использовать компонент `Query` в основном при переносе имеющихся приложений под BDE на платформу технологии InterBase Express. При разработке новых приложений взамен него предлагается использовать компонент `IBDataSet`.

Получение и редактирование данных

В рассматриваемой технологии для выполнения SQL-запросов по выборке, вставке, удалению и обновлению записей в таблицах и представления полученных наборов данных используется компонент `IBDataSet`. Этот компонент реализует практически все возможности описанных выше компонентов доступа к данным по рассматриваемой технологии, именно его и целесообразно использовать при разработке приложений БД.

На этапе разработки и при выполнении приложения для компонента `IBDataSet` доступны следующие свойства типа `TStrings`, задающие различные типы запросов:

- `InsertSQL` — на добавление записей в набор данных;
- `DeleteSQL` — на удаление записей из набора данных;
- `ModifySQL` — на изменение записей в наборе данных;
- `SelectSQL` — запрос на выборку всех записей набора данных;
- `RefreshSQL` — на обновление текущей записи набора данных.

Тип запроса к набору данных определяется с помощью свойства `StatementType` типа `TIBSQLTypes`, который описан так:

```
type TIBSQLTypes = set of (SQLUnknown, SQLSelect, SQLInsert, SQLUpdate,
   SQLDelete, SQLDDL, SQLGetSegment, SQLPutSegment,
   SQLExecProcedure, SQLStartTransaction, SQLCommit,
   SQLRollback, SQLSelectForUpdate, SQLSetGenerator);
```

Здесь:

- `SQLCommit` — фиксация текущей (активной) транзакции;
- `SQLDDL` — выполнение команды DDL;
- `SQLDelete` — удаление записей в таблице или курсоре;
- `SQLExecProcedure` — вызов хранимой процедуры;
- `SQLGetSegment` — чтение сегмента BLOB;
- `SQLInsert` — добавление записей в таблицу;

- `SQLPutSegment` — запись сегмента BLOB;
- `SQLRollback` — восстановление состояния БД, предшествовавшего началу текущей транзакции;
- `SQLSetForUpdate` — хранимая процедура, устанавливается для обновления;
- `SQLSetGenerator` — установка нового значения для существующего генератора;
- `SQLSelect` — возвращение данных из одной или нескольких таблиц (SQL-запрос);
- `SQLStartTransaction` — начало новой транзакции;
- `SQLUnknown` — неизвестный тип SQL;
- `SQLUpdate` — изменение записи в таблице, представлении или курсоре.

В целом компонент `IBDataSet` предназначен для выполнения модифицирующих запросов с целью изменения данных, полученных с помощью запроса на выборку, задаваемого в свойстве `SelectSQL`.

Рассмотрим технику выполнения модифицирующих запросов с использованием свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`. Предположим, что с помощью запроса на выборку, заданного в свойстве `SelectSQL`, мы получили набор данных, содержащий записи с нужными полями заданной таблицы. Пусть нам требуется отредактировать содержимое одного из или нескольких полей некоторой записи. При этом после внесения изменений, например, в компоненте `DBGrid`, эти изменения будут внесены в локальный буфер, а в базе данных на сервере ничего не изменится. Для внесения соответствующих изменений в базе данных на сервере нужно выполнить оператор UPDATE языка SQL. Причем этот запрос должен быть указан в качестве значения свойства `ModifySQL` рассматриваемого компонента.

К примеру, если редактируемый набор данных для таблицы с именем `'COUNTRY'` содержит поля с именами `'COUNTRY'` и `'CURRENCY'`, то соответствующий запрос на изменение записи в свойстве `ModifySQL` может содержать следующий текст:

```
UPDATE COUNTRY
SET
COUNTRY = :COUNTRY,
CURRENCY = :CURRENCY
where
  COUNTRY = :OLD_COUNTRY
```

В приведенном запросе вместо значений указаны параметры, названия которых совпадают с названиями полей редактируемой таблицы. Это означает, что при изменении содержимого полей записи рассматриваемый компонент автоматически установит значения всех параметров через текущие значения полей или не-

которым другим образом. Префикс OLD_ в названии параметра означает, что в параметр будет подставлено старое содержимое поля до его изменения пользователем.

Разработчик должен указать, каким именно образом происходит определение параметров. К примеру, возможный вариант вызова оператора UPDATE с заданием параметров в обработчике события нажатия кнопки может иметь следующий вид:

```
procedure TForm1.Button1Click(Sender: TObject);
begin     {изменение записи}
with  IBDataSet1 do
  begin
   Edit;
   FieldByName('COUNTRY').AsString := Edit1.Text;
   FieldByName('CURRENCY').AsString :=
trim(FieldByName('CURRENCY').AsString);
   Post;
  end;
end;
```

Как видно из приведенного примера, вызов оператора UPDATE осуществляется с помощью метода Edit класса TIBDataSet. А именно, путем вызова метода Edit выполняется подготовка буфера текущей записи для редактирования, с помощью метода FieldByName задаются значения параметров оператора UPDATE и тем самым изменяются значения полей. С помощью метода Post осуществляется сохранение внесенных изменений в базе данных на сервере.

Для вставки и удаления записей в используемом нами классе TIBDataSet1 служат методы Insert и Delete соответственно. Для запросов, указываемых в свойствах InsertSQL и DeleteSQL, задается аналогичная последовательность действий.

Рассмотренные нами запросы для компонента IBDataSet называют *динамическими* или *"живыми"* (live query).

Важным достоинством рассматриваемого компонента является то, что после выполнения любого действия по редактированию набора данных он выполняет запрос, указанный в свойстве RefreshSQL. Этот запрос выполняет возвращение из базы данных одной — текущей записи и предназначен для обновления полей этой записи после внесенных изменений. Для нашего примера в свойстве RefreshSQL должен быть следующий код:

```
Select
  COUNTRY,
  CURRENCY
```

```
from COUNTRY
where
  COUNTRY = :COUNTRY
```

Пояснить достоинство такого механизма можно следующим образом. Если в базе данных используются триггеры, модифицирующие значения полей базы данных, то внесенные в результате выполнения триггеров изменения пользователю не видны. Это означает, что для просмотра возможных изменений нужно выбрать нашу запись из базы данных с помощью оператора SELECT или полностью повторить выборку всех записей с помощью запроса из свойства `SelectSQL`. Такой вариант работы с измененными записями реализован при работе с помощью механизма BDE. Использование свойства `RefreshSQL` в рассматриваемой технологии позволяет избежать необходимости повторной выборки записей всего набора данных.

Для автоматизации построения модифицирующих запросов, т. е. задания свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`, служит редактор. Вызов редактора выполняется с помощью команды **Dataset Editor** контекстного меню компонента `IBDataSet`. Причем делается это после того, как задано его свойство `SelectSQL`.

На вкладке **Options** окна редактора модифицирующих запросов (рис. 15.4) выполняется выбор таблицы из списка **Table Name**. Далее с помощью кнопки **Get Table Fields** осуществляется формирование списков **Key Fields** и **Update Fields**. В списке **Key Fields** нужно выделить ключевые поля, на основе которых будет задаваться условие WHERE в создаваемых запросах. Если в выбранной таблице ключевые поля существуют, то для их выделения нужно нажать кнопку **Select Primary Keys**.

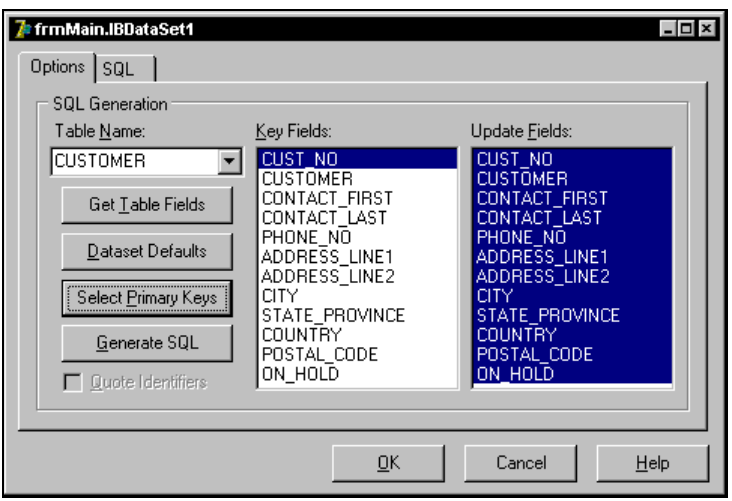


Рис. 15.4. Окно редактора модифицирующих запросов

Выделение полей, для которых допускается редактирование, выполняется с помощью списка **Update Fields**. В состав этого списка нельзя включать вычисляемые поля, поскольку их содержимое редактировать не допускается.

Нажатие кнопки **Generate SQL** вызывает генерацию запросов. На вкладке **SQL** окна редактора отображаются тексты автоматически сгенерированных модифицирующих запросов, т. е. значения свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`.

В компоненте `IBDataSet` реализованы обработчики трех событий:

```
property DatabaseDisconnected: TNotifyEvent;
property DatabaseDisconnecting: TNotifyEvent;
property DatabaseFree: TNotifyEvent;
```

Соответствующие события возникают после отключения БД, во время отключения БД и после высвобождения памяти компонентом соединения соответственно.

Компонент *IBSQL*

Компонент `IBSQL` обеспечивает объект для выполнения SQL-запроса с минимальными накладными расходами. Так, для каждого запроса к набору данных, заданного в компоненте `TIBDataSet`, формируется объект `TIBSQL`. Рассматриваемый компонент получает набор данных в виде однонаправленного курсора и не обеспечивает навигацию по набору данных. Компонент `IBSQL` не имеет локального буфера для набора данных и несовместим с визуальными компонентами.

Рассмотрим основные отличия компонента `IBSQL` от его аналогов для технологий BDE, dbExpress и ADO Express.

Список имен параметров запроса создает свойство `GenerateParamNames` типа `Boolean`, план запроса содержит свойство `Plan` типа `String`.

Число возвращенных после выполнения запроса записей содержит свойство `RecordCount` типа `Integer`.

Число записей, обработанных SQL-операторами `INSERT`, `DELETE` или `UPDATE`, содержит свойство `RowsAffected` типа `Integer`.

При работе приложения для баз данных зачастую полезно знать состояние набора данных. С этой целью при открытии запроса могут быть использованы следующие методы вызова исключений:

- `CheckClosed;` — если набор данных не закрыт;
- `CheckOpen;` — если набор данных не открыт;
- `CheckValidStatement;` — если запрос синтаксически неправилен.

Выскажем короткое соображение по тому, в каких случаях предпочтительно использовать компонент `IBSQL`. Для примера можно отметить вариант приложе-

ния, в котором требуется выполнять массовую вставку записей в базу данных и при этом нет необходимости отображать вставляемые записи в пользовательских компонентах. В этом случае целесообразно использовать компонент IBSQL, поскольку он не буферизует записи в отличие от IBDataSet. Тем самым мы достигаем более быстрой работы приложения.

Пример приложения

Рассмотрим пример приложения (рис. 15.5), построенного с использованием компонентов InterBase Express: IBDataSet, IBDataBase, IBTransaction, DataSource и нескольких вспомогательных компонентов DBEdit, DBGrid и Button.

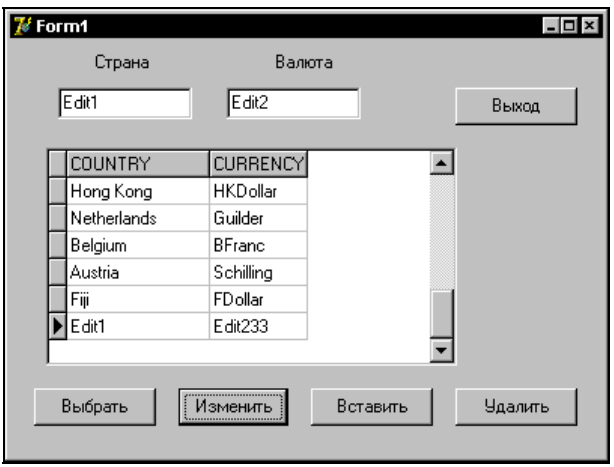


Рис. 15.5. Вид формы при выполнении приложения

В примере демонстрируется возможность выборки, изменения, вставки и удаления записей БД, отображаемой с помощью компонента DBGrid.

Код модуля формы Unit1DataSet для решения поставленной задачи имеет вид:

```
unit Unit1DataSet;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, Grids, DBGrids, IBCustomDataSet, IBDatabase, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    IBDatabase1: TIBDatabase;
    IBTransaction1: TIBTransaction;
    IBDataSet1: TIBDataSet;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Edit1: TEdit;
    Label1: TLabel;
    Edit2: TEdit;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button2Click(Sender: TObject);
begin     {вставка записи}
with  IBDataSet1 do
begin
 Insert;
 FieldByName('COUNTRY').AsString := Edit1.Text;
 FieldByName('CURRENCY').AsString := Edit2.Text;
```

```
 Post;
end;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin     {выборка записей}
  IBDataSet1.Active := False;
  IBDataSet1.Active := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin     {изменение записи}
with  IBDataSet1 do
  begin
   Edit;
   FieldByName('COUNTRY').AsString := Edit1.Text;
   FieldByName('CURRENCY').AsString :=
trim(FieldByName('CURRENCY').AsString);
   Post;
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin     {удаление записи}
 IBDataSet1.Delete;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin     {выход}
 IBDataSet1.Active := False;
 Form1.Close;
end;
end.
```

В обработчике события нажатия кнопки `Button4` (с заголовком **Выбрать**) компонент `IBDataSet1` устанавливается активным, тем самым выполняется запрос на выборку всех записей (свойство `SelectSQL`).

В обработчике события нажатия кнопки `Button1` (с заголовком **Изменить**) новое значение поля с именем `'COUNTRY'` берется из свойства `Edit1.Text`, в то время

как новое значение поля с именем `'CURRENCY'` определяется как результат редактирования этого же поля.

В обработчике события нажатия кнопки `Button2` (с заголовком **Вставить**) с помощью метода `Insert` выполняется запрос из свойства `InsertSQL` на вставку записи в набор данных, здесь же задаются значения параметров запроса. Имена полей для вставляемой записи берутся из свойств `Edit1.Text` и `Edit2.Text` редакторов.

В обработчике события нажатия кнопки `Button3` (с заголовком **Удалить**) с помощью метода `Delete` выполняется запрос из свойства `DeleteSQL` на удаление текущей записи из набора данных.

Глава 16

Инструменты для работы с удаленными базами данных

Ранее были рассмотрены программные инструменты для работы с БД. Большинство этих программ можно использовать и для работы с удаленными БД. Так, создать псевдоним для InterBase позволяют программы Administrator BDE и Database Desktop, а редактировать и просматривать записи можно с помощью программы SQL Explorer. В дополнение к ним в данной главе мы познакомимся с программами, предназначенными специально для удаленных БД.

Программа IBConsole

Программа IBConsole предназначена для управления сервером InterBase и является его *консолью*. Консоль устанавливается совместно с сервером InterBase и находится в его каталоге BIN, ее главный файл называется IBConsole.exe. Программу также можно запустить через меню **Пуск** Windows, выбрав команду **Программы\InterBase\IBConsole**.

Для запуска и остановки сервера служит программа InterBase Server Manager, функциональность которой начиная с версии Delphi 6 значительно уменьшилась.

Программа IBConsole обеспечивает:

- управление локальными и удаленными серверами;
- управление БД;
- интерактивное выполнение SQL-запросов.

Основную часть окна IBConsole занимают две панели (рис. 16.1). В левой панели в виде дерева представлены зарегистрированные серверы и их БД, а также элементы структуры, например, таблицы или пользователи. Отметим, что в качестве имени сервера или БД отображается его псевдоним (alias), задаваемый при регистрации сервера или БД (этот псевдоним никак не связан с псевдонимом BDE). В правой панели выводится информация об объекте, выбранном в левой панели.

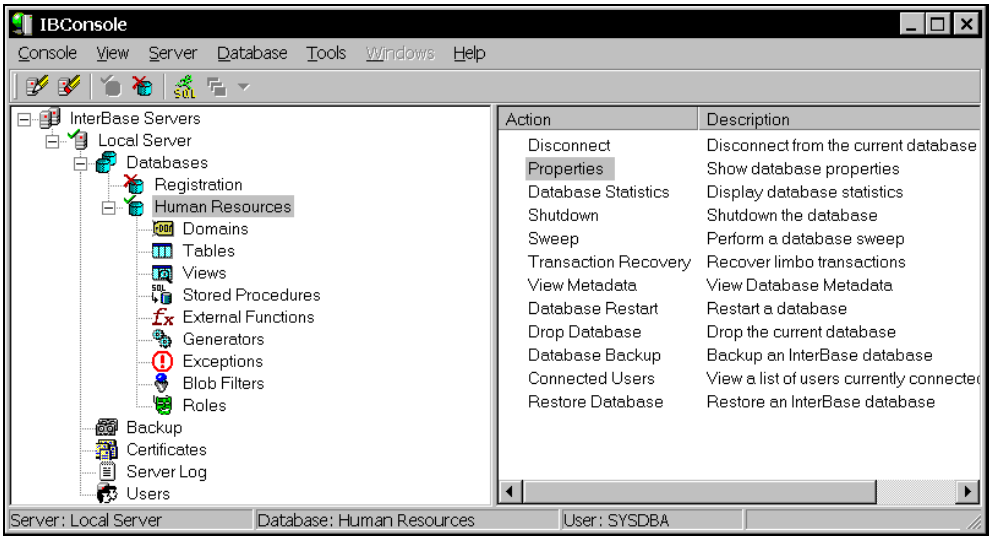


Рис. 16.1. Главное окно программы IBConsole

Управление сервером

Управление сервером заключается в:

- регистрации сервера;
- подключении сервера;
- просмотре протокола работы;
- управлении сертификатами;
- определении пользователей.

Для управления сервером используются команды меню **Server** главного окна программы IBConsole, а также команды контекстного меню сервера и его элементов.

После запуска консоли в левой панели отображается список зарегистрированных серверов, которые первоначально отключены, о чем свидетельствует красный крестик в значке сервера.

Подключение к серверу

Подключение к серверу, выбранному в левой панели, выполняется командой **Login**. При этом появляется окно **Server Login** (рис. 16.2), в котором необходимо указать имя пользователя (**User Name**) и его пароль (**Password**). После указания имени `SYSDBA` системного администратора, его пароля `masterkey` и нажатия кнопки **Login** осуществляется подключение к серверу, имя (alias) которого отображается в надписи **Server**, а к значку сервера добавляется зеленая галочка.

Отключение от выбранного сервера выполняется командой **Logout**. При этом выдается запрос на продолжение операции и в случае подтверждения сервер отключается.

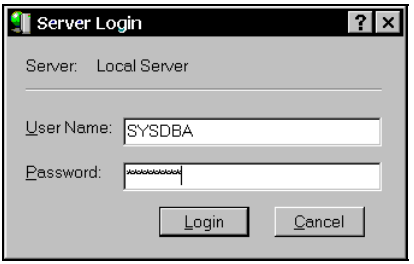


Рис. 16.2. Подключение к серверу

После подключения к серверу можно выполнить *проверку подключения* к одной из его БД. Командой **Diagnose Connection** открывается окно проверки соединения **Communication Diagnostics** (рис. 16.3), в котором указывается информация о сервере. В качестве БД задается ее файл, который можно выбрать в окне **Open**, открываемом нажатием кнопки с тремя точками.

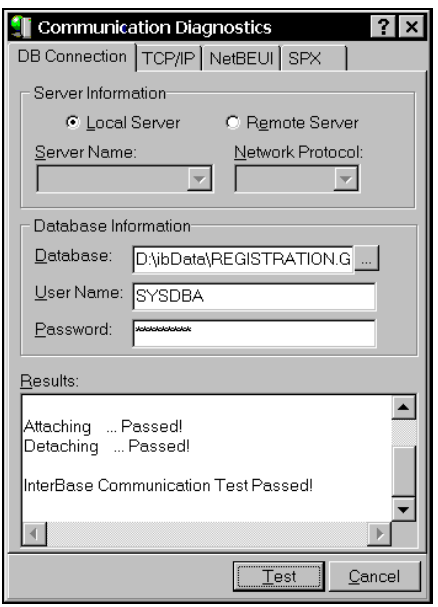


Рис. 16.3. Проверка подключения сервера к БД

Кнопка **Test** инициирует проверку, результаты которой выводятся в поле **Results**.

На страницах **TCP/IP**, **NetBEUI**, **SPX** можно выполнить настройку соответствующего сетевого протокола.

Регистрация сервера

Для *регистрации* в консоли нового сервера необходимо выполнить команду **Register**, после чего открывается окно регистрации и соединения с сервером **Register Server and Connect** (рис. 16.4).

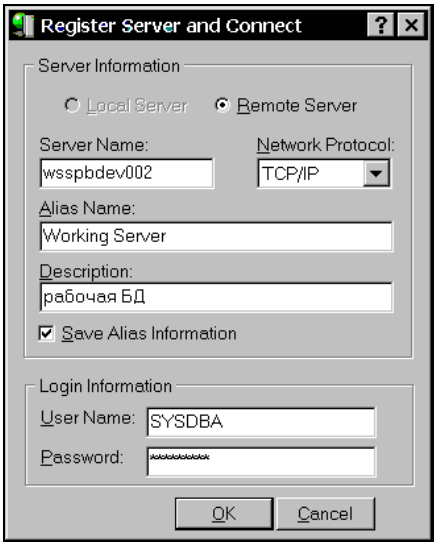


Рис. 16.4. Регистрация сервера

При регистрации локального сервера устанавливается переключатель **Local Server**. При необходимости в поле описания (**Description**) можно ввести краткую информацию, поясняющую назначение и особенности сервера. После нажатия кнопки **OK** локальный сервер регистрируется, а его имя добавляется к списку в левой панели.

В случае регистрации удаленного сервера устанавливается переключатель **Remote Server** и заполняются поля, которые заблокированы при подключении локального сервера: в поле **Server Name** указывается сетевое имя сервера, в списке **Network Protocol** выбирается протокол связи, а в поле **Alias Name** задается имя (псевдоним), под которым сервер регистрируется в консоли.

Одновременно с регистрацией можно выполнить подключение сервера, для чего должны быть заполнены поля **User Name** и **Password** группы **Login Information**.

Отмена регистрации выбранного сервера выполняется командой **Un-Register**. При этом выдается запрос на продолжение операции, и в случае подтверждения сервер исключается из консоли. Перед отменой регистрации сервера его необходимо отключить.

Просмотр протокола работы сервера

В процессе работы сервера ведется *протокол* (log file), просмотреть который можно, вызвав команду **View Logfile**. При этом появляется окно протокола **Server Log** (рис. 16.5), в котором выводится краткий отчет о работе сервера.

Протокол можно распечатать или сохранить в текстовом файле.

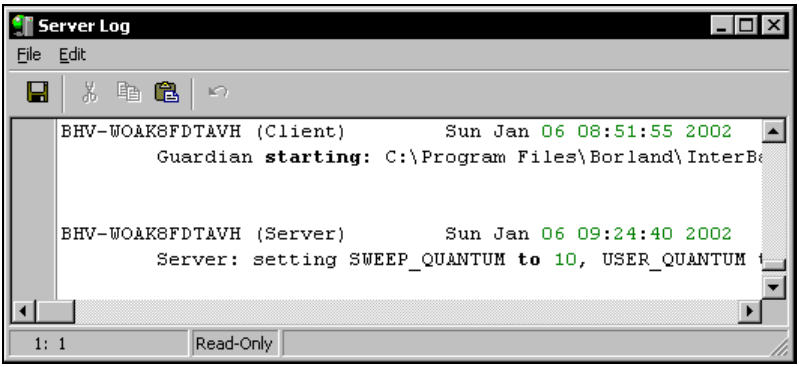


Рис. 16.5. Просмотр протокола работы сервера

Операции с сертификатами

Список сертификатов (certificate — удостоверение), действительных для сервера, можно просмотреть в правой панели, выбрав в левой панели элемент **Certificates**. Сертификат можно *добавить*, открыв командой **Add Certificate** одноименное окно **Add Certificate** (рис. 16.6) и указав в нем код (**ID**) и ключ (**Key**) сертификата. После нажатия кнопки **OK** выполняется проверка сертификата, и, если ошибки не найдены, новый сертификат добавляется к серверу.

Выбранный в списке сертификат *удаляется* командой **Remove Certificate**. Перед удалением выдается запрос на подтверждение операции.

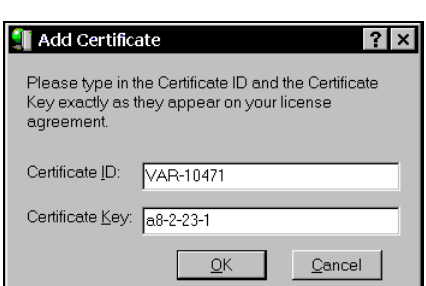


Рис. 16.6. Добавление сертификата к серверу

Управление пользователями

Для каждого сервера есть список пользователей, имеющих право доступа к нему. Список таких пользователей можно просмотреть в правой панели, выбрав в левой панели элемент **Users**. Консоль позволяет добавить или удалить пользователя, а также просмотреть и изменить его атрибуты. Эти действия (кроме удаления) выполняются в окне **User Information** (рис. 16.7), открываемом командой **User Security** или командами контекстного меню списка пользователей.

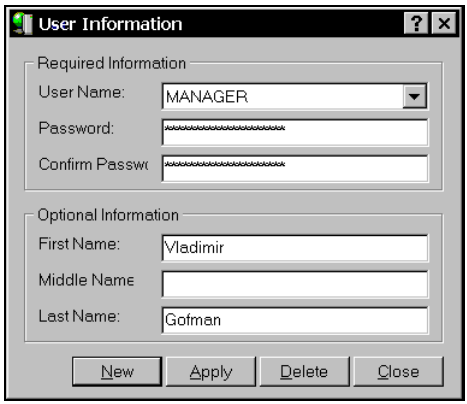


Рис. 16.7. Атрибуты пользователя

Для пользователя необходимо ввести обязательные атрибуты (**Required Information**), которыми являются его системное имя (**User Name**), пароль (**Password**) и повтор пароля (**Confirm Password**). Имя пользователя также можно выбрать в раскрывающемся списке. Остальные атрибуты являются дополнительными (**Optional Information**) и не требуют обязательного ввода: имя (**First Name**), отчество (**Middle Name**) и фамилия (**Last Name**).

Управление БД

Список баз данных, зарегистрированных для сервера, отображается в левой панели. Так, на рис. 16.1 для локального сервера (Local Server) такими БД являются `Registration` и `Human Resources`. Зарегистрированная БД может быть подключена или отключена от сервера, что отмечается, соответственно, зеленой галочкой или красным крестом в значке БД.

Управление БД заключается в:

- регистрации БД;
- подключении БД;
- создании и удалении БД;
- просмотре метаданных;
- проверке состояния БД;
- анализе статистики;
- сохранении и восстановлении БД.

Для управления БД используются команды меню **Database** главного окна программы IBConsole, а также команды контекстного меню баз данных и их элементов.

Регистрация базы данных

Регистрация БД начинается командой **Register**, которая открывает окно **Register Database and Connect** (рис. 16.8). В этом окне необходимо указать (выбрать) главный файл БД (**File**), а также псевдоним (имя) БД (**Alias**), под которым она будет зарегистрирована на сервере, обозначенном надписью **Server**. По умолчанию в качестве псевдонима БД предлагается имя ее главного файла с расширением.

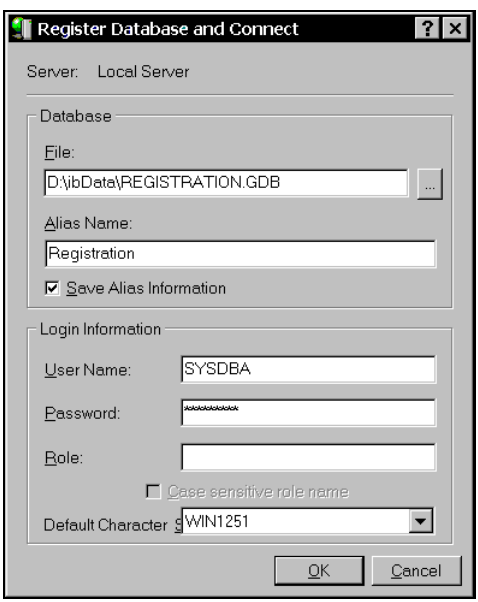


Рис. 16.8. Регистрация БД на сервере

Остальные данные не являются обязательными. Однако если задать имя и пароль пользователя, то после регистрации выполняется подключение указанной БД.

Исключение БД из списка регистрации сервера выполняется командой **Unregister**, при этом БД предварительно должна быть отключена от сервера. При выполнении операции запрашивается подтверждение.

Подключение базы данных

Подключение БД к серверу и *отключение* от него выполняется командами **Connect** и **Disconnect** соответственно. Команда **Connect As** позволяет подключиться к БД с новыми параметрами, указываемыми в окне **Database Connect** (рис. 16.9).

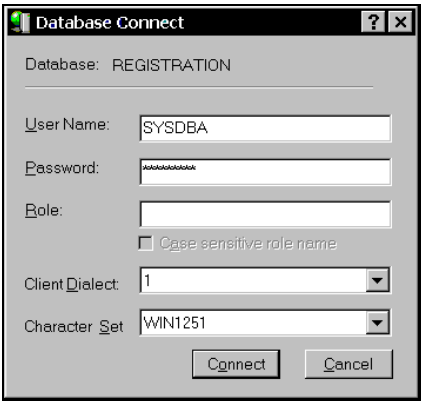


Рис. 16.9. Подключение БД к серверу

Создание базы данных

Консоль позволяет достаточно удобно и быстро *создавать* БД, в том числе многофайловые. Создание БД, а также ее удаление, для которых предназначены команды **Create Database** и **Drop Database**, рассмотрены в *главе 14*.

Просмотр метаданных

Метаданные представляют собой элементы структуры БД. Для выбранной БД их можно просмотреть с помощью команды **View Metadata**, которая открывает окно **Database Metadata**. На рис. 16.10 показаны метаданные БД `emloyee.gdb`, которая поставляется вместе с сервером InterBase в качестве примера.

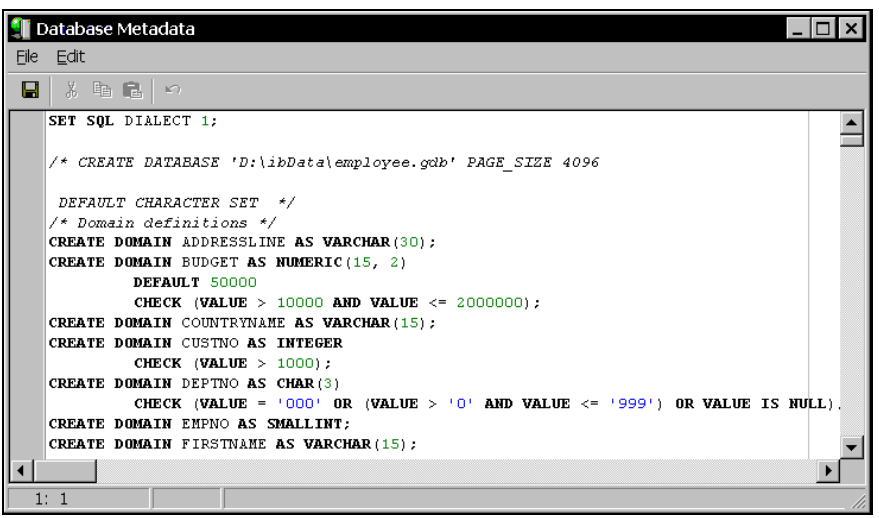


Рис. 16.10. Просмотр метаданных

Метаданные представляют собой сценарий (скрипт), написанный на языке SQL, который можно распечатать или сохранить в текстовом файле. Сохраненный сценарий впоследствии можно *выполнить*, создав БД со всей ее структурой и данными.

Сбор мусора

В процессе интенсивной многопользовательской работы в БД накапливается так называемый "мусор", под которым понимают старые версии записей, которые могут образовываться при одновременном доступе к записям нескольких транзакций. Наличие мусора увеличивает размер и фрагментацию БД, поэтому БД надо периодически чистить — "удалять мусор" (удаление мусора также называют "сбором мусора").

Удаление мусора можно выполнять в ручном или автоматическом режимах. В ручном режиме удаление мусора начинается командой **Maintenance\Sweep** (Обслуживание\Чистка). Во втором режиме удаление мусора начинается автоматически, когда общее число примененных к БД транзакций достигает предельного значения. По умолчанию это значение установлено равным 20 000, его можно изменить в поле **Sweep Interval** (Интервал чистки) окна свойств БД (**Database Properties**).

Если при чистке БД работают активные пользователи, то это снижает эффективность удаления мусора, т. к. используемые транзакциями записи не могут быть обработаны "уборщиком". Поэтому удаление мусора следует проводить в периоды наименьшей загрузки БД, например, в ночные часы или в режиме монопольного доступа к ней системного администратора.

Мусор удаляется также при резервном копировании и последующем восстановлении БД.

Проверка состояния базы данных

БД должна находиться в целостном и непротиворечивом состоянии, т. е. содержать правильные данные. Для проверки состояния БД нужно выбрать команду **Maintenance\Validation**, которая открывает диалоговое окно проверки БД **Database Validation** (рис. 16.11).

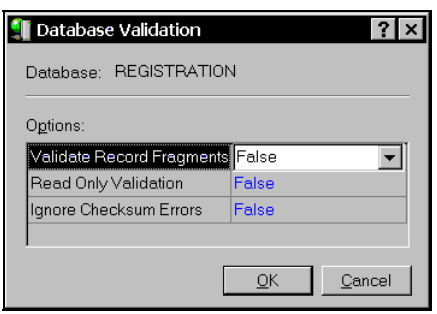


Рис. 16.11. Проверка состояния БД

В надписи **Database** отображается имя проверяемой БД, а группа **Options** позволяет задать параметры поверки:

- **Validate Record Fragments** (проверка структуры БД и структуры страниц);
- **Read Only Validation** (в процессе проверки допускается только читать, но не изменять записи);
- **Ignore Checksum Errors** (ошибки контрольных сумм игнорируются).

По умолчанию все параметры имеют значение `False`, т. е. выключены. После нажатия кнопки **OK** выполняется проверка, о результатах которой выдается соответствующий отчет в окне **Validation Report**.

Анализ статистики

В процессе управления БД собирается определенная информация, характеризующая ее работу и функционирование. Эта информация называется *статистикой*, несмотря на то что часть ее является управляющей информацией, определяемой в том числе и при создании БД (например, размер страницы или дата создания БД). К собственно статистике относятся такие сведения, как частота обновления заголовка БД и гистограмма заполнения страниц.

Для вывода статистики нужно выбрать команду **Maintenance\Database Statistics**, которая открывает диалоговое окно **Database Statistics** (рис. 16.12).

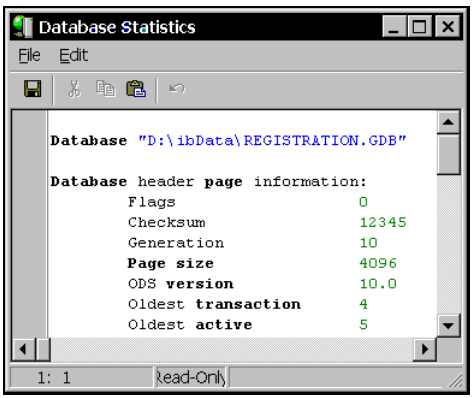


Рис. 16.12. Статистика БД

Выводимые в окне сведения о БД сгруппированы по секциям:

- `Database` (имя БД);
- `Database header page information` (страница заголовка БД):
  - `Flags` (флаги);
  - `Checksum` (контрольная сумма);
  - `Generation` (счетчик обновлений заголовка);
  - `Page size` (размер страницы);

- `ODS version` (версия формата файла БД);
- `Oldest transaction` (номер самой старой незавершенной (активной, отмененной или сбойной) транзакции);
- `Oldest active` (номер самой старой активной транзакции);
- `Next transaction` (номер, который будет назначен следующей транзакции);
- `Sequence number` (номер первой страницы);
- `Next attachment ID` (номер, который будет назначен следующему соединению);
- `Implementation ID` (идентификатор операционной системы, в которой создана БД);
- `Shadow count` (число теневых файлов, определенных для БД);
- `Page buffers` (номер страницы в кэше БД);
- `Next header page` (номер, который будет назначен следующей странице заголовка);
- `Creation date` (дата создания БД);
- `Attributes` (атрибуты БД);

❒ `Database file sequence` (список файлов БД):

- `File` (имя файла);

❒ `Database log page information` (страница журнала БД).

Сохранение и восстановление базы данных

Сохранение БД заключается в создании резервной копии БД, которую впоследствии можно использовать для восстановления данных при сбое. Предварительно БД необходимо отключить.

Для создания резервной копии БД нужно выполнить команду **Maintenance\Backup-Restore\Backup**, которая откроет окно создания резервной копии **Database Backup** (рис. 16.13).

В списке **Alias** группы **Database** выбирается имя сохраняемой БД, а элементы группы **Backup File(s)** определяют файл результата — сохраненной копии БД. В комбинированных списках **Server** и **Alias** выбирается или вводится соответственно имя сервера и БД для сохраняемой БД, а в поле **Filename(s)** вводится полное имя файла копии БД. В приведенном на рис. 16.13 варианте в имени копии указано слово `Copy`, а расширение gdb оставлено без изменений. В примерах, поставляемых совместно с сервером InterBase, имена копий совпадают с именами исходных файлов, а расширения изменены на gbk.

Группа **Options** позволяет задать следующие параметры:

❒ **Format** (Формат) — формат копии, которая по умолчанию создается в переносимом формате, не зависящем от операционной системы: сделанную в этом

формате копию можно перенести (восстановить) на компьютер под управлением любой из операционных систем, где установлен сервер InterBase;

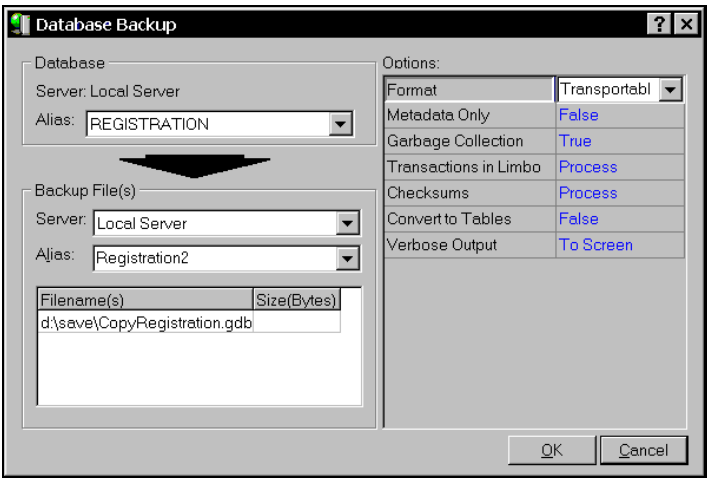


Рис. 16.13. Создание резервной копии БД

- **Metadata Only** (Только метаданные) — копируются только метаданные БД, т. е. ее структура; в результате будет скопирована (создана) пустая БД; по умолчанию параметр имеет значение `False`, поэтому создается полная копия БД;
- **Garbage Collection** (Сбор мусора) — при копировании производится сбор мусора (по умолчанию);
- **Transaction in Limbo** (Транзакция в "забвении") — учитываются сбойные транзакции, которые могут возникнуть при размещении БД на нескольких серверах (значение `Process` по умолчанию); значение `Ignore` предписывает не учитывать сбойные транзакции;
- **Checksums** (Контрольные суммы) — учитываются ошибки, связанные с несовпадением контрольных сумм (значение `Process` по умолчанию); при значении `Ignore` ошибки контрольных сумм игнорируются;
- **Convert to Tables** (Преобразование в таблицы) — преобразование в таблицы, по умолчанию `False`;
- **Verbose Output** (Расширенный вывод) — указывает устройство, на которое в процессе копирования выдается дополнительная информация о ходе процесса, по умолчанию экран (значение `To Screen`); кроме экрана можно указать файл (`To File`) или отменить выдачу сообщений (`None`).

После завершения процесса выдается отчет о результатах, который в случае успешного создания копии выводится в окне **Database Backup** и имеет следующий вид:

```
gbak: readied database D:\ibData\REGISTRATION.GDB for backup
gbak: creating file d:\save\CopyRegistration.gdb
```

```
gbak: starting transaction
gbak: database D:\ibData\REGISTRATION.GDB has a page size of 4096 bytes.
gbak: writing domains
gbak: writing shadow files
gbak: writing tables
gbak: writing functions
...
gbak: writing privilege for user SYSDBA
gbak: writing privilege for user PUBLIC
gbak: writing table constraints
gbak: writing referential constraints
gbak: writing check constraints
gbak: writing SQL roles
gbak: closing file, committing, and finishing. 512 bytes written
```

При наличии ошибок копия не создается, а отчет содержит сообщения об ошибках.

Замечание

Если резервное копирование выполняет не системный администратор, а пользователь, то ошибки могут быть связаны с отсутствием у него прав доступа к информации БД.

Резервная копия представляет собой архивный файл, размер которого в несколько раз меньше, чем размер исходного файла (файлов) БД.

Процесс восстановления БД из резервной копии БД инициируется командой **Maintenance\Backup-Restore\Restore**, открывающей диалоговое окно **Database Restore** (рис. 16.14).

В списке **Alias** группы **Backup File(s)** выбирается имя сохраненной БД, после чего имя ее файла-копии автоматически выводится в поле **Filename(s)**. Если имя БД отсутствует в списке, то можно задать ее файл, выбрав в списке элемент **File** и указав его имя в открывшемся окне **Open**. Элементы группы **Database** определяют сервер (**Server**), имя БД (**Alias**) и главный файл БД (**Filename(s)**).

Группа **Options** содержит следующие параметры восстановления:

- **Page Size (Bytes)** — размер в байтах страницы восстанавливаемой БД (по умолчанию 1024);
- **Overwrite** (Перезаписать) — если расположение и имя восстанавливаемой копии совпадают с расположением и именем существующей БД, то последняя будет заменена копией; по умолчанию имеет значение `False`, т. е. сохраняется существующая БД;
- **Commit After Each Table** (Подтверждение после каждой таблицы) — при восстановлении каждой таблицы выдается запрос на подтверждение этой операции; по умолчанию этот параметр имеет значение `False`, и запрос не выдается;

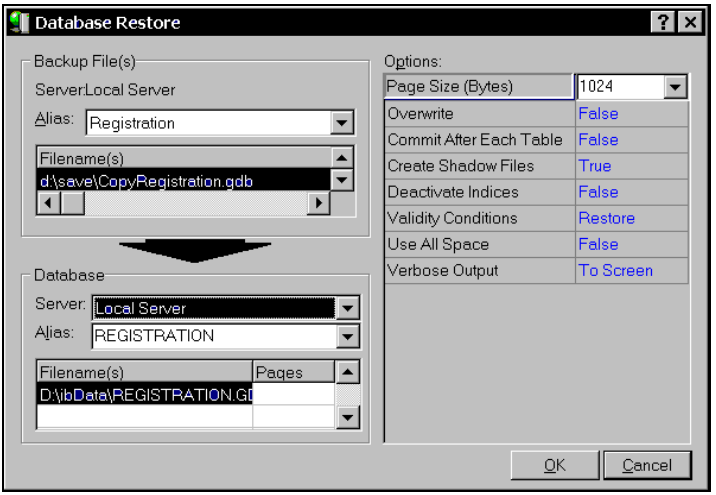


Рис. 16.14. Восстановление БД

- **Create Shadow Files** (Создание теневых файлов); по умолчанию `False`, т. е. восстановление выполняется без создания теневой (зеркальной) копии БД;
- **Deactivate Indices** (Отключить индексы) — восстановление выполняется с отключенными индексами; по умолчанию этот параметр имеет значение `False`, и индексы активны;
- **Validity Conditions** (Условия проверки) — при значении `Restore` (по умолчанию) выполняется восстановление ограничений ссылочной целостности, при значении `Ignore` ограничения не восстанавливаются;
- **Use All Space** (Использовать все имеющееся пространство); по умолчанию `False`;
- **Verbose Output** (Расширенный вывод) — указывает устройство, на которое в процессе копирования выдается дополнительная информация о ходе процесса, по умолчанию экран (значение `To Screen`); кроме экрана, можно указать файл (`To File`) или отменить выдачу сообщений (`None`).

После завершения процесса отчет о результатах, который в случае успешного восстановления БД выводится в окне **Database Restore** и имеет следующий вид:

```
gbak: opened file d:\save\CopyRegistration.gdb
gbak: transportable backup -- data in XDR format
gbak:     backup file is compressed
gbak: Reducing the database page size from 4096 bytes to 1024 bytes
gbak: created database D:\ibData\REGISTRATION.GDB, page_size 1024 bytes
gbak: started transaction
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
```

```
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user SYSDBA
gbak:     restoring privilege for user PUBLIC
gbak: creating indexes
gbak: finishing, closing, and going home
```

Интерактивное выполнение SQL-запросов

Консоль IBConsole позволяет в интерактивном режиме выполнять инструкции, заданные на языке SQL. Выполнение SQL-запросов и получение их результатов выполняются в окне **Interactive SQL** (рис. 16.15), открываемом командой **Tools\Interactive SQL**. Это окно реализует практически ту же функциональность, которая в предыдущих версиях сервера InterBase была реализована в программе Windows Interactive SQL (WISQL). В заголовке окна отображается имя файла БД, а в строке состояния — его полное имя. Одновременно можно открыть несколько окон **Interactive SQL** для различных БД.

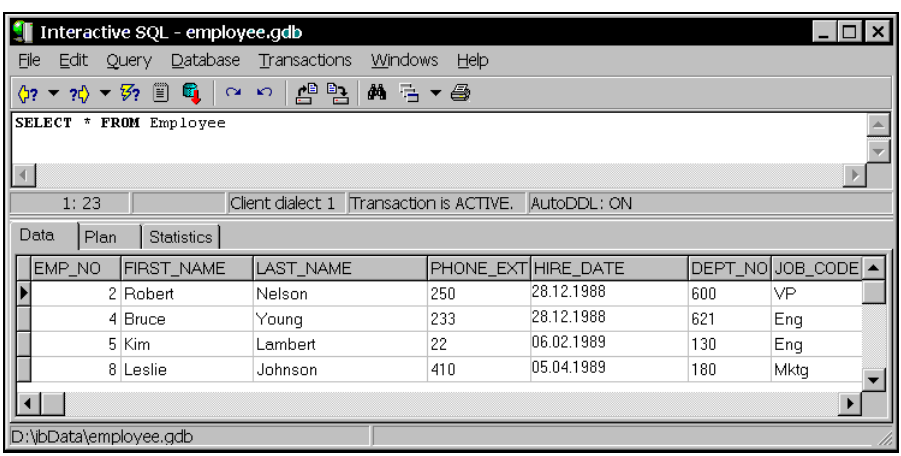


Рис. 16.15. Окно интерактивного выполнения SQL-запросов

В окне **Interactive SQL** можно выполнять различные операции с БД, включая создание и удаление БД и ее таблиц, соединение с БД, просмотр и изменение данных. Выполнение операций с БД основано на выполнении соответствующих инструкций языка SQL (SQL-запросов). Эти инструкции формируются и выполняются автоматически при выборе определенных команд меню. Так, при создании БД на основании указанных параметров формируется инструкция `CREATE DATABASE`.

Кроме того, можно набирать и выполнять инструкции вручную. Инструкции вводятся в верхней части окна, а в нижней его части выдаются результаты их выполнения.

Инструкции можно набирать и выполнять поочередно. Перемещение между отдельными инструкциями осуществляется нажатием кнопок с изображением желтых стрелок. Кнопка со стрелкой влево выводит в окне предыдущую, а со стрелкой вправо — следующую инструкцию. Выполнение инструкции осуществляется при нажатии кнопки с изображением желтой молнии.

Результаты работы инструкции можно запомнить в обычном текстовом файле, выбрав команду сохранения результата **Query\Save Output**.

Последовательность действий, заданных после запуска окна **Interactive SQL**, можно сохранить в виде "истории". Для этого предназначена команда **File\Save SQL History**. "История" запоминается в виде текстового файла с расширением hst или txt.

Последовательность инструкциий SQL представляет собой сценарий, или скрипт, его удобно использовать для автоматизации операций с БД. Сохранение сценария выполняется командой **Query\Save Script**. Скрипт запоминается в виде текстового файла с расширением sql или txt.

Сохраненный сценарий впоследствии можно загрузить, для этого надо выбрать команду **Query\Load Script** и указать имя сценария.

При выполнении операций, связанных с изменением БД, автоматически используется механизм транзакций, т. е. одновременно с началом изменяющей БД операции запускается транзакция. Такой режим запуска транзакций, как вы уже знаете, называется неявным. Для фиксации выполненных изменений нужно выполнить оператор `COMMIT`, а для их отмены (отката) — `ROLLBACK` (это же можно сделать с помощью команд **Transactions\Commit** и **Transactions\Rollback**).

Параметры выполнения SQL-запросов устанавливаются в окне **SQL Options**, открываемом командой **Edit\Options** (рис. 16.16).

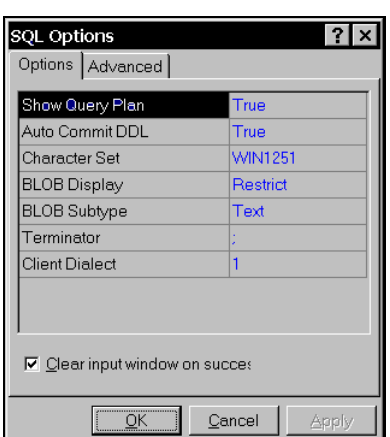


Рис. 16.16. Установка параметров выполнения SQL-запросов

На странице **Options** можно задать следующие параметры:

- **Show Query Plan** (Показывать план выполнения запроса);
- **Auto Commit DDL** (Автоматическое подтверждение операций DDL) — инструкции определения данных, например создания таблиц, автоматически подтверждаются, т. е. связанная с инструкциями неявная транзакция не требует дополнительного подтверждения;
- **Character Set** (Набор символов, используемый для кодировки строк); в нашей стране рекомендуется задавать набор `WIN1251`;
- **BLOB Display** (Режим отображения BLOB-объектов); может принимать значения `Restrict`, `Enabled` и `Disabled`;
- **BLOB Subtype** (Тип данных, содержащихся в объекте BLOB); по умолчанию `Text`;
- **Terminator** (Разделитель, используемый в качестве знака окончания SQL-запроса) — по умолчанию точка с запятой (`;`);
- **Client Dialect** (Диалект (версия) языка SQL).

Флажок **Clear input window on success** управляет режимом очистки верхней половины окна с текстом SQL-запроса в случае его успешного выполнения. По умолчанию флажок установлен, и окно очищается.

Замечание

В коде хранимых процедур и триггеров в качестве разделителя используется знак `;`, который не является окончанием SQL-запроса. Поэтому при их отладке для параметра **Terminator** необходимо установить другое значение, например, пробел.

Дополнительные параметры устанавливаются на странице **Advanced** (рис. 16.17). Наибольший интерес представляет группа переключателей **Transactions**, которые определяют режим подтверждения незавершенной транзакции при прекращении работы с БД: подтверждать незавершенную транзакцию (**Commit on exit**) или отменять ее (**Rollback on exit**).

Управление БД в окне **Interactive SQL** фактически происходит в ручном режиме и заключается в вводе и выполнении инструкций SQL. Единственным средством автоматизации, например, при создании таблиц или триггеров, является использование сценариев, хотя и в этом случае сценарии должны быть предварительно подготовлены. По удобству работы окно **Interactive SQL** уступает даже относительно простым программам типа Database Desktop. Несмотря на это, управление БД рассмотрено нами именно с использованием интерактивного SQL. Это сделано для того, чтобы показать особенности работы с удаленными БД, а также продемонстрировать язык SQL для удаленного сервера.

Кроме консоли IBConsole и подобных ей программ, есть так называемые средства CASE, с помощью которых разработчик может управлять БД в интерактивном режиме.

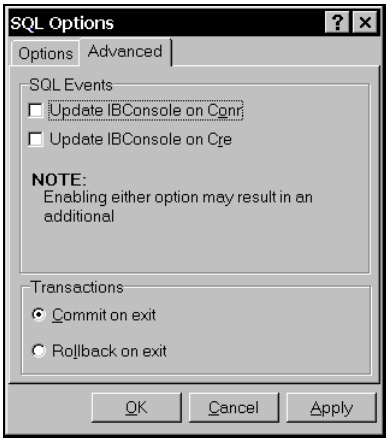


Рис. 16.17. Установка дополнительных параметров выполнения SQL-запроса

Программа SQL Monitor

Программа SQL Monitor представляет собой инструмент, позволяющий отслеживать выполнение SQL-запросов к удаленным БД. Для запуска программы нужно выбрать команду **Database\SQL Monitor** или запустить файл sqlmon.exe, находящийся в каталоге BIN главного каталога Delphi. Программу также можно запустить через меню **Пуск** Windows командой **Программы\Borland Delphi 7\SQL Monitor**.

Программа SQL Monitor (Монитор) отслеживает операции доступа к удаленным БД с помощью драйверов SQL-Links. Поэтому для локальных БД применять эту программу бесполезно, даже если используются средства языка SQL. Это связано с тем, что для локальных БД применяются локальные драйверы.

Замечание

Если в приложении использованы компоненты страницы **InterBase** Палитры компонентов, например, набор данных `IBQuery`, то доступ к БД выполняется напрямую через BDE без использования драйвера SQL-Links. В этом случае контролировать выполнение SQL-запросов к БД InterBase с помощью программы SQL Monitor невозможно.

После запуска Монитор автоматически отслеживает порядок выполнения SQL-запросов (выполняет трассировку инструкций), ведя журнал работы с удаленной БД. Монитор отслеживает запросы для указанного в команде **Clients** клиента, при отладке Delphi-приложения таким клиентом является Delphi 7. Строки журнала выводятся в верхней части окна программы (рис. 16.18). В нижней части окна отображается строка последнего запроса.

Для сохранения журнала в текстовом файле нужно выполнить команду **File\Save Log** и указать имя файла, для очистки журнала — команду **Edit\Clear**, для копирования его содержимого в буфер — команду **Edit\Copy**.

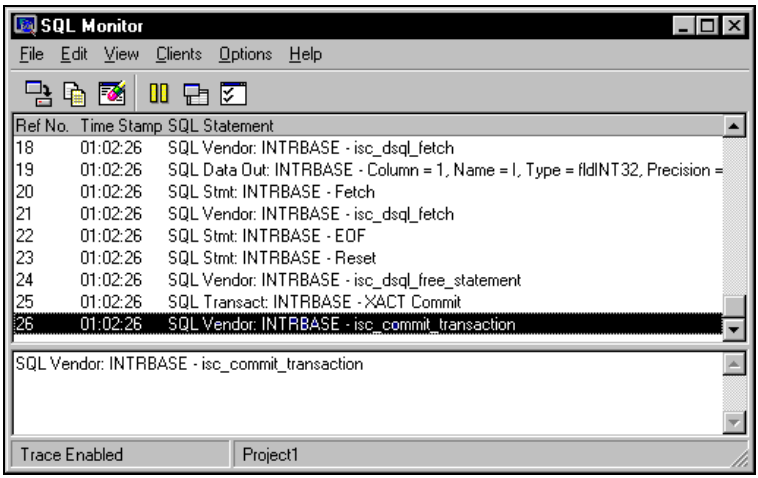


Рис. 16.18. Окно программы SQL Monitor

Параметры Монитора устанавливаются в окне параметров трассировки **Trace Options** (рис. 16.19), открываемом командой **Options\Trace Options**.

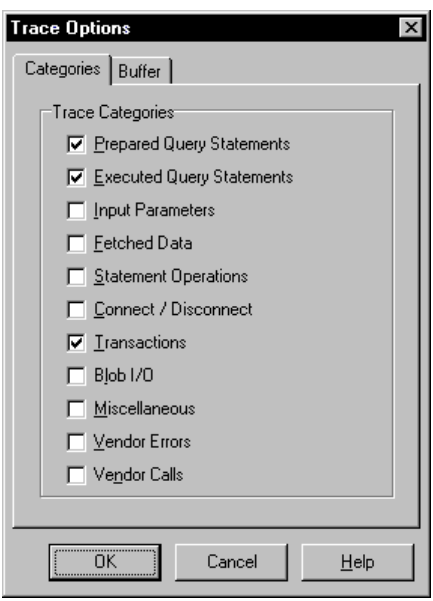


Рис. 16.19. Диалоговое окно установки параметров трассировки

С помощью параметров Монитора можно регулировать степень подробности информации, заносимой в журнал. Монитор имеет следующие параметры:

- **Prepared Query Statements** (подготовленные запросы, передаваемые на сервер);
- **Executed Query Statements** (выполненные на сервере запросы);
- **Input Parameters** (входные параметры) — данные, передаваемые на сервер в качестве параметров запросов;
- **Fetched Data** (данные, возвращаемые сервером);
- **Statement Operations** (операции с запросами) — `ALLOCATE`, `PREPARE`, `EXECUTE` и `FETCH`;
- **Connect/Disconnect** (операции соединения с сервером и отключения от него);
- **Transactions** (операции управления транзакциями);
- **Blob I/O** (ввод/вывод данных типа `BLOB`);
- **Miscellaneous** (остальные операции);
- **Vendor Errors** (сообщения об ошибках, возвращаемые сервером);
- **Vendor Calls** (вызовы API-функций сервера).

По умолчанию включены все параметры, и отслеживание выполнения запроса осуществляется максимально подробно. При этом даже простой запрос приводит к появлению в журнале значительного числа строк. Например, после выполнения запроса на отбор записей

```
SELECT * FROM ti
```

в журнал будет занесено 26 строк, что не облегчает, а скорее затрудняет проверку выполнения запроса. Поэтому в данном случае можно ограничиться только следующими параметрами:

- **Prepared Query Statements**
- **Executed Query Statements**
- **Transactions**

Тогда после выполнения приведенного запроса в журнал будут занесены такие данные:

```
1       01:10:47  SQL Prepare: INTRBASE - SELECT I FROM ti
2       01:10:47  SQL Transact: INTRBASE - XACT (UNKNOWN)
3       01:10:47  SQL Execute: INTRBASE - SELECT I FROM ti
4       01:10:47  SQL Transact: INTRBASE - XACT Commit
```

Монитор можно разместить поверх всех окон командой **Options\Always on Top**.

Глава 17

Трехуровневые приложения

Развитие архитектуры "клиент-сервер" привело к появлению трехуровневой архитектуры, в которой кроме сервера и приложений-клиентов (клиентов) дополнительно присутствует сервер приложений. *Сервер приложений* является промежуточным уровнем, обеспечивающим организацию взаимодействия клиентов ("тонких" клиентов) и сервера, например, выполнение соединения с сервером, разграничение доступа к данным и реализацию бизнес-правил. Сервер приложений реализует работу с клиентами, расположенными на различных платформах, т. е. функционирующими на компьютерах различных типов и под управлением различных операционных систем. Сервер приложений также называют *брокером данных* (broker — посредник).

Основные достоинства трехуровневой архитектуры "клиент-сервер":

- снижение нагрузки на сервер;
- упрощение клиентских приложений;
- единое поведение всех клиентов;
- упрощение настройки клиентов;
- независимость от платформы.

Информационные системы, основанные на трехуровневой сетевой архитектуре, называют также *распределенными*.

Принципы построения трехуровневых приложений

В системе Delphi постоянно совершенствуются используемые технологии программирования и соответствующие им наборы компонентов. Так, в предыдущих версиях системы применяемые при разработке многоуровневых приложений технологии и средства объединялись под общим названием MIDAS (Multi-tier distributed application services — службы многоуровневых распределенных приложений). В последней версии системы соответствующие средства (компоненты и объекты) получили общее название DataSnap.

В Delphi 7 поддерживается создание многоуровневых приложений, основанных на перечисленных ниже технологиях межпрограммного и межкомпьютерного взаимодействия.

- Модель DCOM (Distributed Component Object Model — модель распределенных компонентных объектов) позволяет использовать объекты, расположенные на другом компьютере.
- Сервер MTS (Microsoft Transaction Server — сервер транзакций Microsoft) является дополнением к технологии COM, разработанной фирмой Microsoft, и предназначен для управления транзакциями.
- Модель COM+ (Component Object Model+ — усовершенствованная объектная модель компонентов) фирмы Microsoft введена в Windows 2000 и интегрирует технологии MTS в стандартные службы COM.
- Сокеты TCP/IP (Transport Control Protocol/Protocol Internet — транспортный протокол/протокол Интернета) используются для соединения компьютеров в различных сетях, в том числе в Интернете.
- CORBA (Common Object Request Broker Architecture — общедоступная архитектура с брокером при запросе объекта) позволяет организовать взаимодействие между объектами, расположенными на различных платформах.
- SOAP (Simple Object Acces Protocol — простой протокол доступа к объектам) служит универсальным средством обеспечения взаимодействия с клиентами и серверами Web-служб на основе кодирования XML и передачи данных по протоколу HTTP.

При создании трехуровневого приложения разработка БД и использование сервера принципиально не отличаются от уже рассмотренного случая двухуровневых приложений. Главные особенности трехуровневого приложения связаны с созданием сервера приложений и клиентского приложения, а также с организацией взаимодействия между ними. Для разработки многоуровневых приложений, кроме рассмотренных ранее средств, используются удаленные модули данных и компоненты, размещенные на странице **DataSnap** Палитры компонентов.

В трехуровневой архитектуре с использованием механизма доступа BDE процессор баз данных в обязательном порядке устанавливается совместно с сервером приложений, при этом на клиентском компьютере должна быть установлена только библиотека DBClient.dll относительно небольшого размера (210 Кбайт). Таким образом, на компьютере пользователя "тонким" является не только клиент, но и процессор баз данных. Соответствующая трехуровневая архитектура схематично представлена на рис. 17.1. В частном случае два или все три уровня могут располагаться на одном компьютере, что широко используется при отладке приложений.

Взаимодействие между сервером приложений и клиентом организуется через интерфейс провайдера, называемый *интерфейсом оператора* или просто *провайдером.* Этот интерфейс обеспечивает передачу информации в виде *пакетов данных.* Физически пакеты данных представляют собой совокупности двоичных кодов, образующих блоки. Логически пакет данных является подмножеством

набора данных, которое содержит данные записей, а также метаданные (информация об именах и типах полей) и ограничения. Провайдер обеспечивает разбиение данных на пакеты, а также кодирование пакетов в зависимости от используемого сетевого протокола.

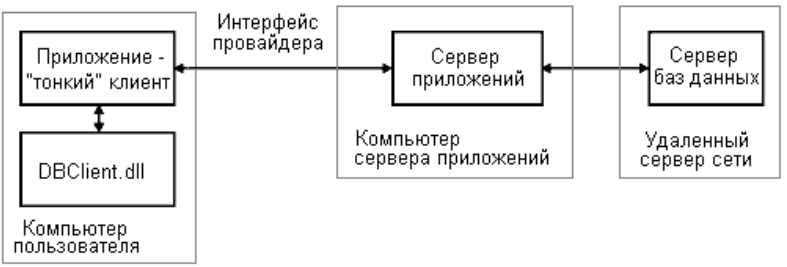


Рис. 17.1. Трехуровневая архитектура типа "клиент-сервер"

Данные, отредактированные клиентом, пересылаются обратно в так называемых *дельта-пакетах.* В дельта-пакете содержится информация о старых и новых значениях записей. Перед внесением в записи БД требуемых изменений (т. е. перед пересылкой записей серверу БД) сервер приложений выполняет проверку корректности и допустимости изменений. Изменения могут быть отвергнуты, например, если запись уже была изменена другим пользователем. В этом случае сервер приложений генерирует событие `OnReconcilError`, которое используется для распознавания и обработки конфликтов между клиентами.

Сервер приложений

Сервер приложений создается на основе удаленного модуля данных, который служит для размещения компонентов, а также для обеспечения взаимодействия с сервером и клиентами. Для создания различных серверов приложений предназначены следующие разновидности удаленных модулей данных:

- **Remote Data Module** — для серверов DCOM, TCP/IP и OLEnterprise;
- **Transactional Data Module** — для сервера MTS;
- **WebSnap Data Module** — для сервера Web;
- **SOAP Server Data Module** — для серверов SOAP.

Удаленные модули данных расположены на страницах **Multitier**, **WebSnap** и **WebServices** Хранилища объектов. В удаленном модуле данных размещаются те же компоненты, что и в простом модуле данных, например, для механизма доступа с помощью BDE такие компоненты, как `Query`, `Database`, `Session`, предназначенные для организации доступа к данным.

Рассмотрим создание простейшего сервера приложений — сервера DCOM, взаимодействие с которым основано на технологии DCOM. Для работы этого

сервера необходимо, чтобы в системе была установлена программная поддержка функционирования распределенных COM-объектов, которая имеется в операционных системах Windows 98/NT/2000. Для Windows 95 ее нужно устанавливать отдельно. Поддержка распределенных COM-объектов устанавливается автоматически при инсталляции ряда программ Windows, кроме того, соответствующие средства можно загрузить из Интернета по адресу:

http://www.microsoft.com/com/dcom95/download-f.htm

Добавление к проекту удаленного модуля данных выполняется выбором объекта **Remote Data Module** страницы **Multitier** Хранилища объектов. При добавлении модуля выводится диалоговое окно мастера **Remote Data Module Wizard**, в котором нужно задать параметры модуля (рис. 17.2).

Рис. 17.2. Добавление удаленного модуля данных

В поле редактирования **CoClass Name** вводится имя модуля данных.

В списке **Instancing** (Создание экземпляров) выбирается способ запуска модуля:

- **Internal** — экземпляр модуля данных создается на сервере в случае, когда модуль данных является частью библиотеки DLL;
- **Single Instance** — для каждого клиента в его адресном пространстве создается один экземпляр удаленного модуля данных, и каждое клиентское соединение запускает этот свой экземпляр;
- **Multiple Instance** — один экземпляр приложения (процесс) представляет все удаленные модули данных, созданные для клиентов (по умолчанию); каждый удаленный модуль данных предназначен для одного клиентского соединения, но все они разделяют одно и то же адресное пространство.

В списке **Threading Model** (Потоковая модель) выбирается способ вызова интерфейса клиента, если модуль данных является частью библиотеки DLL:

- **Single** — библиотека получает запросы клиента по одному;
- **Apartment** — одновременно обрабатывается несколько запросов клиентов, для каждого из которых создан отдельный экземпляр модуля данных (по умолчанию);

- **Free** — отдельный экземпляр модуля данных одновременно может отвечать на несколько запросов клиентов;
- **Both** — отдельный экземпляр модуля данных одновременно может отвечать на несколько запросов клиентов, результаты обработки также возвращаются одновременно;
- **Neutral** — разные клиенты могут одновременно вызывать удаленный модуль данных из нескольких потоков, при этом модель COM следит за тем, чтобы не было конфликта вызовов (однако нужно иметь в виду возможный конфликт потоков: он отслеживается только в версии COM+, при отсутствии ее используется потоковая модель типа Apartment).

После нажатия кнопки **OK** модуль данных с установленными параметрами добавляется к проекту. В приведенном на рис. 17.2 примере модулю присвоено имя `ServerDCOM`, а два других параметра оставлены без изменений.

На этапе проектирования внешний вид удаленного модуля не отличается от вида простого модуля данных, рассмотренного в *главе 4*, посвященной созданию информационной системы. Как и в простом модуле, в удаленном модуле данных размещаются невизуальные компоненты, используемые для доступа к данным. Здесь могут быть наборы компонентов для механизмов доступа ADO, BDE, dbExpres и InterBase Express. К примеру, для технологии BDE чаще всего этими компонентами являются рассмотренные ранее `Query`, `Table`, `Database`, `Session`, а также провайдер `DataSetProvider`. В самом простом случае достаточно разместить в модуле только набор данных. Например, разместим в удаленном модуле набор данных `Query` и зададим для него значения свойств `DataBaseName` и `SQL` так, чтобы включить в набор все поля всех записей таблицы `Personnel`. Указанным свойствам присвоим значения:

- свойству `DataBaseName` — значение `BDPlace`;
- свойству `SQL` — значение `SELECT * FROM Personnel.db`.

На этом создание простейшего сервера DCOM закончено. Перечислим еще раз действия, которые были при этом выполнены:

- к проекту добавлен удаленный модуль данных;
- в модуле размещен компонент набора данных и присвоены значения его свойствам.

Созданное приложение сервера состоит из следующих частей:

- проект;
- главная форма приложения;
- удаленный модуль данных;
- модуль библиотеки типов.

Разработка проекта и главной формы приложения не имеют принципиальных отличий от разработки обычного приложения Delphi. Отметим, что для сервера приложений основная функциональная нагрузка приходится на удаленный мо-

дуль данных. В главной форме можно разместить вспомогательные компоненты и выполнять некоторые сервисные действия, например, вести подсчет клиентов, подключенных к серверу, и выводить показания этого счетчика в надписи `Label`, размещенной в главной форме сервера.

Библиотека типов создается автоматически, а ее модуль сохраняется на диске при сохранении других файлов проекта. Библиотека занимает два файла: *Project*.tlb и *Project*_TLB.pas, где *Project* — имя проекта.

После создания сервера DCOM его нужно зарегистрировать как сервер автоматизации. Регистрация сервера выполняется системой Windows автоматически при запуске приложения сервера.

По умолчанию интерфейс провайдера обеспечивает набор данных, в нашем случае это `Query1`. Кроме того, Delphi включает в свой состав компонент `DataSetProvider`, который предоставляет большие возможности по управлению интерфейсом провайдера, включая обмен XML-данными.

Простейший сервер DCOM представляет собой *удаленный брокер данных*, который обеспечивает соединение с сервером БД и передачу данных клиенту и обратно. Для расширения функциональности сервера приложений к нему добавляются бизнес-правила, предназначенные для поддержания БД в целостном состоянии и реализующие ограничения, применяемые к данным.

Поддержка механизма ограничений обеспечивается *брокером ограничений*. Для набора данных и его отдельных полей можно задавать ограничения на значения полей не только в приложении клиента, но и в сервере приложений (удаленном модуле данных). Ограничения, заданные в сервере приложений, пересылаются клиенту вместе с данными в пакете данных, и эти ограничения действуют наряду с ограничениями, заданными в приложении клиента.

Для реализации ограничений в сервере приложений при использовании технологии BDE можно использовать свойство `Constraints` типа `TCheckConstraints` наборов данных `Table` и `Query`. Тип `TCheckConstraints` представляет собой коллекцию (набор) отдельных ограничений типа `TCheckConstraint`, имеющих следующие свойства:

- `CustomConstraint` типа `String` — код SQL, описывающий ограничение;
- `ErrorMessage` типа `String` — текст, выдаваемый пользователю при нарушении данного ограничения;
- `FromDictionary` типа `Boolean` — признак, значение `True` которого указывает, что ограничение выбирается из словаря данных; по умолчанию свойство имеет значение `False`, и словарь данных не используется;
- `ImportedConstraint` типа `String` — код SQL, описывающий ограничение, которое импортировано из словаря данных.

Для задания ограничений нужно выделить набор данных и в окне Инспектора объектов щелчком в области значения свойства `Constraints` открыть окно, показанное на рис. 17.3, справа. Центральную часть окна занимает список ограни-

чений, применяемых к набору данных, имя которого выводится в заголовке окна (на рисунке — `Query1`). Добавление к списку нового ограничения выполняется командой **Add** контекстного меню, нажатием клавиши <Insert> или нажатием крайней левой кнопки на панели инструментов. Существующие ограничения можно удалять и перемещать в пределах списка, эти действия выполняются с помощью команд контекстного меню, нажатия клавиш или кнопок панели инструментов.

Сразу после добавления ограничение "пустое", и в списке выводится название его типа `TCheckConstraint` (на рис. 17.3 это третье ограничение). Для задания ограничения нужно его описать, например, присвоив значения свойствам `CustomConstraint` и `ErrorMessage`. После того как свойство `CustomConstraint` получит значение, оно будет выведено в списке ограничений. Свойства ограничения становятся доступными через Инспектор объектов после выбора ограничения в списке.

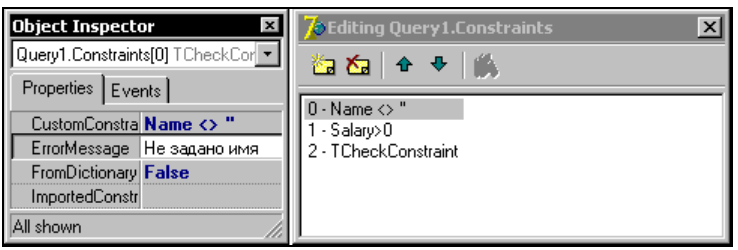


Рис. 17.3. Определение ограничений для набора данных `Query1`

В приведенном на рис. 17.3 примере для данных о сотрудниках организации (таблица `Personnel`) установлены ограничения на значения полей `Name` и `Salary`: поле имени не может быть пустым, а значение оклада должно быть положительным. При нарушении этих ограничений пользователю выдаются соответствующие сообщения: например, если не задано значение поля `Name`, то выдается сообщение `Не задано имя!`.

Как уже было сказано, ограничения сервера приложений действуют в дополнение к ограничениям, заданным в приложении клиента. Так обеспечивается распределение ограничений, применяемых к данным, между отдельными уровнями информационной системы. Достоинством размещения бизнес-правил на сервере приложений является то, что они одинаковы для всех клиентов и что облегчаются внесение изменений в информационную систему и ее настройка.

Свойства `CustomConstraint`, `ConstraintErrorMessage` и `ImportedConstraint` объектов типа `TField` позволяют задать ограничения для отдельных полей набора данных. Эти свойства аналогичны свойствам `CustomConstraint`, `ErrorMessage` и `ImportedConstraint` объекта типа `TCheckConstraints`.

Приложение клиента

Приложение "тонкого" клиента отличается от ранее рассмотренного приложения "толстого" клиента в первую очередь тем, что для "тонкого" клиента нужно выполнить следующие действия:

- организовать связь между приложением клиента и сервером приложений;
- обеспечить обмен информацией между наборами данных клиента и сервера.

Для этого используются компоненты соединения и клиентский набор данных `ClientDataSet`, размещаемые в форме клиента.

Выбор компонента, используемого для соединения с сервером приложений, зависит от типа сервера (типа коммуникационного протокола):

- `DCOMConnection` — для соединения с помощью DCOM;
- `SocketConnection` — для соединения через сокеты TCP/IP;
- `WebConnection` — для соединения с помощью HTTP;
- `SOAPConnection` — для соединения с помощью SOAP (HTTP и XML).

Создадим приложение клиента, подключаемого к рассмотренному выше серверу DCOM, для чего разместим в главной форме компонент `DCOMConnection` со страницы **DataSnap** Палитры компонентов. Основные свойства этого компонента:

- `ComputerName` типа `String` (имя компьютера, на котором расположен сервер приложений);
- `ServerName` типа `String` (имя сервера приложений);
- `ServerGUID` типа `String` (универсальный уникальный идентификатор GUID сервера приложений);
- `Connected` типа `Boolean` (признак, управляющий активностью соединения).

Для указания компьютера, на котором расположен сервер приложений, удобно использовать окно **Browse for Computer** (рис. 17.4), открываемое через Инспектор объектов. После выбора сетевого компьютера и нажатия кнопки **OK** имя выбранного компьютера присваивается в качестве значения свойству `ComputerName`. Если сервер расположен на одном компьютере с приложением клиента (что удобно при отладке приложений), то значение свойства `ComputerName` не задается.

После того как компьютер задан, имена доступных (зарегистрированных) серверов автоматизации можно выбирать с помощью Инспектора объектов в списке значений свойства `ServerName`. Имя сервера является составным и включает в себя имя проекта приложения сервера и имя модуля данных, задаваемое для удаленного модуля данных, например, `Server.ServerDCOM`.

Задание имени сервера приводит к автоматической установке для выбранного сервера идентификатора GUID, который присваивается в качестве значения свойству `ServerGUID`. GUID (Globally Unique Identifier — глобальный уникаль-

ный идентификатор) представляет собой 128-битную константу, присваиваемую объекту COM для его однозначной идентификации. Значение GUID отображается в модуле библиотеки типов сервера приложений. Сервер можно также задать, установив значение свойству `ServerGUID`, в этом случае значение свойства `ServerName` заполняется автоматически. Однако первый путь, связанный с выбором имени сервера, более удобен.

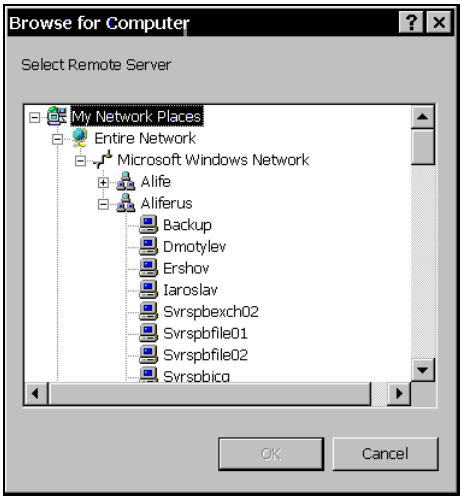


Рис. 17.4. Выбор сетевого компьютера

Чтобы протестировать соединение с сервером приложений, свойство `Connected` устанавливается в значение `True`. В этом случае сервер запускается автоматически, и с ним устанавливается соединение. В общем случае значение этого свойства можно не трогать, т. к. оно автоматически устанавливается в `True` при выборе провайдера для клиентского набора данных `ClientDataSet`.

Клиентский набор данных `ClientDataSet` предназначен для работы с записями, поступающими с сервера приложений. Перечислим свойства этого компонента:

- `RemoteServer` типа `TCustomRemoteServer` (соединение, используемое для связи с сервером);
- `ProviderName` типа `String` (провайдер, обеспечивающий передачу данных);
- `Active` типа `Boolean` (признак, указывающий, открыт или закрыт набор данных);
- `PacketRecords` типа `Integer` (размер пакета данных);
- `FileName` типа `String` (имя файла для обмена данными с диском).

В качестве значения свойства `RemoteServer` можно указывать любой из компонентов, используемых для соединения с сервером: `DCOMConnection`, `SocketConnection`, `WebConnection`, `SOAPConnection`. Нужное значение удобно

выбирать в списке Инспектора объектов. Поскольку для соединения с сервером DCOM в форме размещен компонент `DCOMConnection`, в списке нужно выбрать его имя `DCOMConnection1`, присвоенное компоненту по умолчанию.

После того как соединение выбрано, с помощью свойства `ProviderName` задается провайдер, обеспечивающий передачу данных клиенту. При раскрытии в окне Инспектора объектов списка значений этого свойства автоматически запускается сервер приложений, если он еще не был запущен, обеспечивая выдачу клиенту списка доступных провайдеров (наборов данных). Выбор имени провайдера приводит к соединению клиентского набора данных `ClientDataSet` с соответствующим набором данных сервера.

В удаленном модуле данных рассмотренного ранее сервера приложений расположен набор данных `Query1`, предоставляющий интерфейс `Iprovider`, имя которого можно выбрать в качестве значения, присваиваемого свойству `ProviderName`.

Для работы с данными в приложении клиента размещаются визуальные компоненты и источник данных `DataSource`, которые связываются между собой, а также с клиентским набором данных аналогично тому, как это выполнялось для рассмотренных ранее локальных приложений и для приложений "толстого" клиента. В приведенном на рис. 17.5 примере в форме приложения размещены компоненты `DBGrid` (сетка) и `DataSource`, свойство `DataSet` которого имеет значение `ClientDataSet1` (имя компонента `ClientDataSet` по умолчанию). После установки свойства `Active` клиентского набора данных в значение `True` в сетке отображаются записи набора данных `Personnel`, отбор которых обеспечивает инструкция `SELECT` набора данных `Query1` сервера приложений.

Рис. 17.5. Форма приложения "тонкого" клиента на этапе разработки

Чтобы расширить функциональность рассмотренного простейшего "тонкого" клиента, к нему добавляют возможности, связанные с модификацией записей, передачей изменений на сервер приложений (записью их в БД), обработкой конфликтов между клиентами, а также с выполнением ряда других действий.

При работе с данными клиент может сохранять их на своем компьютере, работая в автономном режиме и не загружая сеть передачей информации. Обновленные данные передаются на сервер, а с сервера новые данные загружаются по

мере необходимости. Этот принцип работы напоминает работу с кэшированными изменениями и реализуется с помощью компонента `ClientDataSet`.

Для просмотра состояния текущей записи клиентского набора данных используется метод `UpdateStatus: TUpdateStatus`, возвращающий следующие значения:

- `usUnmodified` (в записи нет изменений);
- `usModified` (запись изменена (отредактирована));
- `usInserted` (запись вставлена (является новой));
- `usDeleted` (запись удалена).

Проанализировав состояние текущей записи, можно вывести соответствующее сообщение для пользователя, например, в тексте надписи `Label`. Если код, выполняющий вызов метода и анализ его результата, расположить в обработчике события `DataChange` компонента `DataSource`, который связан с клиентским набором данных, то надпись будет автоматически отображать состояние текущей записи.

Получить доступ ко всем изменениям, сделанным в записях, но еще не отправленным на сервер, позволяют свойства `Data` и `Delta` типа `OleVariant`, первое из которых представляет собой данные клиентского набора данных, а второе — его измененные данные (Delta-данные). Для получения измененных записей в форме располагается еще один компонент `ClientDataSet`, который связывается с первым клиентским набором данных так:

```
ClientDataSet2.Data := ClientDataSet1.Delta;
```

Оба свойства доступны во время выполнения, поэтому их значения можно использовать только программно. После приведенного выше присваивания второй клиентский набор данных будет содержать все неотправленные изменения.

Замечание

Если изменения в записях отсутствуют, то при попытке выполнить присваивание генерируется исключение.

Свойство `ChangeCount` типа `Integer`, доступное во время выполнения, содержит число измененных записей, которое нужно проверять на равенство 0 перед тем, как делать попытку получить эти записи. Т. е. инструкция присваивания должна иметь следующий вид:

```
if ClientDataSet1.ChangeCount > 0
       then ClientDataSet2.Data := ClientDataSet1.Delta;
```

На рис. 17.6 показана форма клиентского приложения во время его выполнения. В верхней сетке `DBGrid1` отображаются записи клиентского набора данных `ClientDataSet1`, которые доступны для просмотра и изменения. В надписи `Label1`, расположенной над сеткой, выводится состояние текущей записи. Во второй сетке `DBGrid2` отображаются изменения, сделанные в записях клиентского набора данных. Эта сетка через источник данных `DataSource2` связана со вторым клиентским набором данных `ClientDataSet2`. Надписи `Label2` и

`Label3` отображают текст `'Изменения в записях - '` и число измененных записей, соответственно.

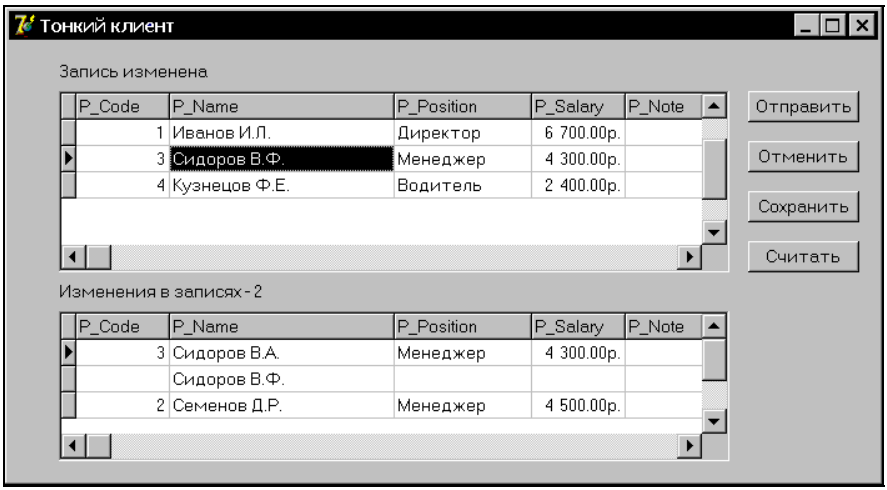


Рис. 17.6. Просмотр состояния текущей записи и изменений в записях

Соответствующий код содержится в обработчике события `DataChange` источника данных `DataSource1`, который связан с клиентским набором данных `ClientDataSet1`:

```
procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
case ClientDataSet1.UpdateStatus of
   usUnModified: Label1.Caption := 'Запись не изменялась';
   usModified:   Label1.Caption := 'Запись изменена';
   usInserted:   Label1.Caption := 'Запись вставлена';
   usDeleted:    Label1.Caption := 'Запись удалена';
   end;
if ClientDataSet1.ChangeCount > 0
       then ClientDataSet2.Data:=ClientDataSet1.Delta;
Label2.Caption:='Изменения в записях - ' +
             IntToStr(ClientDataSet1.ChangeCount);
end;
```

В форме расположены также четыре кнопки:

- **Отправить** (`Button1`);
- **Отменить** (`Button2`);
- **Сохранить** (`Button3`);
- **Считать** (`Button4`).

Назначение кнопок и обработчики событий их нажатия будут рассмотрены ниже.

Сделанные изменения действуют только в приложении клиента и при завершении его работы теряются. Чтобы выполнить обновление данных, изменения нужно отправить на сервер приложений, для чего предназначается метод `ApplyUpdates (MaxErrors: Integer): Integer`. Параметр `MaxErrors` определяет максимальное число ошибок, допустимое при выполнении метода; если для параметра указать значение −1, то на сервер приложений будут переданы все изменения. В качестве результата функция `ApplyUpdates` возвращает число ошибок. Ошибки передачи изменений чаще всего вызваны конфликтами, связанными с редактированием этих же записей другими клиентами.

Рассмотрим в качестве примера следующую процедуру:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Label4.Caption := IntToStr(ClientDataSet1.ApplyUpdates(-1));
end;
```

При нажатии кнопки `Button1` на сервер приложений пересылаются все изменения, сделанные в приложении клиента. Число ошибок, связанных с пересылкой записей, выводится в надписи `Label4`.

Пересылка записей и связанное с этим обновление БД может привести к конфликту записей. В этом случае в клиентском наборе данных для каждого такого конфликта генерируется событие `OnReconcileError` типа `TReconcileErrorEvent`. Тип события описан так:

```
type TReconcileErrorEvent = procedure(DataSet: TClientDataSet;
       E: EReconcileError; UpdateKind: TUpdateKind;
       var Action: TReconcileAction) of object;
```

Параметр `DataSet` определяет клиентский набор данных, обновление чьих записей привело к конфликту и вызвало исключение, объект которого содержит параметр `E`. Этот параметр указывает тип операции обновления данных:

- `usModified` (редактирование);
- `usInserted` (вставка);
- `usDeleted` (удаление).

Параметр `Action` определяет действие, которое должно быть выполнено сервером приложений для устранения ошибки:

- `raSkip` — запись остается в приложении клиента без изменений, т. е. остается в Delta-данных (по умолчанию);
- `raAbort` — операция обновления данных прекращается;
- `raMerge` — изменения этого клиента объединяются с изменениями других клиентов; принимаются изменения значений только тех полей, которые не были изменены другими клиентами;

- `raCorrect` — данные, имеющиеся на сервере, заменяются данными этого клиента; данные, поступившие от других клиентов, будут изменены;
- `raCancel` — операция обновления данных прекращается, все изменения удаляются и восстанавливаются первоначальные значения записей;
- `raRefresh` — изменения, сделанные в приложении клиента, сбрасываются и заменяются значениями, имеющимися на сервере.

Для упрощения работы с возникающими ошибками обновления данных удобно использовать стандартный диалог, который добавляется к проекту выбором объекта **Reconcile Error Dialog** (Диалог устранения ошибок) на вкладке **Dialogs** Хранилища объектов.

Для пользователя отображаются: информация о типе операции обновления данных, описание возникшей ошибки, данные конфликтной записи. Составом отображаемых полей управляют два флажка:

- **Show conflicting fields only** (Показывать только конфликтные поля);
- **Show changed fields only** (Показывать только измененные поля).

Пользователь определяет направленное на устранение ошибки действие с помощью группы переключателей **Reconcile Action**:

- **Skip** (Пропуск);
- **Cancel** (Отмена);
- **Correct** (Подтверждение);
- **Refresh** (Обновление);
- **Merge** (Объединение).

Выбор переключателя приводит к установке соответствующего значения параметра `Action` обработчика события `OnReconcileError`.

Рассмотрим следующий пример:

```
// Подключение модуля формы диалога Reconcile Error Dialog
uses reUnit;
...
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;
E: EReconcileError; UpdateKind: TUpdateKind;
var Action: TReconcileAction);
begin
Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

При возникновении ошибки обновления данных вызывается обработчик события `OnReconcileError`. Для предоставления пользователю информации о возникшей ошибке и обеспечения возможности реагировать на нее в теле обработчика вызывается функция `HandleReconcileError`. Возвращаемый функцией результат зависит от действий пользователя и присваивается параметру `Action` обработчика со-

бытия `OnReconcileError`. Чтобы можно было реализовать вызов функции `HandleReconcileError`, ее модуль должен быть включен в список `uses`.

При возникновении конфликта, связанного с обновлением записи, диалоговое окно устранения ошибки имеет вид, показанный на рис. 17.7.

При отладке приложения на локальном компьютере конфликт обновления можно вызвать, запустив две копии приложения клиента и выполнив попытку редактирования одной и той же записи.

При изменении записей клиентского набора данных можно отменить изменения без передачи их на сервер приложений.

Метод `CancelUpdates` отменяет все изменения, не отправленные на сервер. Так, выполнение процедуры-обработчика

```
procedure TForm1.Button2Click(Sender: TObject);
begin
ClientDataSet1.CancelUpdates;
end;
```

приводит к тому, что при нажатии кнопки `Button2` все изменения, сделанные в приложении клиента, отменяются.

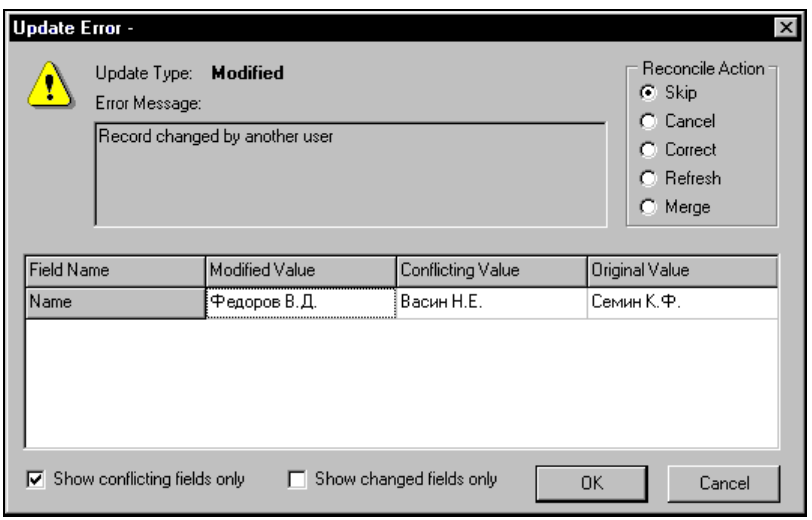


Рис. 17.7. Диалоговое окно устранения ошибок обновления данных

Метод `UndoLastChange (FollowChange: Boolean): Boolean` отменяет последнее изменение независимо от его вида (редактирование, вставка или удаление записи). Параметр `FollowChange` управляет позиционированием указателя текущей записи после выполнения отмены. При значении `True` происходит позиционирование указателя текущей записи на восстановленную запись, а при

значении False положение этого указателя не изменяется. В качестве результата функция UndoLastChange возвращает признак успешности выполнения операции отмены изменения: True — операция завершилась успешно, False — операция не выполнена.

При организации автономной работы клиента удобно использовать методы LoadFromFile и SaveToFile.

Метод LoadFromFile (const FileName: string = '') загружает в клиентский набор данных данные, ранее сохраненные в файле, имя которого указано параметром FileName. Если значением параметра является пустая строка, то данные читаются из файла, задаваемого свойством FileName.

Метод SaveToFile (const FileName: string = ''; Format TDataPacketFormat = dfBinary) сохраняет данные в файле с именем FileName. Необязательный параметр Format указывает формат файла:

- dfBinary (двоичный файл) — по умолчанию;
- dfXML (XML, escape-последовательность);
- dfXMLUTF8 (XML, последовательность UTF8).

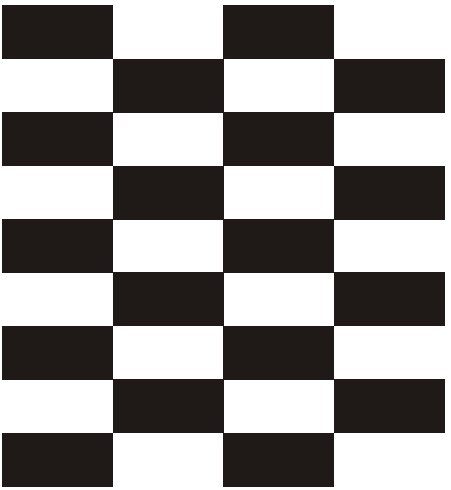
Если на диске нужно сохранить все записи набора данных, то с сервера должны быть получены все данные. С этой целью свойство PacketRecords клиентского набора данных следует установить в значение -1.

Рассмотрим в качестве примера обмен данными с диском:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
if SaveDialog1.Execute then
       ClientDataSet1.SaveToFile(SaveDialog1.FileName);
end;
procedure TForm1.Button4Click(Sender: TObject);
begin
if OpenDialog1.Execute then
       ClientDataSet1.LoadFromFile(OpenDialog1.FileName);
end;
```

При нажатии кнопки Button3 открывается диалог SaveDialog1 выбора файла для сохранения данных. Выбор файла и нажатие кнопки **Сохранить** приводят к записи данных на диск в указанном файле. По умолчанию файл имеет двоичный формат. Нажатие кнопки Button4 вызывает диалог OpenDialog1 выбора файла для чтения данных. Выбор файла и нажатие кнопки **Открыть** приводит к загрузке в клиентский набор ранее сохраненных данных.

Для указания имени файла можно использовать свойство FileName типа String. Если это свойство задано, то методы LoadFromFile и SaveToFile вызываются без параметров.



ЧАСТЬ IV

Базы данных и Интернет

Глава 18

Основные элементы интернет-технологий

В последних версиях системы Delphi имеются развитые средства для работы в глобальной сети Интернет и локальных сетях интранет. В частности, имеются средства для использования электронной почты, применения Web-служб, публикации баз данных и др. Публикация баз данных в Интернете — это размещение информации из баз данных на Web-страницах сети. В этой главе мы рассмотрим важнейшие элементы интернет-технологий, являющиеся основой для разработки Web-приложений.

В Интернете вся информация размещается на Web-страницах, для написания которых используются язык HTML (HyperText Markup Language — язык разметки гипертекста) или его расширения, такие как DHTML (Dynamic HTML — динамический HTML) и XML (eXtensible Markup Language — расширяемый язык разметки). В состав Web-страницы могут входить текстовая информация, ссылки на другие Web-страницы, графические изображения, аудио- и видеоинформация и другие данные. Web-страницы хранятся на Web-сервере.

Для доступа к Web-страницам используются специальные клиентские программы — Web-обозреватели (программы просмотра, называемые также Web-браузерами или Web-броузерами — от англ. *browser*), находящиеся на компьютерах пользователей. Обозреватель формирует запрос на получение требуемой страницы или другого ресурса с помощью специального адреса URL (Uniform Resource Locator — универсальный указатель ресурса). Этот адрес определяет тип протокола для передачи ресурса, имя домена, используемое для доступа к Web-узлу, номер порта, локальный путь к файлу и дополнительные аргументы. Соединение с Web-узлом устанавливается с помощью протокола передачи данных HTTP (HyperText Transfer Protocol — протокол передачи гипертекста).

Для расширения возможностей клиента (обозревателя) и Web-сервера создаются соответствующие программы расширения. Схема взаимодействия клиента и Web-сервера с использованием программ расширения приведена на рис. 18.1.

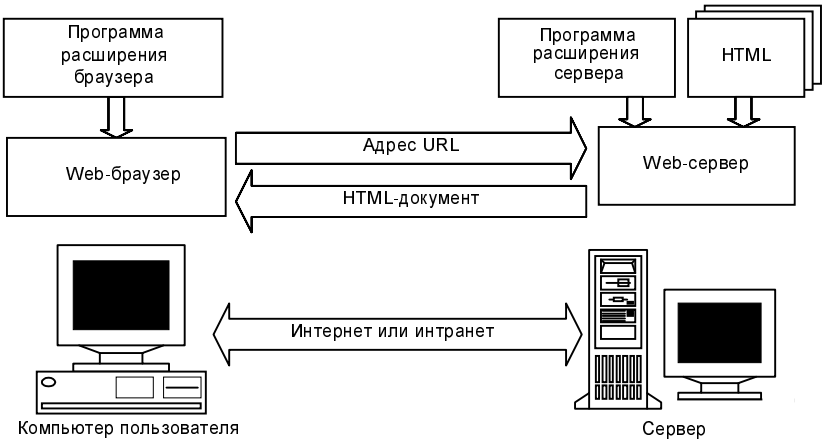


Рис. 18.1. Схема взаимодействия обозревателя и Web-сервера

При организации такого взаимодействия могут использоваться следующие средства:

- сценарии (скрипты), подготавливаемые на различных языках сценариев (JavaScript, JScript и VBScript) и вставляемые в Web-документ;
- апплеты и сервлеты на языке Java;
- элементы управления ActiveX;
- консольные исполняемые программы, реализованные с использованием интерфейса CGI;
- исполняемые программы, реализованные с использованием интерфейса WinCGI;
- динамические библиотеки, реализованные с использованием интерфейса ISAPI;
- динамические страницы IDC/HTX;
- активные серверные страницы ASP.

Дадим краткую характеристику перечисленным выше средствам, которые хотя и выходят за рамки Delphi, но составляют основные элементы интернет-технологий и используются при публикации баз данных.

Сценарии JavaScript, JScript и VBScript

Сценарии (скрипты), написанные на языках JavaScript, JScript или VBScript, используются для динамического управления интерфейсными объектами (компонентами) Web-документа. Эти языки являются интерпретируемыми, т. е. исполняемый файл не создается, а код выполняется непосредственно в процессе интерпретации. Интерпретацию и выполнение сценариев осуществляет обозреватель или Web-сервер. Скрипты рассматриваются как расширение языка HTML и могут включаться в тело HTML-документа. Часть сценария может исполняться во время загрузки документа, а часть сценария, реализованная, как

правило, в виде функции, выполняться в ответ на действия пользователя. Использование того или иного языка сценариев определяется типом применяемого обозревателя.

Язык JavaScript представляет собой объектно-ориентированный язык, имеющий С-подобный синтаксис. Сценарии на JavaScript используются в обозревателье Netscape Navigator, а сценарии на языке JScript — в Microsoft Internet Explorer. Лексика и синтаксис этих языков практически идентичны, различия заключаются в объектных моделях, используемых обозревателями (т. е. в совокупностях элементов, их атрибутов и событий Web-документа). Internet Explorer дополнительно поддерживает язык VBScript, функционально эквивалентный языку JScript, но более простой в освоении. Этот язык наиболее привычен для программистов, имеющих опыт работы с Microsoft Visual Basic.

Сценарии могут применяться как расширение обозревателя (клиентское расширение) или как расширение Web-сервера. В первом случае сценарии включаются в Web-документ и используются для создания динамических эффектов при просмотре Web-страницы, а во втором они требуются для динамического создания Web-документов в ответ на запрос пользователя. Для уменьшения нагрузки на Web-сервер часть функций, связанных с предварительной обработкой запросов и введенных данных, целесообразно выполнять на стороне клиента (в Web-обозревателе). В этом случае говорят об *активности на стороне клиента.* Кроме сценариев, такая активность может быть реализована и другими средствами (апплеты Java, элементы управления ActiveX).

Элементы управления ActiveX

Элементы управления ActiveX представляют собой модули расширения, которые могут использоваться и на стороне клиента, и на стороне сервера. Они реализуются с помощью динамических библиотек DLL и встраиваются в Web-документ как дополнительные интерфейсные элементы. Механизм работы элементов управления ActiveX позволяет получать с их помощью неограниченный доступ к локальным ресурсам компьютера пользователя, включая возможность передачи на сервер любой информации с этого компьютера. Поэтому использование элементов ActiveX на стороне клиента в Интернете не всегда оправданно с точки зрения обеспечения безопасности данных.

Код объектов ActiveX может содержать вирус. Если загрузить такой объект, то компьютеру клиента в принципе может быть причинен вред. Для частичного устранения этого недостатка компания Microsoft добавляет в свои программы цифровую подпись (signing), которая гарантирует "чистоту" объектного кода. Кроме того, обозреватель Internet Explorer способен идентифицировать "подозрительные" модули и выдает при необходимости предупреждающее сообщение, предоставляя пользователю возможность отменить выполнение кода. Однако этот механизм не устраняет целиком опасность заражения вирусом компьютеров пользователей, использующих объекты ActiveX.

Если в вызванной Web-странице встречаются новые элементы управления ActiveX, то их предварительно требуется загрузить и установить, а это нарушает универсальность обозревателя. Поэтому элементы управления ActiveX рекомендуется использовать преимущественно на стороне сервера.

Апплеты и сервлеты Java

Апплеты Java (applet — "маленькое приложение") применяются для создания динамически формируемого интерфейса пользователя. Язык Java является объектно-ориентированным языком с синтаксисом, похожим на синтаксис языка C++. Однако возможности языка Java по доступу к локальным ресурсам пользователей сильно урезаны, что делает его безопасным для использования в сети. Апплеты предназначены для выполнения на любых платформах. Их код интерпретируется виртуальной Java-машиной, входящей в состав обозревателя. Использование такого механизма гарантирует целостность локальных данных пользователя. Для включения апплета в Web-страницу применяется специальный тег (tag — метка) языка HTML. *Сервлеты*, в отличие от апплетов, выполняются на стороне сервера и служат для обработки запросов, передаваемых от обозревателя.

Интерфейсы CGI и WinCGI

Для создания модулей расширения Web-сервера могут использоваться *интерфейсы CGI* (Common Gateway Interface — общий шлюзовой интерфейс) и *API* (Application Program Interface — интерфейс прикладного программирования).

Интерфейс CGI является стандартным протоколом взаимодействия между Web-сервером и модулями расширения, которые могут применяться для выполнения дополнительных функций, не поддерживаемых сервером. Например, такие модули используются для обработки получаемой от пользователя информации, для динамического формирования Web-документа, публикации БД на Web-странице и т. д.

Интерфейсу CGI соответствуют обычные консольные приложения операционной системы DOS. Обмен информацией между сервером и модулем расширения осуществляется с помощью стандартного потокового ввода/вывода, а передача управляющих параметров организуется через переменные окружения операционной системы или через параметры URL-адреса модуля расширения.

Для запуска модуля расширения достаточно задать его URL-адрес в строке обозревателя и начать загрузку документа. При получении запроса обозревателя к CGI-приложению сервер запускает это приложение и передает ему данные из командной строки запроса. CGI-приложение формирует ответ и помещает его в выходной поток (на стандартном устройстве вывода), затем сервер посылает этот ответ с использованием протокола HTTP обратно обозревателю. В случае

параллельной обработки нескольких запросов сервер запускает отдельный процесс для обработки каждого запроса, причем для каждого процесса создается копия модуля расширения в памяти компьютера, на котором находится Web-сервер. Поэтому недостатками протокола CGI является невысокая скорость обработки запросов и повышенная загрузка Web-сервера.

Существует адаптированный вариант общего CGI-протокола для среды Windows 3.1 — WinCGI. Он отличается от протокола CGI тем, что управляющие параметры передаются через INI-файл, а входной и выходной потоки данных перенаправлены в специальные файлы. В остальном механизм взаимодействия с сервером аналогичен механизму, используемому протоколом CGI.

Интерфейсы ISAPI/NSAPI

Более перспективными интерфейсами являются *интерфейсы ISAPI/NSAPI* (Internet Server API/Netscape Server API), разработанные фирмами Microsoft и Netscape соответственно. Эти интерфейсы также предназначены для разработки дополнительных модулей расширения Web-сервера. В случае использования этих интерфейсов модули расширения реализуются в виде библиотек DLL. Запуск модуля расширения выполняется сервером в ответ на запрос обозревателя на загрузку URL-адреса этого модуля. Взаимодействие между сервером и модулем расширения осуществляется с помощью специальных объектов (Request, Response). Сервер передает параметры запроса модулю расширения и получает сформированный Web-документ, который с помощью протокола HTTP пересылается обратно обозревателю.

При многопользовательском режиме работы сервера загрузка ISAPI-модуля расширения (библиотеки DLL) происходит один раз при первом обращении. При обработке последующих запросов к модулю расширения сервер использует уже загруженный экземпляр динамической библиотеки. Такой механизм взаимодействия сервера и модуля расширения обеспечивает экономию ресурсов сервера и увеличение скорости обработки запросов. Однако при возникновении ошибок в коде модуля расширения сам Web-сервер аварийно завершит работу (в отличие от интерфейса CGI, при использовании которого в случае возникновения ошибки в модуле расширения сервер может продолжать нормально функционировать). Поэтому для отладки модулей расширения, реализуемых с помощью интерфейса ISAPI, создают аналог модуля на базе интерфейса CGI. После успешной отладки этого аналога модуль расширения переводится на базу ISAPI.

Страницы ASP, PHP и IDC/HTX

Страницы ASP, PHP и IDC/HTX — это специальные типы страниц, используемые для динамического формирования на сервере Web-документов, содержащих информацию из БД.

Страница IDC (Internet Database Connector) содержит псевдоним БД (системную запись, используемую операционной системой для связи с базой данных), SQL-запрос к базе данных, идентификатор пользователя и пароль для доступа к БД.

Страница HTX (страница гипертекста) содержит HTML-шаблон, определяющий, какую информацию и в каком формате будет иметь результирующий файл. Этот файл поддерживает все теги языка HTML и дополнительные теги для размещения информации из БД.

Страница ASP (Active Server Pages, Активные серверные страницы) содержит одновременно HTML-шаблон и SQL-запрос к БД. В ASP-странице используются средства языка JScript и объектная модель ASP, с помощью которых организуется доступ к БД и формируется внешний вид создаваемой Web-страницы. В ASP-страницах так же, как в IDC-страницах, поддерживаются все теги языка HTML и используются дополнительные теги для размещения кода на языке JScript.

Страницы PHP (Personal Home Page, Персональная домашняя страница) разрабатываются с помощью одноименного языка обработки сценариев, команды которого включаются в документ HTML.

ASP-, PHP- и IDC/HTX-страницы обрабатываются Web-сервером, в результате чего генерируется Web-страница, содержащая информацию из БД, которая отсылается обозревателю.

Формирование Web-страниц

В Интернете информация находится на Web-узлах (Web-сайтах), на которых для организации взаимодействия с пользователями сети устанавливается специальное программное обеспечение, в том числе Web-сервер. В функции Web-сервера входит обработка запросов Web-обозревателей пользователей сети. В результате обработки запроса сервер формирует Web-документ, который отсылается обозревателю в формате протокола HTTP.

Web-сервер может формировать динамические Web-страницы и отсылать готовые Web-страницы различными способами. Для формирования динамической Web-страницы, содержащей информацию из БД, дополнительно используются модули расширения сервера (см. рис. 18.1).

Различают пассивное и активное состояния Web-сервера. Так, Web-сервер находится в *пассивном* состоянии, если формируемый им документ содержит статическую текстовую, графическую, мультимедийную информацию и гиперссылки. В таком документе отсутствуют средства ввода и обработки запросов к серверу.

В случае, когда на Web-странице находятся интерфейсные элементы, которые могут в ответ на реакцию пользователя обращаться с запросами к серверу, сервер переходит в *активное* состояние. Для публикации БД основной интерес представляет активный Web-сервер, реализуемый с помощью программных расширений.

Для создания программных расширений Web-сервера, формирующих на Web-странице содержимое БД, используются следующие средства:

- консольные исполняемые программы, использующие интерфейс CGI;
- исполняемые программы, использующие интерфейс WinCGI;
- динамические библиотеки, использующие интерфейс ISAPI;
- динамические страницы IDC/HTX;
- активные серверные страницы ASP.

Кроме того, для организации связи программных расширений Web-сервера с БД используются современные интерфейсы доступа к данным, такие как OLE DB, ADO и ODBC. Эти интерфейсы представляют промежуточный уровень между источником данных и приложением, в качестве которого здесь выступают программные расширения Web-сервера.

Интерфейсы OLE DB, ADO, ODBC

Интерфейсы доступа к источникам данных OLE DB, ADO, используемые в продуктах Microsoft, существенно отличаются от предшествующих интерфейсов, подобных ODBC.

Интерфейс ODBC (Open Database Connectivity — совместимость открытых баз данных) применяется операционной системой для доступа к источникам данных, как правило, к реляционным БД, использующим язык структурированных запросов SQL для организации управления данными.

Интерфейс OLE DB (Object Linking and Embedding DataBase — связывание и внедрение объектов баз данных) является более универсальной технологией для доступа к любым источникам данных через стандартный интерфейс COM (Component Object Model — объектная модель компонентов). Данные могут быть представлены в любом виде и формате (например, реляционные БД, электронные таблицы, документы в RTF-формате и т. д.). В интерфейсе OLE DB используется механизм провайдеров, под которыми понимаются поставщики данных, составляющих надстройку над физическим форматом данных. Такие провайдеры еще называются сервис-провайдерами, благодаря им, можно объединять в однотипную совокупность объекты, связанные с разными источниками данных.

Кроме того, есть OLE DB-провайдер, который реализует интерфейс доступа OLE DB поверх конкретного сервис-провайдера данных. При этом поддерживается возможность многоуровневой системы OLE DB-провайдеров, когда OLE DB-провайдер может находиться поверх группы OLE DB-провайдеров или сервис-провайдеров.

Интерфейс OLE DB может использовать для доступа к источникам данных интерфейс ODBC. В этом случае применяется OLE DB-провайдер для доступа к ODBC-данным. Таким образом, интерфейс OLE DB не заменяет интерфейс

ODBC, а позволяет организовывать доступ к источникам данных через различные интерфейсы, в том числе и через ODBC.

Интерфейс ADO (ActiveX Data Objects — объекты данных ActiveX) предоставляет иерархическую модель объектов для доступа к различным OLE DB-провайдерам данных. Он характеризуется еще более высоким уровнем абстракции и базируется на интерфейсе OLE DB. Объектная модель ADO включает небольшое количество объектов, которые обеспечивают соединение с провайдером данных, создание SQL-запроса к данным, создание набора записей на основе запроса и др. Разрабатывая интерфейс ADO, фирма Microsoft предназначала его для использования в сетях Интернет/интранет для доступа к различным источникам данных.

Статическая публикация БД

При публикации БД на Web-страницах используются следующие способы формирования Web-страниц:

- статическая публикация Web-страниц, содержащих информацию из БД;
- динамическая публикация Web-страниц, содержащих информацию из БД.

Рассмотрим особенности формирования каждого типа страниц.

В случае *статической* публикации БД Web-страницы с соответствующей информацией создаются и хранятся на Web-сервере до поступления запроса пользователя на их получение (в виде файлов на жестком диске в формате Web-документа). Генерацию таких страниц может выполнять обычное приложение Windows, имеющее доступ к БД. Этот способ используется при публикации информации, редко обновляемой в базе данных (обновление БД можно выполнять с требуемой периодичностью или при внесении в нее изменений). Такая организация публикации БД в Интернете имеет ряд преимуществ, заключающихся в получении более быстрого доступа к Web-документам и уменьшении нагрузки на сервер при обработке запросов.

Отметим, что при обработке запроса на получение Web-страницы статическим способом сервер может находиться как в пассивном, так и в активном состоянии. В активном состоянии сервер находится в случае, если Web-страницы содержат интерактивные элементы, которые в ответ на реакцию пользователя обращаются с запросами к серверу.

Динамическая публикация БД

Динамическая публикация используется в случаях, когда необходимо публиковать информацию из БД, содержимое которой часто обновляется. Таким способом публикуется информация из БД для интернет-магазинов и информацион-

ных систем, работающих в реальном масштабе времени, например систем продажи билетов.

При динамической публикации страницы создаются после поступления запроса пользователя на сервер. Сервер передает запрос на генерацию таких страниц своей программе-расширению, которая формирует требуемый документ. Затем сервер отсылает готовые Web-страницы обратно обозревателю. Для формирования динамических страниц используются различные средства и технологии: ASP и IDC/HTX-страницы, программы расширения сервера на основе интерфейсов CGI и ISAPI.

В случае использования ASP и IDC/HTX-страниц запрос на получение динамически формируемой Web-страницы передается специальным динамическим библиотекам, входящим в состав Web-сервера. Например, если используется Personal Web Server и публикация осуществляется средствами IDC/HTX, то применяется динамическая библиотека httpodbc.dll. Такие библиотеки анализируют файл ASP или файлы IDC и HTX, которые используются в качестве шаблона.

Путь к файлу ASP или IDC задается в строке запроса. В зависимости от расширения имени файла в строке запроса сервер принимает решение о передаче управления тому или иному модулю расширения: если указывается расширение exe, то используется интерфейс CGI, а если dll — то интерфейс ISAPI.

Отметим, что при формировании динамической Web-страницы сервер находится в активном состоянии. После отправки страницы обозревателю сервер может перейти в пассивное состояние, если сформированная страница содержит только статическую информацию. На одном Web-сервере могут использоваться страницы, создаваемые как статическим, так и динамическим способом.

Web-приложения

Среди программных средств, используемых для получения информации из Интернета, выделилась новая категория программ — *Web-приложения*, или интернет-приложения. Web-приложение — это совокупность Web-страниц, клиентских и серверных сценариев, расположенных на одном или нескольких компьютерах и выполняемых в рамках одной информационной системы (целевой задачи). Для разработки Web-приложений могут использоваться различные комбинации рассмотренных технологий.

Информационные системы, построенные на основе Web-приложений, характеризуются многоуровневой архитектурой и позволяют использовать все достоинства Интернета. Приложения, реализующие технологию публикации БД в Интернете, составляют отдельный класс Web-приложений. В принципе такие программы представляют собой многоуровневые клиент-серверные приложения. Однако Web-приложения имеют свои особенности, связанные с организацией

работы Интернета. Разработчику Web-приложения приходится учитывать следующие аспекты:

- совместимость Web-обозревателей;
- разграничение доступа и обеспечение безопасности данных;
- надежность линий связи.

Web-приложения, публикующие БД на Web-страницах, выполняются на стороне сервера. Сервер обрабатывает запросы обозревателя. Запросы к БД Web-сервер передает серверу приложений или серверу баз данных. Обработав запрос, сервер БД передает нужные данные Web-серверу, который формирует Web-документ и отсылает его обозревателю.

Функции обозревателя заключаются в отображении Web-страниц, сгенерированных сервером или модулями расширения, и отправке запросов пользователя Web-приложению, т. е. обозреватель является связующим звеном между пользователем и Web-приложением.

Протоколы передачи данных

В Интернете могут использоваться различные платформы и Web-серверы: для операционных систем Windows NT/2000/2003 — это Microsoft Internet Information Server (MS IIS), для UNIX — Apache Server (этот сервер распространяется в исходных кодах, поэтому после перекомпиляции может выполняться и в других операционных системах), а для Windows 98 — Microsoft Personal Web Server.

Запросы от обозревателя поступают на Web-сервер в соответствии с установленным протоколом обмена, например HTTP. Вообще, в Интернете поддерживается более десятка протоколов. Протокол HTTP является одним из самых распространенных и предназначен для передачи данных различных форматов между обозревателем и Web-сервером. Соединение между компьютером-отправителем и компьютером-получателем осуществляется с помощью протокола низкого уровня TCP/IP (Transport Control Protocol/Internet Protocol — протокол управления передачей/межсетевой протокол).

Протокол TCP/IP представляет собой универсальный платформо-независимый протокол передачи данных по сети. За физическую передачу данных отвечает протокол IP. TCP представляет собой протокол более высокого уровня. Он разбивает в компьютере-отправителе файлы на пакеты, добавляет в каждый из них адрес получателя и порядковый номер пакета в группе пакетов. В компьютере-получателе протокол TCP собирает файлы из пакетов, проверяет их целостность и при необходимости посылает запрос на повторную трансляцию.

При отправке серверу HTTP-запроса на формирование динамической страницы, содержащей отчет из БД, сервер включает в работу модули расширения. Взаимодействие Web-сервера и модуля расширения основывается на интерфейсах CGI или API. Модуль расширения получает запрос и обрабатывает его, формируя ответ (обычно Web-страницу).

Универсальный указатель ресурсов (URL)

Обозреватель формирует запрос на получение нужной страницы с помощью универсального указателя ресурса — URL (Universal Resource Locator). Для загрузки требуемой страницы в окне обозревателя указывается строка адреса. Загрузку требуемого документа можно осуществить и из Web-документа с помощью специальных тегов.

Приведем структуру URL для протокола HTTP:

```
http://<хост>/<порт>:<путь>?<поиск>
```

Здесь:

- `http` — имя протокола доступа к ресурсу;
- `<хост>` — имя домена поиска требуемого Web-узла в Интернете;
- `<порт>` — номер порта, который задает номер логического канала связи в Интернете;
- `<путь>` — локальный путь к файлу;
- `<поиск>` — дополнительные параметры запроса.

В обозревателях используются следующие два типа адресов указателей:

- *абсолютные указатели* задают адрес, полностью определяющий компьютер, каталог и файл. В отличие от относительных, абсолютные адреса могут ссылаться на файлы, расположенные на других компьютерах;
- *относительные указатели* используются для получения доступа к файлу, расположенному на том же компьютере, что и документ, в котором находится указатель на этот файл. Например, если обозреватель загрузил страницу, находящуюся по адресу **http://myweb**, то при задании относительного адреса **/my** реальный адрес будет **http://myweb/my**. Относительные адреса удобны для использования в пределах одного компьютера.

Основу Web-приложений составляет совокупность Web-документов и различных шаблонов, использующих формат языка HTML или его расширений. В связи с этим для разработки Web-приложений прежде всего необходимо знать HTML.

Язык разметки гипертекста HTML

Язык HTML представляет собой универсальный язык подготовки документов для Интернета. HTML-документ является разновидностью Web-документа и в общем случае представляет собой обычный ASCII-файл, который можно отредактировать в любом текстовом редакторе. Он состоит из текстовых данных,

снабженных специальными управляющими метками — тегами. Теги несут в себе служебную информацию для обозревателя, задавая различные режимы форматирования.

Теги заключаются в угловые скобки. Например, начальным тегом документа является следующий тег:

```
<HTML>
```

Теги могут быть парными и непарными. Парные теги делятся на открывающий и закрывающий. Пара таких тегов называется *контейнером.*

Например, весь текст HTML-документа заключен в теги `<HTML>` и `</HTML>`:

```
<HTML>
...
<!-- Это комментарий, который в обозревателе не отображается.
     Между тегами <HTML> и </HTML> располагается
     текст HTML-документа с другими управляющими тегами --!>
...
</HTML>
```

Здесь `<HTML>` — открывающий тег, `</HTML>` — закрывающий тег, между ними находятся все данные. `<!-- Комментарий --!>` — пример непарного тега.

HTML-документ состоит из двух частей — заголовка и тела документа. *Заголовок* является необязательным элементом, он может использоваться для того, чтобы задать название документа, определить отношения между несколькими документами и т. д. Заголовок и тело документа обрамляются специальными тегами. В *теле* HTML-документа находятся текст документа и различные управляющие теги.

Расширяемый язык разметки XML

Расширяемый язык разметки XML представляет собой дальнейшее развитие языка HTML и по сравнению с ним обеспечивает ряд дополнительных возможностей:

- подготовка и настройка XML-документов со сложной структурой;
- использование информационных объектов (entity) для работы с группами данных;
- описание типа документа DTD (Document Type Definition);
- контроль правильности документа;
- использование XML-формата для хранения структурированных однотипных данных (ранее для этой цели применялись только БД).

Отметим, что XML принадлежит подмножеству стандартного обобщенного языка разметки SGML (Standard Generalized Markup Language). Хотя язык SGML предоставляет более развитые возможности по сравнению с XML, его примене-

ние требует много времени на подготовку и обработку соответствующего SGML-документа. Поэтому он не получил такого широкого распространения, как XML или HTML.

Главное отличие XML от HTML заключается в том, что с помощью XML можно не только наполнить создаваемый документ размеченным содержимым, но и определить структуру документа и типы хранимых в нем данных. Средства XML обеспечивают стандартизованное представление данных, облегчая этим задачу использования (импорта) данных из других источников.

Документы XML можно разделить на правильно построенные (well-formed) и действительные (valid). *Правильно построенные* документы XML удовлетворяют спецификации XML, но не имеют определения типа документа DTD. *Действительные* документы XML, в отличие от правильно построенных, содержат также определение типа документа DTD (определяющее типы используемых данных и возможную структуру документа) или ссылаются на него.

Спецификацию XML поддерживает ряд современных программ просмотра, в частности, обозреватели Internet Explorer версии 4.0 и выше и Netscape Navigator версии 5.0 и выше.

При описании XML-документа накладываются следующие дополнительные (по сравнению с HTML) ограничения:

- каждому открывающему тегу должен соответствовать закрывающий тег;
- пары открывающих и закрывающих тегов могут вкладываться друг в друга, однако пересечение пар открывающих и закрывающих тегов не допускается;
- при определении элементов и атрибутов учитывается регистр символов;
- все значения атрибутов должны указываться в апострофах (`''`) или кавычках (`""`).

Рассмотрим более подробно состав XML-документа, а также назначение, объявление и примеры использования его составляющих.

В XML-документе обычно присутствуют: элементы, атрибуты, информационные объекты и комментарии. *Элементы* представляют собой части документа. Элемент объявляется так:

```
<!ELEMENT Имя Содержание>
```

Объявление элементов может размещаться внутри документа или в определении типа документа DTD. В DTD можно объявить произвольное число элементов, при этом не все из них должны использоваться в документе XML. Отдельные элементы могут содержать другие элементы. Например, чтобы указать, что элемент `employee` (сотрудник) содержит элементы `surname` (фамилия) и `post` (должность), нужно поместить в DTD определения элементов в следующем порядке:

```
<!ELEMENT surname(#PCDATA)>
<!ELEMENT post(#PCDATA)>
<!ELEMENT employee(surname, post)>
```

Здесь строка `#PCDATA` является *атрибутом* и указывает, что соответствующий элемент содержит символьные данные.

Другие символы служат для задания режима использования элемента, например, определяют, что элемент в содержании документа необязателен или может использоваться несколько раз. Основные символы, применяемые при описании элемента, приведены в табл. 18.1.

Таблица 18.1. *Основные символы в описании элемента*

Символ	Имя символа	Назначение
`()`	Круглые скобки	Содержит альтернативные элементы или элементы списка
`|`	Вертикальная линия	Разделение альтернативных элементов
`?`	Знак вопроса	Элемент может отсутствовать или использоваться один раз
`,`	Запятая	Разделение элементов в списке
`*`	Звездочка	Элемент может отсутствовать или использоваться любое число раз
`+`	Плюс	Элемент должен использоваться не менее одного раза

Например, в строке

```
<!ELEMENT employee(surname, post+)>
```

знак `+` указывает на то, что в составе элемента `employee` элемент `post` должен присутствовать не менее одного раза.

Атрибуты служат для уточнения характеристик элементов, вида содержащейся в них информации и порядка ее размещения. Атрибуты можно указывать в строке объявления элемента (как показано выше) или в описании DTD (в задаваемом с помощью тега `<!ATTLIST>` списке атрибутов). Для действительного документа XML объявление атрибута в DTD задается так:

```
<!ATTLIST Имя_элемента Имя_атрибута Тип Значение_по_умолчанию>
```

Атрибут элемента может быть строковым, маркированным или перечислимым и иметь следующие типы и назначение:

- `CDATA` (символьные данные);
- `Enumerated` (набор возможных значений);
- `ID` (уникальный идентификационный номер);
- `IDREF` (указатель на элемент с заданным значением `ID`);
- `IDREFS` (указатель на несколько элементов в документе в виде номеров `ID`, разделенных пробелами);

- `ENTITY` (имя внешнего информационного объекта);
- `ENTITIES` (указатель на несколько внешних информационных объектов в виде имен `ENTITY`, разделенных пробелами);
- `NMTOKEN` (имя XML);
- `NOTATION` (имя в обозначении, объявленном в DTD);
- `NMTOKENS` (множество имен XML, разделенных пробелами).

Предположим, что в документе содержится тег вида:

```
<greeting language="Russia">
```

Приведенный тег может быть объявлен в DTD так:

```
<!ELEMENT greeting #PCDATA
<!ATTLIST greeting language CDATA "English"
```

Здесь в первой строке указывается, что элемент `greeting` может содержать управляемые символьные данные. Во второй строке для этого элемента задается атрибут `language` типа `CDATA`, и для атрибута устанавливается значение `English`, используемое по умолчанию.

Комментарии в XML, как и в HTML, пишутся в обрамлении открывающего `<!--` и закрывающего `-->` тегов.

Обычно документ XML состоит из двух частей: пролога и данных. *Пролог* включает в себя объявления XML и описание типа документа DTD, а *данные* представляют собой размеченный текст. В общем случае документ XML может содержать данные и объявления из различных источников: баз данных, сценариев и др., т. е. документ XML состоит из пролога и следующих за ним ***информационных объектов*** (entity). В качестве информационного объекта может выступать файл, запись базы данных или другой элемент данных.

Информационные объекты позволяют задавать имена для наборов символьных данных или файлов так, чтобы их можно было включать в состав XML-документа. Например, если в документе многократно используется символьная строка с информацией об издательстве, то целесообразно создать информационный объект с помощью следующего кода:

```
<!ENTITY redBHV "СПб: БХВ-Петербург"
```

Теперь в документе достаточно указывать ссылки вида `&redBHV`, синтаксический анализатор при обработке документа заменит их на строку `СПб: БХВ-Петербург`.

Приведенный пример показывает использование *внутреннего* информационного объекта, описание которого содержится в самом документе. Кроме того, допустимо использовать *внешний* информационный объект, который включает имя и ссылку на контейнер, хранящий его содержание. Объявление внешних

ссылку на контейнер, хранящий его содержание. Объявление внешних информационных объектов выполняется с помощью инструкции следующего формата:

```
<!ENTITY name (PUBLIC | SYSTEM) "content")
```

Здесь строка `name` задает имя информационного объекта, а ключевые слова `PUBLIC` или `SYSTEM` указывают на доступность использования информационного объекта в документах.

В XML имеется 5 *предопределенных* общих ссылок: `<`, `>`, `'`, `&quote;` и `&`, которые ссылаются на текстовые информационные объекты (символы) <, >, ', " и & соответственно. Они применяются в документах XML, чтобы избежать неправильной интерпретации соответствующих символов как части разметки документа. Например, символы < и > при их непосредственном использовании могут интерпретироваться как начало и конец тега.

Создание отдельного ***определения типа документа DTD*** для каждого документа XML обеспечивает:

- возможность проверять корректность документа для синтаксического анализатора;
- удобство задания и изучения структуры документа определенного типа;
- возможность применять не только элементы и атрибуты, но и информационные объекты.

В состав DTD входят пять частей:

- типы документов;
- элементы;
- атрибуты;
- информационные объекты;
- комментарии.

Определение типа документа может размещаться внутри документа XML, в этом случае говорят о *внутреннем* DTD, или быть задано во внешнем файле (*внешнее* DTD). Применительно к внешним DTD различают системное (`SYSTEM`) и общее (`PUBLIC`) определения типа документа. Системное DTD может использоваться только одним или несколькими документами XML, а общее DTD — любым документом XML.

Одно из важнейших назначений DTD заключается в определении структуры документа как иерархии составляющих его элементов. В частности, применительно к книге соответствующее внутреннее DTD и сам документ XML могут быть заданы так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book [
<!ENTITY % phras "PCDATA | emphasis">
<!ELEMENT book (title_page, content)>
<!ELEMENT title_page (author, title)>
<!ELEMENT author (#PCDATA)>
```

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT content (part, chapter+)>
<!ELEMENT part (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
]>
<book>
<title_page>
<author> Ivanov Dm. </author>
<title> Name of book </title>
</title_page>
<content>
<part> Name of Part </part>
<chapter> Text of Chapter1 </chapter>
<chapter> Text of Chapter2 </chapter>
<chapter> Text of Chapter3 </chapter>
</content>
</book>
```

Здесь первая строка кода указывает, что это XML-документ, и определяет используемую систему кодирования — 8-битовую кодировку Unicode, соответствующую символам ASCII. Тег `<!DOCTYPE book [` указывает на начало описания DTD с именем `book`, а тег `]>` — на окончание DTD. В строке

```
<!ELEMENT content (part, chapter+)>
```

знак `+` говорит о том, что элементов `chapter` (глава) в родительском элементе `content` (содержание) книги может быть несколько.

При обнаружении ошибок в исходном коде документа XML обозреватель выдает сообщение о характере и местоположении первой из них, как показано, например, на рис. 18.2. Отметим, что при подготовке документов HTML возможность проверки синтаксических ошибок отсутствует в принципе.

Для приведенного документа XML и его описания DTD в окне обозревателя Microsoft Internet Explorer будет отображаться структура, изображенная на рис. 18.3.

Для создания *внешнего* DTD следует поместить объявления всех размещаемых в нем элементов, атрибутов и информационных объектов в отдельный файл с расширением dtd, например, в файл book.dtd. В состав внешнего DTD нужно поместить также инструкцию по обработке кода XML, включив строку

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Чтобы получить доступ к внешнему DTD, размещенному в сети, в XML-документ нужно включить следующее описание:

```
<!DOCTYPE book SYSTEM "http://www.xom.com/dtds/book.dtd"
```

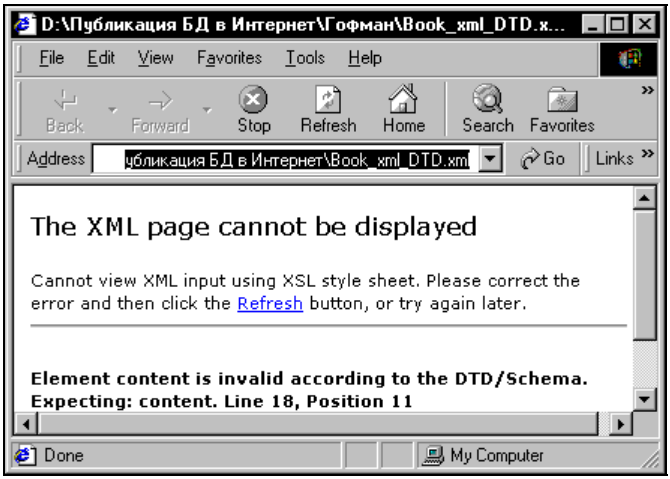


Рис. 18.2. Сообщение об ошибке в окне обозревателя

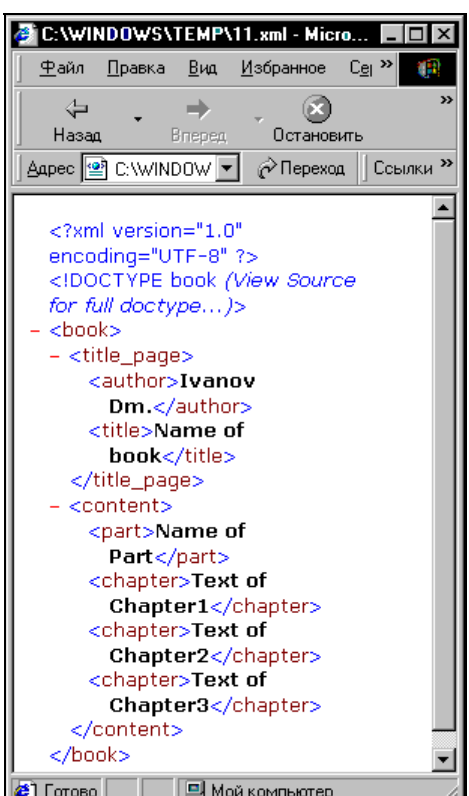


Рис. 18.3. Вид структуры в окне обозревателя

Здесь `http://www.xom.com/dtds/book.dtd` — адрес размещения файла DTD в сети. Если DTD является общим, в приведенной строке вместо слова `SYSTEM` нужно указать `PUBLIC`.

Как отмечалось, расширяемый язык разметки XML представляет собой развитие языка HTML и по сравнению с ним обеспечивает ряд дополнительных возможностей. Главное отличие XML от HTML заключается в том, что он позволяет определить структуру документа и типы хранимых в нем данных.

Напомним, что одно из достоинств XML состоит в том, что в разрабатываемых с его помощью документах описание структуры хранимых данных отделено от собственно данных. В связи с этим XML представляет собой удобное ***средство обмена данными*** между отдельными приложениями, т. к. позволяет обеспечить согласованный обмен данными в случаях, когда структуры данных (например, имена и типы полей) в приложениях различаются.

Кроме того, с помощью XML можно упростить доступ к данным, хранимым в базах данных. Например, для доступа к данным персональных БД или табличного процессора Excel пользователю требуется установить соответствующие программные инструменты. Вместо этого можно создать активные серверные страницы (ASP) или сценарии на языке JScript или VBScript, которые будут извлекать данные из БД и помещать их в документ XML. В дальнейшем информацию из полученного таким образом документа XML можно использовать в других приложениях или отображать на Web-страницах. Т. е. полученные данные становятся доступными для всех пользователей, имеющих обозреватель, независимо от наличия СУБД или табличного процессора.

Документы XML могут использоваться как на стороне клиента, так и на стороне сервера. Применение XML-документов на стороне *клиента* в настоящее время сдерживается в основном отсутствием соответствующих инструментов: не все обозреватели поддерживают просмотр и работу с документами XML, хотя в большинстве современных продуктов такая возможность уже имеется. Как уже говорилось, к ним относятся обозреватели Microsoft Internet Explorer, начиная с версии 4.0, и Netscape Navigator версии 5.0 и выше.

При работе с документами XML на *стороне сервера* используются языковые средства, обычно применяемые для расширения возможностей сервера, такие как Java и JScript. Отметим еще, что последние версии современных систем программирования, ориентированных на разработку Web-приложений, также поддерживают средства создания и обработки документов XML. В частности, такие средства имеются в составе JBuilder 4.0 и Delphi 7.

Более того, в системе Delphi 7 XML является языком кодирования данных, передаваемых при использовании Web-служб с помощью протокола SOAP, объединяющего XML и протокол HTTP и обеспечивающего тем самым универсальность обмена данными.

Программа XML Mapper

Программа XML Mapper (XML-преобразователь) задает отображение (определяет соответствие) между XML-документами и пакетами данных, используемыми клиентскими наборами данных. Эту программу можно запустить из среды Delphi командой **Tools\XML Mapper** или запуском файла xmlmapper.exe, находящегося в каталоге BIN главного каталога Delphi, а также через меню **Пуск** Windows, выбрав команду **Программы\Borland Delphi 7\XML Mapper**.

Пакет данных содержит метаданные, описывающие структуру XML-документа. *Отображение* (трансформация) представляет собой соответствие между элементами XML-документа и полями в пакете данных. С помощью программы XML Mapper можно задать отображения:

- существующей XML-схемы или документа на клиентский набор данных;
- существующего пакета данных на новую XML-схему;
- существующей XML-схемы на существующий пакет данных.

Заданное отображение можно сохранить в трансформационном файле (файле соответствия) с расширением xtr. Обычно трансформационные файлы создаются в паре: для прямого и обратного преобразования. Полученные с помощью программы XML Mapper трансформационные файлы могут использоваться компонентами типа `TXMLTransform`, `TXMLTransformProvider` и `TXMLTransformClient` для выполнения преобразования данных в приложении.

Окно программы XML Mapper состоит из трех панелей (рис. 18.4):

- панель документа (**Document**) — отображает XML-документ и 3 варианта его схем;
- панель отображения (**Transformation**) — задает соответствие между XML-документом и пакетом данных;
- панель пакета данных (**Datapacket**) — отображает метаданные пакета данных.

Панель документа содержит две вкладки, позволяющие просматривать содержимое открытого (загруженного) XML-документа в виде дерева элементов (вкладка **Document View**) и в виде схем DTD, XDR или XSD (вкладка **Schema View**). Открытие документа выполняется командой **File\Open** главного меню программы или командой **Open XML Document** контекстного меню панели документа.

Замечание

Файлы с расширениями dtd (Data Type Definition), xdr (reduced XMLData) и xsd (XML schema) служат для хранения соответствующих вариантов схем XML-документа, отображаемых на вкладке **Schema View**.

Панель пакета данных содержит две вкладки, позволяющие просматривать содержимое текущего пакета данных в виде дерева полей (вкладка **Field View**) и структуры (вкладка **Datapacket View**). Открытие пакета данных выполняется командой **File\Open** главного меню программы или командой **Open Datapacket**

контекстного меню панели пакета данных. Файл пакета данных имеет расширение xml, кроме того, можно открывать пакет данных, сохраненный в двоичном формате (файл с расширением cds).

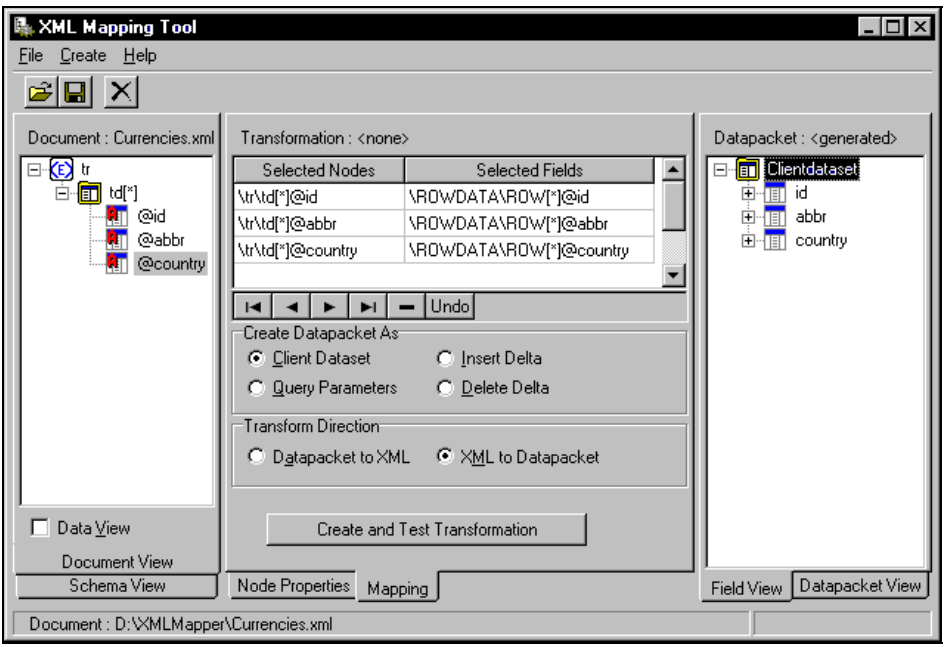


Рис. 18.4. Главное окно программы XML Mapper

Панель отображения определяет соответствие между элементами XML-документа и полями пакета данных. Эта панель содержит две вкладки: **Node Properties** и **Mapping**.

Вкладка **Node Properties** позволяет просматривать и изменять свойства элемента (Node), выбранного в открытом XML-документе. Свойства элементов открытого документа можно сохранить в файле с расширением xrp. Сохранение свойств выполняется командой **File\Save\Repository** главного меню программы или командой **Save Repository** контекстного меню панели отображения. Впоследствии сохраненные свойства элементов XML-документа можно загрузить из файла .xrp.

Свойства выбранного элемента отображаются в таблице, содержащей названия свойств (столбец **Properties**) и значения свойств (столбец **Values**). Отметим следующие свойства:

- `Attribute Name` (имя элемента);
- `Mapped Name` (имя поля в пакете данных, которому соответствует элемент);
- `Data Type` (тип данных, например, `String`, `Float` или `Integer`);

- `Data Format` (формат данных);
- `Max Length` (максимальная длина (для строковых данных));
- `Value Required` (требование обязательности значения);
- `In Primary Key` (признак включения поля в состав первичного ключа пакета данных);
- `Default Value` (значение по умолчанию);
- `Sample Value` (пример значения).

Первоначально свойства имеют значения по умолчанию, определяемые на основании информации из XML-документа. Эти значения можно изменять, например, задать другой тип данных.

На вкладке **Mapping** находится таблица, отображающая соответствие между элементами XML-документа (столбец **Selected Nodes**) и полями пакета данных (столбец **Selected Fields**). Добавление нового элемента к таблице выполняется двойным щелчком на нем или командой **Select** контекстного меню элемента. Команда **Select All** включает в таблицу все элементы документа. Под таблицей соответствия расположен Навигатор, позволяющий перемещаться по элементам, а также удалять элементы из таблицы.

Создание пакета данных из XML-документа и, наоборот, получение XML-документа из пакета данных выполняется, соответственно, командами **Create\Datapacket from XML** и **Create\XML from Datapacket** главного меню или командами **Create Datapacket from XML** и **Create XML from Datapacket** контекстных меню панелей документа и пакета данных. Ниже приводится текст файла Currencies.xml XML-документа и созданного на его основе файла пакета данных CurrenciesDatapacket.xml.

Файл Currencies.xml:

```
<?xml version="1.0" standalone="yes"?>
 <tr>
   <td id="1" abbr="CHF" country="Swiss"/>
   <td id="2" abbr="RUB" country="Russia"/>
   <td id="3" abbr="GBP" country="Great Britain"/>
   <td id="4" abbr="USD" country="USA"/>
 </tr>
```

Файл CurrenciesDatapacket.xml:

```
<?xml version="1.0" standalone="yes"?>
<DATAPACKET Version="2.0">
<METADATA>
  <FIELDS>
    <FIELD attrname="id" fieldtype="string" WIDTH="1"/>
    <FIELD attrname="abbr" fieldtype="string" WIDTH="3"/>
```

```
    <FIELD attrname="country" fieldtype="string" WIDTH="13"/>
  </FIELDS>
  <PARAMS/>
</METADATA>
<ROWDATA/>
</DATAPACKET>
```

Отображение (трансформация) XML-документа на пакет данных и обратное преобразование выполняются командой **Create\Transformation** главного меню или командой **Create Transformation** контекстного меню панели отображения. Преобразование также можно выполнить нажатием кнопки **Create and Test Transformation**, при этом направление преобразования задается переключателями группы **Transform Direction**. Созданное отображение можно сохранить в трансформационном файле командой **File\Save\Transformation** главного меню или командой **Save Transformation** контекстного меню панели отображения.

Глава 19

Web-приложения и интерфейсы

Современные Web-приложения применяются для создания информационных систем в сетях Интернет/интранет и обычно строятся как многозвенные (многоуровневые) приложения, публикующие БД. Это уменьшает нагрузку на сервер баз данных и линии связи. В этой главе описаны характеристика и структура Web-приложений, особенности Web-приложений, публикующих БД в сетях Интернет/интранет. Кроме того, рассматриваются интерфейсы, используемые при программировании Web-приложений.

Характеристика Web-приложений

Развитие сетей Интернет/интранет привело к появлению новой категории программ — Web-приложений. К Web-приложениям относят набор Web-страниц, сценариев и других программных средств, расположенных на одном или нескольких компьютерах (клиентских и серверных) и объединенных для выполнения прикладной задачи. При этом Web-приложения, публикующие БД в Интернете, представляют собой отдельный класс Web-приложений.

Современные информационные системы, построенные на основе Web-приложений, использующих БД, базируются на многоуровневой клиент-серверной архитектуре и принципах функционирования Интернета. С основами этой архитектуры мы ознакомились в предыдущей главе. Здесь мы кратко повторим, как осуществляется работа приложений в Интернете.

Web-приложения выполняются на стороне Web-сервера, который находится на Web-узлах сети Интернет. Web-сервер обрабатывает запросы обозревателя на получение Web-страниц и отсылает ему требуемые данные в формате Web-документов.

Обмен данными в Интернете осуществляется на основе коммуникационного протокола TCP/IP и протокола более высокого уровня (приложений) HTTP. Упрощенная схема функционирования Web-приложения приведена на рис. 19.1.

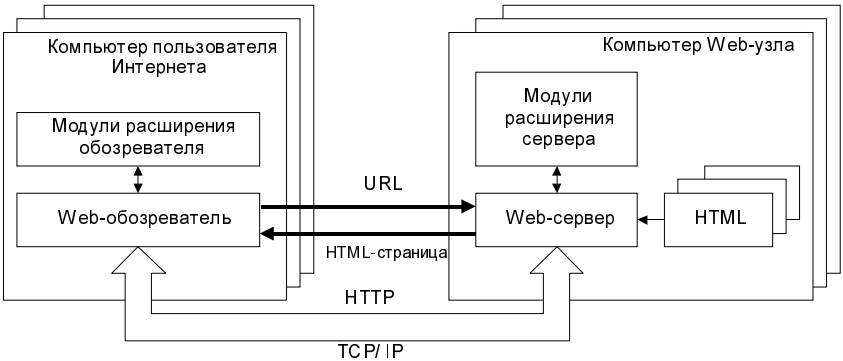


Рис. 19.1. Схема функционирования Web-приложения с модулями расширения

Напомним, что под Web-документом (Web-страницей) понимаются используемые в Интернете документы в форматах HTML, XML, шаблоны в форматах ASP, HTX и т. д.

Для доступа к Web-страницам используются специальные клиентские программы — Web-обозреватели, находящиеся на компьютерах пользователей Интернета. Обозреватель формирует запрос на получение требуемой страницы или другого ресурса с помощью адреса URL. Функции обозревателя заключаются в отображении Web-страниц, сгенерированных сервером или модулями расширения, и отправке запросов пользователя Web-приложению, т. е. обозреватель является связующим звеном между пользователем и Web-приложением. При этом Web-обозреватель устанавливает соединение с требуемым Web-узлом, используя различные протоколы передачи данных, в частности уже рассмотренный протокол HTTP.

Web-приложения в сетях интранет

Развитие технологий глобальной сети Интернет привело к появлению архитектурных и технологических решений для локальных корпоративных сетей, называемых также сетями *интранет*. В общем случае под сетью интранет понимают выделенную часть Интернета, в которой выполняется Web-приложение (информационная система).

Применение интернет-технологий в корпоративных интранет-сетях позволяет повысить эффективность функционирования сетей и используемых в них информационных систем. Приложения, построенные на основе интернет-технологий, характеризуются надежностью и масштабируемостью, открытостью архитектуры, простотой освоения и использования.

Кроме того, использование сетей интранет характеризуется значительным снижением затрат на обслуживание, модернизацию и наращивание сети по сравне-

нию с традиционными корпоративными сетями, построенными на клиент-серверных технологиях. Важным достоинством сетей интранет является возможность развертывания корпоративных локальных и глобальных сетей на существующей инфраструктуре.

При использовании Web-приложений в сети интранет может использоваться архитектура, показанная на рис. 19.2. Сети интранет в общем случае имеют различные внутренние структуры, причем они могут и не иметь выхода в Интернет.

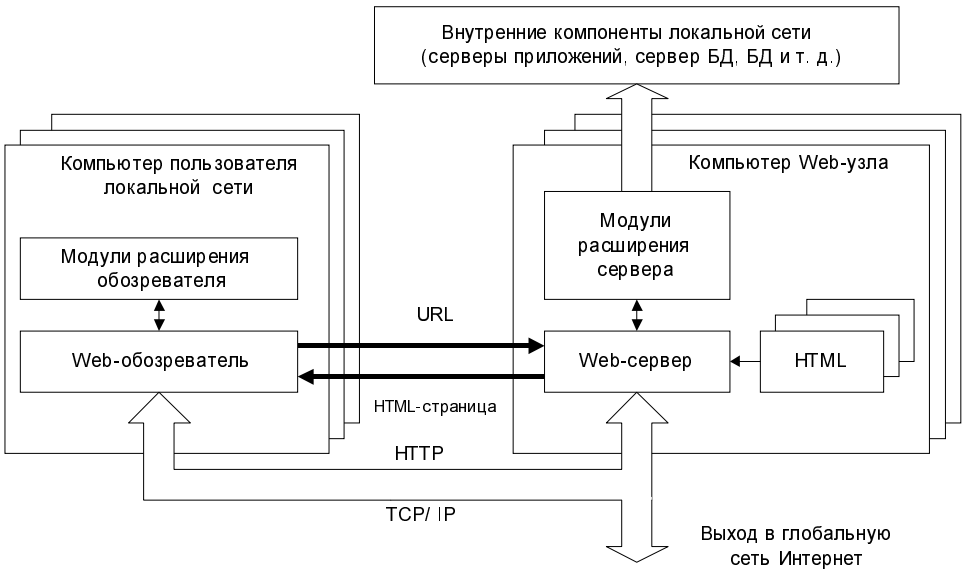


Рис. 19.2. Схема функционирования
Web-приложения с модулями расширения

В качестве клиентских приложений в этой архитектуре выступают Web-обозреватели, которые обращаются с запросами к серверу БД или к серверу приложений через Web-сервер. В зависимости от используемой конфигурации Web-сервер может находиться на сервере БД или на сервере приложений.

В функции Web-сервера в сети интранет входит обработка запросов Web-обозревателей на получение информации из разделяемых БД, преобразование этих запросов (может выполняться модулями расширения Web-сервера) в SQL-запросы или другие форматы, понятные для сервера БД или сервера приложений.

Кроме того, интранет-приложение предоставляет следующие дополнительные возможности:

- удаленные доступ и управление. Концепция *удаленного доступа* подразумевает возможность подключения к интранет-сети извне, т. е. с любого компьютера из Интернета. Под *удаленным управлением* понимается подключение к

локальной сети и выполнение функциональных операций по управлению ее ресурсами с удаленного компьютера;

- выход в Интернет клиентов сети. При этом становятся доступными услуги, предоставляемые глобальной сетью: получение актуальной информации в различных сферах, электронная почта, обмен данными с внешними источниками, использование приложений, находящихся в Интернете, и т. д.

Сеть интранет может быть построена на основе использования Web-сервера в локальной сети или на основе услуг, предоставляемых внешним (интернетовским) Web-сервером. Такие услуги, как правило, предоставляет интернет-провайдер, обеспечивающий возможность использования функций своего Web-сервера. Многие компании используют совмещенную структуру Web-серверов, при которой сама локальная сеть строится с использованием собственного Web-сервера, а выход в глобальную сеть осуществляется через Web-сервер провайдера.

Применение архитектуры Интернета в сетях интранет по сравнению с традиционными архитектурами локальных сетей дает следующие преимущества: стандартизация пользовательского интерфейса; удобное администрирование и конфигурирование; удешевление установки и лицензирования клиентских компьютеров.

Для расширения возможностей клиентской части (обозревателя) и серверной части разрабатывают модули расширения обозревателя и сервера, используемые для динамического управления интерфейсными объектами (компонентами) Web-документа.

Web-приложения с модулями расширения серверной части

Архитектура Web-приложений с модулями расширения сервера (рис. 19.3) может включать в себя стандартные модули расширения — DLL-библиотеки, реализующие, например, технологии ASP, IDC/HTX, объекты ActiveX и пр. Кроме того, допускается подключение дополнительных модулей, разработанных с использованием интерфейсов CGI, ISAPI и др.

В этом случае в функции Web-сервера входят обработка запросов Web-обозревателей пользователей сети, вызов (загрузка) соответствующего модуля расширения сервера и передача ему параметров запроса. В результате обработки запроса модулями расширения сервера формируется Web-документ с использованием различных HTML-шаблонов. Готовый Web-документ Web-сервера отсылается обратно Web-обозревателю в формате протокола HTTP.

Как отмечалось, различают пассивное и активное состояние Web-сервера. Так, Web-сервер находится в пассивном состоянии, если формируемый им документ содержит статическую информацию, т. е. на Web-странице отсутствуют средства ввода и обработки запросов к серверу.

Если Web-документ динамически создается в ответ на запрос пользователя (рис. 19.3) или если в обозреватель загружены различные интерактивные визуальные элементы управления, то сервер является активным. Для публикации БД интерес представляет активный Web-сервер, реализуемый с помощью модулей расширения.

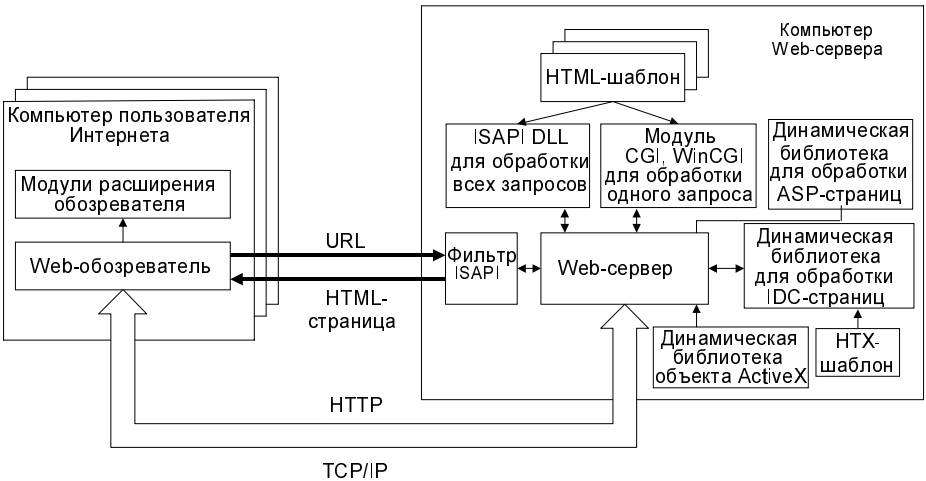


Рис. 19.3. Архитектура Web-приложения с модулями расширения сервера

Для создания модулей расширения Web-сервера используются интерфейсы CGI, WinCGI или API.

Интерфейс *CGI* является стандартным протоколом взаимодействия между Web-сервером и модулями расширения, которые могут применяться для выполнения дополнительных функций, не поддерживаемых сервером. Напомним, что интерфейсу CGI соответствуют обычные консольные приложения операционной системы DOS. Для написания CGI-программ подходит практически любой язык программирования, обеспечивающий доступ к переменным среды и ввод/вывод через стандартные потоки STDIN и STDOUT. В частности, для написания CGI-программ подходит среда Delphi.

Для запуска CGI-модуля обозреватель должен сформировать запрос к серверу с указанием адреса URL этого модуля. Это можно сделать следующими способами:

- задать URL модуля CGI в адресной строке обозревателя;
- послать серверу запрос на выполнение CGI-модуля путем выбора ссылки, атрибут `HREF` которой содержит адрес этого модуля;
- нажать в форме кнопку типа `Submit`, у которой атрибут `Action` содержит URL-адрес CGI-модуля.

После передачи запроса обозревателя CGI-приложению сервер передает ему также данные из командной строки запроса. CGI-приложение формирует ответ и помещает его в выходной поток (на стандартном устройстве вывода), затем сервер посылает этот ответ с использованием протокола HTTP обратно обозревателю.

В случае параллельной обработки нескольких запросов сервер запускает отдельный процесс для каждого запроса, причем для каждого процесса создается копия модуля расширения в памяти компьютера, на котором находится Web-сервер. При большом размере исполняемого CGI-файла сильно увеличивается время отклика сервера на запрос обозревателя, поскольку потребуется время для загрузки модуля CGI с диска в память. Причем, если CGI-программа является интерпретируемой (например, написана на языке PHP или Perl), она будет выполняться еще медленнее, т. к. в этом случае, кроме загрузки модуля CGI, загружается интерпретатор, производится интерпретация команд. При наличии сотен или тысяч обращений к серверу произойдет запуск сотен или тысяч CGI-программ, каждая из которых будет обрабатывать соответствующий запрос.

В некоторых операционных системах, в частности, в UNIX, могут повторно использоваться уже загруженные программные коды модуля CGI первого процесса с созданием для каждого нового обращения только своего экземпляра данных.

Интерфейс *WinCGI* (протокол) отличается от интерфейса CGI тем, что управляющие параметры передаются через INI-файл, а входной и выходной потоки данных перенаправлены в специальные файлы. Этот интерфейс является реализацией интерфейса CGI для операционной системы Windows 3.1.

Более перспективными интерфейсами для разработки дополнительных модулей расширения Web-сервера являются интерфейсы ISAPI/NSAPI. При использовании этих интерфейсов модули расширения реализуются в виде библиотек DLL. Такой механизм обеспечивает экономию ресурсов сервера и увеличение скорости обработки запросов.

Интерфейс ISAPI может применяться также для создания ISAPI-фильтров, которые, в отличие от модулей ISAPI, используются для контроля всего потока данных между сервером и обозревателем на уровне протокола HTTP. ISAPI-фильтры можно применять для динамической перекодировки, шифрования, сбора статистической информации о работе сервера.

Web-приложения с модулями расширения клиентской части

Для создания динамических эффектов при просмотре Web-страниц в модулях расширения клиентской части (активность на стороне клиента) используют апплеты, подключаемые программы, объекты ActiveX и сценарии.

Исполняемые программы являются первым поколением клиентских расширений. Для организации работы такой программы в Web-приложении необходимо ее предварительно установить и сконфигурировать обозреватель. Поэтому при

использовании исполняемых программ очень трудно обеспечить совместимость Web-приложения с различными обозревателями. Кроме того, использование такой технологии нарушает принцип стандартности клиентского интерфейса.

Апплеты Java применяются для создания динамически формируемого интерфейса пользователя. Апплет представляет собой байтовый код, который интерпретируется виртуальной Java-машиной, входящей в состав обозревателя. Загрузка апплета производится при загрузке Web-документа. Принцип функционирования апплета делает возможным подключение классов Java, хранящихся на сервере. Код апплета может начать выполняться сразу после загрузки в компьютер клиента или активизироваться с помощью специальных команд.

Используя библиотеку классов Java, можно создавать мультимедийные страницы и организовывать распределенные процедуры обработки данных с использованием различных серверов и протоколов.

Апплет может быть специализирован для работы с внешними базами данных. С этой целью в Java включены наборы классов для поддержки графического пользовательского интерфейса и классы для доступа к БД.

Для включения дополнительного действия в Web-приложение достаточно включить тег апплета в Web-документ и поместить апплет-класс в библиотеку апплетов на сервере. При этом изменения в конфигурацию Web-сервера вносить не нужно.

Для взаимодействия апплета Java с внешним сервером баз данных разработан специализированный протокол JDBC (Java DataBase Connectivity — совместимость Java с базами данных), построенный на принципах интерфейса ODBC и применяется для стандартизации кода апплета Java при организации доступа к различным СУБД.

Архитектура Web-приложений, публикующих БД

Для публикации БД на Web-страницах в архитектуру приложений вводятся дополнительные уровни, включающие сервер БД, сервер приложений и источник БД.

При такой архитектуре Web-сервер передает запрос на генерацию Web-страниц своей программе-расширению, которая на основе информации БД формирует требуемый документ. Затем Web-сервер отсылает готовые страницы обратно обозревателю. Как мы знаем, для формирования динамических страниц используются различные средства и технологии: ASP и IDC/HTX-страницы, программы-расширения сервера на основе интерфейсов CGI и ISAPI.

При использовании ASP и IDC/HTX-страниц запрос на получение динамически формируемой Web-страницы передается специальным динамическим библиотекам, входящим в состав Web-сервера. Например, если в качестве Web-сервера используется Personal Web Server и публикация осуществляется средствами IDC/HTX, то применяется динамическая библиотека httpodbc.dll. Такие библио-

теки анализируют файл ASP или IDC и HTX-файлы, которые используются в качестве шаблонов.

Напомним, что публикация БД на Web-страницах бывает статической и динамической. Программы-расширения Web-сервера, публикующие базы данных, включают в себя элементы, характерные для обычных приложений БД. При статической или динамической публикации должна использоваться архитектура приложений БД, дополненная характерными компонентами архитектуры Web-приложений. Рассмотрим подробнее архитектуру Web-приложений, использующих БД.

Двухуровневые Web-приложения

При двухуровневой архитектуре Web-приложений источник БД хранится на том же компьютере, где находится Web-сервер. Для доступа к источнику БД используются модули расширения. В простейшем случае в архитектуру Web-приложений добавляется источник БД.

Функционирование Web-приложения при такой архитектуре заключается в следующем. Обозреватель для начала работы с Web-приложением отсылает URL-адрес главной страницы приложения Web-серверу. Последний, обработав запрос URL, высылает требуемую страницу в формате HTML обратно обозревателю. Эта страница несет общую информацию о Web-приложении и позволяет пользователю выбрать нужную ему функцию из предоставляемых приложением. Далее возможны несколько вариантов работы Web-приложения.

Если пользователю нужна определенная информация из БД, то обозреватель по ссылке, находящейся в загруженной HTML-странице, формирует URL-запрос к модулю расширения сервера. Используемые при этом технологии различаются в зависимости от типа Web-сервера и других особенностей Web-приложения. Например, если на Web-узле установлен Web-сервер Microsoft Internet Information Server, то это может быть технология ASP- или IDC/HTX-страниц, интерфейсы CGI или ISAPI, а если установлен сервер Apache, то интерфейс CGI.

Если необходимо сформировать запрос URL с параметрами, то на уровне обозревателя могут использоваться сценарии JavaScript для проверки правильности ввода параметров запроса.

После того как пользователь выбрал ссылку, обозреватель отсылает URL Web-серверу. Для обработки запроса сервер вызывает требуемый модуль расширения и передает ему параметры URL. Модуль расширения сервера формирует SQL-запрос к БД.

Из модуля расширения сервера доступ к БД может осуществляться различными способами и на основе различных интерфейсов. Например, в случае использования технологии ASP-страниц применяются объектная модель ADO, объектный интерфейс OLE DB, интерфейс ODBC. Также возможен вариант непосредственного доступа к БД. Например, в случае модуля ISAPI, разработанного в среде Delphi, для доступа к БД может использоваться один посредник — драй-

вер BDE (Borland DataBase Engine), входящий в состав программных средств модуля расширения сервера.

Недостатки рассмотренной двухуровневой архитектуры: повышенная нагрузка на Web-сервер, поскольку все действия по обработке URL-запросов, извлечению информации из БД и формированию HTML-страниц выполняются Web-сервером и модулями расширения Web-сервера; низкий уровень безопасности из-за невозможности обеспечить требуемый уровень защиты информации в БД. Для преодоления указанных недостатков применяются Web-приложения с большим числом уровней.

Трехуровневые Web-приложения

При включении в Web-приложение промежуточного уровня, основанного на технологии "клиент-сервер", его архитектура расширяется до трехуровневой. При такой архитектуре клиентский уровень занимает обозреватель, на уровне сервера находится сервер БД, а на промежуточном уровне размещаются Web-сервер и модули расширения сервера. Модуль расширения сервера выступает преобразователем протоколов между клиент-серверным приложением БД и Web-сервером (рис. 19.4).

Введение уровня Web-сервера в клиент-серверные приложения БД расширяет возможность их применения как межплатформенного приложения. Принципы взаимосвязи обозревателя и Web-сервера остаются те же, что и в предыдущей архитектуре. Отличия этой архитектуры заключаются в организации взаимодействия модуля расширения сервера и источника БД.

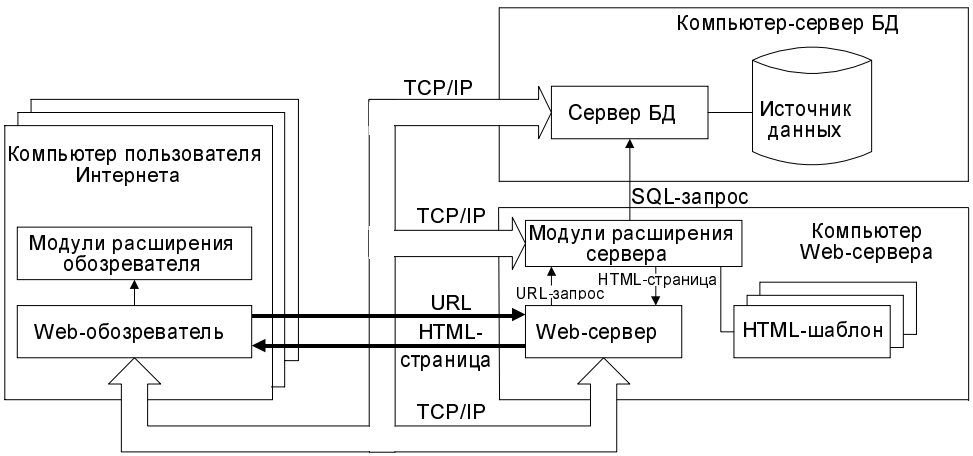


Рис. 19.4. Архитектура трехуровневого Web-приложения, использующего БД

Для получения данных модуль расширения Web-сервера формирует и отсылает SQL-запрос удаленному серверу БД. На компьютере, где установлен удаленный

сервер БД, содержится и сама база данных. После получения SQL-запроса удаленный сервер направляет его SQL-серверу (серверу баз данных). SQL-сервер обеспечивает выполнение запроса и выдачу модулю расширения Web-сервера результатов запроса.

Таким образом, в трехуровневой архитектуре вся обработка SQL-запроса выполняется на удаленном сервере. Достоинства такой архитектуры по сравнению с предыдущей состоят в следующем: уменьшение сетевого трафика — в сети циркулирует минимальный объем информации; увеличение уровня безопасности информации, поскольку обработка запросов к базе данных выполняется сервером БД; повышение устойчивости Web-приложения к сбоям; взаимозаменяемость компонентов архитектуры приложения; снижение сложности модулей расширения Web-сервера.

Недостатком рассмотренной архитектуры является увеличение времени обработки запросов, связанное с дополнительным обращением по сети к серверу БД. Для устранения этого недостатка между сервером БД и Web-сервером должны использоваться высокоскоростные надежные линии связи.

Дальнейшее развитие архитектуры Web-приложений и технологии "клиент-сервер" привело к появлению *многоуровневой архитектуры*, в которой между модулем расширения Web-сервера и базой данных, кроме сервера БД, дополнительно вводится сервер приложений. *Сервер приложений* является промежуточным уровнем, который обеспечивает организацию взаимодействия клиентов ("тонких клиентов") и сервера БД.

Отметим, что в некоторых архитектурах информационных систем Web-сервер может структурно объединяться с сервером приложений.

При функционировании Web-приложений с использованием многоуровневой архитектуры сохраняется возможность параллельной работы Web-обозревателей и клиентских приложений БД. В этом случае говорят о *смешанном* Web-приложении.

Интерфейсы Web-приложений

В следующих разделах описаны общепринятые интерфейсы программирования Web-приложений — общий интерфейс взаимодействия CGI и интерфейс программирования серверных приложений ISAPI.

Общий интерфейс взаимодействия CGI

CGI-модуль представляет собой консольный исполняемый файл, загружаемый сервером для обработки данных обозревателя и генерации HTML-документа. Обмен данными между сервером и CGI-модулем происходит через стандартные входной и выходной потоки. Кроме того, для передачи параметров и получения системной информации могут задействоваться переменные окружения. При ис-

пользовании CGI-интерфейса (протокола) можно формировать HTML-документ динамически (в ответ на запрос пользователя из обозревателя).

CGI-модуль может вызываться из обозревателя для динамического формирования HTML-документов на основании данных, введенных посетителями Web-сервера при помощи форм или путем выбора ими ссылок. Напомним, что обработку данных, введенных пользователем в интерфейсные элементы формы, можно выполнить с помощью обработчика, URL которого задается атрибутом `ACTION` тега формы, а способ передачи данных из формы на Web-сервер определяет атрибут `METHOD` тега формы.

При разработке CGI-модуля необходимо учитывать, каким методом HTTP-протокола будут передаваться данные между обозревателем и сервером. Основными методами передачи параметров запроса являются `POST` и `GET`. Метод `POST` применяется для посылки информации, включенной в HTTP-запрос и относящейся к ресурсу, указанному URL-адресом CGI-модуля. Метод `GET` используется для передачи информации в URL-адресе CGI-модуля. В зависимости от метода, примененного в HTTP-запросе, сервер формирует передаваемые для CGI-модуля параметры различными способами. В случае метода `GET` для получения параметров запроса привлекаются переменные окружения (переменные операционной системы MS-DOS, устанавливаемые командой `SET`), если же задействован метод `POST`, данные доставляются через стандартный поток ввода (рис. 19.5).

Например:

```
<HTML><HEAD><TITLE>Пример CGI-модуля</TITLE>
</HEAD>
<BODY>
<A HREF="http://localhost/scripts/my_cgi.exe?TEST=modul">
Загрузить CGI-модуль по запросу
http://localhost/Scripts/my_cgi.exe?TEST=modul</A>
<FORM ACTION=http://localhost/scripts/my_cgi.exe METHOD=POST>
<B>Введите ваше имя:</B>
<INPUT MAXLENGTH=60 NAME=NAME SIZE=40 VALUE="My_login">
<BR>Тип действия: <INPUT CHECKED NAME=TYPE TYPE=radio
                            VALUE=Registry>- Зарегистрировать
<INPUT NAME=TYPE TYPE=radio VALUE=Execute>Выполнить сценарий
<BR><INPUT NAME=Submit TYPE=Submit VALUE=Послать запрос>
<BR><INPUT NAME=Reset TYPE=Reset VALUE=Сброс><BR>
</FORM></P></BODY></HTML>
```

В окне обозревателя будут отображены ссылка (`Загрузить CGI-модуль по запросу http://localhost/Scripts/my_cgi.exe?TEST=modul`) и элементы управления, входящие в состав формы. При нажатии специальной кнопки **Послать запрос**, имеющей тип `Submit`, обозреватель отправляет запрос Web-серверу. Последний, в свою очередь, запускает CGI-модуль, находящийся по URL-адресу **http://localhost/Scripts/my_cgi.exe**. При этом данные, введенные пользователем в интерфейсные элементы формы, переносятся через стандартный поток ввода.

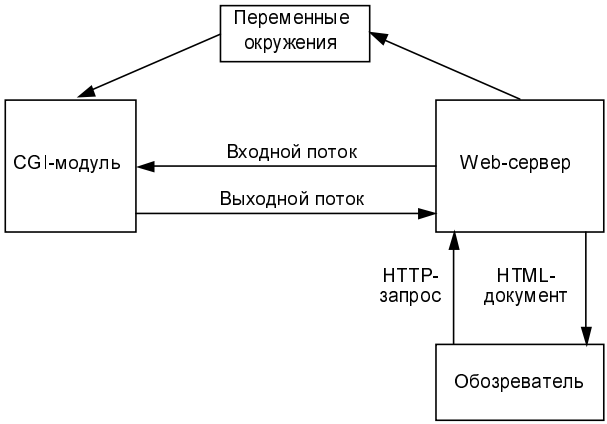


Рис. 19.5. Схема взаимодействия сервера и CGI-модуля

При выборе пользователем ссылки `Загрузить CGI-модуль по запросу` `http://localhost/Scripts/my_cgi.exe?TEST=modul` обозреватель передает запрос Web-серверу на загрузку CGI-модуля, расположенного по URL-адресу **http://localhost/Scripts/my_cgi.exe**. При этом для посылки запроса используется метод `GET`, и сервер отправляет CGI-модулю только данные, находящиеся в командной строке (`TEST=modul`) через переменную окружения `QUERY_STRING`.

Метод `GET` может использоваться и для передачи данных формы. В этом случае в теге формы указывается `METOD=GET`. При использовании метода `GET` данные размещаются в переменной окружения `QUERY_STRING` в следующем формате:

```
"Имя1=Значение1&Имя2=Значение2&Имя3=Значение3"
```

Здесь `Имя1`, `Имя1`, `Имя3` — значения параметров `NAME`, задающих имена полей формы. При этом количество параметров определяется, исходя из числа элементов формы (в данном случае таких элементов три). На место значений параметров подставляются данные из соответствующих полей.

Отметим, что не все значения интерфейсных элементов формы передаются на сервер. Например, значение служебного элемента типа `reset`

```
<INPUT NAME=Reset TYPE=reset VALUE=Сброс>
```

не передается на сервер, т. к. оно не несет полезной информации.

Если в приведенном выше примере HTML-документа параметр `METHOD` сопоставить с методом `GET`, то переменная `QUERY_STRING` примет значение

```
NAME=My_login&TYPE=Registry&Submit=Send
```

Это значение будет закодировано, т. е. символы пробела будут замещены символами `+`, а специальные и управляющие коды (в случае их использования) будут заменены на последовательности символов вида `Xxx`, где `xx` — шестнадцатеричное представление символа.

Если в имена или в поля интерфейсных элементов входят символы, набранные в альтернативной кодировке, то при их передаче сервер будет также подставлять вместо каждого такого символа шестнадцатеричный код, предваряемый символом %.

Если в приведенном выше примере первый интерфейсный элемент оформить как <INPUT MAXLENGTH=60 NAME=ИМЯ SIZE=40 VALUE="МОЕ ИМЯ">, то переменная QUERY_STRING будет иметь следующее значение:

```
%C8%CC%DF=%CC%CE%C5+%C8%CC%DF&TYPE=Registry&Submit=Send
```

В результате CGI-модуль генерирует HTML-документ, который помещается в стандартный поток вывода. Web-сервер задействует для управления и передачи различной информации CGI-модулю многочисленные переменные окружения.

Переменные окружения

Переменные окружения применяются в CGI-модуле для получения от Web-сервера служебной информации о самой программе Web-сервера, параметрах HTTP-запроса и другой информации, передаваемой сервером модулю. Например, в случае применения метода GET при формировании HTTP-запроса в переменную QUERY_STRING помещаются передаваемые пользовательские данные, а при альтернативном подходе (метод POST) в переменных окружения CONTENT_TYPE и CONTENT_LENGTH содержатся тип и длина передаваемой информации соответственно. Сами данные в последнем случае доставляются через стандартный входной поток.

Переменные окружения могут включать, к примеру, следующие данные:

- SERVER_NAME — символическое имя или IP-адрес компьютера, на котором запущен Web-сервер (задается в URL при обращении к этому Web-серверу);
- GATEWAY_INERFACE — версия CGI-интерфейса;
- SERVER_PORT — номер порта, на который обозреватель посылает запросы Web-серверу;
- QUERY_STRING — строка символов, следующая за знаком ? в URL данного запроса;
- REMOTE_IDENT — имя пользователя, полученное от сервера (если сервер поддерживает аутентификацию);
- HTTP_REFER — URL документа HTML, из которого осуществляется вызов CGI-модуля;
- HTTP_CONNECTION — тип соединения;
- HTTP_HOST — имя узла, на котором работает Web-сервер;
- HTTP_ACCEPT_ENCODING — тип схемы кодирования, используемой обозревателем для формирования запроса Web-серверу;
- HTTP_FROM — имя пользователя, установленное в настройках обозревателя.

Замечание

Тип MIME (Multipurpose Internet Mail Extensions, многоцелевые расширения почтового стандарта Интернета) определяет протокол передачи почтовых сообщений, используемый взамен стандартного.

Для Web-сервера Microsoft PWS 95/2.0 в среде Windows 98 и адреса

```
HREF="http://igin/scripts/my_cgi.exe?TEST=modul"
```

задаваемого в адресной строке обозревателя, переменные окружения будут иметь следующие значения:

```
SERVER_SOFTWARE = Microsoft-PWS-95/2.0
SERVER_NAME = igin
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = POST
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
PATH_TRANSLATED = C:\WebShare\wwwroot\
SCRIPT_NAME = /scripts/my_cgi.exe
QUERY_STRING = TEST=modul
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
CONTENT_TYPE = application/x-www-form-urlencoded
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
HTTP_USER_AGENT = Mozilla/4.7 [en] (Win98; I)
HTTP_ACCEPT_LANGUAGE = en,ru
HTTP_CONNECTION = Keep-Alive
HTTP_HOST = igin
HTTP_ACCEPT_ENCODING = gzip
```

Остальные переменные принимают нулевое значение (`Null`).

Здесь: `igin` — имя компьютера, на котором выполняется CGI-модуль; `C:\WebShare\wwwroot\` — каталог, в котором находится CGI-модуль; `my_cgi.exe` — имя файла, содержащего модуль.

Для Web-сервера MS IIS 5.0 в среде Windows 2000 Server и адреса, задаваемого в адресной строке обозревателя:

```
http://igin/Scripts/my_cgi.exe
```

следующие переменные окружения могут иметь значения, отличные от представленных выше:

```
SERVER_SOFTWARE = Microsoft-IIS/5.0
SERVER_NAME = igin
```

```
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.1
PATH_TRANSLATED = c:\inetpub\wwwroot
SCRIPT_NAME = /Scripts/my_cgi.exe
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
```

Стандартный вывод

После перекодировки входных данных в CGI-модуле необходимо сформировать HTML-документ для передачи его на стандартное устройство вывода. При этом HTML-страница динамически генерируется на основании информации, хранимой в БД, которая может обновляться в реальном масштабе времени.

Интерфейс программирования серверных приложений ISAPI

Перспективным направлением, в котором развиваются технологии создания моделей расширения сервера, является интерфейс (протокол) программирования серверных приложений ISAPI. Этот интерфейс разработан для преодоления ограничений и недостатков интерфейса CGI. В отличие от CGI, интерфейс ISAPI предусматривает реализацию модулей расширения сервера в виде *динамической библиотеки.* При этом для обработки каждого запроса обозревателя к модулю расширения сервера ISAPI не создается отдельный процесс, а вызываются функции модуля ISAPI, которые один раз загружаются в память при первом обращении.

Есть несколько вариантов реализации протоколов серверных расширений. Наибольшее распространение получили два из них — NSAPI компании Netscape и ISAPI фирмы Microsoft. Из-за ограниченности объема книги мы рассмотрим только возможности протокола ISAPI.

Приложения, построенные с использованием протокола ISAPI, основаны на аналогичных с CGI-модулями принципах взаимодействия с сервером. Отличия указанных протоколов заключаются в осуществлении обмена данными между модулем и сервером.

Модули расширения ISAPI, как и модули CGI, принимают от сервера данные, которые передает модулю обозреватель, обрабатывают эти данные и динамически формируют HTML-документ. Однако вместо чтения содержимого переменных окружения и стандартного потока ввода модуль расширения ISAPI получает данные при помощи специально предназначенных для этого функций. Аналогично вместо записи выходных данных в стандартный поток вывода модуль расширения ISAPI также вызывает специальные функции.

Модули ISAPI в зависимости от назначения подразделяются на две группы: обычные модули расширения ISAPI и фильтры ISAPI. Первая группа предназначена, как и CGI-модули расширения сервера, для динамического формиро-

вания HTML-документов. Фильтры ISAPI обеспечивают выполнение нестандартных задач управления всеми данными, проходящими через сервер (например, кодирование данных, сбор статистической информации об использовании ресурсов сервера и т. д.).

Рассмотрим основные принципы создания и работы модулей расширений ISAPI. Модуль ISAPI реализуется в виде библиотеки DLL. Эта библиотека загружается в одно адресное пространство с Web-сервером при первом обращении к ней из обозревателя. Причем вызов ISAPI-модуля осуществляется подобно вызовам CGI-модулей расширения — из форм или ссылок. ISAPI-модуль исполняется в рамках единого адресного пространства программных ресурсов сервера. Такая организация ISAPI-модуля позволяет значительно повысить производительность сервера в целом по сравнению с сервером, использующим CGI-модули.

Преимущество ISAPI-интерфейса становится особенно заметным при одновременном активном использовании модуля расширения несколькими пользователями.

Отметим, что CGI-модули являются более надежным средством с точки зрения устойчивости Web-сервера к сбоям. Так, при возникновении ошибки в ISAPI-модуле может произойти аварийное завершение работы всего Web-сервера, тогда как в случае возникновения ошибки в CGI-модуле будет завершен только отдельный процесс. По упомянутой причине особенности отладки ISAPI-модуля состоят в том, что для перезапуска ISAPI-модуля требуется его переименовать или перезагрузить Web-сервер.

Рассмотрим структуру модулей ISAPI. Библиотека DLL, представляющая собой ISAPI-модуль, экспортирует как минимум две функции: `GetExtensionVersion` и `HttpExtensionProc`. Кроме этих функций, ISAPI-модуль может использовать функцию `TerminateExtension`, которой передается управление перед выгрузкой ISAPI-модуля из памяти. Эта функция освобождает ресурсы, выделенные ISAPI-модулю.

Функция `GetExtensionVersion(var Ver: THSE_VERSION_INFO): BOOL` предназначена для того, чтобы ISAPI-модуль мог сообщить серверу версию спецификации ISAPI-интерфейса и строку описания ISAPI-модуля. Параметр-переменная `Ver` указывает управляющую структуру, тип которой описан так:

```
type
  THSE_VERSION_INFO = packed record
    // Версия спецификации интерфейса ISAPI
    dwExtensionVersion: DWORD;
    // Строка описания модуля ISAPI
    lpszExtensionDesc: array [0..HSE_MAX_EXT_DLL_NAME_LEN-1] of Char;
  end;
```

Функция `HttpExtensionProc(var ECB: TEXTENSION_CONTROL_BLOCK): DWORD` обеспечивает основную функциональность ISAPI-модуля и используется для

обмена данными между ISAPI-модулем и сервером. Параметр-переменная `ECB` указывает управляющую структуру, тип которой описан так:

```
type
  TEXTENSION_CONTROL_BLOCK = packed record
    // Размер блока в байтах
    cbSize: DWORD;
    // Версия спецификации интерфейса ISAPI
    dwVersion: DWORD;
    // Идентификатор соединения
    ConnID: HCONN;
    // Код состояния HTTP
    dwHttpStatusCode: DWORD;
    // Данные протоколирования
    lpszLogData: array [0..HSE_LOG_BUFFER_LEN - 1] of Char;
    // Переменная окружения REQUEST_METHOD
    lpszMethod: PChar;
    // Переменная окружения QUERY_STRING
    lpszQueryString: PChar;
    // Переменная окружения PATH_INFO
    lpszPathInfo: PChar;
    // Переменная окружения PATH_TRANSLATED
    lpszPathTranslated: PChar;
    // Полный размер данных, полученных от обозревателя
    cbTotalBytes: DWORD;
    // Размер доступного блока данных
    cbAvailable: DWORD;
    // Указатель на доступный блок данных
    lpbData: Pointer;
    // Тип полученных данных
    lpszContentType: PChar;
    // Функция для получения значений переменных окружения
    GetServerVariable: TGetServerVariableProc;
    // Функция для посылки данных обозревателю
    WriteClient: TWriteClientProc;
    // Функция для получения данных от посетителя
    ReadClient: TReadClientProc;
    // Служебная функция
    ServerSupportFunction: TServerSupportFunctionProc;
  end;
```

Поясним особенности использования основных полей этой структуры данных.

Поле `cbTotalBytes` предназначено для записи суммарного количества байтов данных, принимаемых от обозревателя. Блок данных размером не более

48 Кбайт считывается сервером автоматически. Эти данные становятся доступны при вызове функции `HttpExtensionProc`. Но полностью данные дочитываются в цикле при помощи функции `ReadClient`.

Поле `cbAvailable` служит для сохранения длины блока данных, полученных автоматически от обозревателя (не более 48 Кбайт).

Поле `LpbData` содержит указатель на область памяти, в которой размещен полученный сервером от обозревателя блок данных размером `cbAvailable` байтов.

В управляющий блок помимо данных включены следующие указатели на методы.

Поле `GetServerVariable` содержит указатель на функцию, средствами которой ISAPI-модуль определяет значения переменных среды окружения.

В поле `WriteClient` находится указатель на метод, используемый для отправки данных обозревателю, в отличие от интерфейса CGI, в случае применения которого модуль расширения помещает результат в стандартный поток вывода.

Поле `ReadClient` хранит указатель на функцию, предназначенную для считывания данных в буфер предварительного чтения, имеющий адрес `LpbData` и размер, не превышающий 48 Кбайт.

В поле `ServerSupportFunction` содержится указатель на функцию, служащую для пересылки стандартного заголовка протокола HTTP и осуществления других действий.

Глава 20

Публикация баз данных средствами Delphi

Система Delphi предоставляет программисту мощные средства для разработки сложных Web-приложений. Специальные компоненты Delphi позволяют помещать информацию из БД непосредственно на HTML-страницы. С помощью Delphi можно создавать Web-приложения на основе интерфейсов CGI, WinCGI, ISAPI и NSAPI. В этой главе рассматривается использование интерфейсов CGI и ISAPI как наиболее часто встречающихся.

Компоненты, используемые при разработке Web-приложений

При разработке Web-приложений используются компоненты Палитры компонентов, расположенные на странице **Internet** (рис. 20.1). Рассмотрим назначение этих компонентов.

Рис. 20.1. Страница **Internet** Палитры компонентов

- `WebDispatcher` (Web-диспетчер) предназначен для создания Web-модуля данных и позволяет модулю расширения Web-сервера обрабатывать HTTP-запросы.
- `PageProducer` (генератор страниц) предназначен для преобразования HTML-шаблона в HTML-документ. При этом в HTML-шаблоне производится замена специальных тегов на данные, которые генерируются динамически при выполнении программы с помощью обработки события `OnHTMLTag`.
- `DataSetTableProducer` (генератор таблицы набора данных) предназначен для генерации HTML-документа, который содержит таблицу из данных, полу-

ченных в результате обработки запроса к БД на получение всех записей. Наследует свойства объекта `TDataSet`.

- `DataSetPageProducer` (генератор страницы набора данных) предназначен для преобразования HTML-шаблона, содержащего ссылки на поля БД, вместо которых динамически при выполнении приложения подставляются значения из текущей записи БД.
- `QueryTableProducer` (генератор таблицы по запросу) предназначен для генерации HTML-документа, который содержит таблицу из данных, полученных в результате обработки запроса к БД.
- `SQLQueryTableProducer` (генератор таблицы по SQL-запросу) предназначен для генерации HTML-документа, который содержит таблицу из данных, полученных в результате обработки SQL-запроса к БД.
- `TcpClient` (клиент TCP) добавляет объект типа `TTcpClient` к форме или модулю данных для преобразования приложения в TCP/IP-клиента; определяет требуемое соединение с сервером TCP/IP, управляет открытым соединением и завершает законченное соединение.
- `TcpServer` (сервер TCP) добавляет объект типа `TTcpServer` к форме или модулю данных для преобразования приложения в сервер TCP/IP; ожидает запросы на TCP/IP-соединения от других машин и устанавливает соединения при получении запросов.
- `UdpSocket` (сокет UDP) добавляет объект типа `TUdpSocket` к форме или модулю данных для преобразования приложения в UDP/IP-клиент или UDP/IP-сервер.
- `XMLDocument` (документ XML). Класс `TXMLDocument` может использоваться непосредственно для доступа к узлам (элементам) документа с помощью интерфейса IXMLNode или применяться совместно с классами и интерфейсами, генерируемыми Мастером XML Data Binding.
- `WebBrowser` (Web-обозреватель) создает собственный обозреватель для отображения HTML-страниц. Для использования этого компонента требуется Microsoft Internet Explorer версии не ниже 4.0.

Замечание

UDP (User Datagram Protocol) представляет собой протокол передачи прикладных пакетов дейтаграммным методом. Подобно TCP, использует протокол IP для доставки данных, но, в отличие от TCP, обеспечивает обмен дейтаграммами без подтверждения.

В справке указано назначение еще двух компонентов: `ClientSocket` (сокет клиента, служит для создания приложений TCP/IP-клиента, может быть добавлен к форме или модулю данных) и `ServerSocket` (сокет сервера, служит для создания приложений TCP/IP-сервера, может быть добавлен к форме или модулю данных), которых на странице **Internet** Палитры в действительности нет.

При разработке Web-приложений наиболее часто используются компоненты, предназначенные для генерации HTML-документов (`PageProducer`, `QueryTableProducer`,

DataSetTableProducer, DataSetPageProducer) и для создания модулей расширения Web-сервера (WebDispatcher). Более подробно особенности их применения будут рассмотрены ниже.

Статическая публикация

Как отмечалось, статическая публикация базы данных в Интернете используется при редком обновлении информации в БД. При использовании такого типа публикации информация из БД помещается в HTML-документ, который хранится постоянно в файле на диске. При этом запрос на получение данных не приводит к большой загрузке сервера, т. к. пользователю возвращается уже готовый HTML-документ.

Для статической публикации можно использовать обычное приложение Windows, которое запускают периодически (после обновления информации в БД). Вообще говоря, для такой публикации достаточно создать простейшее приложение БД, которое обеспечивает:

- доступ к БД;
- генерацию HTML-документа на основе информации, получаемой из БД;
- сохранение созданного файла на диске.

Рассмотрим особенности использования средств Delphi для решения этих задач на следующем примере.

Пусть требуется выполнить статическую публикацию содержимого демонстрационной БД, входящей в состав стандартной поставки Delphi. Эта БД имеет псевдоним DBDEMOS и имя файла таблицы Animals.dbf (при полной установке Delphi псевдоним создается автоматически).

Разместим в форме 2 кнопки Button и установим с помощью Инспектора объектов их свойства Name в значения Gen и Show соответственно. Кроме кнопок, поместим в форму компонент Table. Ниже приведен код модуля DbHTML генератора HTML-страниц.

```
unit DbHTML;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, DBTables, StdCtrls, ComCtrls;
type
  TDbHTMLForm = class(TForm)
    Table1: TTable;
       Gen: TButton;
      Show: TButton;
    procedure GenClick(Sender: TObject);
    procedure ShowClick(Sender: TObject);
  end;
```

```
var
  DbHTMLForm: TDbHTMLForm;
implementation

{$R *.DFM}

uses
  Shellapi, FileCtrl;

// Обработчик события OnClick кнопки Gen
procedure TDbHTMLForm.GenClick(Sender: TObject);
var HTMLStr: TStringList;
          I: Integer;
begin
// Инициализация объекта Table1
Table1.Close;
Table1.DatabaseName := 'DBDEMOS';
Table1.TableName := 'Animals.dbf';
Table1.Open;

// Создание заголовка HTML-документа
HTMLStr := TStringList.Create;
HtmlStr.Clear;
HtmlStr.Add ('<HTML>');
HtmlStr.Add ('<HEAD>');
HtmlStr.Add ('<TITLE>' + 'HTML-страница, отчет из БД' +
             '</TITLE>');
HtmlStr.Add ('</HEAD>');
HtmlStr.Add ('<BODY BGCOLOR="#FFFFEE">');
HtmlStr.Add ('<H1><CENTER> Данные из файла ' + Table1.TableName +
             '</CENTER></H1>');

// Вывод тега, открывающего таблицу
HtmlStr.Add('<TABLE BORDER>');

// Вывод тега, открывающего строку таблицы
HtmlStr.Add('<TR>');
// Создание заголовков таблицы
for I := 0 to Table1.FieldCount - 1 do
    HtmlStr.Add('<TH>' + Table1.Fields[I].FieldName + '</TH>');
// Вывод тега, закрывающего строку таблицы
HtmlStr.Add('</TR>');
```

```
// Вывод данных
Table1.First;
// Перебор всех записей
while not Table1.EOF do begin
      HtmlStr.Add('<TR>');
      for I := 0 to Table1.FieldCount - 1 do
          // Вывод тега ячейки данных таблицы
          if Table1.Fields[I].DisplayText = ''
             then HtmlStr.Add('<TD>' + '____'+ '</TD>')
             else HtmlStr.Add('<TD>' + Table1.Fields[I].DisplayText +
                  '</TD>');
      HtmlStr.Add('</TR>');
      Table1.Next;
      end;

// Вывод тега, закрывающего таблицу
HtmlStr.Add('</TABLE>');
// Вывод тега, закрывающего тело документа
HtmlStr.Add ('</BODY>');
// Вывод тега, закрывающего документ
HtmlStr.Add ('</HTML>');

// Сохранение сформированного HTML-документа в файле Htmltest.htm
HtmlStr.SaveToFile ('Htmltest.htm');
HtmlStr.Free;
end;

// Обработчик события OnClick кнопки Show
procedure TDbHTMLForm.ShowClick(Sender: TObject);
begin
// Запуск просмотра файла HTMLTEST.htm
ShellExecute (Handle, 'open', pChar('HTMLTEST.htm'), '', '', sw_ShowNormal);
end;

end.
```

В приведенном примере HTML-страницы генерируются без использования специальных средств библиотеки VCL. Формирование и сохранение в файле HTML-страниц задается в явном виде. Интерфейсная часть приложения представляет собой окно с двумя кнопками: `Gen` и `Show`.

Рассмотрим подробнее, как происходит обработка события нажатия кнопки.

В процедуре `GenClick` используется объект `HTMLStr` типа `TStringList`, в котором хранятся строки формируемого HTML-документа. В начале процедуры выполняется инициализация объекта `Table1` с помощью следующего кода:

```
Table1.Close;
Table1.DatabaseName := 'DBDEMOS';
Table1.TableName := 'Animals.dbf';
Table1.Open;
```

Здесь для объекта `Table1` устанавливаются значения свойств, задающих источник БД. Свойство `DatabaseName` определяет псевдоним (`DBDEMOS`), а свойство `TableName` задает таблицу БД (в рассматриваемом случае используемая таблица находится в файле Animals.dbf). Методы `Close` и `Open` соответственно закрывают и устанавливают соединение с БД. При этом вызов метода `Close` перед инициализацией объекта `Table1` обеспечивает успешное выполнение кода при повторном обращении к процедуре `GenClick`.

Далее в процедуре осуществляются создание объекта `StringList` с помощью конструктора `Create`:

```
HTMLStr := TStringList.Create;
```

и его инициализация с помощью метода `Clear`:

```
HtmlStr.Clear;
```

Формирование HTML-документа выполняется с помощью метода `Add`:

```
HtmlStr.Add ('<HTML>');
```

Информация из БД выводится в формируемый HTML-документ в табличном виде. В частности, заголовок таблицы выводится с помощью следующего кода:

```
HtmlStr.Add('<TABLE BORDER>');
HtmlStr.Add('<TR>');
for I := 0 to Table1.FieldCount - 1 do
    HtmlStr.Add('<TH>' + Table1.Fields[I].FieldName + '</TH>');

HtmlStr.Add('</TR>');
```

Свойство `FieldCount` задает число полей в записи таблицы. Строка `Table1.Fields[I].FieldName` возвращает указатель на строку, содержащую имя поля. При этом объект `Fields` является массивом полей записи. Выражение `Fields[I]` позволяет обратиться к полю записи с номером `I` (нумерация с нуля).

Далее в обработчике стоит итерационный цикл, содержащий вложенный цикл с параметром:

```
Table1.First;
while not Table1.EOF do begin
      HtmlStr.Add('<TR>');
      for I := 0 to Table1.FieldCount - 1 do
          if Table1.Fields[I].DisplayText = ''
```

```
            then HtmlStr.Add('<TD>' + '____'+ '</TD>')
            else HtmlStr.Add('<TD>' + Table1.Fields[I].DisplayText +
                 '</TD>');
      HtmlStr.Add('</TR>');
      Table1.Next;
      end;
```

Перед началом цикла с помощью метода `First` указатель текущей записи устанавливается на первую запись таблицы. Во внешнем цикле указатель текущей записи последовательно перемещается по записям таблицы, а во внутреннем цикле осуществляется перебор всех полей текущей записи. Выход из внешнего цикла происходит при достижении указателем текущей записи конца таблицы (значение выражения `not Table1.EOF` становится ложным).

Метод `Table1.Fields[I].DisplayText` возвращает указатель на строку, содержащую данные из `I`-го поля текущей записи. Чтобы обозреватель не нарушал структуру таблицы, при отсутствии в поле данных в формируемый документ выводится строка `'<TD>' + '____'+ '</TD>'` (если данные есть, то выводится `'<TD>' + Table1.Fields[I].DisplayText + '</TD>'`).

Последняя строка в теле цикла содержит вызов метода `Next`, который переводит указатель текущей записи на следующую запись в таблице.

Сохранение сформированного HTML-документа в файле Htmltest.htm на диске выполняется с помощью метода `SaveToFile` объекта `StringList`:

```
HtmlStr.SaveToFile ('Htmltest.htm');
```

NAME	SIZE	WEIGHT	AREA	BMP
Angel Fish	2	2	Computer Aquariums	(TYPEDBINARY)
Boa	10	8	South America	(TYPEDBINARY)
Critters	30	20	Screen Savers	(TYPEDBINARY)
House Cat	10	5	New Orleans	(TYPEDBINARY)
Ocelot	40	35	Africa and Asia	(TYPEDBINARY)
Parrot	5	5	South America	(TYPEDBINARY)
Tetras	2	2	Fish Bowls	(TYPEDBINARY)

Рис. 20.2. Вид сгенерированного приложением HTML-документа в окне обозревателя

В процедуре `ShowClick` вызывается API-функция Windows `ShellExecute`:

```
ShellExecute (Handle, 'open', pChar ('Htmltest.htm'), '', '',
   sw_ShowNormal);
```

которая загружает файл Htmltest.htm в обозреватель.

Вид сформированного приложением HTML-документа в окне обозревателя показан на рис. 20.2. Как видно из рисунка, данные в HTML-документе представлены в виде таблицы, столбцы которой соответствуют полям, а строки — записям БД.

Компоненты генерации HTML-страниц

В следующих разделах рассматриваются специальные компоненты `PageProducer`, `DataSetPageProducer`, `DataSetTableProducer` и `QueryTableProducer`, предназначенные для эффективной работы с HTML-страницами в Delphi, а также пример генератора HTML-страниц.

Компонент *PageProducer*

Компонент `PageProducer` генерирует HTML-страницы на основе специального HTML-шаблона, связанного с этим компонентом. *HTML-шаблон* представляет собой последовательность HTML-команд и специальных тегов-транспарантов, имеющих следующий формат:

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

Содержимое шаблона может формироваться заранее (на этапе проектирования) или динамически — перед генерацией HTML-страницы.

Для доступа к шаблону используется свойство `HTMLDoc` типа `TStrings` или `HTMLFile` типа `TFileName`. Свойства `HTMLDoc` и `HTMLFile` являются взаимоисключающими — при выборе одного из них другое свойство обнуляется.

При вызове метода `Content` производится синтаксический анализ HTML-строк. При обработке подлежащих замене тегов возникает событие `OnHTMLTag` типа `THTMLTagEvent`, в обработчике которого производится замена всех тегов-транспарантов на необходимые текстовые значения. Кроме того, метод `Content` возвращает указатель на буфер, в котором находится сформированный документ. Например:

```
Memo1.Text := PageProducer1.Content;
```

Свойство `Dispatcher` типа `TCustomWebDispatcher` компонента `PageProducer` служит для связи с Web-сервером. Оно содержит ссылку на Web-модуль `WebModule` или `WebDispatcher`, в функции которых входит обслуживание текущего запроса. Особенности работы с этим свойством рассматриваются ниже.

Компонент *DataSetPageProducer*

Компонент `DataSetPageProducer` аналогичен предыдущему компоненту, но дополнительно содержит свойство `DataSet` типа `TdataSet`, в котором указывается используемый набор данных. При применении этого компонента можно в тегах-транспарантах HTML-шаблона (связанного с компонентом `DataSetPageProducer` через свойства `HTMLDoc` или `HTMLFile`) в качестве имени тега (`TagName`) указывать имена полей в записи БД. Тем самым обеспечивается отображение значений полей текущей записи в определяемом свойством `DataSet` наборе данных. Например:

```
Имя:    <#name>   <BR>
Размер: <#size>   <BR>
Вес:    <#weight> <BR>
Место:  <#area>   <BR>
```

Здесь `name`, `size`, `weight`, `area` — имена полей в записи БД.

Следующий фрагмент кода формирует в редакторе `Memo1` HTML-документ по данным первой записи из БД:

```
Table1.First;
Memo1.Clear;
Memo1.Text := DataSetPageProducer1.Content;
```

Инструкция `Table1.First` устанавливает указатель текущей записи на первую запись набора данных, инструкция `Memo1.Clear` очищает все строки редактора `Memo1`, после чего он заполняется содержимым сформированного HTML-документа с помощью метода `Content` объекта `DataSetPageProducer1`.

Компонент *DataSetTableProducer*

Компонент `DataSetTableProducer` позволяет выводить содержимое БД в табличном виде. Рассмотрим наиболее часто используемые свойства и методы этого компонента.

Свойство `Header` типа `TStrings` задает заголовок HTML-документа, а свойство `Footer` типа `TStrings` задает строки, заканчивающие HTML-документ.

Свойство `DataSet` определяет набор данных, на основании которого формируется HTML-документ.

Свойство `RowAttributes` типа `THTMLTableRowAttributes` позволяет отдельно форматировать строки таблицы, а свойство `TableAttributes` типа `THTMLTableAttributes` служит для форматирования всей таблицы.

Метод `Content` компонента `DataSetTableProducer`, как и одноименный метод перечисленных выше компонентов, предназначен для генерации документа. Однако в отличие от предыдущих компонентов, событие `OnHTMLTag` в компоненте `DataSetTableProducer` не генерируется. Кроме того, при форматировании ячей-

ки таблицы возникает событие OnFormatCell типа THTMLFormatCellEvent, обработчик которого позволяет индивидуально форматировать каждую ячейку. Тип события описан следующим образом:

```
type THTMLFormatCellEvent =
  procedure THTMLForm.DataSetTableProducer1FormatCell(Sender: TObject;
      CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
      CellData: String);
```

Здесь параметры CellRow и CellColumn содержат номера редактируемой строки и столбцы таблицы соответственно, при этом параметр CellRow имеет значение 0 при формировании заголовка таблицы, а нулевое значение CellColumn соответствует формированию первого столбца таблицы.

Параметры BgColor, Align и VAlign задают цвет фона ячейки и горизонтальное и вертикальное выравнивание. Поскольку они входят в процедуру с описателем var, их значения можно изменить программно. Например:

```
if (CellRow = 0) then BgColor:='gray'
   else if (CellRow > 5) BgColor:='silver';
```

Допускается присваивать параметру Align значения haLeft, haRight, haCenter, что обеспечивает выравнивание влево, вправо и по центру ячейки соответственно.

Параметр VAlign может принимать значения haTop, haMiddle, haBottom, haBaseline (выравнивание по верху, посередине, по низу и по базовой линии в ячейке соответственно).

Параметр CellData содержит выводимые в ячейке данные. Этот параметр можно программно задавать для каждой ячейки. Например:

```
if (CellColumn > 2) then
   CellData := '<FONT size=6>' + CellData + '</FONT>';
```

Свойство Columns компонента DataSetTableProducer связано с Редактором столбцов. При двойном щелчке мыши в Инспекторе объектов на свойстве Columns или на значке компонента DataSetTableProducer в рабочей форме открывается окно Редактора столбцов (рис. 20.3). Отметим, что Редактор столбцов не имеет ничего общего с Редактором, используемым для настройки столбцов компонента DBGrid.

Загрузить Редактор столбцов можно также выбрав пункт **Response Editor** контекстного меню компонента DataSetTableProducer.

Замечание

Информация из БД становится доступной в Редакторе столбцов, если в связанном с компонентом DataSetTableProducer объекте Table свойству Active установлено значение True.

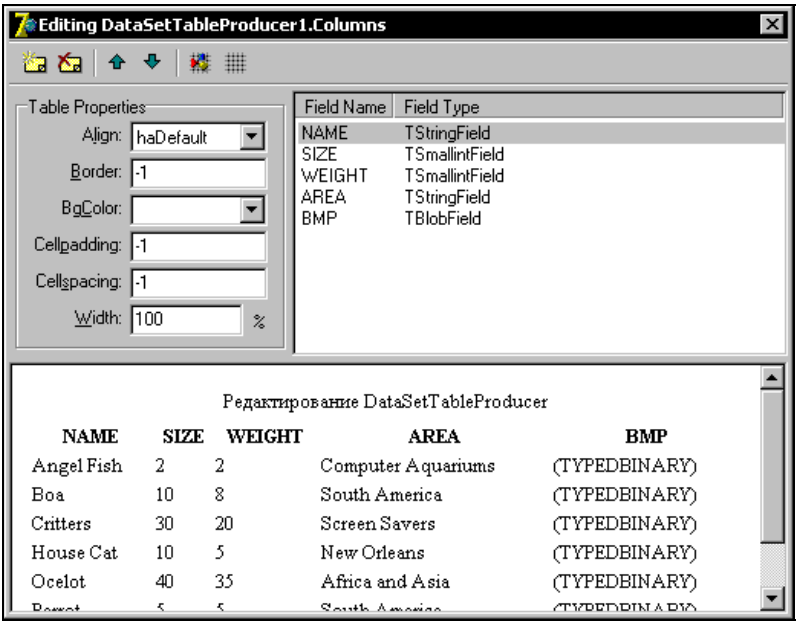


Рис. 20.3. Вид Редактора столбцов компонента `DataSetTableProducer`

В верхней части окна Редактора столбцов находится панель с кнопками для управления порядком вывода полей БД в результирующую HTML-таблицу. Кроме того, можно изменять количество и порядок следования полей БД с помощью команд контекстного меню Редактора столбцов. Перечислим назначение некоторых кнопок панели инструментов:

- **Add All Fields** — добавляет все поля таблицы, при этом появляется запрос на удаление уже присутствующих полей;
- **Delete Selected** — удаляет выбранные поля;
- **Add New** — добавляет новые поля;
- **Move Selected Up** и **Move Selected Down** — перемещают выбранное поле вверх или вниз;
- **Restore Defaults** — устанавливает для выбранного поля таблицы значения свойств форматирования по умолчанию.

При успешном открытии таблицы БД ее поля отображаются в нижней части Редактора столбцов.

Редактор столбцов позволяет определять группу выводимых столбцов и параметры таблицы с помощью средств визуального программирования. При этом создается HTML-код, в котором атрибутам табличных тегов присваиваются соответствующие значения параметров, установленные в Редакторе столбцов.

С помощью панели **Table Properties** можно задать следующие свойства, относящиеся ко всем полям таблицы:

- `Align` — горизонтальное выравнивание текста в ячейке таблицы, допустимые значения: `haDefault`, `haCenter`, `haLeft`, `haRight`;
- `Border` — толщина рамки (при значении `Border`, равном `-1`, рамка не выводится);
- `BgColor` — цвет фона ячеек таблицы, может принимать значения констант стандартных цветов, используемых в Delphi; по умолчанию цвет ячеек совпадает с цветом фона HTML-страницы;
- `CellPadding` — отступ в пикселах между текстом ячейки и рамкой таблицы;
- `CellSpacing` — отступ между ячейками таблицы в пикселах;
- `Width` — ширина таблицы в процентах от ширины HTML-страницы.

Редактирование отдельного столбца таблицы происходит с одновременным использованием Редактора столбцов и Инспектора объектов. При выделении поля в правом верхнем окне Редактора столбцов в Инспекторе объектов становятся доступными свойства этого поля. Задание свойств `Align` и `BgColor` для столбца аналогично их заданию для всей таблицы. Кроме того, становятся доступными дополнительные свойства. Вот наиболее часто используемые из них:

- `FieldName` — имя поля в таблице БД;
- `Title` — настройка свойств заголовка столбца, имеет аналогичные всей таблице свойства `Align` и `BgColor`. Свойство `Caption` задает заголовок столбца, свойство `Custom` добавляет дополнительные атрибуты к генерируемому элементу HTML, а свойство `VAlign` определяет вертикальное выравнивание заголовка и может принимать значения `haDefault`, `haTop`, `haBottom`, `haMiddle`, `haBaseLine`.

Компонент *QueryTableProducer*

Компонент `QueryTableProducer` отличается от компонента `DataSetTableProducer` наличием свойства `Query` типа `TQuery`, которое используется для выполнения SQL-запроса к базе данных.

Пример генератора HTML-страниц

В примере выполняется формирование статического HTML-документа, аналогичного документу из *разд. "Статическая публикация" данной главы*. Однако здесь генерация документа происходит с использованием стандартных компонентов Delphi.

В форме находятся четыре кнопки с именами `GenPageProducer`, `GenDataSetPageProducer`, `GenTableProducer`, `GenQueryTableProducer`, предназначенные для генерации HTML-документа с помощью размещенных в форме компонентов `PageProducer`, `DataSetPageProducer`,

компонентов PageProducer, DataSetPageProducer, DataSetTableProducer и QueryTableProducer соответственно. Для отображения сформированной HTML-страницы в обозревателе предназначена кнопка с именем ShowHtml. В форме также размещены многострочный редактор Memo1 и наборы данных Table1 и Query1.

В качестве источника данных для компонентов DataSetPageProducer и GenTableProducer используется набор данных Table1, связанный с этими компонентами через их свойство DataSet.

В качестве значения свойства HTMLDoc компонента PageProducer установлен следующий текст:

```
<TITLE>Демонстрация возможностей PageProducer </TITLE>
</HEAD><BODY>
<H1> Демонстрация PageProducer </H1>
Эта страница демонстрирует возможности PageProducer <P>
Имя базы данных <B> <#dbname> </B>
Дата: <B> <#date> </B>. <P>
<HR>
</BODY></HTML>
```

При формировании HTML-документа (методом Content) теги-транспаранты <#dbname> и <#date> в этом шаблоне будут заменены на конкретные значения.

Значением свойства HTMLDoc компонента DataSetPageProducer является следующий текст:

```
<HTML> <HEAD>
<TITLE> Данные из файла <#dbname> </TITLE>
</HEAD> <BODY>
<H1> <CENTER> Данные из файла <#dbname> </CENTER> </H1>
Данные первой записи из БД <BR>
Имя:    <#name>   <BR>
Размер: <#size>   <BR>
Вес:    <#weight> <BR>
Место:  <#area>   <BR>
Дата:   <#date>   <BR>
</BODY> </HTML>
```

Здесь теги-транспаранты <#dbname>, <#name>, <#size>, <#weight>, <#area> и <#date> при формировании HTML-документа также будут заменены на конкретные значения полей БД.

В качестве источника данных для компонента QueryTableProducer используется набор данных Query1, связанный с этим компонентом через свойство DataSet последнего. Свойству SQL объекта Query1, используемому для хранения SQL-запроса к БД, необходимо задать значение, которое обеспечит получение тре-

буемых данных. В рассматриваемом примере свойство `SQL` объекта `Query1` содержит следующий запрос:

```
SELECT * FROM Authors
WHERE YearBorn > 1960
```

Ниже приведен код модуля `DBHForm` главной формы `HTMLForm` приложения:

```
unit DBHForm;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, DBCtrls, StdCtrls, DBTables,
  ExtCtrls, Mask, Db, Dialogs, HTTPApp, DSProd, DBWeb;
type
  THTMLForm = class(TForm)
                ShowHtml: TButton;
         GenPageProducer: TButton;
 GenDataSetPageProducer: TButton;
        GenTableProducer: TButton;
  GenQueryTableProducer: TButton;
                  Memo1: TMemo;
           PageProducer1: TPageProducer;
    DataSetPageProducer1: TDataSetPageProducer;
  DataSetTableProducer1: TDataSetTableProducer;
    QueryTableProducer1: TQueryTableProducer;
                 Table1: TTable;
                 Query1: TQuery;

    procedure FormCreate(Sender: TObject);
    procedure GenPageProducerClick(Sender: TObject);
    procedure GenDataPageProducerClick(Sender: TObject);
    procedure GenTableProducerClick(Sender: TObject);
    procedure GenQueryTableProducerClick(Sender: TObject);
    procedure ShowHtmlClick(Sender: TObject);
    procedure DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure DataSetTableProducer1FormatCell(Sender: TObject;
      CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
      var Align: THTMLAlign; var VAlign: THTMLVAlign; CustomAttrs,
      var CellData: String);
  end;
```

```
var
  HTMLForm: THTMLForm;
implementation
{$R *.DFM}
uses
  ShellAPI;
procedure THTMLForm.FormCreate(Sender: TObject);
begin
// Инициализация наборов данных
Table1.Close;
Table1.DatabaseName := 'DBDEMOS';
Table1.TableName := 'Animals.dbf';
Table1.Open;
Query1.DatabaseName := 'Authors';
Query1.Open;
end;
procedure THTMLForm.ShowHtmlClick(Sender: TObject);
begin
// Сохранение сгенерированного HTML-документа
Memo1.Lines.SaveToFile ('htmltest.htm');
// Загрузка HTML-документа в обозреватель
ShellExecute (Handle, 'open', PChar('htmltest.htm'), '', '', sw_ShowNormal);
end;
// Генерация HTML-страницы компонента PageProducer
procedure THTMLForm.GenPageProducerClick(Sender: TObject);
begin
Memo1.Clear;
Memo1.Text := PageProducer1.Content;
end;
// Генерация HTML-страницы компонента DataSetPageProducer
procedure THTMLForm.GenDataSetPageProducerClick(Sender: TObject);
begin
Table1.First;
Memo1.Clear;
Memo1.Text := DataSetPageProducer1.Content;
end;
// Генерация HTML-страницы компонента TablePageProducer
procedure THTMLForm.GenTableProducerClick(Sender: TObject);
begin
Memo1.Clear;
Memo1.Text := DataSetTableProducer1.Content;
end;
```

```
// Генерация HTML-страницы компонента QueryTableProducer
procedure THTMLForm.GenQueryTableProducerClick(Sender: TObject);
begin
Memo1.Clear;
Memo1.Text := QueryTableProducer1.Content;
end;
// Обработка события OnHTMLTag компонента PageProducer1,
// используемая для замены тегов-транспарантов
procedure THTMLForm.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
                      const TagString: String; TagParams: TStrings;
                      var ReplaceText: String);
begin
// Замена тегов <#dbname> и <#date>
if TagString = 'dbname'
    then ReplaceText := Table1.TableName
    else if TagString = 'date' then ReplaceText := DateToStr(Date);
end;
// Обработка события OnHTMLTag компонента DataSetPageProducer1,
// используемая для замены тегов-транспарантов
procedure THTMLForm.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
                      const TagString: String; TagParams: TStrings;
                      var ReplaceText: String);
begin
// Замена тегов <#dbname> и <#date>
if TagString = 'dbname'
   then ReplaceText := Table1.TableName
   else if TagString = 'date' then ReplaceText := DateToStr(Date);
// Замена тегов <#name>, <#size>, <#weight>, <#area> на значения
// соответствующих полей текущей записи БД производится автоматически
end;
// Обработка события OnFormatCell компонента DataSetTableProducer1
procedure THTMLForm.DataSetTableProducer1FormatCell(Sender: TObject;
            CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
            var Align: THTMLAlign; var VAlign: THTMLVAlign;
            var CustomAttrs, CellData: String);
begin
if CellColumn > 2
    then CellData := '<FONT size=6>' + CellData + '</FONT>';
if Length(CellData) > 2
    then CellData := '<B>' + CellData + '</B>'
if CellRow > 5 BgColor := 'silver';
end;
end.
```

Инициализация наборов данных `Table1` и `Query1` осуществляется в процедуре `FormCreate`. Псевдоним `Authors` используется для связи набора данных `Query1` с БД, входящей в состав учебника по ASP-страницам, устанавливаемого вместе с операционной системой Windows 2000 Server. Этот псевдоним необходимо создать самостоятельно с помощью Администратора источников данных ODBC, установив связь псевдонима `Authors` с указанной БД. Напомним, что названный Администратор входит в состав утилит Панели управления операционной системы.

Для формирования HTML-документа с использованием компонента `PageProducer` предназначена кнопка `GenPageProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 20.4.

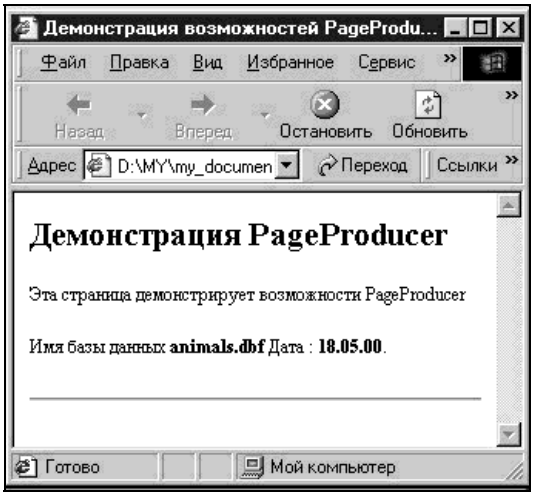


Рис. 20.4. HTML-страница, сгенерированная с помощью компонента `PageProducer`

HTML-страница содержит статический текст, состоящий из имени таблицы БД и строки с обозначением текущего времени. Страница формируется при вызове метода `Content`, при этом генерируется событие `OnHTMLTag` и выполняется его обработчик (процедура `PageProducer1HTMLTag`). В коде обработчика происходит замена тегов-транспарантов на значения имени таблицы БД и текущего времени, осуществляемая на основании анализа шаблона HTML-документа:

```
if TagString = 'dbname'
    then ReplaceText := Table1.TableName
    else if TagString = 'date' then ReplaceText := DateToStr(Date);
```

Здесь параметр `TagString` обработчика задает имя тега, при обработке которого произошло событие `OnHTMLTag`, а параметр `ReplaceText` содержит текст, на который заменяется тег.

Формирование HTML-документа с использованием компонента `DataSetPageProducer`, обращающегося к данным компонента `Table1`, происхо-

дит при нажатии кнопки `GenDataSetPageProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 20.5.

Рис. 20.5. HTML-страница, полученная с использованием компонента `DataSetPageProducer`

На этой странице выводится информация из первой записи БД. Для установки указателя на первую запись используется метод `First` набора данных `Table1`.

В процессе генерации HTML-документа возникает событие `OnHTMLTag`. Обработчик этого события для компонента `DataSetTableProducer` в основном совпадает с обработчиком аналогичного события рассмотренного выше компонента `PageProducer`, однако замена специальных тегов, имеющих имена, совпадающие с именами соответствующих полей таблицы БД, на значения этих полей производится автоматически.

Для формирования HTML-документа с помощью компонента `DataSetTableProducer` предназначена кнопка `GenTableProducer`. Вид полученного HTML-документа в окне обозревателя показан на рис. 20.6.

Показанный HTML-документ содержит данные таблицы БД, находящиеся в файле Animals.dbf. Формирование документа выполняется вызовом метода `Content` компонента `DataSetTableProducer`. При генерации HTML-документа используются параметры, установленные в Редакторе столбцов. При форматировании каждой ячейки таблицы возникает событие `OnFormatCell`, в обработчике которого редактируются параметры отображения.

В рассматриваемом примере форматирование заключается в следующем:

- устанавливается шестой размер шрифта для ячеек четвертого и следующих столбцов таблицы:

```
CellData := '<FONT size=6>' + CellData + '</FONT>';
```

- устанавливается полужирный шрифт для ячеек, размер содержимого которых превышает 2:

```
CellData := '<B>' + CellData + '</B>'
```

- для строк, начиная с 6-й, задается серебристый цвет фона:

```
BgColor:='silver';
```

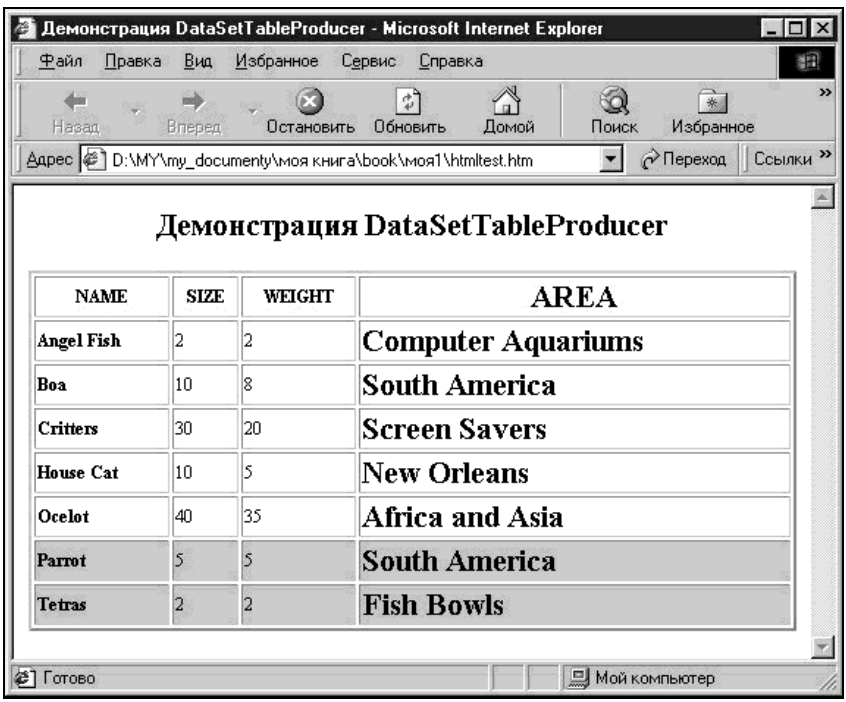


Рис. 20.6. HTML-страница, полученная с использованием компонента `DataSetTableProducer`

Для формирования HTML-документа с помощью компонента `QueryTableProducer` используется кнопка `GenQueryTableProducer`. Компонент `QueryTableProducer` отличается от `DataSetTableProducer` тем, что для получения данных применяется набор данных `Query`, а не `Table`. Вид сгенерированного HTML-документа в окне обозревателя показан на рис. 20.7.

В приведенном HTML-документе содержатся записи из таблицы `Authors`, отбор которых обеспечивает компонент `Query1`. Его SQL-запрос выбирает записи, для которых значение года превышает 1960.

Перейдем теперь к описанию динамической публикации, которая, как уже говорилось, используется для БД, обновляемых в реальном масштабе времени. Примером таких БД являются интернет-магазины, информационные системы в крупных сетях интранет и т. д.

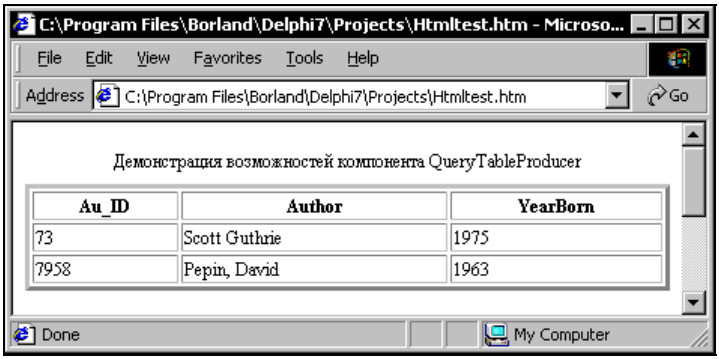


Рис. 20.7. HTML-страница, полученная с использованием компонента `QueryTableProducer`

Динамическая публикация

Для Web-приложений, использующих БД, могут применяться технологии активных серверных страниц (ASP), модули расширения на основе Java или интерфейсов CGI, ISAPI/NSAPI, WinCGI. При этом применение интерфейсов является предпочтительным по сравнению с другими технологиями при решении нестандартных задач, например, при обеспечении взаимодействия Web-сервера с сервером, осуществляющим прием платежей для обслуживания интернет-магазина. Кроме того, модули расширения сервера, созданные на основе интерфейсов, позволяют реализовать более быструю обработку запросов пользователей Web-приложения к БД.

В этом разделе рассматриваются средства Delphi, предназначенные для создания модулей расширения на основе интерфейсов CGI и ISAPI.

Создание модуля CGI

Познакомимся с особенностями разработки модуля CGI средствами Delphi на примере обычного консольного приложения.

Создадим модуль CGI с именем cgifirst.exe, который будет осуществлять генерацию HTML-страниц на основе информации из базы данных Animals.dbf, входящей в демонстрационный пакет программ Delphi. Для запуска этого модуля нужно выполнить следующие действия:

1. Установить Web-сервер (Personal WS или MS IIS).
2. Задать в параметрах проекта путь для выходного каталога (OutPut), где размещаются сценарии (для Personal WS — это каталог Webshare\Scripts, а для MS IIS — Inetpub\Scripts).
3. Создать в среде Delphi новое консольное приложение.

Загрузка модуля cgifirst.exe производится с помощью HTML-страницы, содержащей ссылку:

```
<P> <A HREF="http://igin/scripts/cgifirst.exe">
  Пример CGI-модуля </A> </P>
```

Исходный текст программы (консольного приложения) `CgiFirst` приводится ниже.

```
program CgiFirst;
{$APPTYPE CONSOLE}
uses
  Windows, SysUtils, DBTables, DB;
var
  // Массив для хранения URL CGI-модуля
  ScriptName: array [0..100] of Char;
  // Массив для хранения параметров строки запроса
  PathName: array [0..30] of Char;
  // Массив для хранения типа метода
  MethodName: array [0..30] of Char;
  // Флаг, используемый для возвращения на главную страницу
  k: Integer;
  // Внутренняя переменная
      ii: Integer;
  Table1: TTable;
// Процедура вывода заголовка страницы
procedure Header; forward;
// Процедура вывода окончания страницы
procedure Footer; forward;
// Процедура вывода списка ссылок
procedure ShowList; forward;
// Процедура вывода таблицы данных
procedure ShowTable; forward;
// Процедура вывода результатов запроса
procedure Showquery(it: Integer); forward;
// Вывод заголовка страницы
procedure Header;
begin
writeln('Content type: text/html');
writeln;
writeln('<HTML> <HEAD>');
writeln('<TITLE> Cgi-приложение </TITLE>');
writeln('</HEAD> <BODY>');
writeln('<H1> Публикация БД CGI-приложением </H1>');
writeln('<HR>');
end;
```

```
// Вывод окончания страницы
procedure Footer;
begin
writeln('<HR>');
writeln('<CENTER> <B> БД из файла ' +
   Table1.TableName + '</B> </H2> </CENTER>');
writeln('</BODY>');
writeln('</HTML>');
end;
// Вывод списка ссылок
procedure ShowList;
var i: Integer;
begin
Table1.First;
// Вывод заголовка списка
writeln('<CAPTION> <B> <FONT size=6> Список животных' +
   '</FONT> </B> </CAPTION>');
writeln ('<P> <TABLE BORDER>');
writeln ('<TR> <TH>' + 'Данные' + '</TH>');
// Вывод списка без последнего столбца
for i := 0 to Table1.FieldCount - 2 do begin
    writeln ('<TH>');
    writeln ('<A HREF="' + ScriptName + '/' +
       Table1.Fields [i].FieldName + '"> <B>' +
       Table1.Fields [i].FieldName + '</B> </A>');
    writeln ('</TH>');
    end;
writeln('</TR> </TABLE>');
writeln('<P> <P> <TABLE BORDER>');
writeln('<TR> <TH>');
writeln('<A HREF="' + ScriptName +
   '"> <B> Вернуться назад на основную страницу </B> </A>');
writeln ('</TR> </TH>');
writeln ('</TABLE>');
end;
// Вывод таблицы данных
procedure ShowTable;
var i: Integer;
begin
Table1.First;
writeln('<CAPTION> <B> <FONT SIZE=6>' +
   'Таблица с данными о животных </FONT> </B> </CAPTION>');
writeln('<P>< TABLE BORDER>');
writeln('<TR>');
```

```
// Вывод заголовков таблицы
for i := 0 to Table1.FieldCount - 2 do
    writeln ('<TH>' + Table1.Fields [i].FieldName + '</TH>');
writeln('</TR>');
// Вывод строк таблицы
while not Table1.EOF do begin
      writeln ('<TR>');
      for i := 0 to Table1.FieldCount - 2 do
          writeln ('<TD>' + Table1.Fields [i].AsString + '</TD>');
      writeln ('</TR>');
      Table1.Next;
      end;
writeln('</TABLE>');
writeln('<P> <P> <TABLE BORDER>');
writeln('<TR> <TH>');
// Вывод ссылки на запуск cgifirst.exe без параметров
writeln('<A HREF="'+ScriptName +
   '"> <B> Вернуться назад на основную страницу </B> </A>');
writeln ('</TR> </TH>');
writeln ('</TABLE>');
end;
// Вывод результатов запроса
procedure Showquery(it: Integer);
var i: Integer;
begin
Table1.First;
writeln('<CAPTION> <B> <FONT SIZE=6>' +
   'Данные о животных </FONT> </B> </CAPTION>');
writeln('<P> <TABLE BORDER>');
writeln('<TR>');
// Вывод заголовка списка
writeln('<TH>' + Table1.Fields[it].FieldName + '</TH>');
// Вывод данных требуемого столбца
while not Table1.EOF do begin
      writeln ('<TD>' + Table1.Fields [it].AsString + '</TD>');
      Table1.Next;
      end;
writeln('</TR>');
writeln('</TABLE>');
writeln('<P> <P> <TABLE BORDER>');
writeln('<TR> <TH>');
// Вывод специальной ссылки, которая при ее выборе (событие
// onClick) возвращает предыдущую страницу
// ("window.history.back()" — текст на Javascript)
```

```
writeln('<A HREF="" onClick="window.history.back()">' + '
   '<B> Вернуться назад </B> </A>');
writeln ('</TR> </TH>');
riteln ('<TR> <TH>');
writeln ('<A HREF="'+ScriptName +
   '"> <B> Вернуться назад на основную страницу </B> </A>');
writeln ('</TR> </TH>');
writeln ('</TABLE>');
end;
// Главная программа
begin
// Вывод заголовка страницы
Header;
// Считывание переменных окружения
// URL
GetEnvironmentVariable('SCRIPT_NAME', ScriptName,
   sizeof(ScriptName));
// Параметры из командной строки
GetEnvironmentVariable('PATH_INFO', PathName, sizeof(PathName));
// Создание таблицы
Table1 := TTable.Create(nil);
try
   Table1.DatabaseName := 'DBDEMOS';
   Table1.TableName := 'Animals.dbf';
   Table1.Open;
   k := 0;
   // Анализ параметров запроса
   if StrComp(PathName, '/List') = 0 then begin
      k := 1;
      ShowList;
      end
      else if StrComp(PathName, '/Table') = 0
         then begin ShowTable; k := 1; end
         else begin
            for ii := 0 to Table1.FieldCount - 2 do begin
               if StrComp (PathName,Pchar('/' +
                  Table1.Fields [ii].FieldName)) = 0) then
                     begin Showquery(ii);k := 1 end
               end;
            end;
            if k = 0 then begin
               writeln ('<A HREF="' + ScriptName + '/List"> <B>' +
                  'Загрузить список животных </B> </A> <P>');
```

```
                writeln ('<A HREF="' + ScriptName +'/TABLE"> <B>' +
                    'Загрузить данные о животных </B> </A> <P>');
                end;
  // Вывод окончания страницы
  Footer;
  finally
    Table1.Close;
    Table1.Free;
  end;
end.
```

В зависимости от параметров запроса приведенная программа обеспечивает вывод четырех HTML-страниц. При обработке на сервере URL-запроса вида

```
http://igin/scripts/cgifirst.exe
```

выводится главная страница (первая), содержащая следующие две гиперссылки на подчиненные страницы:

```
Загрузить список животных
Загрузить данные о животных
```

Первая из этих гиперссылок связана с URL-адресом **http://igin/scripts/cgifirst.exe/List** и адресует ко второй HTML-странице, вид которой показан на рис. 20.8.

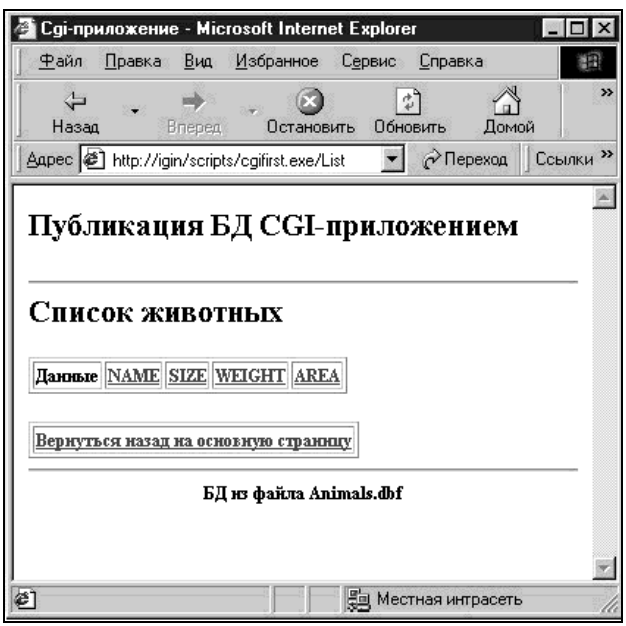


Рис. 20.8. Вид первой подчиненной HTML-страницы

Вторая гиперссылка содержит URL **http://igin/scripts/cgifirst.exe/Table** и адресует к третьей HTML-странице (рис. 20.9).

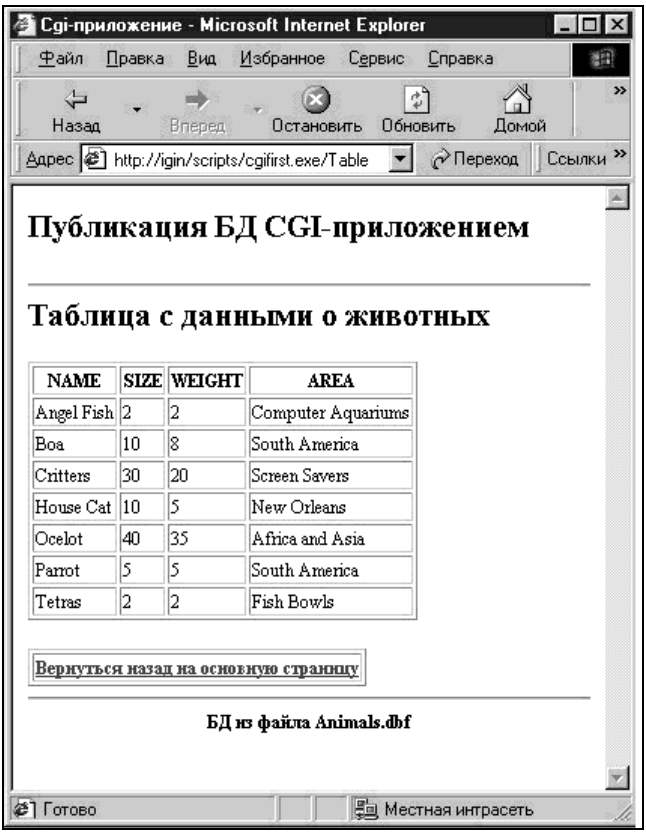


Рис. 20.9. Вид второй подчиненной HTML-страницы

Четвертая HTML-страница вызывается при выборе гиперссылок, содержащихся в ячейках верхней таблицы (см. рис. 20.8), причем строка запроса в URL параметризуется в зависимости от выбранного в этой таблице поля. Например, для первого поля (`NAME`) URL принимает значение **http://igin/scripts/cgifirst.exe/NAME** и адресует к HTML-странице, показанной на рис. 20.10.

Таким образом, приведенное CGI-приложение обеспечивает формирование четырех страниц при обработке на стороне сервера следующих URL-запросов:

- **http://igin/scripts/cgifirst.exe** — главная страница, выполняется основная часть программы;
- **http://igin/scripts/cgifirst.exe/List** — первая подчиненная таблица (см. рис. 20.8), выполняется процедура `ShowList`;

- **http://igin/scripts/cgifirst.exe/Table** — вторая подчиненная таблица (см. рис. 20.9), выполняется процедура `ShowTable`;
- **http://igin/scripts/cgifirst.exe/NAME** — подчиненная страница нижнего уровня (рис. 20.10); кроме параметра `NAME`, в запросе могут использоваться названия и других полей таблицы — `SIZE`, `WEIGHT`, `AREA`, в этом случае загружающая эту страницу ссылка может изменяться.

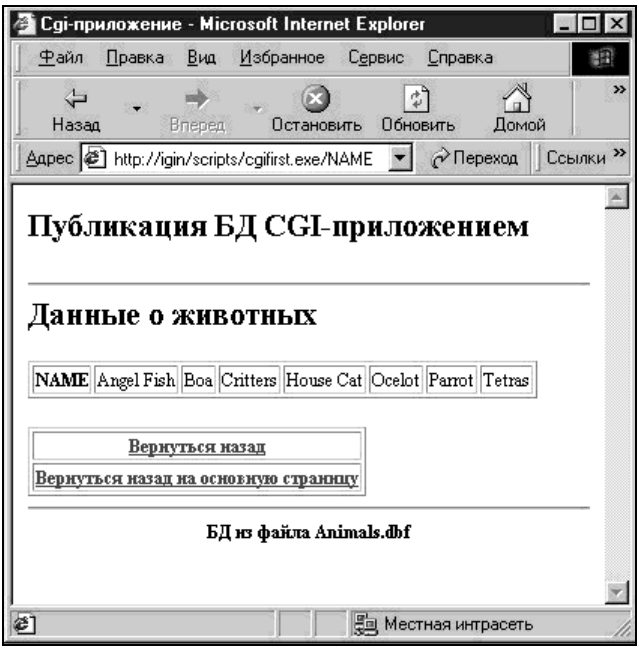


Рис. 20.10. Вид подчиненной HTML-страницы нижнего уровня

Поясним назначение наиболее важных частей программы. В начале основной части программы вызывается процедура `Header`, которая выводит заголовок для каждой HTML-страницы.

Для доступа к переменным окружения используется функция Windows API, имеющая следующий прототип:

```
DWORD GetEnvironmentVariable(
    LPCTSTR lpName,    // Адрес имени переменной окружения
    LPTSTR  lpBuffer,  // Адрес буфера для значения переменной
    DWORD   nSize      // Размер буфера в символах
  );
```

В программе эта функция вызывается дважды — для считывания значений системных переменных `SCRIPT_NAME` и `PATH_INFO`. Переменная `SCRIPT_NAME` содержит URL-адрес (без подстроки параметров) выполняемого в данный момент

модуля расширения, а переменная `PATH_INFO` содержит подстроку передаваемых параметров, которая входит в состав URL-адреса после символа `/`. Например, для URL-адреса **http://igin/scripts/cgifirst.exe/NAME** переменная `SCRIPT_NAME` содержит подстроку `http://igin/scripts/cgifirst.exe`, а переменная `PATH_INFO` — подстроку `/NAME`.

Для доступа к данным используется переменная `Table1` типа `TTable`. Управление объектом `Table1` осуществляет следующий код:

```
Table1 := TTable.Create (nil);
try
    Table1.DatabaseName := 'DBDEMOS';
    Table1.TableName := 'Animals.dbf';
    Table1.Open;
...
finally
    Table1.Close;
    Table1.Free;
  end;
```

Здесь метод `Create` класса `TTable` применяется для создания экземпляра объекта `Table1`. Для обработки исключений, связанных с открытием БД, используется конструкция `try... finally`.

Для определения страницы, которую должен формировать CGI-модуль, используется функция `StrComp`, которая сравнивает значение параметра URL-запроса, хранящегося в переменной `PathName`, с соответствующей строкой. При совпадении содержимого переменной `PathName` со строкой `/List` для генерации HTML-документа вызывается процедура `ShowList`, а при совпадении содержимого переменной `PathName` со строкой `/Table` — процедура `ShowTable`.

В цикле проверяется совпадение значения переменной `PathName` со строкой, содержащей названия полей таблицы БД:

```
for ii := 0 to Table1.FieldCount - 2 do begin
    if StrComp(PathName, Pchar('/' +
         Table1.Fields [ii].FieldName)) = 0 then
         begin Showquery(ii); k := 1; end
```

При совпадении вызывается процедура `Showquery`, в которую передается индекс соответствующего поля. Процедура `PChar` возвращает указатель на строку

```
'/' + Table1.Fields[ii].FieldName
```

Если ни одна из строк не совпадает с содержимым переменной `PathName`, то загружается главная страница CGI-приложения.

Для возврата на главную страницу (верхнего уровня) используется ссылка вида:

```
<A HREF="http://igin/scripts/cgifirst.exe">
<B> Вернуться назад на основную страницу </B> </A>
```

Возврат на предыдущие страницы осуществляется по ссылке:

```
<A HREF="" onClick="window.history.back()">
<B> Вернуться назад </B> </A>
```

Выбор этой гиперссылки вызывает событие `onClick`, при обработке которого выполняется функция `JavaScript.back()` объекта `window.history`.

Создание ISAPI-модуля расширения сервера

Web-приложение, выступающее в роли модуля расширения сервера и поддерживающее протокол программирования ISAPI, является динамически загружаемой библиотекой (DLL). Delphi обладает удобными средствами визуального проектирования таких приложений.

Рассмотрим схему работы Web-приложения, которое создано с помощью средств Delphi и является модулем расширения сервера (рис. 20.11).

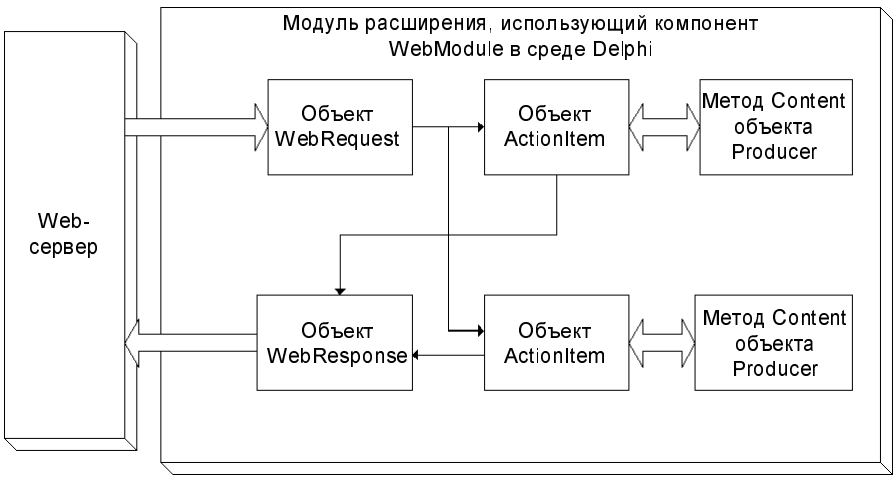


Рис. 20.11. Схема взаимодействия объектов модуля расширения сервера

При получении Web-приложением HTTP-запроса создается объект `WebRequest`, который обрабатывается в Web-модуле `WebModule`. Этот объект является связующим звеном между сервером и модулем расширения.

Для обработки запроса в модуле `WebModule` имеется набор объектов `WebActionItems`. Модуль `WebModule` определяет, какой из этих объектов отвечает за обработку поступившего запроса, при этом вызывается соответствующий обработчик события `OnActions`, в котором для ответа формируется объект `WebResponse`.

Модуль `WebModule` сравнивает значения свойств `MethodType` и `Pathinfo` объектов `WebActionItem` с соответствующими значениями параметров в HTTP-запросе. При совпадении значений этих свойств вызывается обработчик собы-

тия `OnAction` соответствующего объекта `WebActionItem`. При этом свойству `Enable` объекта `WebActionItem` в процессе проектирования должно быть присвоено значение `True`. Если ни у одного из объектов `WebActionItem` свойства `MethodType` и `Pathinfo` не совпали с соответствующими значениями параметров в HTTP-запросе, то модуль расширения прекращает свою работу, и сервер прерывает соединение с клиентом.

В случае использования компонентов-генераторов HTML-страниц для обработки запроса может вызываться их метод `Content`, который помещает формируемый HTML-документ в объект `WebResponse`.

Перечислим свойства объектов `WebRequest` и `WebResponse`, которые могут потребоваться при создании обычного Web-приложения.

Для получения передаваемых параметров из обозревателя в модуле расширения сервера `WebModule` используются следующие свойства объекта `WebRequest`:

- `URL` типа `String` (URL-запрос);
- `ScriptName` типа `String` (URL-путь к приложению, выполняемому на сервере);
- `QueryFields` типа `String` (параметры запроса, переданные в строке URL);
- `ContentFields` типа `String` (параметры запроса при использовании методов `Post` или `Put`).

Возвращение серверу сформированного HTML-документа осуществляется с помощью перечисленных ниже свойств объекта `WebResponse`.

- `Content` типа `String` (буфер для информации, возвращаемой обозревателю в качестве ответа на запрос);
- `ContentStream` типа `Stream` (указатель на поток, используемый для возвращения обозревателю большого объема информации);
- `ContentType` типа `String` (указатель типа возвращаемых обозревателю данных).

Замечание

ISAPI-модули расширения предназначены для многопользовательского доступа. При обработке нескольких запросов к одному и тому же модулю расширения управление передается единственному экземпляру ISAPI-модуля. Для обеспечения коллективного доступа к БД требуется объект `Session`. Если поместить в Web-модуль компонент класса `TSession` и установить его свойства `Active` и `AutoSessionName` в значение `True`, то для каждого нового запроса к БД будет устанавливаться новое подключение.

Рассмотрим создание простого Web-приложения, использующего интерфейс ISAPI.

Обратимся для этого к Хранилищу объектов и выберем на его странице **New** объект **Web Server Application**. После выбора этого объекта появляется окно (рис. 20.12) задания типа используемого серверного приложения (протокола), в нашем случае это `ISAPI/NSAPI`.

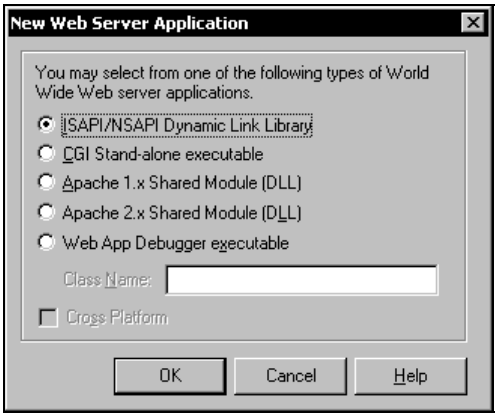


Рис. 20.12. Окно выбора типа серверного приложения

В результате открывается окно Web-модуля **WebModule1** (рис. 20.13), в котором можно размещать требуемые компоненты. Разместим в модуле два набора данных `Table` и свяжем эти компоненты с таблицами демонстрационной БД. Для генерации HTML-документов добавим к модулю два компонента `DataSetTableProducer`.

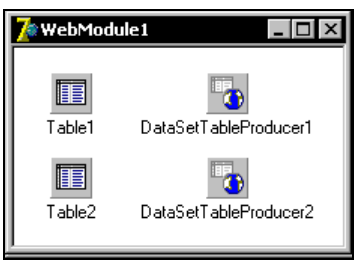


Рис. 20.13. Окно Web-модуля

Иерархию (дерево) составляющих модуль `WebModule` объектов можно отображать в окне **Object TreeView** (рис. 20.14). Здесь же можно удалить объекты, добавить и удалить элементы (поля) объектов, а также изменить порядок следования элементов.

Действия с объектами `WebActionItem` выполняются в окне редактирования действий модуля **Editing WebModule1.Actions** (рис. 20.15), открываемом двойным щелчком на свободном месте окна модуля командой **Action Editor** контекстного меню модуля или через его свойство `Actions` в Инспекторе объектов.

Добавим в этом окне требуемые действия ISAPI-модуля, осуществляемые в зависимости от параметров командной строки. Добавление действия выполняется с помощью крайней левой кнопки панели инструментов. Обработчик каждого параметра задается как обработчик события `OnActions` в каждом компоненте

`WebActionItems`. Имя обработчика события `OnActions` для объекта `WebActionItem` можно определить с помощью Инспектора объектов (рис. 20.16).

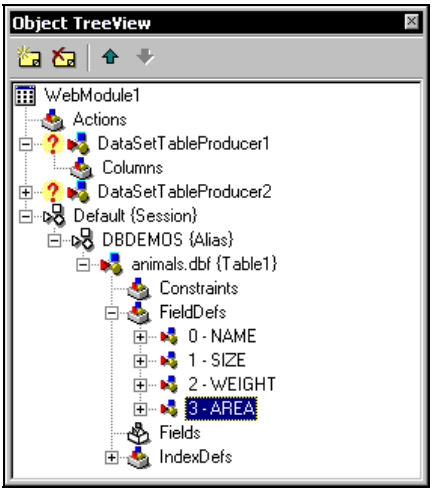


Рис. 20.14. Иерархия составляющих модуль `WebModule` компонентов

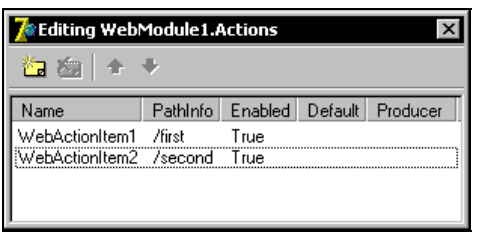


Рис. 20.15. Окно редактирования действий Web-модуля

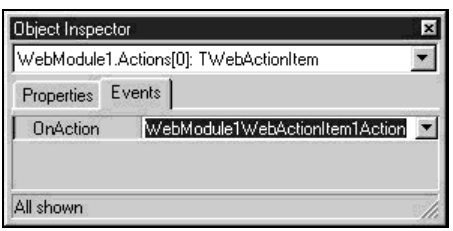


Рис. 20.16. Событие объекта `WebActionItem`

Для параметризации вызова модуля указываются значения свойств `PathInfo` для каждого объекта `WebActionItem` (также через Инспектор объектов).

Замечание

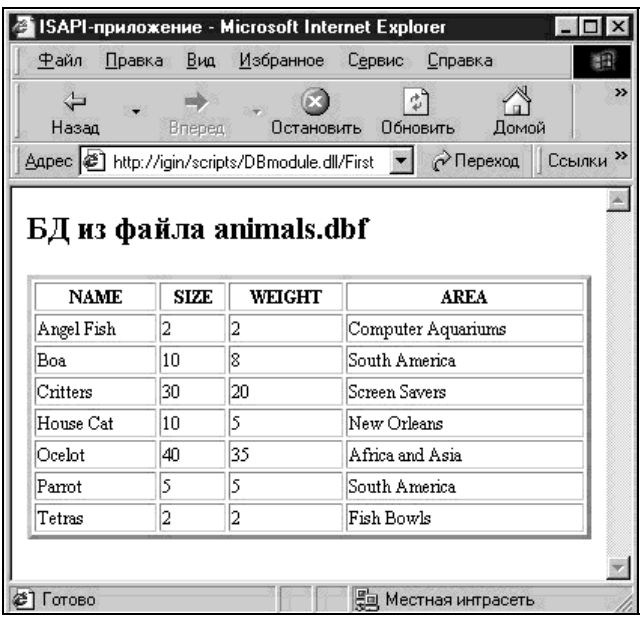
При отладке ISAPI-модуля возникает проблема, связанная с внесением изменений в соответствующую библиотеку DLL. Чтобы перегрузить созданную заново библиотеку, требуется перезапускать Web-сервер. Поэтому рекомендуется либо переименовывать ISAPI-модуль при повторном обращении к обновленной библиотеке, либо сначала создать для отладки CGI-приложение. После завершения отладки приложения нужно перекомпилировать его в ISAPI-модуль.

Модуль Dbmodule.dll выводит на HTML-страницы отчет из двух БД в виде таблиц. Для загрузки модуля используется следующая HTML-страница:

```
<HTML><HEAD>
</HEAD>
   <BODY>

<A HREF="http://igin/scripts/DBmodule.dll/First">
Вывести БД из файла biolife.db </A> </P>
<P> <A HREF="http://igin/scripts/DBmodule.dll/Second">
   Вывести БД из файла animals.dbf </A> </P>

</BODY>
</HTML>
```



NAME	SIZE	WEIGHT	AREA
Angel Fish	2	2	Computer Aquariums
Boa	10	8	South America
Critters	30	20	Screen Savers
House Cat	10	5	New Orleans
Ocelot	40	35	Africa and Asia
Parrot	5	5	South America
Tetras	2	2	Fish Bowls

Рис. 20.17. Вывод модулем Dbmodule.dll данных из файла animals.dbf

Исходный текст модуля `Dbmodule` имеет следующий вид.

```
unit Dbmodule;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBTables;
type
  TWebModule1 = class(TWebModule)
                   Table1: TTable;
    DataSetTableProducer1: TDataSetTableProducer;
                   Table2: TTable;
    DataSetTableProducer2: TDataSetTableProducer;

    procedure WebModule1WebActionItem1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure WebModule1WebActionItem2Action(Sender: TObject;
              Request: TWebRequest; Response: TWebResponse;
              var Handled: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  WebModule1: TWebModule1;
implementation
{$R *.DFM}
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
                     Request: TWebRequest; Response: TWebResponse;
                     var Handled: Boolean);
begin
Response.Content := '<HTML><HEAD>' +
  '<TITLE>ISAPI-приложение</TITLE>' +
  '</HEAD> <BODY> <H1> БД из файла ' + Table1.TableName + ' </H1>';
Response.Content := Response.Content + DataSetTableProducer1.Content
      + '</BODY> </HTML>';
end;
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
                      Request: TWebRequest; Response: TWebResponse;
                      var Handled: Boolean);
begin
Response.Content := '<HTML> <HEAD> <TITLE> ISAPI-приложение </TITLE>'
     +'</HEAD> <BODY> <H1> БД из файла ' + Table2.TableName+' </H1>';
```

```
Response.Content := Response.Content + DataSetTableProducer2.Content;
Response.Content := Response.Content + '</BODY> </HTML>';
end;
end.
```

При выборе первой ссылки будет загружаться HTML-страница, вид которой приведен на рис. 20.17.

При выборе второй ссылки выводится HTML-страница, содержащая отчет из БД biolife.dbf, по формату аналогичная первой странице.

Обработка пользовательского ввода в модуле ISAPI

При создании информационных систем на основе Web-приложений, публикующих БД в Интернете, для обеспечения взаимодействия с пользователем необходимо реализовать обработку вводимых им данных в формах на HTML-страницах. Получение передаваемых из обозревателя параметров наиболее часто выполняется с помощью методов `GET` и `POST`. Особенности работы сервера при использовании каждого из этих методов были описаны в *главе 19*.

В среде Delphi для получения передаваемых из обозревателя параметров используется объект типа `TWebRequest`. Рассмотрим наиболее важные свойства этого объекта.

Для получения параметров, переданных методом `GET`, предназначено свойство `QueryFields` типа `TStrings`. В этом свойстве переданные параметры записываются в следующем формате:

```
Параметр1=Значение_Параметра1
Параметр2=Значение_Параметра2
...
ПараметрN=Значение_ПараметраN
```

Доступ к данным, переданным методом `GET`, осуществляется с помощью следующей инструкции:

```
Response.Content := Request.QueryFields.Values['N1'];
```

где `N1` — имя переданного параметра.

Для получения параметров, переданных методом `POST`, используется свойство `ContentFields` типа `TStrings`. Доступ к данным при этом происходит так:

```
Response.Content := Request.ContentFields.Values['N1'];
```

Рассмотрим особенности использования свойств объекта `TWebRequest` на примере. Создадим обычное ISAPI-приложение, содержащее два объекта `WebActionItem`, параметры `PathInfo` которых задаются равными `\1` и `\2` соответственно. Поместим на поле Web-диспетчера компоненты `Query` и `QueryTableProducer` и установим свойство `Query` объекта `QueryTableProducer` в значение `Query1`. С помощью Редактора столбцов установим параметр `Border`

компонента `QueryTableProducer` в значение `5`, что обеспечит отображение рамки вокруг таблицы.

Для ввода пользовательских данных может использоваться следующая HTML-страница:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<BR>
<A HREF="http://localhost/Scripts/input.dll/1?
   N1=Ганс Андерсен&N2=1990">
Передача параметров методом GET
</A>
<FORM NAME=Form1 METHOD="Post"
ACTION="http://localhost/Scripts/Input.dll/2>
<BR>
Эта форма позволяет вводить новую строку в БД "Authors"
<BR>
Поля для ввода имени автора: <INPUT TYPE=Text NAME="N1"> <BR>
Поля для ввода года издания: <INPUT TYPE=Text NAME="N2"> <BR>
<BR>
<INPUT TYPE=Submit VALUE="Выполнить запрос">
</FORM>
</BODY>
</HTML>
```

Эта страница позволяет ввести два параметра: имя автора (`N1`) и год издания книги (`N2`).

Для добавления записей используем БД, имеющую псевдоним `Authors`. Напомним, что в названной БД присутствуют поля `author` и `YearBorn`.

Следующий код, находящийся в процедуре `WebModule1WebActionItem2Action`, позволяет сформировать посылаемый с помощью объекта `Query1` запрос к БД на добавление записи:

```
Query1.Close;
Query1.DatabaseName := 'Authors';
Query1.SQL.Clear;
Query1.SQL.Add ('INSERT INTO authors(author, YearBorn) values (''' +
   Request.ContentFields.Values['N1'] + ''', ' +
   Request.ContentFields.Values['N2'] + ')';
Query1.ExecSQL;
Query1.SQL.Clear;
Query1.SQL.Add('SELECT * FROM authors');
```

```
Query1.Open;
Response.Content := '<HTML> <HEAD>' +
   '<TITLE> Пример публикации БД </TITLE>' +
   '</HEAD> <BODY>' + QueryTableProducer1.Content;
```

Процедура `WebModule1WebActionItem2Action` вызывается, если передача параметров осуществляется с применением метода `POST`, а если для передачи параметров используется метод `GET`, то в этом случае вызывается процедура `WebModule1WebActionItem1Action` с аналогичным кодом, за исключением того, что в нем вместо свойства `ContentFields` применяется свойство `QueryFields`.

Для запуска ISAPI-модуля с передачей параметров методом `POST` нужно загрузить приведенную выше HTML-страницу, заполнить поля ввода и нажать кнопку **Выполнить запрос**. Запуск ISAPI-модуля с методом `GET` происходит при выборе ссылки **Передача параметров методом GET**.

Вид окна обозревателя с результатом работы ISAPI-модуля с передачей параметров методом `GET` показан на рис. 20.18.

В результате обработки запроса обозревателя рассматриваемым ISAPI-модулем будет сформирован HTML-документ, содержащий в виде таблицы все строки обновленной БД.

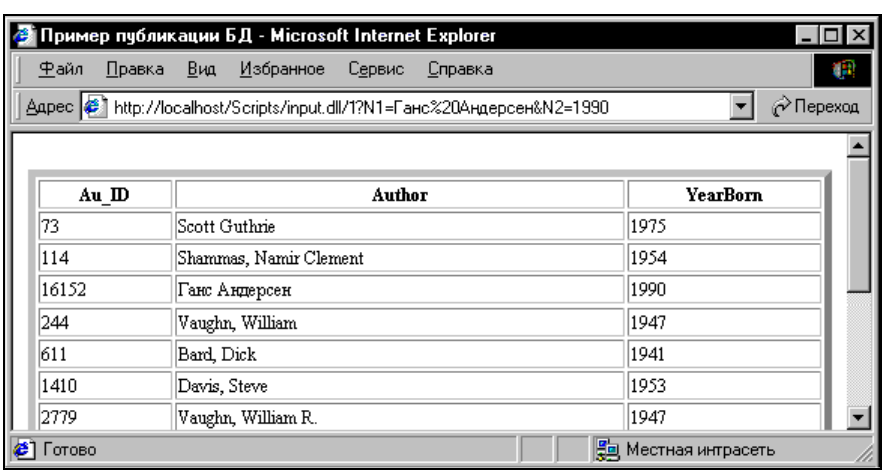


Рис. 20.18. Окно обозревателя с результатом работы ISAPI-модуля

Публикация графики

Рассмотрим применение стандартных средств Delphi для динамической публикации графической информации из БД на следующем примере.

Разрабатываемый модуль реализуем в виде ISAPI-библиотеки, использующей таблицу из файла animals.dbf. Для запуска данного приложения нужно создать

ISAPI-модуль, как описано выше, для которого потребуются 2 объекта `WebActionItem`. Свойства `PathInfo` этих объектов-действий устанавливаются в значения `text` и `image`, кроме того, подготавливаются обработчики для события `OnAction`. В модуле необходимо также разместить объекты `DataSetTableProducer` и `Table`.

В примере предварительно созданы три HTML-файла, текст которых приводится ниже. Управляющий файл proton.html содержит следующий код:

```
<HTML>
<HEAD>
<TITLE> Демонстрация работы с графикой </TITLE>
</HEAD>
<FRAMESET COLS="70%,30%">
<FRAME SRC="main.html" NAME=main>
<FRAME SRC="image.html" NAME=image>
</FRAMESET>
</HTML>
```

Совместное отображение графики и текста в HTML-документе удобно реализовать с помощью кадров. Тег `<FRAMESET COLS="70%,30%">` задает набор кадров. Параметр `COLS` используется для разбивки рабочей области обозревателя на два столбца шириной 70 и 30% соответственно.

Тег `<FRAME SRC="main.html" name=main>` определяет параметры кадра: параметр `SRC` задает отображаемый в кадре файл, а параметр `name` служит для доступа к созданным в этом кадре объектам.

Файл первого кадра (с именем main.html) содержит следующий код:

```
<HTML>
<BODY>
<SCRIPT>
location="http://igin/scripts/isag.dll/text"
</SCRIPT>
</BODY>
</HTML>
```

В теге `<SCRIPT>` для передачи управления модулю расширения isag.dll используется свойство `location` объекта `windows`.

Файл второго кадра (с именем image.html) содержит код:

```
<HTML>
<HEAD>
<TITLE>Демонстрация работы с графикой</TITLE>
</HEAD>
<BODY>
<SCRIPT>
location="http://igin/scripts/isag.dll/image?0"
```

```
</SCRIPT>
</BODY>
</HTML>
```

Исходный текст программы `isag` (табличное отображение отчета из БД) содержит следующий код:

```
unit isag;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, DBWeb, Db, DBTables;
type
  TWebModule1 = class(TWebModule)
                Table1: TTable;
  DataSetTableProducer1: TDataSetTableProducer;
  procedure DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
                     CellColumn: Integer; var BgColor: THTMLBgColor;
                     var Align: THTMLAlign; var VAlign: THTMLVAlign;
                     var CustomAttrs, CellData: String);
  procedure WebModuleAfterDispatch(Sender: TObject;
                            Request: TWebRequest;
                            Response: TWebResponse;
                            var Handled: Boolean);
    procedure WebModuleBeforeDispatch(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  procedure WebModule1WebActionItem1Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  procedure WebModule1WebActionItem2Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
  private
    ScriptName: string;
end;
var
  WebModule1: TWebModule1;
implementation
{$R *.DFM}
uses
  Graphics, Jpeg, ExtCtrls,dbctrls;
// Обработчик события OnFormatCell
// Параметр Target указывает на соседний кадр
procedure TWebModule1.DataSetTableProducer1FormatCell(
        Sender: TObject; CellRow, CellColumn: Integer;
```

```
        var BgColor: THTMLBgColor; var Align: THTMLAlign;
        var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
if (CellColumn = 4) and (CellRow > 0) then
   CellData := '<A HREF="' + ScriptName + '/image?' +
      IntToSTR(CellRow - 1) + ' "Target="image"> Show Image </A>';
end;
// Обработчик события AfterDispatch
procedure TWebModule1.WebModuleAfterDispatch(Sender: TObject;
                   Request: TWebRequest; Response: TWebResponse;
                   var Handled: Boolean);
begin
Table1.close;
end;
// Обработчик события BeforeDispatch
procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
                  Request: TWebRequest; Response: TWebResponse;
                  var Handled: Boolean);
begin
Table1.Open;
ScriptName := Request.ScriptName;
end;
// Обработчик события OnAction объекта WebActionItem1
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
                   Request: TWebRequest; Response: TWebResponse;
                   var Handled: Boolean);
begin
// Формирование ответа обозревателю с помощью
// компонента DataSetTableProducer1
Response.Content:= DataSetTableProducer1.Content;
end;
// Обработчик события OnAction объекта WebActionItem2
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
                    Request: TWebRequest; Response: TWebResponse;
                    var Handled: Boolean);
var
   DataSource1: TDataSource;
      DBImage1: TDBImage;
         Jpeg1: TJpegImage;
        Stream: TMemoryStream;
           num: Integer;
begin
DataSource1 := TDataSource.Create(nil);
DBImage1 := TDBImage.Create(nil);
```

```
try
   Table1.Open;
   DataSource1.DataSet := Table1;
   DBImage1.DataSource := DataSource1;
   DBImage1.DataField := 'BMP';
   // Извлечение параметров запроса
   num := StrToInt(Request.Query);
     if num = 0 then Table1.First
                else Table1.MoveBy(num);
    Jpeg1 := TJpegImage.Create;
    try
      Jpeg1.CompressionQuality := 50;
      Jpeg1.Assign(DBImage1.Picture.Bitmap);
      Stream := TMemoryStream.Create;
      try
        Jpeg1.SaveToStream (Stream);
        Stream.Position := 0;
        Response.ContentStream := Stream;
        Response.ContentType := 'image/jpeg';
        Response.SendResponse;
        finally
          Stream.Free;
        end;
      finally
        Jpeg1.Free;
      end;
    finally
      Table1.Close;
    end;
end;
end.
```

Результаты обработки запросов модулем isag.dll приведены на рис. 20.19.

В первом кадре (рис. 20.19) в виде таблицы отображаются текстовые данные. Для формирования таблицы в левом кадре используется следующий URL-запрос:

```
http://igin/scripts/isag.dll/text
```

Для обработки этого запроса вызывается процедура `WebModule1WebActionItem1Action`, в которой генерируется HTML-документ, содержащий таблицу. Формирование HTML-документа выполняет метод `Content` компонента `DataSetTableProducer1`:

```
Response.Content:=DataSetTableProducer1.Content;
```

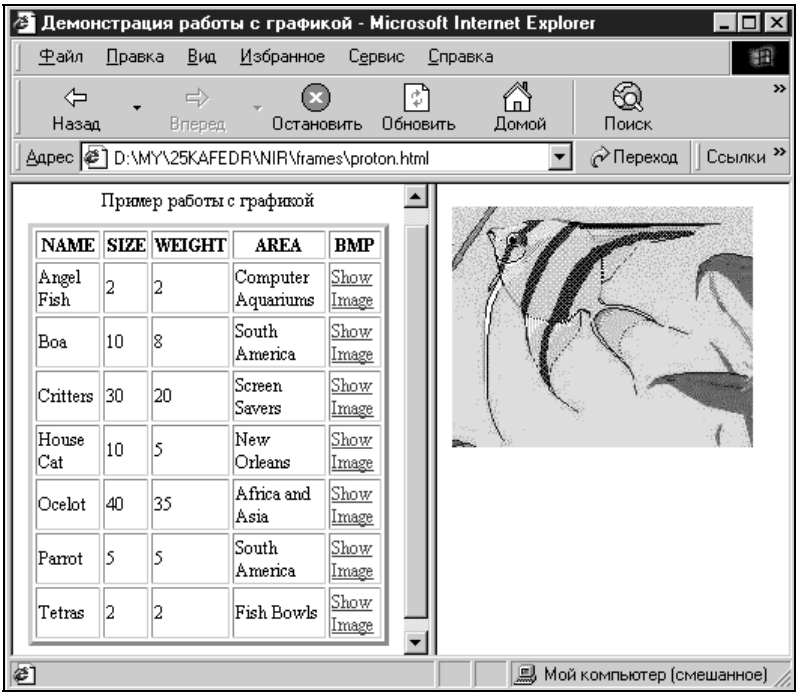


Рис. 20.19. Результаты работы модуля isag.dll

При форматировании каждой ячейки этой таблицы в методе `Content` объекта `DataSetTableProducer1` возникает событие `OnFormatCell`, в обработчике которого содержится условная инструкция

```
if (CellColumn = 4) and (CellRow > 0) then ...
```

Тем самым допускается редактирование только пятого столбца всех строк, начиная со второй, и для них формируются ссылки вида

```
<A HREF="http://igin/scripts/isag.dll/image?0" TARGET="image"> Show Image </A>
```

Цифра после знака `?` изменяется в зависимости от номера строки. Атрибут `TARGET="image"` задает целевой кадр для вывода графики после обработки запроса.

Собственно публикация графической информации осуществляется в процедуре `WebModule1WebActionItem2Action`.

Для извлечения параметров URL-запроса, передаваемых в самом запросе, используется свойство `Query` параметра `Request`. Для преобразования строкового значения переданного параметра применяется процедура `StrToInt`:

```
num := StrToInt(Request.Query);
```

С помощью метода `MoveBy` объекта `Table` указатель текущей записи устанавливается на запись номер `num`.

Графическая информация извлекается из БД с помощью объекта `DBImage`:

```
DataSource1 := TDataSource.Create(nil);
DBImage1 := TDBImage.Create(nil);
Table1.Open;
DataSource1.DataSet := Table1;
DBImage1.DataSource := DataSource1;
DBImage1.DataField := 'BMP';
```

Информация в поле `BMP` базы данных хранится в формате BMP. Чтобы обозреватель мог отобразить графическую информацию, ее необходимо преобразовать в формат JPG. Для этого используется объект класса `TJPEG`:

```
Jpeg1 := TJpegImage.Create;
Jpeg1.CompressionQuality := 50;
```

Преобразование файла из формата BMP в формат JPG осуществляется с помощью метода `Assign` объекта `JpegImage`:

```
Jpeg1.Assign(DBImage1.Picture.Bitmap);
```

Для отправки изображения обозревателю используется промежуточный поток

```
Stream := TMemoryStream.Create;
Jpeg1.SaveToStream(Stream);
```

Указатель текущей позиции устанавливается на начало потока с помощью свойства `Position` объекта `Stream`:

```
Stream.Position := 0;
```

Далее указателю на возвращаемый поток Web-сервера присваивается значение указателя на промежуточный поток:

```
Response.ContentStream := Stream;
```

Тип возвращаемых данных устанавливается следующим образом:

```
Response.ContentType := 'image/jpeg';
```

С помощью метода `SendResponse` находящиеся в потоке данные отсылаются обозревателю:

```
Response.SendResponse;
```

Замечание

Событие `BeforeDispatch` возникает прежде, чем Web-диспетчер приступит к обработке поступившего HTTP-запроса. Событие `AfterDispatch` возникает после того как HTTP-ответ успешно сформирован, но еще не отослан. Эти события удобно использовать для выполнения соответственно инициализации и удаления различных ресурсов, необходимых для обработки запроса.

Использование технологии ADO

В завершение главы рассмотрим средства доступа к источникам данных с использованием технологии (интерфейса) ADO, являющейся надстройкой над интерфейсом OLE DB. Напомним, что основа интерфейса ADO — набор объектов, на вершине иерархии которого находятся объекты `Connection`, `Command` и `Recordset`, обеспечивающие соединение с электронными источниками информации, поддерживающими этот интерфейс.

Для работы с интерфейсом ADO предназначены компоненты, находящиеся на странице **ADO** Палитры компонентов (см. рис. 2.8). Назначение указанных компонентов приводится в *главе 2*. Кроме того, работа с базами данных по технологии ADO освещалась в *главе 10*.

Рассмотрим особенности использования компонентов ADO на примере модуля ISAPI, публикующего БД в Интернете.

Для запуска примера требуется создать обычное ISAPI-приложение, как это было сделано в предыдущих разделах. После создания основного модуля, реализующего функции Web-диспетчера, добавим к нему один объект `WebActionItem`.

В данном приложении для получения доступа к источнику данных достаточно использовать два компонента: `ADOQuery` и `ADOConnection`. Поместим их на поле объекта `WebDispatcher`, добавим еще компонент `DataSetTableProducer`.

Установим свойство `DataSet` объекта `DataSetTableProducer1` в значение `ADOQuery1` и зададим рамку вокруг таблицы, установив с помощью Редактора столбцов ее толщину в значение 5.

Рассмотрим порядок настройки компонентов ADO на примере компонентов `ADOQuery` и `ADOConnection` (остальные компоненты ADO настраиваются аналогично). Сначала настроим компонент `ADOConnection`. Для этого выберем его свойство `ConnectionString`. После щелчка на этом свойстве в окне Инспектора объектов появляется окно Мастера настройки соединения с источником данных (рис. 20.20).

В этом окне можно выбрать вариант соединения с помощью файла связи (переключатель **Use Data Link File**) или строки соединения (переключатель **Use Connection String**). Выберем строку соединения.

Для продолжения настройки следует нажать кнопку **Build**, открывающую окно **Data Link Properties** (Свойства связи с данными). В нем нужно выбрать тип поставщика услуг (OLE DB-провайдера данных). В нашем случае мы публикуем информацию из уже использованной в других примерах демонстрационной базы данных учебника по ASP-страницам. Для нее с помощью Администратора источника данных ODBC создан псевдоним `Authors`. Поэтому выберем OLE DB-провайдер для ODBC-драйвера.

Для продолжения настройки нажмем кнопку **Next**, после чего появляется очередное окно **Data Link Properties** (Свойства связи с данными). В нем необходимо указать псевдоним для ODBC-источника данных, в нашем случае — `Authors`.

Здесь же можно задать параметры соединения. В случае соединения с БД с псевдонимом `Authors` параметры соединения отсутствуют.

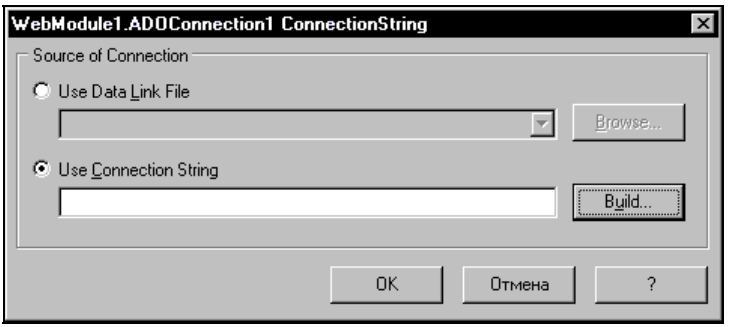


Рис. 20.20. Окно Мастера настройки соединения с источником данных

Для проверки соединения можно воспользоваться кнопкой **Test connection** (Проверить соединение). С помощью вкладки **Advanced** (Дополнительно) можно установить режимы доступа к источнику данных. Параметры соединения с ADO-источником данных можно просмотреть на вкладке **All** (Все).

После того как настройка соединения объекта `ADOConnection` завершена, в объекте `ADOQuery` для получения доступа к источнику данных достаточно установить свойство `Connection` в значение `ADOConnection1`.

Далее необходимо создать обработчик события `OnAction` для объекта `WebActionItem1`, в который нужно поместить следующий код:

```
ADOQuery1.Close;
ADOQuery1.SQL.Clear;
ADOQuery1.SQL.Add('SELECT * FROM Authors WHERE YearBorn >' +
   Request.ContentFields.Values['N1']);
ADOQuery1.Open;
Response.Content := '<HTML> <HEAD>' +
   '<TITLE> Пример публикации БД с использованием интерфейса ADO ' +
   '</TITLE> </HEAD> <BODY> <H1> Данные из БД, начиная с ' +
   Request.ContentFields.Values['N1'] + '</H1>' +
   DataSetTableProducer1.Content + '</BODY> </HTML>';
```

Здесь инструкция

```
ADOQuery1.SQL.Add('SELECT * FROM Authors WHERE YearBorn >' +
   Request.ContentFields.Values['N1']);
```

формирует строку запроса на выборку записей из БД, для которых значение поля `YearBorn` превышает величину, введенную пользователем и получаемую с

помощью выражения `Request.ContentFields.Values['N1']` из поля формы с именем `N1`.

Для загрузки модуля ISAPI используется следующий HTML-документ:

```
<HTML>
<HEAD></HEAD>
<BODY>
<FORM NAME=Form1 METHOD="Post"
ACTION="http://localhost/Scripts/adow.dll">
<BR>
<H1> Эта форма позволяет посылать запрос <BR>
  на выборку данных из БД "Authors"
</H1>
<BR>
Поля для ввода года издания: <INPUT TYPE=Text NAME="N1"> <BR>
<BR>
<INPUT TYPE=Submit VALUE="Выполнить запрос">
</FORM>
</BODY>
</HTML>
```

Например, если пользователь ввел в поле `N1` значение `1954` и нажал кнопку **Выполнить запрос**, в обозревателе будет отображен HTML-документ, представленный на рис. 20.21.

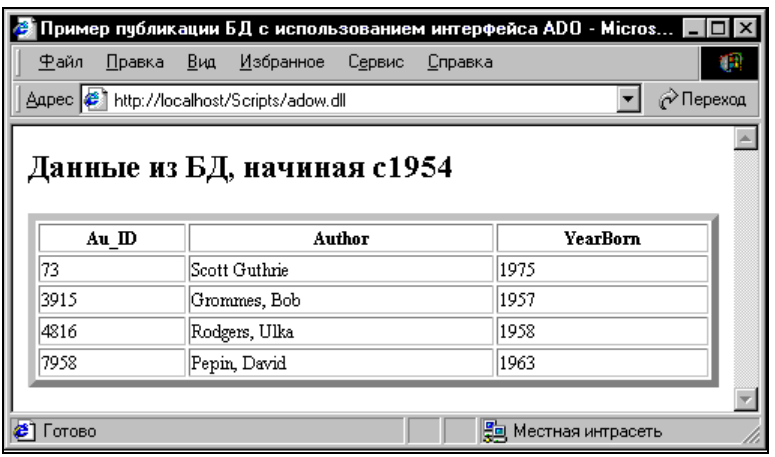


Рис. 20.21. Окно обозревателя с результатом работы модуля adow.dll

На этой странице выведены записи, для которых значения поля `YearBorn` превышают `1954`.

Глава 21

Работа с электронной почтой

Система Delphi обеспечивает возможность работы с электронной почтой (E-mail) в разрабатываемом приложении. Для этого можно использовать функцию `ShellExecute` или интерфейс MAPI (Mail API).

Использование функции *ShellExecute*

Наиболее простым способом вызова почтового клиента, в окне которого можно подготовить (создать) отправить письмо, является применение функции `ShellExecute`.

Вот пример вызова почтового клиента:

```
// Не забудьте подключить этот модуль
uses ShellApi;
ShellExecute(Form1.Handle, Nil, 'MailTo: vladimir@mail.ru', Nil,
        Nil, SW_SHOWNORMAL);
```

В результате вызова этой функции открывается окно почтового клиента, установленного на компьютере в качестве почтовой программы по умолчанию.

Замечание

Для реализации возможности работы с электронной почтой на компьютере должна быть установлена какая-либо почтовая программа (почтовый клиент), например, Microsoft Outlook, Outlook Express, The Bat.

Строка `MailTo:` в третьем параметре функции в приведенном выше примере указывает Windows, что необходимо запустить установленную по умолчанию почтовую программу, а необязательный адрес, идущий после нее (в нашем примере это `vladimir@mail.ru`), автоматически заносится в поле получателя.

Если необходимо автоматически заполнить поле темы письма, то оно указывается после адреса как значение параметра `Subject`. Также можно автоматически заполнить текст сообщения, для чего используется параметр `Body`.

Параметры отделяются от адреса знаком `?`, а друг от друга — знаком `&`.

Рассмотрим пример вызова почтового клиента с автоматическим заполнением адреса, темы и текста сообщения

```
ShellExecute(Form1.Handle, nil, 'MailTo: vladimir@mail.ru' +
      '?Subject=The first message' + '&Body=It is just an example',
      Nil, Nil, SW_SHOWNORMAL);
```

Если почтовой программой по умолчанию является приложение Microsoft Outlook, и в нем задано использование Microsoft Word в качестве редактора сообщений, то в результате выполнения приведенного метода откроется окно, приведенное на рис. 21.1 (это диалоговое окно является немодальным).

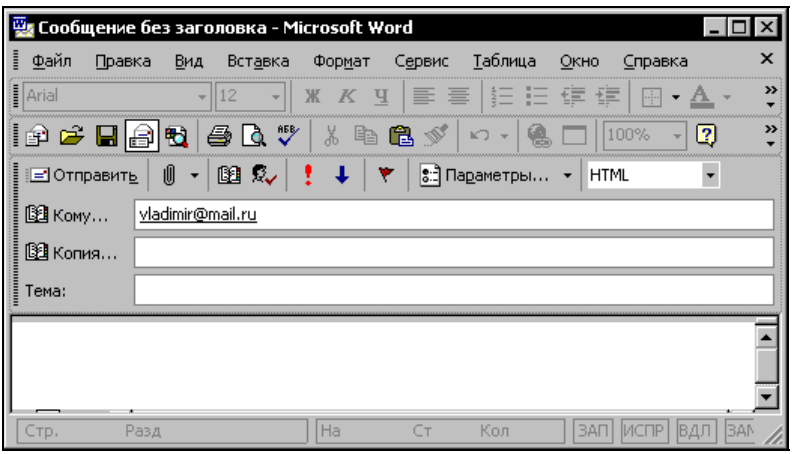


Рис. 21.1. Диалоговое окно подготовки почтового сообщения

Использование интерфейса MAPI

Для работы с электронной почтой можно также использовать интерфейс MAPI, который предоставляет больше возможностей, однако программировать работу с ним труднее, чем в случае использования функции `ShellExecute`. Для работы с интерфейсом MAPI следует подключить модуль `Mapi` с помощью строки вида

```
Uses Mapi;
```

Рассмотрим пример отправки почтового сообщения из приложения с использованием MAPI. Для решения поставленной задачи в форму приложения, кроме прочих, поместим следующие компоненты: `edtReceiver`, `edtTitle`, `memMessageText`, `edtAttachment`, `OpenDialog1` и `btnInsertFile`. Поле ввода **Получатель** (`edtReceiver`) предназначено для ввода почтового адреса получателя (адресата), поле ввода **Название** (`edtTitle`) содержит название (тему) сообщения. Текст почтового сообщения набирается в многострочном редакторе `memMessageText`. Форма приложения на этапе разработки приведена на рис. 21.2.

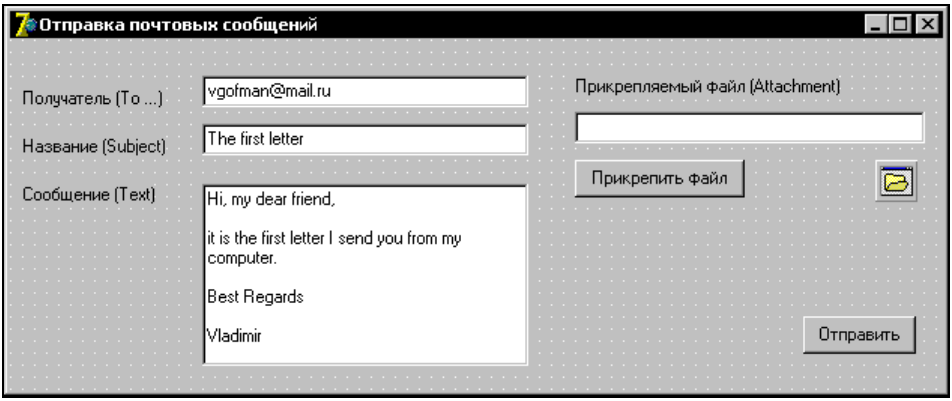


Рис. 21.2. Вид формы приложения на этапе разработки

К отправляемому сообщению можно присоединить файл, полный путь которого вводится в поле ввода **Прикрепляемый файл** (`edtAttachment`). Файл удобно выбирать, используя компонент `OpenDialog1` выбора файла. Стандартное диалоговое окно выбора файла открывается нажатием кнопки **Прикрепить файл** (`btnInsertFile`). После выбора требуемого файла его полный путь заносится в поле `edtAttachment`.

Ниже приведен код модуля `Unit1` приложения.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
        btnSend: TButton;
    edtReceiver: TEdit;
       edtTitle: TEdit;
         Label1: TLabel;
         Label3: TLabel;
  edtAttachment: TEdit;
 memMessageText: TMemo;
         Label4: TLabel;
         Label5: TLabel;
  btnInsertFile: TButton;
    OpenDialog1: TOpenDialog;
```

```
    procedure btnSendClick(Sender: TObject);
    procedure btnInsertFileClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

// Этот модуль обеспечивает возможность работы с электронной почтой
uses Mapi;

{$R *.DFM}

// Подготовить и отправить почтовое сообщение
function SendEmail(Receiver, Title, MessageText,
                   AttachmentPath: string): boolean;
var MapiMessage: TMapiMessage;
         MError: Cardinal;
      Recipient: TMapiRecipDesc;
File_Attachment: TMapiFileDesc;
begin
// Подготовить информацию получателя
FillChar(Recipient, SizeOf(Recipient), 0);
Recipient.ulRecipClass := MAPI_TO;
Recipient.lpszName := PChar(Receiver);

// Подготовить информацию прикрепленного файла
if Trim(AttachmentPath) <> '' then begin
   FillChar(File_Attachment, SizeOf(File_Attachment), 0);
   File_Attachment.nPosition := ULONG(-1);
   File_Attachment.lpszPathName := PChar(AttachmentPath);
   end;

// Сформировать сообщение
with MapiMessage do begin
    ulReserved := 0;
    lpszSubject := PChar(Title);
```

```
    lpszNoteText := PChar(MessageText);
    lpszMessageType := nil;
    lpszDateReceived := nil;
    lpszConversationID := nil;
    flFlags := 0;
    lpOriginator := nil;
    nRecipCount := 1;
    lpRecips := @Recipient;
    if Trim(AttachmentPath) <> '' then begin
       nFileCount := 1;
       lpFiles := @File_Attachment;
       end
    else begin
       nFileCount := 0;
       lpFiles := nil;
       end;

  end;

// Попробовать отправить сообщение
MError := MapiSendMail(0, 0, MapiMessage, MAPI_DIALOG, 0);
if not (MError = SUCCESS_SUCCESS) then begin
   ShowMessage('Сообщение не было отправлено!');
   Result := False;
   end
   else Result := True;
end;

// Отправить сообщение
procedure TForm1.btnSendClick(Sender: TObject);
begin
SendEmail(edtReceiver.Text, edtTitle.Text,
         memMessageText.Text, edtAttachment.Text);
end;

// Выбрать прикрепляемый файл
procedure TForm1.btnInsertFileClick(Sender: TObject);
begin
if OpenDialog1.Execute then
   edtAttachment.Text := OpenDialog1.FileName;
end;

end.
```

После ввода в соответствующие поля требуемой информации письмо можно отправить, нажав кнопку **Отправить** (`btnSend`). В обработчике событий этой кнопки вызывается функция `SendEmail`, выполняющая требуемые действия по подготовке и отправке почтового сообщения (вызов почтового клиента).

Подготовка сообщения заключается в заполнении соответствующих структур типа `TMapiRecipDesc` — для получателя, типа `TMapiFileDesc` — для присоединяемого файла и типа `TMapiMessage` — для самого отправляемого сообщения.

Вызов почтового клиента выполняет функция `MapiSendMail(lhSession: LHANDLE; ulUIParam: Cardinal; var lpMessage: TMapiMessage; flFlags: FLAGS; ulReserved: Cardinal): Cardinal`.

Параметр `lhSession` содержит ссылку на сеанс, используемый для отправки данного сообщения. Если параметр имеет значение 0, то для отправки сообщения временно создается новый сеанс.

Параметр `ulUIParam` содержит ссылку на родительское окно открываемого модального диалогового окна почтовой программы. Если параметр имеет значение 0, то родителем модального диалогового окна является приложение, из которого это диалоговое окно было вызвано.

Параметр `lpMessage` содержит отправляемое сообщение.

Параметр `flFlags` может принимать значения, важнейшие из которых:

- `MAPI_DIALOG` — указывает, что при отправке сообщения отображается диалоговое окно, в котором пользователь может ввести или изменить данные сообщения. Это окно является модальным, т. е. для продолжения работы с приложением необходимо закрыть окно сообщения. Если эта опция не установлена, то сообщение сразу отправляется получателю. При этом как минимум должен быть задан получатель сообщения;
- `MAPI_LOGON_UI` — указывает, что должно быть открыто диалоговое окно для ввода имени пользователя и пароля, если таковые установлены.

Параметр `ulReserved` зарезервирован и должен иметь значение 0.

Функция возвращает результат, основными возможными значениями которого являются следующие:

- `SUCCESS_SUCCESS` — сообщение отправлено успешно;
- `MAPI_E_ATTACHMENT_NOT_FOUND` — не найден указанный прикрепленный файл;
- `MAPI_E_ATTACHMENT_OPEN_FAILURE` — прикрепленный файл нельзя открыть;
- `MAPI_E_FAILURE` — произошла неизвестная ошибка;
- `MAPI_E_INSUFFICIENT_MEMORY` — недостаточно памяти для выполнения операции;
- `MAPI_E_TEXT_TOO_LARGE` — текст сообщения слишком велик;
- `MAPI_E_TOO_MANY_RECIPIENTS` — слишком много получателей;
- `MAPI_E_USER_ABORT` — пользователь отменил диалог.

Из приведенных вариантов следует, что если результатом функции является значение, отличное от `SUCCESS_SUCCESS`, то сообщение электронной почты не было отправлено.

Замечание

Если на компьютере установлен почтовый клиент, отличный от Microsoft Outlook, то отправка сообщений с использованием модуля MAPI может выполняться некорректно.

Перед попыткой отправить сообщение с помощью кнопки **Отправить** почтовый клиент, например, Microsoft Outlook, должен быть запущен. В противном случае будет выдано предупреждение "Your message hase not been sent!" (Ваше сообщение не было отправлено!).

Глава 22

Характеристика Web-служб

Web-службы, получившие дальнейшее развитие в Delphi 7, предназначены для обеспечения универсальности доступа к данным Интернета.

Основные понятия

Для разработки и использования Web-служб в Delphi используются широко распространеный протокол HTTP и ставший своего рода стандартом обмена данными язык XML. Тем самым обеспечивается универсальная двусторонняя связь между клиентом и сервером. При этом от клиента не требуется использование определенной платформы или языка программирования.

Web-службы представляют собой модульные приложения, которые могут быть опубликованы и вызваны в Интернете. Возможен программный доступ к ним со стороны клиентских приложений.

Работа с Web-службами ведется с помощью протокола SOAP (Simple Object Access Protocol, простой протокол доступа к объектам), являющегося стандартом легковесного протокола обмена информацией в децентрализованной распределенной среде. Этот протокол использует язык XML для кодирования удаленных вызовов процедур и обычно использует HTTP в качестве коммуникационного протокола.

Компоненты, поддерживающие Web-службы в Delphi, ориентированы на использование протоколов SOAP и HTTP, но основа их является достаточно общей и позволяет использование других протоколов кодирования и коммуникации.

С помощью средств Delphi 7 можно создавать приложения-серверы Web-служб, а специальные компоненты и Мастера позволяют создавать клиенты для Web-служб, которые используют SOAP-кодирование.

Приложения Web-служб публикуют информацию о том, через какой интерфейс они доступны и как осуществлять их вызов, с помощью документов WSDL (Web Service Definition Language, язык определения Web-службы). На стороне сервера приложение может опубликовать документ WSDL, описывающий Web-службу. На стороне клиента Мастер или утилита командной строки позволяют импортировать опубликованный документ WSDL, обеспечивая клиента определения-

ми интерфейса и требуемой для связи информацией. Если уже имеется документ WSDL, описывающий предполагаемую к выполнению Web-службу, то можно сгенерировать код на стороне сервера как и при импортировании документа WSDL.

При создании клиентского приложения, использующего Web-службу, требуется иметь описание выполняемых ею действий и сведений, передаваемых Web-службе при обращении к ней. К примеру, клиенту требуется знать предоставляемые Web-службой методы, их параметры и используемые протоколы. В определенном смысле названная информация аналогична информации из библиотеки типов, используемой стандартным компонентом COM. Недостатком использования библиотеки типов Microsoft COM является то, что ее клиентами могут быть только приложения, работающие под управлением операционных систем фирмы Microsoft.

Использование Web-служб устраняет названный недостаток, при этом их услугами могут пользоваться клиентские приложения, являющиеся разработками любых фирм. Важно только, чтобы при обращении к серверу Web-службы использовались указанные выше протоколы и кодирование.

Документ WSDL

Как отмечалось, описывающий Web-службу документ WSDL кодируется на языке XML. Фрагменты кода XML WSDL-документа для примера приводятся ниже:

```
// ************************************************************************
//
// The types declared in this file were generated from data read from
// the WSDL File described below:
// WSDL: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?WSDL
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=sitesDataSet
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=eventGroupDataSet
...
// >Import: https://www.software-events.net/mSiteBuilderServices/
//   events.asmx?schema=revocationListDataSet
// Encoding : utf-8
// Version  : 1.0
// (13.11.2002 15:37:15 - 1.33.2.5)
//*************************************************************//
unit events;
interface
uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;
```

```
const
  sUDDIOperator   = 'http://test.uddi.microsoft.com/inquire';
  sUDDIBindingKey = 'f1471305-b656-405b-940c-688bce0f6d12';
type
//*****************************************************************//
// The following types, referred to in the WSDL document are not
// being represented in this file. They are either aliases[@] of
// other types represented or were referred to but never[!] declared
// in the document. The types from the latter category typically map
// to predefined/known XML or Borland types; however, they could also
// indicate incorrect WSDL documents that failed to declare or import
// a schema type.
//*****************************************************************//
  // !:boolean         - "http://www.w3.org/2001/XMLSchema"
  // !:int             - "http://www.w3.org/2001/XMLSchema"
  // !:string          - "http://www.w3.org/2001/XMLSchema"
  // !:dateTime        - "http://www.w3.org/2001/XMLSchema"
  // !:decimal         - "http://www.w3.org/2001/XMLSchema"
  // !:short           - "http://www.w3.org/2001/XMLSchema"

  loginState = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMySitesResult  = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMyGroupsResult = class;
                    { "http://www.software-events.net/mSiteBuilder" }
  GetMyVenuesResult = class;
                    { "http://www.software-events.net/mSiteBuilder" }
...
{ "http://www.software-events.net/mSiteBuilder" }
  statusType = (All, True, False);
  { "http://www.software-events.net/mSiteBuilder" }
  accessType = (All2, Owned, Shared, ReadOnly, Public);
//*****************************************************************//
  // Namespace : http://www.software-events.net/mSiteBuilder
//*****************************************************************//
  loginState = class(TRemotable)
  private
    FIsAuthenticated: Boolean;
    FLoggedInAs: WideString;
  published
    property IsAuthenticated: Boolean read FIsAuthenticated
                                          write FIsAuthenticated;
```

```
    property LoggedInAs: WideString read FLoggedInAs
                                          write FLoggedInAs;
  end;
...
//*****************************************************************//
// Namespace : http://www.software-events.net/mSiteBuilder
// soapAction: http://www.software-events.net/mSiteBuilder/
//   %operationName%
// transport : http://schemas.xmlsoap.org/soap/http
// style     : document
// binding   : mSiteBuilderSoap
// service   : mSiteBuilder
// port      : mSiteBuilderSoap
// URL       : https://www.software-events.net/mSiteBuilderServices/
//   events.asmx
//*****************************************************************//
  mSiteBuilderSoap = interface(IInvokable)
  ['{8B2B09D4-39E0-ECD4-F2A2-7243249A2B69}']
    function  Login(const mPUID: WideString): Boolean; stdcall;
    ...
    function  AddMyEventToGroup(const groupID: Integer;
                      const eventID: Integer; const
                      accessLevel: accessType): Boolean; stdcall;
  end;

function GetmSiteBuilderSoap(UseWSDL: Boolean=System.False;
       Addr: string=''; HTTPRIO: THTTPRIO = nil): mSiteBuilderSoap;

implementation

function GetmSiteBuilderSoap(UseWSDL: Boolean; Addr: string;
       HTTPRIO: THTTPRIO): mSiteBuilderSoap;
const
  defWSDL =
  'https://www.software-events.net/mSiteBuilderServices/events.asmx?WSDL';
  defURL  =
  'https://www.software-events.net/mSiteBuilderServices/events.asmx';
  defSvc  = 'mSiteBuilder';
  defPrt  = 'mSiteBuilderSoap';
var
  RIO: THTTPRIO;
begin
  Result := Nil;
  if (Addr = '') then
```

```
  begin
    if UseWSDL then
      Addr := defWSDL
    else
      Addr := defURL;
  end;
  if HTTPRIO = nil then
    RIO := THTTPRIO.Create(nil)
  else
    RIO := HTTPRIO;
  try
    Result := (RIO as mSiteBuilderSoap);
    if UseWSDL then
    begin
      RIO.WSDLLocation := Addr;
      RIO.Service := defSvc;
      RIO.Port := defPrt;
    end else
      RIO.URL := Addr;
  finally
    if (Result = Nil) and (HTTPRIO = Nil) then
      RIO.Free;
  end;
end;

initialization
  InvRegistry.RegisterInterface(TypeInfo(mSiteBuilderSoap),
       'http://www.software-events.net/mSiteBuilder', 'utf-8');
  ...
  RemClassRegistry.RegisterXSInfo(TypeInfo(statusType),
       'http://www.software-events.net/mSiteBuilder', 'statusType');
  RemClassRegistry.RegisterXSInfo(TypeInfo(accessType),
       'http://www.software-events.net/mSiteBuilder', 'accessType');

  RemClassRegistry.RegisterXSClass(GetRevocationListResult,
       'http://www.software-events.net/mSiteBuilder',
       'GetRevocationListResult');

end.
```

Как видно из приведенного кода, документ WSDL включает объявления классов, прототипы функций, объявления вызываемого интерфейса и другие элементы. В разделе `implementation` содержится описание функции с именем

Get<имя службы>, возвращающей вызываемый интерфейс. В приведенном листинге функция имеет имя GetmSiteBuilderSoap.

Дадим для объекта RIO короткие пояснения по поводу его свойств WSDLLocation, Service, Port типа string, которые используются в функции GetmSiteBuilderSoap для организации связи с Web-службой. Свойство WSDLLocation служит для указания адреса местоположения документа WSDL, свойства Service и Port содержат доступные по этому адресу наборы служб и портов соответствено. С помощью порта мы получаем доступ к конкретной Web-службе.

Вызываемый интерфейс

Серверы, поддерживающие Web-службы, построены с использованием *вызываемых интерфейсов*, которые скомпилированы для включения информации о типе времени выполнения (RunTime Type Information, RTTI). На стороне сервера эта информация используется при интерпретации поступающих вызовов методов от клиента, так чтобы они могли быть корректно введены. На стороне клиента информация о типе времени выполнения используется для того, чтобы динамически сгенерировать таблицу методов для подготовки вызова методов интерфейса.

Для создания вызываемого интерфейса требуется скомпилировать интерфейс с параметром компиляции {$M+}. Потомок любого вызываемого интерфейса также является вызываемым. Однако если вызываемый интерфейс происходит от другого интерфейса, не являющегося вызываемым, то Web-служба может использовать только методы, определенные в вызываемом интерфейсе и его потомках. Методы, наследуемые от невызываемых предков, не компилируются с информацией о типе и не могут использоваться как часть Web-службы.

При использовании вызываемого интерфейса автоматически поддерживается работа со следующими встроенными скалярными типами данных: Boolean, ByteBool, WordBool, LongBool, Char, Byte, ShortInt, SmallInt, Word, Integer, Cardinal, LongInt, Int64, Single, Double, Extended, String, WideString, Currency, TDateTime, Variant. При работе с ними от программиста не требуются никакие дополнительные действия. При необходимости использовать другие типы данных требуется выполнить их регистрацию с помощью реестра удаляемых типов.

Страница *WebServices* Палитры компонентов

На странице **WebServices** Палитры компонентов (рис. 22.1) содержатся следующие компоненты, обеспечивающие работу с Web-службами:

- HTTPRIO — использует сообщения HTTP для вызова объекта удаленного интерфейса с помощью протокола SOAP;

- `HTTPReqResp` — выполняет обращение к методу на вызываемом интерфейсе путем посылки сообщения SOAP на сервер;
- `OPToSoapDomConvert` — управляет загрузкой и выгрузкой методов вызова SOAP;
- `SoapConnection` — используется в клиентской части приложения распределенной базы данных для установления и обслуживания соединения между клиентом и удаленным сервером приложения, который выполнен как Web-служба;
- `HTTPSoapDispatcher` — отвечает на сообщения SOAP путем пересылки их программе вызова для интерпретации;
- `WSDLHTMLPublish` — выполняет публикацию списка документов WSDL, описывающих приложение Web-службы;
- `HTTPSoapPascalInvoker` — интерпретирует сообщение запроса SOAP и выполняет соответствующий вызываемый интерфейс.

Рис. 22.1. Страница **WebServices** Палитры компонентов

Рассмотрим кратко особенности использования перечисленных компонентов.

Компонент `HTTPRIO` служит для генерации статически связанных обращений к вызываемым интерфейсам на удаленном приложении Web-службы. Когда приложение назначает зарегистрированному вызываемому интерфейсу компонент `HTTPRIO`, он динамически генерирует в основной памяти таблицу методов, обеспечивающую реализацию этого вызываемого интерфейса. При этом компонент `HTTPRIO` выполняет методы в этой таблице методов путем кодирования вызова метода как запроса SOAP и посылки сообщения с запросом HTTP приложению Web-службы. Кроме того, он распаковывает результирующее сообщение с ответом HTTP, чтобы получить возвращаемое значение и любые выходные параметры или чтобы вызвать исключение, если запрос вызвал исключение на сервере.

Опубликованные свойства компонента `HTTPRIO` используют, чтобы указать, как связываться с приложением Web-службы. Есть два способа указать, где размещено приложение сервера:

- использовать свойство `URL`, чтобы определить URL, по которому расположено приложение сервера;
- для динамического использования информации о соединении из документа WSDL можно установить свойство `WSDLLocation` в нужное значение. После этого нужно выбрать значения для свойств `Service` и `Port` в раскрывающемся списке в окне Инспектора объектов, чтобы определить используемые связи.

Прежде чем компонент HTTPRIO сможет генерировать таблицу методов для вызываемого интерфейса, последний должен быть зарегистрирован в реестре вызовов. Для этого используются глобальная функция InvRegistry и ее метод RegisterInterface.

Компонент HTTPReqResp управляет основанной на протоколе HTTP связью с провайдером Web-службы по требованию компонента HTTPRIO. Компонент HTTPRIO использует этот компонент, чтобы установить соединение с провайдером Web-службы и выполнить следующие две задачи: выдать запрос GET для получения информации из документа WSDL; выдать запрос POST, чтобы передать метод вызова серверу для выполнения и получить обратно результаты.

При этом целевую Web-службу HTTP-сообщений можно задать напрямую с помощью свойств SoapAction и URL или использовать свойство SoapAction в документе WSDL, как определено свойством WSDLView. Дополнительные свойства обеспечивают детали, включенные в заголовок HTTP-запроса сообщения.

Компонент HTTPReqResp в Windows для установления соединения с сервером использует динамическую библиотеку WinInet. Отметим, что библиотека wininet.dll должна быть установлена в клиентской системе, в Windows он обычно размещается в папке System, если установлен Internet Explorer 3 и выше. В Windows 2000 эта библиотека размещена в папке System32.

Компонент OPToSoapDomConvert представляет собой реализацию интерфейса IOPConvert, управляющего загрузкой и выгрузкой методов вызова SOAP. Он использует парсеры (синтаксические анализаторы) DOM для проверки правильности кодирования методов вызова SOAP и их результатов.

Обычно приложения не используют компонент OPToSoapDomConvert напрямую. Вместо этого он создается и используется компонентом HTTPRIO в клиентских приложениях и компонентами HTTPSoapPascalInvoker или THTTPSoapCppInvoker — в серверных приложениях.

Компонент OPToSoapDomConvert выполняет преобразование между кодированием SOAP вызова метода или его результатами и содержимым (контекстом) вызова. Если вызов метода порождает исключение, то компонент OPToSoapDomConvert генерирует ошибочный пакет, так что клиент может вызвать соответствующее исключение удаленно.

Компонент SoapConnection позволяет:

- устанавливать начальное соединение с удаленным сервером приложения с помощью SOAP;
- получать интерфейс для сервера приложения;
- получать список провайдеров на сервере приложения;
- отменять соединение с удаленным сервером приложения.

Для использования компонента SoapConnection в Windows, как и в случае компонента HTTPReqResp, должна быть установлена библиотека wininet.dll в клиентской системе.

Компонент `SoapConnection` использует внутренний объект `HTTPRIO` чтобы получить интерфейс от приложения Web-службы. Им может быть интерфейс `IAppServer` или `IAppServerSOAP`, или один из их потомков в зависимости от значения свойства `UseSOAPAdapter`. Рекомендуется использовать интерфейс `IAppServerSOAP` как наиболее адаптированный для протокола SOAP.

Некоторые приложения сервера, например, создаваемые с помощью Delphi 6 или Kylix 2, поддерживают только интерфейс `IAppServer`. В этом случае свойство `UseSOAPAdapter` компонента `SoapConnection` следует установить в значение `False`.

Независимо от используемого для связи с сервером приложения интерфейса компонент `SoapConnection` делает доступной информацию клиентским наборам данных в приложении клиента посредством обеспечения интерфейса `IAppServer`. Клиентские наборы данных используют интерфейс `IAppServer` из компонента `SoapConnection` для связи с провайдерами на сервере приложения или, наоборот, для вызова интерфейса модуля данных сервера приложения.

Компонент `HTTPSoapDispatcher` используют в приложении, публикующем Web-службу с применением протокола SOAP. Этот компонент отвечает на все обращения к вызываемым интерфейсам, зарегистрированным в приложении. Регистрация их выполняется с помощью метода `RegisterInterface` реестра вызовов `InvRegistry`.

Рассматриваемый компонент `HTTPSoapDispatcher` действует как диспетчер, принимая поступающие сообщения и направляя их другому компоненту, который управляет задачей их интерпретации и обработки. Названный компонент идентифицируется с помощью свойства `Dispatcher`. Отметим, что диспетчер представляет собой интерфейс вызывающего компонента, который интерпретирует сообщение SOAP, идентифицирует вызываемый интерфейс, являющийся целью вызова, выполняет вызов и формирует содержимое ответного сообщения.

Компонент `HTTPSoapDispatcher` автоматически регистрирует себя вместе с модулем или диспетчером Web как автодиспетчируемый объект. Это означает, что модуль или диспетчер Web перенаправляет все поступающие сообщения, не требуя применять элементы Web-действия.

Компонент `WSDLHTMLPublish` помещают в Web-модуль приложения Web-службы, чтобы опубликовать список документов WSDL, описывающих приложение Web-службы. Публикуемые компонентом `WSDLHTMLPublish` определения выводятся из всех вызываемых интерфейсов, зарегистрированных с помощью реестра вызовов `InvRegistry`, как и из всех удаленных классов и типов, зарегистрированных в реестре удаленных типов `RemTypeRegistry`.

По умолчанию адрес, по которому установлено приложение, содержащее компонент `WSDLHTMLPublish`, используется как адрес, где эти службы доступны. В случае реализации Web-службы в другом месте, установив свойство `AdminEnabled` компонента в значение `True` и запустив WSDL-администратор из Web-обозревателя, можно изменить адреса, сгенерированные в документах WSDL.

Компонент `WSDLHTMLPublish` автоматически регистрирует себя с Web-модулем или Web-диспетчером как автодиспетчируемый объект. Это означает, что Web-модуль или Web-диспетчер пересылает все приходящие сообщения, не требуя применять элементы Web-действия. Эти сообщения включают запросы на список документов WSDL и запросы к WSDL-администратору.

Компонент `TTPSoapPascalInvoker` используют в приложении, которое публикует Web-службу с применением протокола SOAP. Этот компонент получает поступающий запрос SOAP от компонента `HTTPSoapDispatcher`, анализирует его, генерирует соответствующее обращение к зарегистрированному вызываемому интерфейсу и кодирует результаты вызова этого интерфейса.

В среде Delphi `HTTPSoapPascalInvoker` используют непосредственно как вызывающий компонент в приложении Web-службы. Этот компонент реализует интерфейс `IHTTPSoapDispatch`, который диспетчер использует для передачи входящих к нему запросов SOAP и приема содержимого ответного сообщения. Чтобы обеспечить диспетчеру возможность использовать этот интерфейс, нужно задать для свойства `Dispatcher` компонента `HTTPSoapPascalInvoker` значение `THTTPSoapDispatcher`.

Схема взаимодействия клиента и сервера

Схематично процесс взаимодействия клиента и сервера Web-службы приведен на рис. 22.2. Как видно из приведенной схемы, для того чтобы обратиться к серверу Web-службы, клиент с помощью программы импорта WSDL-документа предварительно должен получить описание требуемой службы в виде файла .pas.

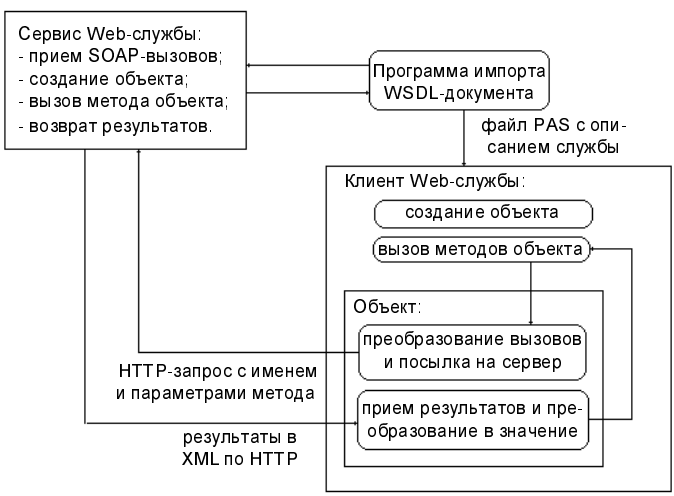


Рис. 22.2. Схема процесса взаимодействия клиента и сервера

На основе названного описания клиент Web-службы создает объект, который является своего рода посредником между клиентом и сервером при обращении именно к той службе, для которой он был создан. Функции объекта указаны на рис. 22.2.

Приведенная схема процесса взаимодействия между клиентом и сервером Web-службы является весьма упрощенной, поэтому на ней, в частности, не получили отображения различные способы передачи параметров.

Разработка клиента для Web-службы

В этой главе мы ограничимся рассмотрением вопроса создания клиента Web-службы. Во-первых, нам может и не потребоваться заниматься созданием собственного сервера Web-службы, а будет достаточно обеспечивать доступ к уже имеющимся службам. Во-вторых, создать приложение-клиент проще, нежели сервер. В-третьих, создание и использование клиента позволяет прояснить смысл программирования и использования Web-служб.

Для разработки приложения-клиента, осуществляющего доступ к определенной в WSDL-документе Web-службе, требуется выполнить следующие шаги:

1. Импортирование определений из документа WSDL.
2. Получение вызываемого интерфейса и вызов его для доступа к Web-службе.
3. Может потребоваться обработка заголовков сообщений SOAP, передаваемых между клиентом и сервером.

Рассмотрим указанные шаги более подробно.

Импортирование документа WSDL

Для вызова программы импортирования документа WSDL нужно выбрать команду **File\New\Other** и на странице **WebServices** Хранилища объектов сделать двойной щелчок на значке **WSDL Importer**. В открывшемся диалоговом окне Мастера импорта WSDL, если известен адрес размещения документа WSDL в Интернете, то следует указать его URL в поле **Location of WSDL file or URL**, нажать кнопку **Next** и после установления связи с провайдером импортировать его из Интернета. В противном случае с помощью кнопки **Search UDDI** откроем диалоговое окно **UDDI Browser** (рис. 22.3).

В раскрывающемся списке **Registry** нужно выбрать имя UDDI-реестра, который планируется использовать. При этом можно выбрать из списка общий (public) реестр или указать URL частного реестра.

В поле **Business Name** следует указать наименование отрасли бизнеса, которой принадлежит импортируемая Web-служба, например **software**.

В списке **Sort** следует выбрать вариант сортировки результатов импорта документов.

Далее, нажав кнопку **Find**, инициируем поиск документа WSDL в Интернете с заданными параметрами. Дерево результатов поиска отображается в правой части

диалогового окна **UDDI Browser** на вкладке **Results** (см. рис. 22.3). Здесь требуется выбрать подходящий вариант точки входа и после активизации кнопки **Import WSDL** сделать щелчок мышью. В результате должна происходить загрузка документа WSDL, описывающего выбранную Web-службу. В действительности вместо успешной загрузки порой выдается сообщение об ошибке (документ пуст).

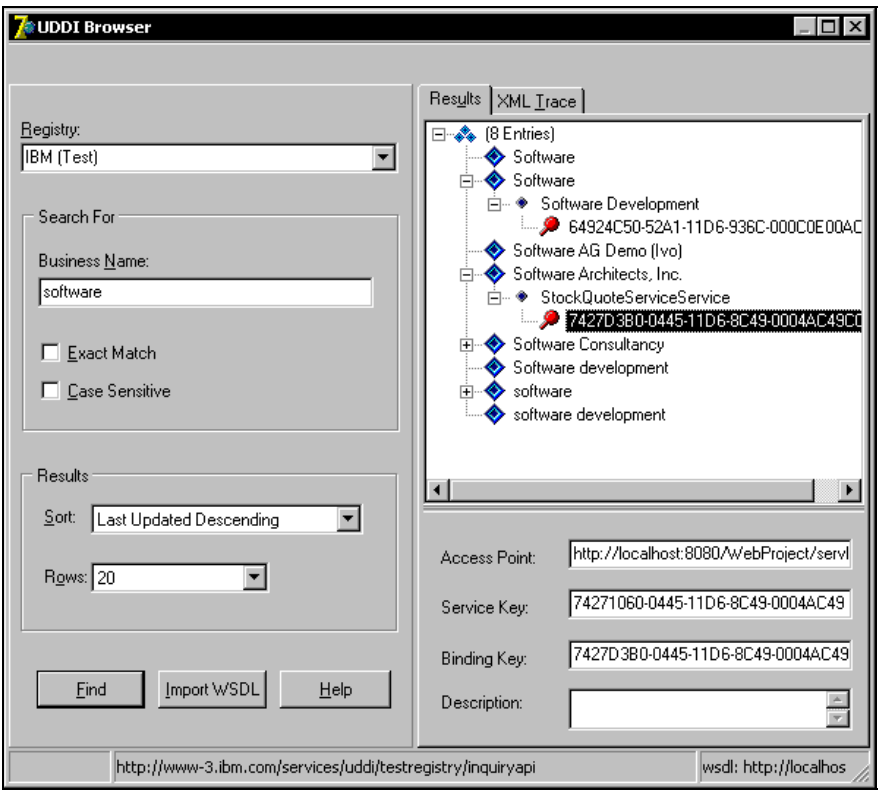


Рис. 22.3. Диалоговое окно **UDDI Browser**

В случае успешной загрузки документа WSDL Мастер импорта выполняет его разбор и генерирует соответствующий модуль с кодом описания службы, помещая его в файл с раширением pas. Вид кода в правой части диалогового окна Мастера импорта приведен на рис. 22.4. В левой части этого окна расположено дерево типов и интерфейсов, имеющихся в документе WSDL. Работа с Мастером импорта документов WSDL завершается нажатием кнопки **Finish**.

Замечание

UDDI (Universal Description, Discovery and Integration, универсальное описание, обнаружение и объединение) представляет собой общий формат, используемый для регистрации Web-служб в Интернете.

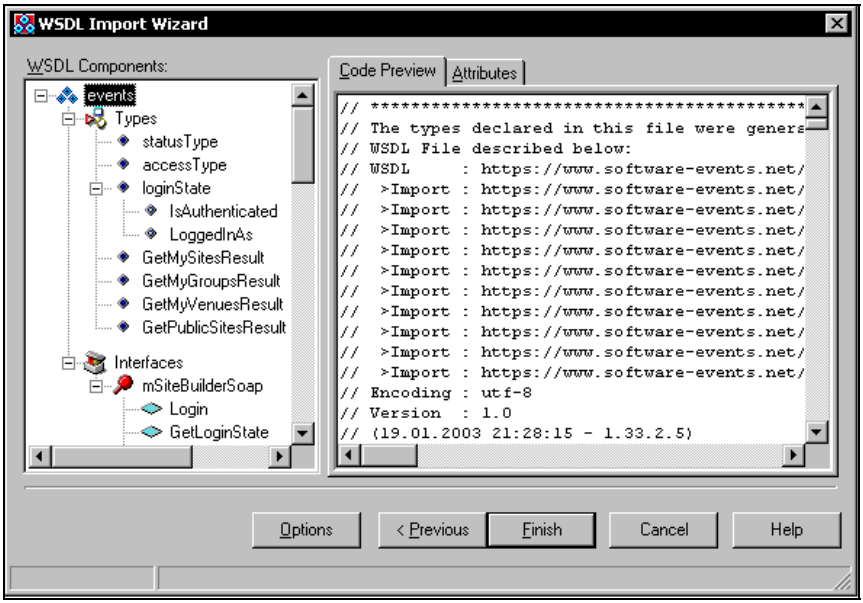


Рис. 22.4. Диалоговое окно **UDDI Browser** с компонентами документа WSDL

Обращение к вызываемому интерфейсу

Для *обращения к вызываемому интерфейсу* клиентское приложение должно включать все определения вызываемых интерфейсов и все удаляемые классы, реализующие сложные типы. Если сервер приложения Web-службы написан на Delphi, то можно использовать те же модули, которые сервер приложения использовал для определения и регистрации соответствующих интерфейсов и классов вместо файлов, генерируемых при импортировании документа WSDL. При этом нужно, чтобы модуль использовал то же самое пространство имен URI и заголовок SOAP Action при регистрации вызываемых интерфейсов. Эти значения можно или просто специфицировать в коде регистрации интерфейсов, или этот код может быть сгенерирован автоматически. Во втором случае определяющий интерфейс модуль должен иметь одно и то же имя у клиента и сервера, кроме того, клиент и сервер должны задать глобальной переменной `AppNameSpacePrefix` одинаковое значение.

При наличии определения вызываемого интерфейса экземпляр для вызова можно получить двумя описанными ниже способами.

При импортировании документа WSDL программа импортирования автоматически генерирует глобальную функцию, которая возвращает интерфейс и которую можно вызвать. Например, при импортировании документа WSDL, определяющего вызываемый интерфейс с именем `mSiteBuilderSoap` с помощью строки

```
mSiteBuilderSoap = interface(IInvokable)
```

сгенерированный модуль будет включать глобальную функцию с прототипом вида:

```
function GetmSiteBuilderSoap(UseWSDL: Boolean; Addr: string;
      HTTPRIO: THTTPRIO): mSiteBuilderSoap;
```

Если параметр `UseWSDL` имеет значение `True`, то местоположение сервера определяется с помощью документа WSDL, в противном случае клиентское приложение предоставляет URL Web-службы сервера с помощью параметра `Addr`. Если `UseWSDL` имеет значение `True`, то параметр `Addr` указывает URL документа WSDL, описывающего Web-службу.

При задании пустой строки для второго параметра названной функции по умолчанию будет использоваться импортированный документ. Такой подход предпочтителен в случаях возможного изменения URL для Web-службы или отдельных составляющих. Здесь информация просматривается динамически при вызове метода приложением.

Сгенерированная функция использует внутренний удаленный интерфейсный объект для реализации вызываемого интерфейса. Если применяется эта функция и выявлена необходимость доступа к лежащему в ее основе удаленному интерфейсному объекту, то интерфейс типа `IRIOAccess` можно получить из вызываемого интерфейса и использовать его для доступа к удаленному интерфейсному объекту:

```
var
  Interf: IServerInterface;
  RIOAccess: IRIOAccess;
  InterfObj: THTTPRIO;
begin
  Interf := GetIServerInterface(True,
          'http://MyServices.org/scripts/AppServer.dll/wsdl');
  RIOAccess := Interf as IRIOAccess;
  InterfObj := RIOAccess.RIO as THTTPRIO;
```

В случае использования удаленного интерфейсного объекта можно создать экземпляр (объект) типа `THTTPRio` для нужного интерфейса:

```
InterfObj := THTTPRio.Create(Nil);
```

Важно, что не требуется явно удалять интерфейсный объект типа `THTTPRio`. Если объект создан без владельца (Owner), как в последней строке кода, то он автоматически становится свободным при освобождении его интерфейса. Если он создан с помощью владельца, то владелец несет ответственность за освобождение экземпляра типа `THTTPRio`.

При наличии экзепляра типа `THTTPRio` следует снабдить его информацией, требуемой для идентификации интерфейса и размещения сервера. Это можно выполнить двумя путями.

- Если не предполагается, что URL, пространство имен или заголовок SOAP Action потребуется изменять, то можно просто задать URL для Web-службы,

к которой нужен доступ. Объект типа `THTTPRio` использует этот URL для поиска определения интерфейса, а также информации о пространстве имен и заголовке на основе информации в реестре вызовов. URL задается путем установки свойства `URL` для размещения сервера:

```
InterfObj.URL := 'http://www.myco.com/MyService.dll/SOAP/IServerInterface';
```

- Если предполагается динамическое определение URL, пространства имен или заголовка SOAP Action в процессе выполнения, то можно использовать свойства `WSDLLocation`, `Service` и `Port`. Это обеспечит извлечение необходимой информации из документа WSDL:

```
InterfObj.WSDLLocation := 'Cryptography.wsdl';
InterfObj.Service := 'Cryptography';
InterfObj.Port := 'SoapEncodeDecode';
```

После определения способа задания размещения сервера и идентификации интерфейса можно получить указатель интерфейса для вызываемого интерфейса из объекта типа `THTTPRio`. Для этого используется оператор `as`. Просто распределяется экземпляр объекта типа `THTTPRio` на вызываемый интерфейс:

```
InterfaceVariable := InterfObj as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
```

При получении указателя интерфейса объект типа `THTTPRio` создает `vtable` (массив указателей на методы — таблица виртуальных функций, ТВФ) для ассоциированного интерфейса динамически в памяти, поддерживая тем самым возможность вызовов интерфейса.

Объект типа `HTTPRio` для получения информации о вызываемом интерфейсе полагается на реестр вызовов. Если у клиентского приложения нет реестра вызовов или вызываемый интерфейс не зарегистрирован, то объект типа `HTTPRio` не может создать таблицу виртуальных функций (ТВФ) в памяти.

Если интерфейс, получаемый от объекта типа `HTTPRio`, назначен глобальной переменной, то перед завершением приложения нужно установить это назначение в `Nil`. К примеру, если `InterfaceVariable` из предыдущего примера есть глобальная переменная, а не переменная стека, то перед тем, как объект типа `HTTPRio` будет свободен, требуется освободить интерфейс. Обычно соответствующий код помещают в обработчик события `OnDestroy` формы или модуля данных:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  InterfaceVariable := Nil;
end;
```

Необходимость задания значения `Nil` для глобальной интерфейсной переменной обусловлена тем, что объект типа `HTTPRio` создает таблицу виртуальных функций в памяти динамически. Причем, таблица виртуальных функций должна еще присутствовать, когда интерфейс освобожден. Если интерфейс не освобожден вместе с формой или с модулем данных, то он освобождается, когда ос-

вобождается глобальная переменная при завершении приложения. Память для глобальных переменных может быть освобождена после освобождения формы или модуля данных, содержащих объект типа `HTTPRio`, в этом случае таблица виртуальных функций не будет доступной, когда интерфейс свободен.

Пример создания клиента Web-службы

Рассмотрим пример создания клиента для Web-службы, размещенной по адресу **https://www.software-events.net/mSiteBuilderServices/events.asmx?WSDL**. При этом предположим, что предварительно мы создали проект приложения Delphi с пока еще пустой формой.

1. Выполним импортирование WSDL-документа: выберем команду **File\New\Other** и на странице **WebServices** Хранилища объектов сделаем двойной щелчок на значке **WSDL Importer**. В открывшемся диалоговом окне Мастера импорта WSDL в поле **Location of WSDL file or URL** укажем адрес размещения документа WSDL в сети Интернет, нажмем кнопку **Next** и после установления связи с провайдером импортируем его.

   Замечание

   Напомним, что фрагменты документа WSDL для этой службы приведены в начале этой главы. Из текста документа видно, что вызываемый интерфейс имеет имя `mSiteBuilderSoap`, а автоматически сформированная глобальная функция — имя `GetmSiteBuilderSoap`. В описании интерфейса содержатся функции (`Login`, `GetLoginState`, `GetMySites` и др.), которые можно использовать при обращении к вызываемому интерфейсу Web-службы.

2. Для организации обращения создаваемого нами клиента к Web-службе поместим в форму кнопку и создадим для нее обработчик события `OnClick`, код которого приведен ниже.

```
procedure TForm1.Button1Click(Sender: TObject);
var Port: mSiteBuilderSoap;
begin
try
Port := GetmSiteBuilderSoap;
 ShowMessage(BoolToStr(Port.Login('kkkk')));
 finally
 Port := Nil;
end;
end;
end.
```

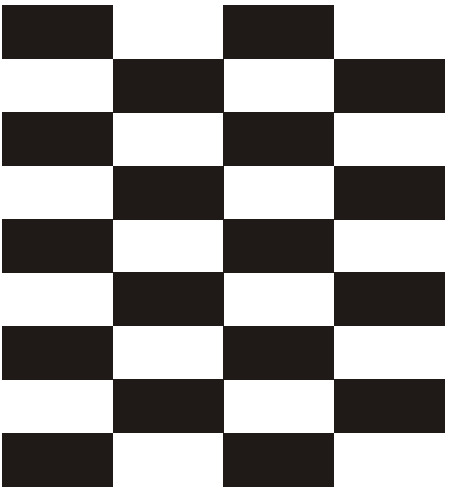
В приведенном коде переменной `Port` типа `mSiteBuilderSoap` присваивается значение глобальной функции `GetmSiteBuilderSoap`, выполняющей обращение к вызываемому интерфейсу. Далее с помощью функции `ShowMessage` отображается результат вызова функции `Login`, возвращающей значение типа `Boolean` и

указывающей тем самым на возможность регистрации. При этом используется функция `BoolToStr`, преобразующая его в тип `String` (строка `'kkkk'` в качестве аргумента при обращении к функции взята произвольно).

Теперь при запуске созданного нами приложения-клиента после нажатия кнопки будет происходить открытие диалогового окна установления связи с провайдером, обращение к Web-службе и выдача сообщения о возможности регистрации (Login). При первом нажатии нашей кнопки появится окно сообщения (рис. 22.5) с значением `-1` (соответствует значению `True`), при всех последующих — с значением `0` (соответствует значению `False`).

Рис. 22.5. Вид окна сообщения

Описанным способом можно организовать обращение к любой функции в составе вызываемого интерфейса рассматриваемой Web-службы.



ЧАСТЬ V

DELPHI 2005

Глава 23

Работа с БД в Delphi 2005

Общая характеристика Delphi 2005

Система программирования Delphi 2005, как и ее предшественник Delphi 8, ориентирована на использование Microsoft .NET. Основу Microsoft .NET составляет .NET Framework, которая представляет собой платформу для разработки и исполнения приложений. В ее состав входят следующие основные компоненты:

- CLR (Common Language Runtime) — общеязыковая исполняемая среда;
- FCL (.NET Framework Class Library) — библиотека классов .NET Framework.

Среда CLR управляет выполнением кода приложений и обеспечивает службы (services), упрощающие процесс разработки приложений. В качестве примера можно назвать службы управления памятью и многоязыковой поддержки.

Библиотека классов FCL содержит множество объектно-ориентированных компонентов, служащих для разработки приложений .NET, позволяющих использовать все достоинства применения служб среды CLR.

Среда Microsoft .NET Framework обеспечивает *многоязыковую совместимость* приложений. Благодаря общеязыковой исполняемой среде CLR приложение .NET может быть написано на любом .NET-совместимом языке программирования высокого уровня, к примеру, Visual Basic .NET и Visual C# .NET. Естественно, что теперь здесь может использоваться язык Delphi.

При создании с помощью компилятора для платформы .NET модуля (двоичного файла) с расширением dll или exe его содержимое представляет собой *сборку* на промежуточном языке Microsoft Intermediate Language (MSIL). Кроме команд на языке MSIL, двоичный модуль .NET содержит метаданные, которые предназначены для описания всех типов в этом модуле (классов, структур, перечислений и т. д.).

В сборку может входить произвольное количество типов. Для указания определенного типа требуется обеспечить уникальность имен по отношению к другим типам в сборке. Для решения этой проблемы и предназначены пространства имен, выполняющие группирование стандартных типов и дающие возможность их использования при разработке собственных типов.

Пространство имен представляет собой логическую структуру, предназначенную для организации используемых в приложении имен с целью предотвращения

конфликтов между именами из разных сборок. Например, пусть в приложении .NET есть обращения к двум разным внешним сборкам. Если при этом в каждой из них имеется тип с одним и тем же именем, то по своим функциональным возможностям эти типы различны, и смешивать их недопустимо. Следовательно, при программировании нужно указать, к какому типу и из какой сборки выполняется обращение. Для этого достаточно к имени типа добавить наименование нужного пространства имен.

Интерфейс системы программирования Delphi 2005, хотя и претерпел определенные изменения по сравнению с Delphi 7, но техника конструирования приложений по большому счету осталась той же. Первоначально выбирается тип разрабатываемого приложения. Затем с помощью Палитры компонентов выполняется перенос нужных компонентов в Конструктор. Отметим, что визуальные компоненты помещаются в Конструктор непосредственно на форму приложения, а невизуальные компоненты помещаются в нижней части (поддон компонентов — component tray) Конструктора, расположенной непосредственно под формой. Таким образом, при конструировании приложения видно, какие компоненты являются визуальными, а какие невизуальными.

С помощью Инспектора объектов настраиваются нужные свойства компонентов, задаются обработчики событий для соответствующих компонентов. Далее по команде **Run/Run** приложение запускается на компиляцию и выполнение.

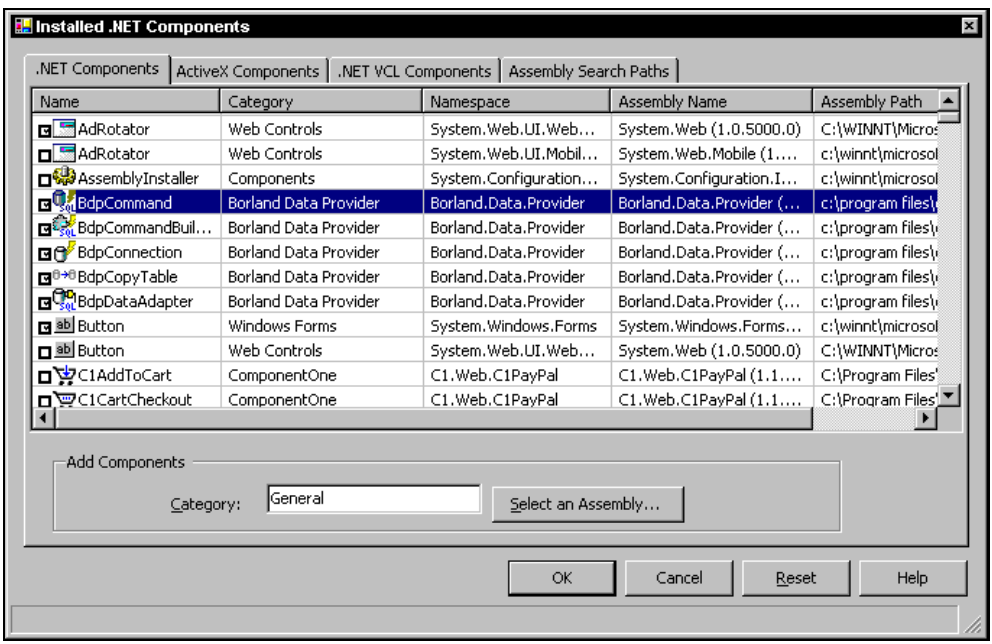


Рис. 23.1. Диалоговое окно **Installed .NET Components**

Для разработки приложения в среде Delphi 2005 можно использовать компоненты из библиотек .NET Framework и VCL, допуская разное их сочетание. При конструировании приложения разработчик может просмотреть, какие компоненты из библиотек .NET Framework и VCL доступны для использования, и выполнить настройку состава нужных страниц Палитры компонентов. Сделать это можно с помощью диалогового окна **Installed .NET Components** (рис. 23.1), вызываемого по одноименной команде меню **Component**.

Например, чтобы включить/исключить определенный компонент (принадлежащий библиотеке .NET Framework) на некоторой странице Палитры компонентов, нужно на вкладке **.NET Components** окна (см. рис. 23.1) в строке с именем этого компонента и категорией (совпадает с названием страницы) установить/сбросить флажок.

Заметим, что состав страниц Палитры компонентов существенно зависит от вида разрабатываемого приложения.

Характеристика VCL для .NET

Библиотека VCL для .NET представляет собой платформу программирования для разработки приложений в среде Delphi 2005 с помощью VCL. С помощью Delphi 2005 и VCL для .NET можно разрабатывать новые приложения, а также переносить имеющиеся приложения Win32 на платформу .NET, причем с сохранением большей части кода.

В среде Delphi 2005 поддерживается разработка приложений на платформе .NET с помощью языка Delphi, а также с использованием элементов управления VCL для .NET и элементов управления Windows Forms. Схематично это показано на рис. 23.2. Как видно на приведенной схеме, в Delphi 2005 поддерживается создание приложений для работы с базами данных для всех известных нам по предыдущей версии системы технологий доступа, но применительно к платформе .NET (ADO.NET, dbExpress .NET, BDE.NET, IBX.NET, DataSnap .NET). В случае использования ADO.NET применяются провайдеры данных фирмы Borland — BDP.NET.

Библиотека VCL для .NET представляет собой большое подмножество наиболее общих классов в составе библиотеки VCL для Win32.

Платформа .NET разработана так, чтобы обеспечить согласование с любым языком, входящим в ее состав. Поэтому код Delphi, оперирующий с классами и функциями VCL для Win32, во многих случаях может быть перекомпилирован на платформу .NET с минимальными изменениями.

Классы библиотеки VCL для .NET принадлежат пространству имен Borland.Vcl. При этом классы, связанные с базами данных, размещены в пространстве имен Borland.Vcl.DB. Библиотека классов времени выполнения (Runtime library classes) находится в пространстве имен Borland.Vcl.Rtl. Файлы модулей (unit) были собраны в соответствующие пространства имен Borland.Vcl. Некоторые

модули были перемещены, тем не менее пространства имен позволяют облегчить поиск объектов, требуемых для достижения нужной функциональности приложения. Файлы с исходным кодом для всех объектов Delphi 2005 размещены в подкаталоге c:\Program Files\Borland\BDS\3.0\Source.

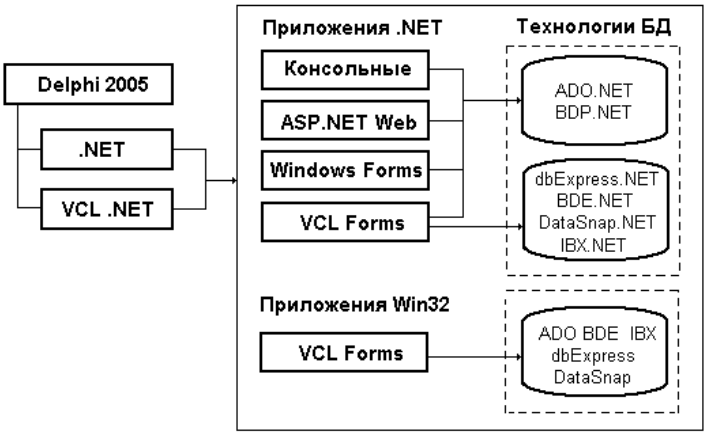


Рис. 23.2. Варианты приложений Delphi 2005

Разработка приложения VCL Forms

Характеристика способов

Разработка приложений VCL Forms в среде Delphi 2005 осуществляется для двух вариантов:

- для платформы .NET;
- для платформы Win32.

Делается это соответственно по командам **File/New/VCL Forms Application — Delphi for .NET** и **File/New/VCL Forms Application — Delphi for Win32**. При задании первой из указанных команд на Палитре компонентов становятся доступны страницы, приведенные на рис. 23.3. В случае задания второй из названных команд на Палитре компонентов присутствуют все те же страницы (см. рис. 23.2). Кроме того, дополнительно появляются страницы **COM+**, **Servers**, **Web Services**, **Internet Express** и **WebSnap**.

Заметим, что, хотя в случае задания любого из названных вариантов команд мы имеем повторяющиеся во многом наименования страниц, но находящиеся на этих страницах компоненты в большинстве случаев принадлежат разным пространствам имен со всеми вытекающими последствиями.

Рис. 23.3. Состав Палитры компонентов для VCL Forms

Приложения БД VCL Forms для .NET

При создании приложений для работы с базами при задании команды **VCL Forms Application — Delphi for .NET** доступны следующие технологии доступа к данным (см. рис. 23.2):

- dbExpress.NET — компоненты доступа к данным расположены на странице **dbExpress** Палитры компонентов и принадлежат пространствам имен Borland.Vcl.SqlExpr и Borland.Vcl.SimpleDS;
- BDE.NET — компоненты доступа к данным расположены на странице **BDE** Палитры компонентов и принадлежат пространству имен Borland.Vcl.DBTables;

- IBX.NET — компоненты доступа к данным расположены на странице **InterBase** Палитры компонентов и принадлежат пространствам имен Borland.Vcl.IBTable, Borland.Vcl.IBQuery, Borland.Vcl.IBProc и т. п.;
- ADO.NET — компоненты доступа к данным расположены на странице **dbGo** Палитры компонентов и принадлежат пространству имен Borland.Vcl.ADODB.

Приложения БД VCL Forms для Win32

При создании приложений для работы с базами при задании команды **VCL Forms Application — Delphi for Win32** доступны следующие технологии доступа к данным (см. рис. 23.2):

- dbExpress — компоненты доступа к данным расположены на странице **dbExpress** Палитры компонентов и принадлежат пространству имен Classes.TPersistent;
- BDE — компоненты доступа к данным расположены на странице **BDE** Палитры компонентов и принадлежат пространству имен Controls.TControl;
- IBX — компоненты доступа к данным расположены на странице **InterBase** Палитры компонентов и принадлежат пространству имен Classes.TPersistent и Controls.TControl;
- ADO — компоненты доступа к данным расположены на странице **dbGo** Палитры компонентов и принадлежат пространству имен Controls.TControl.

Для примера отметим, что при доступе по технологии ADO может применяться аналогичный набор компонентов, что и в случае Delphi 7, для задания набора данных (`ADODataSet`), для установления соединения (`ADOConnection`), для отображения данных (`dbGrid`), для задания источника данных (`DataSource`).

В частности, описанный в *10 главе* пример применения технологии ADO оказывается полностью работоспособным в Delphi 2005 при использовании аналогичных компонентов (`ADOConnection1`, `ADODataSet1`, `DataSource1` и `DBGrid1`) и тех же настройках. В связи с этим мы не приводим вид формы приложения и тексты обработчиков событий, связанных с работой приложения.

Технология ADO.NET

Общая характеристика

ADO.NET (ActiveX Data Object .NET) содержит набор классов, используемый для доступа к источникам данных в платформе .NET. ADO.NET представляет собой новую объектную модель, которая использует стандарт XML для передачи данных. В ADO.NET реализована идея использования *отсоединенных* наборов данных, причем такой способ работы является основным. По сравнению с ADO, ADO.NET предполагает более легкое программирование, лучшую производительность и масштабирование, меньшую зависимость от источников данных.

ADO.NET можно использовать для доступа к реляционным СУБД, таким как SQL Server 2000, и ко многим дополнительным источникам данных, для работы с которыми предназначен провайдер OLE DB.

Одно из ключевых новшеств ADO.NET — замена ADO-объекта `RecordSet` комбинацией объектов `DataTable`, `DataSet`, `DataAdapter` и `DataReader`. Объект `DataTable` представляет набор (`collection`) записей отдельной таблицы и в этом отношении аналогичен `RecordSet`. Объект `DataSet` представляет набор объектов `DataTable`, а также содержит отношения и ограничения, используемые при связывании таблиц. На самом деле `DataSet` — это хранящаяся в памяти реляционная структура данных со встроенной поддержкой XML.

В ADO.NET поддержка реляционной модели осуществляется двумя способами. *Первый* способ реализует подсоединенную модель доступа к данным, в которой функционируют стандартные методы доступа к реляционной базе данных, включая поддержку параметрических запросов, хранимых процедур, SQL-операторов пакетного выполнения и транзакций. Этот способ полностью аналогичен способам, применяющимся в известных интерфейсах доступа к данным, таких как OLE DB, ODBC, JDBC.

При *втором* способе доступ к данным осуществляется в *отсоединенном* режиме с помощью объекта `DataSet`, имитирующего базу данных. К данным, содержащимся в этом объекте, применимы все операции, характерные для баз данных. Объект `DataSet` может содержать любое количество реляционных таблиц, отношений и ограничений и даже позволяет получать подмножество таблиц с помощью SQL-подобного языка запросов. Кроме того, в отсоединенную модель входит объект `DataAdapter`, который играет роль связующего класса между реальной базой данных и объектом `DataSet`.

Технология ADO.NET создана для использования в управляемых проектах. Предыдущая технология ADO основана на технологии COM и при использовании из управляемых приложений требует дополнительных затрат на выполнение кода. К тому же ADO имеет меньшие возможности при работе с отключенными наборами данных и XML. Например, в ADO непросто сохранить изменения, произведенные в отключенном курсоре. Рассмотрим некоторые *преимущества* ADO.NET в сравнении с ADO.

Масштабируемость. При использовании объекта `DataSet` работа происходит с отсоединенными наборами данных. Это означает, что вы используете соединение с источником данных очень короткое время. Во многих системах количество подключений к базам данных является самым узким местом в плане масштабируемости. Для таких систем ADO.NET является хорошим решением, резко повышающим их масштабируемость.

Независимость от источника данных. В ADO возможности объекта `RecordSet` сильно зависели от используемого источника данных. Теоретически ADO обеспечивал доступ к данным независимо от источника данных, на практике всегда необходимо было иметь хорошее представление о возможностях провайдера.

В ADO.NET `DataSet` действительно не зависим от источника данных, и изменение провайдера, с помощью которого заполняется `DataSet`, не влияет на функциональность `DataSet`.

Способность к взаимодействию. Так как ADO.NET использует XML как стандартный формат передачи данных, программа, которой необходимо получить данные из компонента ADO.NET, не обязана сама быть компонентом ADO.NET. В общем случае она вообще может не быть Windows-программой. Единственное требование — эта программа должна понимать XML. И это позволяет компонентам ADO.NET при использовании других компонентов и служб легко взаимодействовать с любой программой на любой платформе.

Компоненты ADO.NET существуют в пяти главных пространствах имен в **библиотеке классов .NET**. Укажем назначение этих пространств имен.

- **System.Data** — содержит фундаментальные классы с базовой функциональностью ADO.NET. В их число входят классы `DataSet` и `DataRelation`, которые позволяют манипулировать структурированными реляционными данными.
- **System.Data.Common** — эти классы применяются другими классами в пространствах имен System.Data.SqlClient и System.Data.OleDb, которые являются их наследниками и предоставляют конкретные версии, настроенные для поставщиков SQL Server и OLE DB.
- **System.Data.OleDb** — содержат классы, которые используются для соединения с поставщиком OLE DB, включая `OleDbCommand` и `OleDbConnection`.
- **System.Data.SqlClient** — содержит классы, которые используются для соединения с базой данных Microsoft SQL Server. Эти классы предоставляют такие же свойства и методы аналогично классам из System.Data.OleDb. Отличие состоит в том, что они оптимизированы для SQL Server.
- **System.Data.SqlTypes** — содержит структуры для специфических типов данных SQL Server, например, `SqlMoney` и `SqlDateTime`. Эти типы не являются обязательными и используются для работы с типами данных SQL Server без необходимости конвертировать их в стандартные эквиваленты .NET. Использование таких структур позволяет увеличить производительность обмена данными за счет устранения автоматического преобразования данных.

Базовые объекты ADO.NET можно разбить на две группы: первая используется для хранения и управлениями данными (классы `DataSet`, `DataTable`, `DataRow` и `DataRelation`), вторая группа — для обеспечения соединения с источниками данных (классы `Connections`, `Commands` и `DataReader`). Объекты ADO.NET второй группы существуют в двух видах: для связи с поставщиками OLE DB и для взаимодействия с SQL Server.

Объекты данных позволяют хранить локальные данные, не имеющие связи с источником. Кроме того, они не хранят соединение с источником данных и, следовательно, их можно создавать самостоятельно, не используя базу данных. Объект `DataSet` не имеет непосредственного соединения с источником данных, и его можно создавать без обращения к базе данных.

Схема доступа к данным

Одна из схем доступа к данным с помощью ADO.NET представлена на рис. 23.4. Необходимая пользователю информация находится в базе данных. Обрабатываемые данные размещаются в объекте `DataSet`, не имеющем соединения с источником данных. Тем самым достигается возможность работы с данными в течение длительного промежутка времени. На рис. 23.4 серым цветом под пунктирной линией для наглядности выделены неподсоединенные объекты.

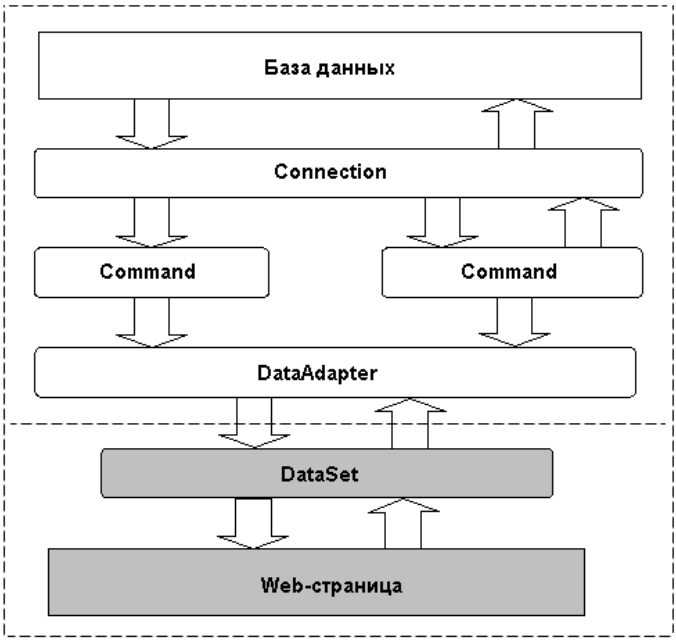


Рис. 23.4. Схема доступа к данным с помощью ADO.NET

Отметим, что технология ADO.NET представляет собой концепцию универсального доступа к данным, которую можно определить как распределенное хранение данных и управление ими.

ADO.NET в Delphi 2005

Для разработки приложений БД по технологии ADO.NET в Delphi 2005 используются компоненты, расположенные на странице **Data Components** Палитры компонентов. Они принадлежат пространству имен System.Data. Как и для всей технологии ADO.NET, наиболее общее использование компонентов **Data Components** связано с построением табличных запросов, выборкой и отображением результатов в приложении и, при необходимости, изменением данных пользователем и сохранением в базе данных результатов внесенных изменений.

На странице **Data Components** Палитры расположены следующие компоненты ADO.NET:

- `SqlDataAdapter` — представляет множество команд данных и соединения с базой данных, которые используются для заполнения компонента `DataSet` и обновления (сохранения измененных данных) базы данных SQL-сервера;
- `SqlConnection` — представляет открытое соединение с базой данных SQL-сервера;
- `DataView` — представляет связываемые данные, настроенный просмотр (view) табличного набора данных `DataTable` для сортировки, фильтрации, поиска, редактирования и навигации;
- `DataSet` — представляет внутренний кэш набора данных, полученного из источника данных;
- `SqlCommand` — представляет оператор на языке Transact-SQL или хранимую процедуру, предназначенные для выполнения на SQL Server.

Как и в случае .NET Framework, в Delphi 2005 различают наборы данных `DataSet` двух типов: типизированные (typed) и нетипизированные (standard). Разница между ними заключается в том, имеется ли схема данных для `DataSet`.

Типизированный `DataSet` имеет место в случае, если при работе с набором данных сначала создается схема, а затем в него помещаются данные. При использовании типизированного `DataSet` требуется выполнить ряд вспомогательных действий по определению правил и форматов, которым должен соответствовать этот набор данных.

Создание нетипизированного объекта `DataSet` не требует дополнительных действий, поскольку он не содержит внутренней схемы размещения данных. В нетипизированном объекте `DataSet` таблицы, столбцы и другие элементы представлены как коллекции. Создание нетипизированного объекта `DataSet` проще, но при его использовании недоступны все возможности его типизированного аналога.

Достоинством применения типизированного объекта `DataSet` является возможность ссылаться на столбцы набора данных с помощью имен. При этом программирование работы с набором данных оказывается проще.

Использование нетипизированных объектов `DataSet` необходимо в случаях, когда схема набора данных отсутствует. Такое может быть, к примеру, при взаимодействии приложения с компонентом, возвращающим набор данных с неизвестной структурой. Если используемый набор данных имеет динамически перестраиваемую структуру, то использование типизированного объекта `DataSet` также может оказаться нецелесообразным.

Провайдеры данных для .NET

Платформа .NET Framework включает провайдеров для MS SQL, OLE DB и Oracle. В системе программирования Delphi 2005 в дополнение к этим провай-

дерам включены провайдеры данных фирмы Borland для Microsoft .NET (Borland Data Providers for .NET — BDP.NET). Провайдеры BDP.NET представляют собой реализацию .NET провайдера и соединяются с наиболее популярными базами данных.

Достоинством использования провайдеров BDP.NET является то, что они обеспечивают высокую производительность при доступе к источникам данных без промежуточного слоя COM (рис. 23.5).

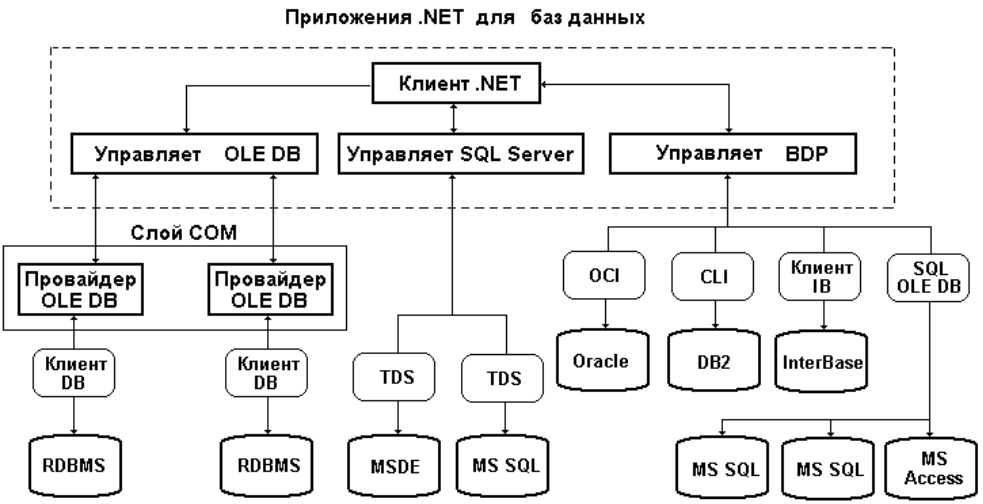


Рис. 23.5. Схема использования провайдеров данных

Для BDP.NET компоненты, доступ к метаданным и конструкторы определены под следующими пространствами имен:

- Borland.Data.Provider — содержит объекты, общие для всех провайдеров BDP, включая объекты ошибок, объекты классов исключений, перечислители типов данных, параметры провайдера и интерфейсы для построения собственных классов `Command`, `Connection` и `Cursor`;
- Borland.Data.Common — содержит ключевые классы BDP.NET, например, `BdpCommand`, `BdpConnection`, `BdpDataAdapter` и другие, которые обеспечивают средства для взаимодействия с внешними источниками данных, такими как базы данных Oracle, DB2, Interbase и MS SQL Server;
- Borland.Data.Schema — содержит интерфейсы для создания собственных классов манипулирования схемой базы данных, таких как ряд типов и перечислителей, которые определяют метаданные;
- Borland.Data.Design — содержит конструкторы.

Компоненты BDP.NET расположены на странице **Borland Data Providers** Палитры компонентов:

- `BdpConnection` — служит для создания одноименного объекта соединения с базой данных;
- `BdpCommand` — служит для создания одноименного объекта, обеспечивающего выполнение SQL-запросов и хранимых процедур;
- `BdpDataAdapter` — задает объект, представляющий собой мост между объектом `DataSet` и базой данных в его основе;
- `BdpCommandBuilder` — генерирует однотабличные команды, как часть операции объекта `BdpDataAdapter`;
- `BdpCopyTable` — служит для копирования таблицы (структуры, первичного ключа и данных) из одного источника данных BDP.NET в другой;
- `DataHub` — служит как проводник между наборами данных `DataSet` и `DataSync`;
- `DataSync` — поддерживает реальные данные от любого провайдера данных .NET на этапе проектирования приложения;
- `RemoteConnection` — обеспечивает соединение между клиентом и удаленным сервером;
- `RemoteServer` — удаленный сервер данных, управляющий запросами данных для многих клиентов в двухзвенных и трехзвенных приложениях.

BDP.NET включает провайдеров для ряда популярных систем управления базами данных, каждому из которых соответствует свое пространство имен:

- InterBase — Borland.Data.InterBase;
- Oracle — Borland.Data.Oracle;
- IBM DB2 — Borland.Data.Db2;
- Microsoft SQL Server — Borland.Data.Mssql;
- Microsoft Access — Borland.Data.Msacc;
- Sybase — Borland.Data.Sybase.

Провайдеры BDP.NET устанавливают соответствие типов данных SQL к типам данных .NET Framework, предотвращая тем самым необходимость изучения типов данных конкретной базы данных. В зависимости от базы данных BDP.NET приводит типы данных к их родным типам данных. Там, где это применимо, BDP.NET обеспечивает:

- установление соответствия совместимых типов по базам данных;
- логические типы данных приводятся к родным типам .NET.

Класс `DataSet` в ADO.NET использует типы данных .NET Framework. Под типы данных BDP.NET задаются типы данных .NET для поддерживаемых баз данных.

Платформа .NET Framework включает широкий диапазон логических типов данных. BDP.NET наследует логические типы данных, обеспечивая встроенное

установление соответствия для поддерживаемых баз данных: DB2, InterBase, MS SQL, MSDL и Oracle.

Пример соответствия типов данных BDP.NET для сервера InterBase приведен в табл. 23.1.

Таблица 23.1. *Соответствие типов для InterBase*

InterBase	Bdp	BdpSub	System.Type
CHAR	String	stFixed	String
VARCHAR	String	NA	String
SMALLINT	Int16	NA	Int16
INTEGER	Int32	NA	Int32
FLOAT	Float	NA	Single
DOUBLE	Double	NA	Double
BLOB Sub_Type 0	Blob	stBinary	Byte[]
BLOB Sub_Type 1	Blob	stMemo	Char[]
TIMESTAMP	Datetime	NA	DateTime

Аналогичные схемы соответствия типов данных имеются и для других баз данных (DB2, MS SQL, MSDL и Oracle).

Приложение Windows Forms с ADO.NET и BDP.NET

Разработка приложений Windows Forms в среде Delphi 2005 осуществляется для платформы .NET. Делается это по команде **File/New/Windows Forms Application — Delphi for .NET**. При этом в Палитре компонентов становятся доступны страницы, приведенные на рис. 23.6.

Рис. 23.6. Состав Палитры компонентов для Windows Forms

Рассмотрим технологию создания простейшего приложения ADO.NET для работы с базами данных с использованием Windows Forms и BDP.NET. После генерации требуемых объектов соединения проект отображает данные в `DataGrid`. Для облегчения разработки приложений для баз данных BDP.NET включает компонент Конструкторов. При этом вместо того, чтобы помещать индивидуальные компоненты в Конструктор и поочередно настраивать их, целесообразно использовать конструкторы BDP.NET, чтобы быстро создать и настроить компоненты базы данных.

Рассмотрим применение основных компонентов Windows Forms, ADO.NET и BDP.NET. Чтобы подтвердить и конфигурировать провайдер данных, можно перетащить (drag-and-drop) объекты из Проводника данных (Data Explorer), который представляет собой окно с деревом баз данных различных провайдеров, расположенное в правой части пользовательского интерфейса среды Delphi (рис. 23.7).

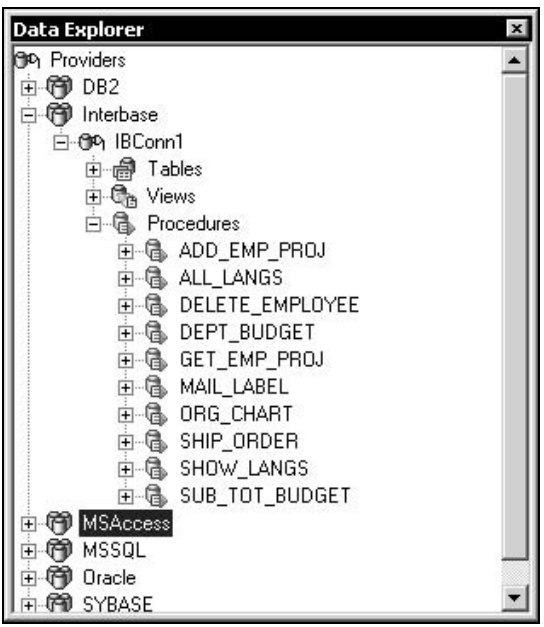


Рис. 23.7. Вид Проводника данных

Построение проекта BDP.NET включает три основных шага:

1. Настройка компонентов соединения BDP.NET и источника данных.
2. Создание и настройка компонента адаптера `BdpDataAdapter`.
3. Связывание компонента сетки данных `DataGrid` с компонентами соединения.

Чтобы настроить компоненты соединения и источник данных, нужно выполнить следующее:

1. Задав команду **File/New/Windows Forms Application — Delphi for .NET**, вызвать Конструктор Windows Forms.
2. Перетащить компонент `BdpConnection` в нижнюю часть (component tray) Конструктора со страницы **Borland Data Provider** Палитры компонентов.
3. В нижней части Инспектора объектов выполнить щелчок в строке **Connection Editor**.
4. При необходимости добавить новое соединение:
   - выполнить щелчок по кнопке **Add**;
   - в открывшемся окне выбрать тип провайдера в раскрывающемся списке **Provider Name**;
   - ввести имя соединения и нажать **OK**.
5. Выбрать соединение.
6. В строке **Database** поля **Connection Settings** задать путь к базе данных и нажать **OK**.

Замечание

При использовании сервера Interbase база данных может размещаться в локальном каталоге c:\Program Files\Common Files\Borland Shared\Data. При соединении с разделяемой сетью нужно указать сетевой путь и необходимо иметь права доступа к удаленному серверу.

7. Заполнить поля **UserName** и **Password** для базы данных. Напомним, что для базы данных сервера Interbase в них указываются значения `SYSDBA` и `masterkey`.
8. Нажать кнопку **Test** для проверки правильности соединения.
9. При получении сообщения об успешности соединения нажать **OK**.

Для создания и настройки адаптера данных (компонента `BdpDataAdapter`) нужно выполнить следующие шаги:

1. Со страницы **Borland Data Provider** Палитры компонентов поместить в нижнюю часть Конструктора компонент `BdpDataAdapter`.
2. В окне Инспектора объектов раскрыть свойство `SelectCommand` в области заполнения.
3. Выбрать объект соединения из раскрывающегося списка в строке свойства `Connection`.
4. Выполнить щелчок в строке **Configure Data Adapter** в нижней части Инспектора объектов.
5. В открывшемся диалоговом окне редактора **Data Adapter Configuration** (рис. 23.8) на вкладке **Command** выбрать имя таблицы, выделить нужные поля таблицы и нажать кнопку **Generate SQL**.

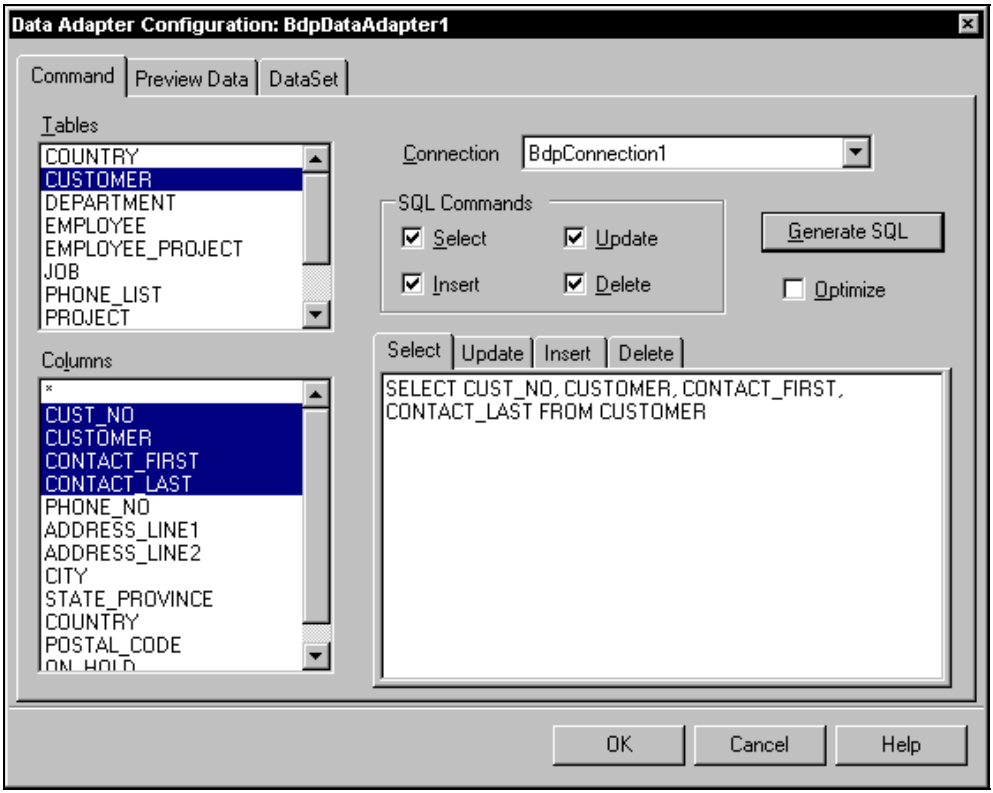


Рис. 23.8. Окно редактора Data Adapter Configuration

Замечание

С помощью вкладки **Preview Data** редактора Data Adapter Configuration можно выполнить предварительный просмотр набора данных, задаваемого автоматически сформированным оператором `SELECT`. Для этого нужно нажать кнопку **Refresh**. В результате в окне предварительного просмотра будут отображаться записи выбранных полей таблицы.

На вкладке **DataSet** рассматриваемого редактора путем установки переключателя **New DataSet** или **Existing DataSet** можно запросить, соответственно, автоматическое создание нового компонента `DataSet` (с указанием нужного имени) или связать настраиваемый адаптер (компонент `BdpDataAdapter`) с имеющимся компонентом `DataSet`.

Для создания нового компонента `DataSet` выберем вкладку **DataSet**, установим переключатель **New DataSet**, уточним имя компонента или оставим имя по умолчанию и нажмем **OK**. В результате новый компонент `DataSet` появится среди уже имеющихся компонентов в нижней части среды Delphi.

Для связывания компонента `DataGrid` с компонентами соединения нужно выполнить следующее.

1. С помощью Инспектора объектов свойству `Active` компонента `BdpDataAdapter` установить значение `True`.
2. Со страницы **Data Controls** Палитры компонентов перетащить компонент `DataGrid` в Конструктор.
3. С помощью Инспектора объектов свойству `DataSource` объекта `DataGrid` выбрать из раскрывающегося списка значение имени сгенерированного нами ранее объекта `DataSet` (по умолчанию `DataSet1`).
4. Кроме того, для свойства `DataMember` объекта `DataGrid` выбрать из раскрывающегося списка имя соответствующей таблицы.

В результате компонент `DataGrid` будет отображать данные из набора `DataSet` (рис. 23.9).

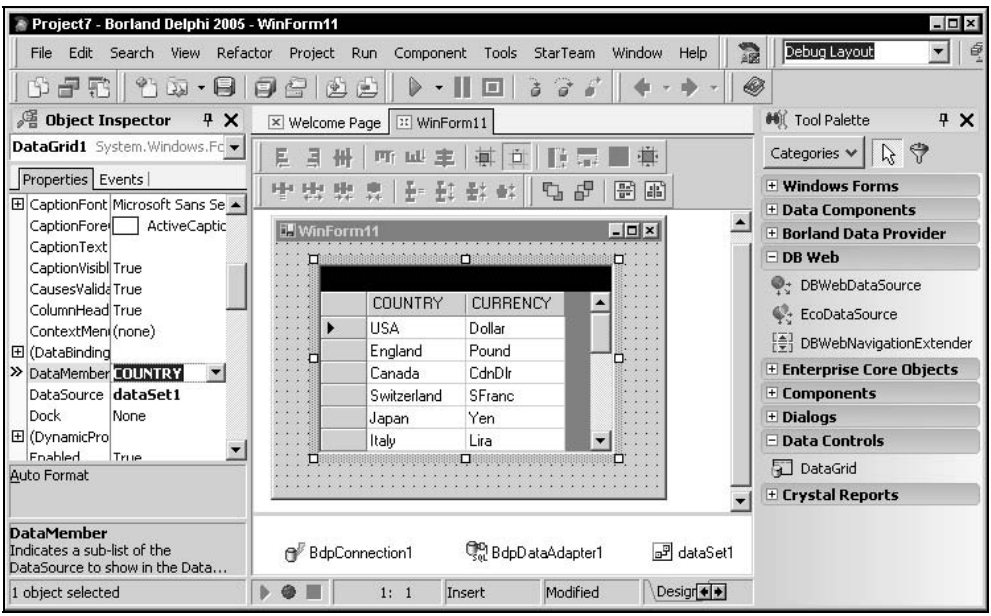


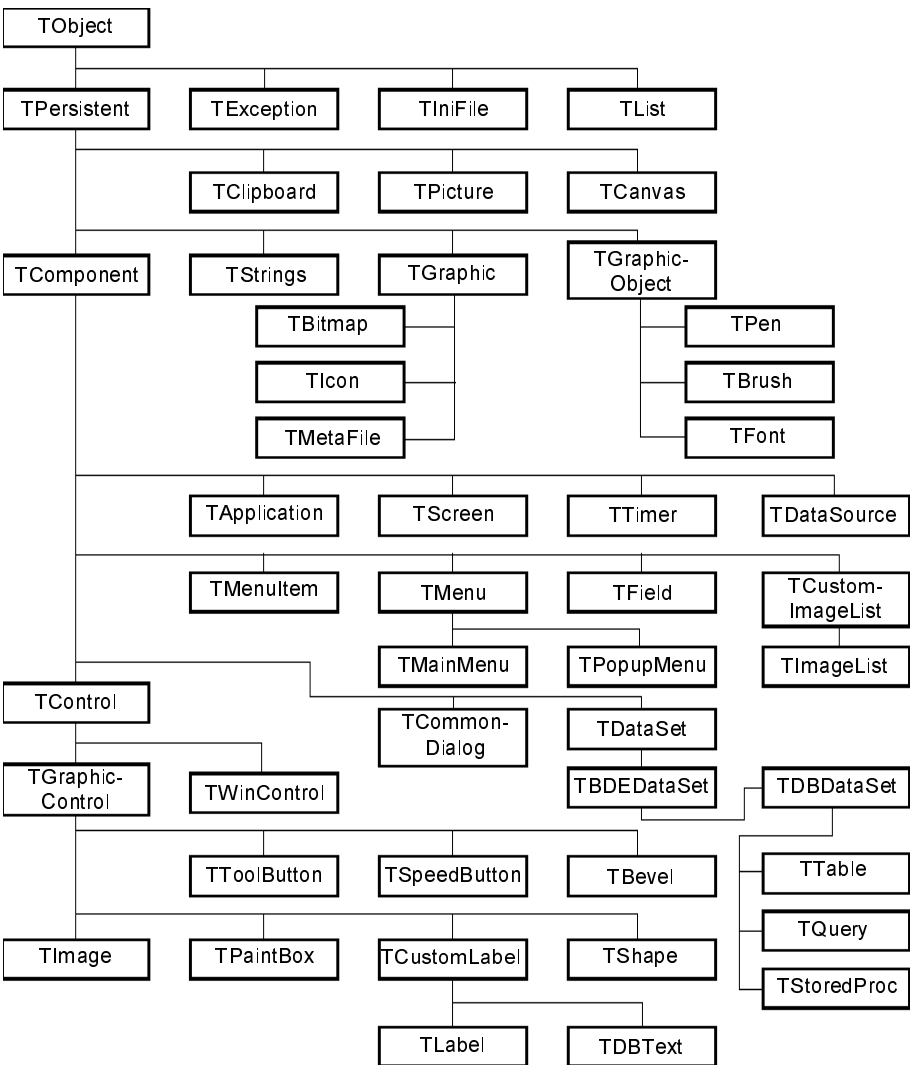
Рис. 23.9. Вид приложения ADO.NET с BDP.NET при проектировании

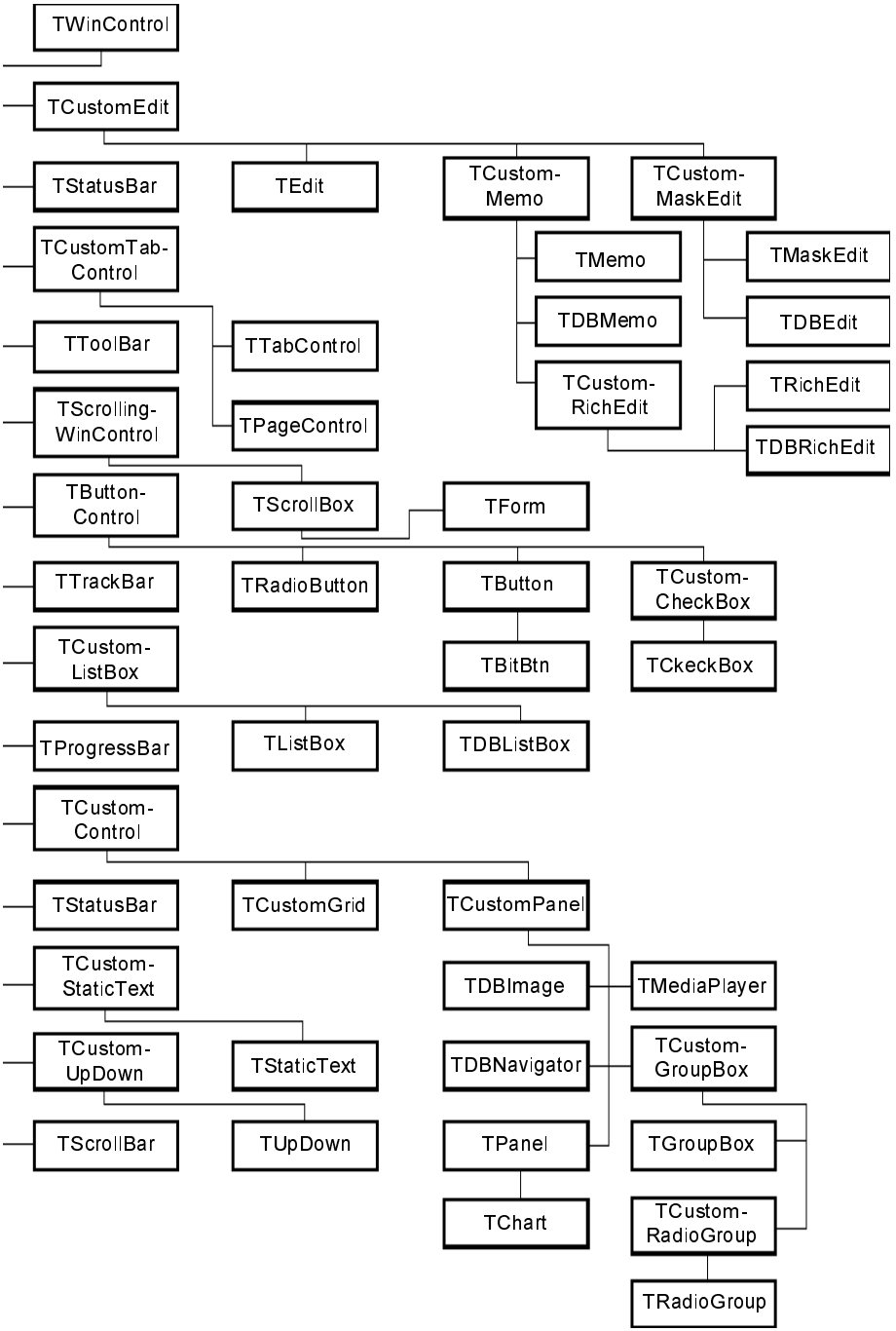
Теперь нам остается задать команду **Run/Run**, чтобы запустить приложение на компиляцию и выполнение. При выполнении приложение будет отображать с помощью компонента `DataGrid` данные из таблицы.

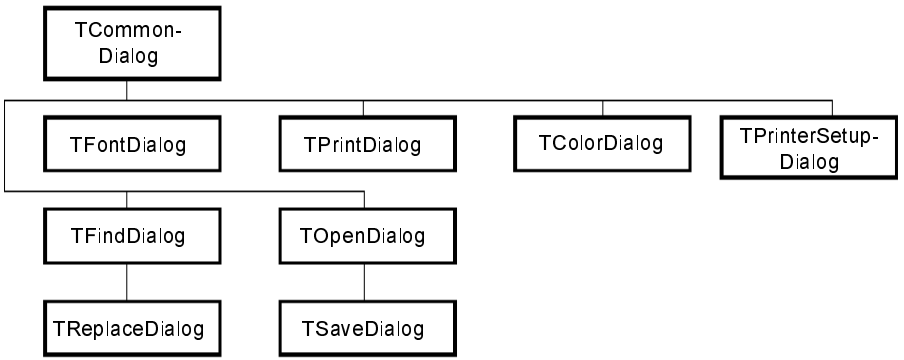
Приведенное нами приложение Windows Forms, хотя и является простейшим в смысле его функциональности, но позволяет получить представление о составе и взаимодействии основных компонентов при использовании технологии ADO.NET и провайдеров BDP.NET.

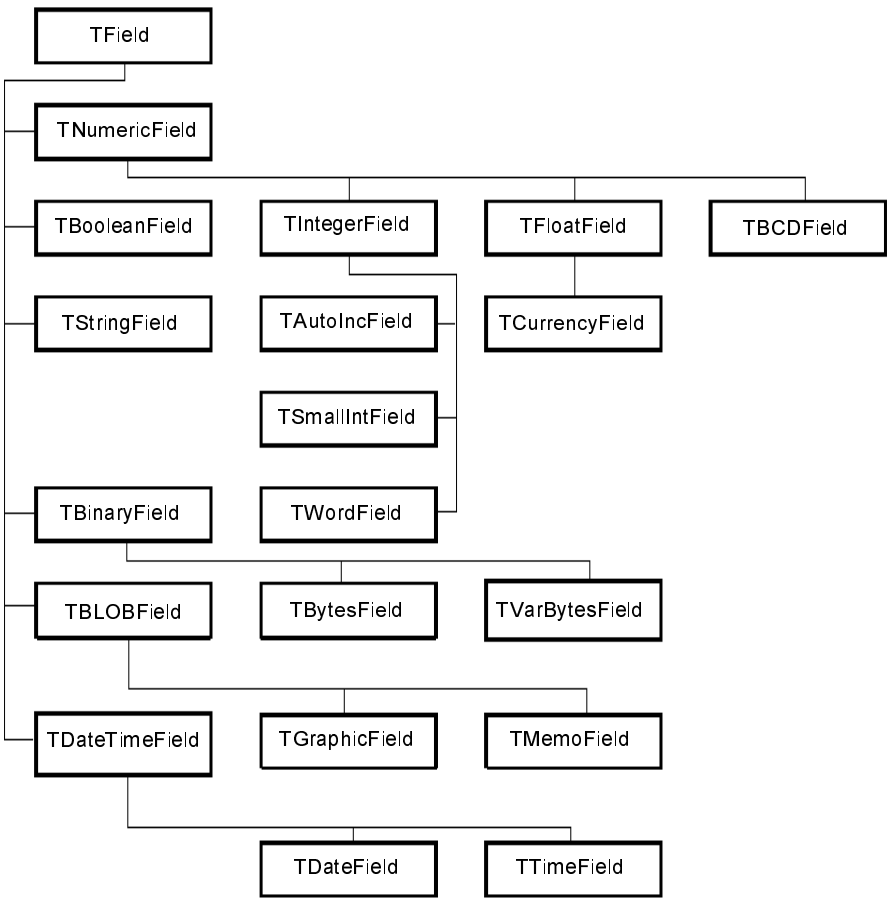
Приложение

Фрагменты иерархии классов VCL




TWinControl
TCustomEdit
TStatusBar
TEdit
TCustom-Memo
TCustom-MaskEdit
TCustomTab-Control
TMemo
TMaskEdit
TDBMemo
TDBEdit
TToolBar
TTabControl
TCustom-RichEdit
TRichEdit
TScrolling-WinControl
TPageControl
TDBRichEdit
TButton-Control
TScrollBox
TForm
TTrackBar
TRadioButton
TButton
TCustom-CheckBox
TCustom-ListBox
TBitBtn
TCkeckBox
TProgressBar
TListBox
TDBListBox
TCustom-Control
TStatusBar
TCustomGrid
TCustomPanel
TCustom-StaticText
TDBImage
TMediaPlayer
TCustom-UpDown
TStaticText
TDBNavigator
TCustom-GroupBox
TScrollBar
TUpDown
TPanel
TGroupBox
TChart
TCustom-RadioGroup
TRadioGroup


TCommon-
Dialog
TFontDialog
TPrintDialog
TColorDialog
TPrinterSetup-
Dialog
TFindDialog
TOpenDialog
TReplaceDialog
TSaveDialog


TField
TNumericField
TBooleanField
TIntegerField
TFloatField
TBCDField
TStringField
TAutoIncField
TCurrencyField
TSmallIntField
TBinaryField
TWordField
TBLOBField
TBytesField
TVarBytesField
TDateTimeField
TGraphicField
TMemoField
TDateField
TTimeField

Предметный указатель

•

.NET Framework 599

A

ActiveX 487
AddDatabase 433
AddPassword 69
ADO 492, 569
ADO.NET 604
Alias 458
ALTER TABLE 242, 382
AppendRecord 214
ApplyRange 192
ApplyUpdates 279, 416
ASP 489

B

BDE 10, 34
BDE Administrator 322
BDP.NET 609

C

CancelRange 192
CancelUpdates 279, 480
CGI 488, 512, 545
ChangeCount 476
CHECK 384
CheckClosed 441
CheckOpen 441
CheckValidStatement 441
Code 20
COLLATE 384
Column 143
Columns Editor 144
COM+ 467
Command 296
CommandText 295, 296
CommandType 272, 295, 296
Commit 234, 271, 433
Connected 289, 432
ConnectionName 270
ConnectionState 270
ConnectOptions 290
Constraints 471
Content 566
CORBA 467
CREATE DOMAIN 391
CREATE GENERATOR 409
CREATE INDEX 243, 390
CREATE PROCEDURE 394
CREATE TABLE 381
CREATE TABLE 240
CREATE TRIGGER 406
CREATE VIEW 392
CreateTable 165

D

Data Controls 133
Data Pump 351
Data View Dictionary 304
Database Desktop 55, 331
DataModule 78
DataSetPageProducer 534
DataSetTableProducer 534
DataSource1 75
DB 40
DBChart 158
DBCheckBox 135
DBComboBox 138
DBCtrlGrid 149, 150
DBGrid 139, 140
DBGrid1 76
DBImage 154
DBListBox 138
DBLookupComboBox 139
Dbmodule 558
DBNavigator 152

DBNavigator1 76
DBRadioGroup 136
DCOM 467
DCOMConnection 473
DefaultAction 433
Delete 216
DELETE 262
Delphi 2005 599
DROP INDEX 244
DROP TABLE 381
DTD 497, 500

E

EDatabaseError 40
EDBClient 41
EDBEditError 40, 42
Edit 210
edtAttachment 573
edtReceiver 573
edtTitle 573
E-mail 572
Event Editor 306
EventStatus 290
ExecDirect 273
ExecSQL 273
Execute 271, 296
EXECUTE PROCEDURE 404
ExecuteDirect 271
ExecuteOptions 296

F

Filter 186
FindKey 204
ForcedRefresh 435
FOREIGN KEY 388

G

gdb 378
GEN_ID 385, 410
GenerateParamNames 441
GET 560
Global Page Catalog 304
GotoBookmark 183
GROUP BY 253
GUID 473

H

HTML 495
HTTP 494, 579
HTTPReqResp 586
HTTPRIO 585
HTTPSoapDispatcher 587

I

IB 40
IBConsole 377, 446
IBTransaction 432
IDC/HTX 489
IN 251
IndexFieldNames 225
IndexName 225
INSERT 260
InsertRecord 215
Interactive SQL 460
InterBase Express 430
InTransaction 433
ISAPI 489, 554
ISAPI/NSAPI 513

J

JavaScript 487
JDBC 514
JScript 487

L

LIKE 251
Locate 197
LockTable 168
Lookup 202

M

MAPI 573
MasterFields 225

MasterSource 225
memMessageText 573
MTS 467

N

NSAPI 489
Number 20

O

ODBC 328, 491
OLE DB 491
OnAfterScroll 180
OnConnectComplete 290
OnDisconnect 291
OnDrawColumnCell 142
OnWillConnect 290
OPToSoapDomConvert 586

P

PageProducer 533
Post 211, 234
POST_EVENT 422
PRIMARY KEY 387

Q

qbe 336
QR 40
Query 167
QueryTableProducer 537

R

Reconcile 279
Reconcile Error Dialog 479
RecordNumber 182
RecordStatus 293
REFERENCES 388
RemoveAllPasswords 69
RemoveDatabase 433
RemovePassword 69
RenameTable 167
Report Library 303
Rollback 433
RTTI 584
RvCustomConnection 313
RvDataSetConnection 313
RvProject 312
RvQueryConnection 313
RvSystem 313
RvTableConnection 313

S

SaveToFile 481
SELECT 244
SendEmail 577
SetFields 213
SetKey 205
SetSchemaInfo 273
SGML 496
ShellExecute 572
SHOW INDEX 390
SOAP 467, 579
SQL 236
SQL Builder 342
SQL Explorer 349
SQL Links 328
SQL Monitor 463
SQLConnection 267
SQLPASSTHRU 413
SQL-запрос 262
StartTransaction 234, 433
StatementType 437
States 296
StoredProc 394

T

Table1 76
TADOCommand 296
TCP/IP 494
TCustomADODataSet 292
TDBError 41
THTTPRio 592
TIBCustomDataSet 434
TIBDataBase 430, 431
TIBDataSet 437

TIBQuery 436
TIBSQL 441
TIBTable 436
TMapiFileDesc 577
TMapiMessage 577
TMapiRecipDesc 577
TraceList 282
TSimpleDataSet 279
TSQLDataSet 272
TSQLMonitor 282
TSQLQuery 277
TSQLStoredProc 278
TSQLTable 277
TTPSoapPascalInvoker 588

U

UDDI 589
UndoLastChange 480
UpdateRecordTypes 435
UpdateSQL 419
URL 495

V

VBScript 487

W

WebModule1 556
WebResponse 555
WebServices 584
Web-приложение 493, 508
  в сети интранет 510
Web-службы 579
Web-страница:
  формирование 490
WinCGI 488
WSDL 580
WSDLHTMLPublish 587

X

XML 496 579
XML Mapper 504

А

Администратор BDE 322
Апплет 488, 514
Архитектура "клиент-сервер" 13, 358
Архитектура "файл-сервер" 13, 357
Атрибуты 29

Б

База данных 7
Банки данных 7
Бизнес-правила 28, 362

В

Визуальные компоненты для работы
  с данными 133
Внешний ключ 388
Вставка записей 339
Вывод графических изображений 154
Вызов хранимой процедуры 400
  выбора 401
Вызываемый интерфейс 584
Вычисляемое поле 111
Вычисляемый столбец 383

Г

Генератор 409
Главная таблица 24, 345, 388
Главный индекс 19, 95
Главный пароль 68
Глубина индекса 391
Группирование записей 253, 347

Д

Диапазон ограничений 385
Динамическая публикация 545
  Web-страниц 492
  БД 492

Динамическое поле 106
Добавление записей 214
Домен 383, 391
Допустимое значение 70
Доступ к данным 81, 106
Доступ к значению поля 116
Доступ к полю 90
Драйвер 326, 368

З

Задание индекса 60
Записная книжка 190
Запись 17
Запрос SQL 101, 333

И

Избыточность данных 44
Изменение данных 94, 128, 415, 427, 460
Изменение:
  поля связи 26
  структуры таблицы 74
Изоляция транзакций 370
Имя:
  индекса 61, 95
  поля 58, 116
  таблицы 93
Индекс 20, 390
Индексные поля 60
Индексы 352
Инструкция:
  выбора строки 398
  цикла 397
Инструменты 34, 322
  для работы с локальными БД 322
Интранет-приложение 510
Использование переключателей 136
Источник данных 127

К

Каскадное удаление 26, 33, 408
Клиент 358
Клиент Web-службы 589
Ключ 19, 58, 59, 95, 387
Ключевые поля 19, 95
Кодировка символов 326, 377
Компоненты:
  для создания приложений БД 35
  отображения данных 309
  управления отчетом 312
Конструирование запросов 335, 342
Конфигурационный файл 330
Копирование данных 351

Л

Логическая таблица 81, 392
Логическое поле 135
Локальная архитектура 12
Локальная БД 11

М

Маска 63, 125
Метаданные 363
Метод доступа к данным 21
Механизм:
  кэшированных изменений 414
  событий сервера 422
  транзакций 370
  транзакций 27, 234, 412
Многофайловая БД 379
Модели БД 9
Модификация:
  записей 259
  набора данных 206
Модифицированная сетка 149
Модуль:
  данных 77
  расширения клиента 513
  расширения сервера 511
Монопольный доступ 94

Н

Набор данных 81, 92, 100
Навигатор 152
Навигационный способ доступа 84
Настройка BDE 322

Нормализация БД 43
Нормальные формы БД 46

О

Обработка ввода пользователя 560
Объект поля 105
Ограничение:
  значения 384
  ссылочной целостности 388, 407
Ограничения:
  БД 28
  на значения полей 62
  ссылочной целостности 352
Окно:
  Обозревателя дерева объектов 34
  Редактора кода 34
Описание:
  полей таблицы 58
  столбца 383
Определение:
  данных 239
  индекса 96
Отбор:
  данных 244, 427
  записей 336, 398
Отношение подчиненности 24
Отображение данных 128

П

Параметры:
  BDE 322, 368, 371
  индекса 61, 96
Пароль 67
Первичный:
  индекс 19
  ключ 19, 387
Переход по закладкам 181
План выполнения запроса 429
Подчиненная таблица 24, 345, 388
Поиск:
  записей 197
  по индексным полям 204
Поле 17, 90, 105
  выбора 110, 113
  связи 65, 113, 224, 345
Построение диаграмм 158
Права доступа 67, 425
Представление записей 139
Префиксы имен компонентов 40
Привилегия 425
Приложение клиента 473
Приложение-клиент 358
Приложения баз данных 8
Проектирование БД 43
Просмотр 392
  данных 460
  отчета 317
Простой и комбинированный
  списки 138
Процедура:
  выбора 401
  действия 401
Псевдоним 323, 333, 350, 368
  таблицы 344
Псевдофильтрация 196
Публикация:
  баз данных 485
  графики 562

Р

Разбалансировка индекса 391
Размер поля 58
Регистрация:
  БД 452
  сервера 449
Редактирование:
  данных 350
  записей 208, 338
Редактируемый набор данных 104
Редактор полей 107
Режим набора данных 88, 127
Реляционная БД 16
Реляционный способ доступа 84, 358
Реструктуризация таблицы 382

С

Сбор мусора 454
Связывание таблиц 23, 341, 345, 389
Сервер 358
Сервер приложений 468, 517
Сервлет 488

Система управления базой данных 7
Системные установки 329
Системный администратор 378
Словарь данных 29, 350
Сложный критерий 253
Соединение:
  с базой данных 365, 460
  таблиц 257
Создание:
  базы данных 460
  приложения 75
  таблиц 56, 381, 460
  удаленной БД 377
Сортировка 254
  набора данных 169
  строк 384
Составная инструкция 397
Состояние набора данных 84
Список подчиненных таблиц 74
Способ:
  взаимодействия с сервером 413
  доступа к данным 22
Средства CASE 51
Ссылочная целостность 65
Статистическая функция 347
Статическая публикация 528
  Web-страниц 492
Статическое поле 107
Столбец 359
Столбцы сетки 143
Структура:
  базы данных 363
  таблицы 380

Т

Таблица 16
  выбора 70
Текущая запись 128
Текущий:
  индекс 94
  сеанс работы 371
Технология ADO 284
Тип:
  поля 58, 109, 116, 119
  столбца 383
  таблицы 29, 93
Транзакция 27
Трехуровневая архитектура "клиент-сервер" 15
Трехуровневые Web-приложения 516
Триггер 395 405

У

Удаление:
  записей 216, 339
  таблицы 381
Управление:
  БД 451
  полями 245
Уровень доступа 168
Условие 384
  отбора 346
  отбора записей 249
Условный инструкцию 396

Ф

Фильтрация 184
  по диапазону 192
Формат:
  поля 125
  таблиц 324
  таблиц dBase 29
  таблиц Paradox 31
Форматирование выводимой информации 124
Функция, определяемая пользователем 410

Х

Хранимая процедура 393

Ш

Шаблон 63

Я

Язык хранимых процедур 395
Языковой драйвер 69, 326