А. ХОМОНЕНКО, С. АДАДУРОВ

bhv®

РАБОТА С БАЗАМИ ДАННЫХ В C++ BUILDER

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
Сергей Ададуров

РАБОТА С БАЗАМИ ДАННЫХ в C++ BUILDER

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26-018.1
Х76

Хомоненко А. Д., Ададуров С. Е.

Х76 Работа с базами данных в C++ Builder. — СПб.: БХВ-Петербург, 2006. — 496 с.: ил.

ISBN 5-94157-690-0

Рассматривается использование средств C++ Builder для разработки приложений баз данных. Даются понятия баз данных, характеризуются элементы и описываются этапы проектирования реляционных баз данных, изложена технология разработки информационных систем. Показаны основные приемы работы с данными при создании таблиц, подготовке SQL-запросов, использовании триггеров и хранимых процедур. Подробно описаны основные визуальные компоненты для разработки приложений, а также инструменты для администрирования локальных и удаленных данных. Рассматриваются навигационный и реляционный способы доступа к данным с помощью BDE, ADO, dbExpress и Interbase Express, основы программирования на SQL. Показывается использование локальных и удаленных баз данных, включая создание многоуровневых информационных систем. Благодаря подробному изложению тем и большому числу примеров книга может служить практическим руководством по работе с базами данных.

Для разработчиков БД

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Татьяна Лапина*
Компьютерная верстка	*Екатерины Трубниковой*
Корректор	*Татьяна Кошелева*
Дизайн серии	*Инна Тачина*
Дизайн обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.11.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 39,99.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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 5-94157-690-0

© Хомоненко А. Д., Ададуров С. Е., 2006
© Оформление, издательство "БХВ-Петербург", 2006

Содержание

Предисловие **1**

ЧАСТЬ I. ОСНОВЫ РАБОТЫ С БАЗАМИ ДАННЫХ **3**

Глава 1. Основные понятия баз данных **5**

Банки данных 5
Модели данных 6
Базы данных и приложения 7
Механизмы доступа приложений 8
BDE 9
ADO 9
dbExpress 9
InterBase Express 9
Варианты архитектуры для BDE 10

Глава 2. Реляционные базы данных и средства работы с ними **15**

Элементы реляционной базы данных 15
Таблицы баз данных 15
Ключи и индексы 18
Методы и способы доступа к данным 21
Связи между таблицами 23
Механизм транзакций 27
Бизнес-правила 28
Словарь данных 29
Таблицы форматов dBase и Paradox 29
Средства для работы с базами данных 34
Инструменты 35
Компоненты 36
Исключения баз данных 41

Глава 3. Проектирование баз данных **45**

Проблемы и подходы к проектированию 45
Функциональные зависимости атрибутов 49
Нормализация баз данных 50
Средства CASE 54

Глава 4. Технология создания информационной системы ... **57**
Варианты создания таблиц ... 57
Создание таблицы с помощью Database Desktop ... 59
Описание полей ... 62
Задание индексов ... 64
Задание ограничений на значения полей ... 66
Задание ссылочной целостности ... 69
Задание паролей ... 71
Задание языкового драйвера ... 74
Задание таблицы для выбора значений ... 74
Просмотр списка подчиненных таблиц ... 78
Изменение структуры таблицы ... 78
Создание приложения BDE ... 79
Использование модуля данных ... 82

Глава 5. Компоненты доступа к данным с помощью BDE ... **87**
Наборы данных ... 87
Состояния наборов данных ... 90
Режимы наборов данных ... 95
Доступ к полям ... 98
Особенности набора данных *Table* ... 100
Особенности набора данных *Query* ... 109
Объекты поля ... 115
Редактор полей ... 117
Операции с полями ... 126
Доступ к значению поля ... 127
Проверка типа и значения поля ... 131
Форматирование отображаемого значения поля ... 137
Источник данных ... 140

ЧАСТЬ II. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ ... **143**

Глава 6. Визуальные компоненты для работы с данными ... **145**
Отображение и редактирование значения логического поля ... 147
Отображение и выбор значения поля ... 148
Отображение и выбор значения поля в списке ... 150
Простой и комбинированный списки ... 151
Списки, сформированные по значениям поля набора данных ... 151
Представление записей в табличном виде с помощью сетки ... 152
Характеристики сетки ... 152
Столбцы сетки ... 156
Компонент "модифицированная сетка" ... 161
Использование навигационного интерфейса ... 165
Компонент "графическое изображение" ... 167
Построение диаграмм ... 171

Глава 7. Навигационный доступ к данным ... **179**
Операции с таблицей БД ... 180
Создание, удаление и переименование ... 180
Установка уровня доступа ... 181

Сортировка набора данных ... 183
Навигация по набору данных ... 185
Перемещение по записям ... 185
Переход по закладкам ... 188
Фильтрация записей ... 192
Фильтрация по выражению ... 192
Фильтрация по диапазону ... 197
Навигация с псевдофильтрацией ... 200
Поиск записей ... 200
Поиск в наборах данных ... 201
Поиск по индексным полям ... 204
Модификация набора данных ... 206
Редактирование записей ... 208
Добавление записей ... 212
Удаление записей ... 214
Пример формы приложения ... 215
Связывание таблиц ... 223

Глава 8. Доступ к данным с помощью запросов ... 227

Основные сведения о языке SQL ... 228
Функции языка ... 229
Определение данных ... 230
Создание и удаление таблицы ... 230
Изменение состава полей таблицы ... 233
Создание и удаление индекса ... 234
Отбор данных из таблиц ... 235
Описание оператора *SELECT* ... 235
Управление полями ... 236
Простое условие отбора записей ... 240
Сложные критерии отбора записей ... 244
Группирование записей ... 245
Сортировка записей ... 246
Соединение таблиц ... 248
Модификация записей ... 250
Редактирование записей ... 251
Вставка записей ... 252
Удаление записей ... 253
Статический и динамический запросы ... 254
Запросы с параметрами ... 255

Глава 9. Технология dbExpress ... 259

Общая характеристика ... 259
Установление соединения с сервером ... 260
Компоненты доступа к данным ... 266
Универсальный доступ к данным ... 266
Просмотр таблиц ... 273
Выполнение SQL-запроса ... 274
Выполнение хранимых процедур ... 276
Компонент редактирования набора данных ... 277
Отладка соединения с сервером ... 281

Глава 10. Технология ADO ... **283**
Общая характеристика ... 283
Установление соединения ... 285
Управление соединением и транзакциями ... 289
Компоненты доступа к данным ... 291
Доступ к таблицам ... 294
Выполнение запросов ... 294
Вызов хранимых процедур ... 295
Компонент ADODataSet ... 295
Команды ADO ... 296
Пример приложения ... 298

Глава 11. Создание и просмотр отчетов с помощью QuickReport ... **303**
Компоненты отчета ... 303
Компонент-отчет ... 303
Полоса отчета ... 311
Компоненты, размещаемые в полосе ... 313
Простой отчет ... 316
Заголовок отчета ... 317
Заголовки столбцов и данные ... 317
Итоговая полоса ... 319
Колонтитулы ... 319

Глава 12. Инструменты ... **321**
Программа BDE Administrator ... 321
Работа с псевдонимами ... 323
Параметры драйвера ... 325
Системные установки ... 329
Использование конфигурационных файлов ... 331
Программа Database Desktop ... 331
Редактирование записей таблиц ... 333
Работа с псевдонимами ... 333
Работа с SQL-запросами ... 334
Визуальное конструирование запросов ... 335
Отбор записей из таблицы ... 336
Связывание таблиц ... 337

ЧАСТЬ III. УДАЛЕННЫЕ БАЗЫ ДАННЫХ ... **339**

Глава 13. Введение в работу с удаленными базами данных ... **341**
Основные понятия ... 341
Архитектура "клиент-сервер" ... 342
Сервер и удаленная БД ... 343
Средства работы с удаленными БД ... 343
Сервер InterBase ... 345
Бизнес-правила ... 346
Организация данных ... 347
Запуск сервера ... 349
Особенности приложения ... 350
Соединение с базой данных ... 350

Глава 14. Работа с удаленными базами данных InterBase ... **353**
Создание базы данных ... 353
Управление структурой таблиц ... 357
Описание столбца ... 359
Ограничения столбца ... 360
Описание ключей ... 363
Определение ограничений ссылочной целостности ... 365
Использование индексов ... 366
Домены ... 368
Представления ... 369
Хранимые процедуры ... 370
Создание и изменение ... 371
Виды хранимых процедур ... 372
Язык хранимых процедур ... 372
Триггеры ... 379
Создание и изменение ... 379
Примеры использования ... 380
Создание генераторов ... 383
Механизм событий сервера ... 384
Управление привилегиями ... 385
Манипулирование данными ... 387

Глава 15. Доступ к удаленным БД с помощью BDE ... **391**
Управление соединениями с базой данных ... 391
Компонент *Database* ... 391
Компонент *Session* ... 395
Соединение с базой данных ... 399
Вызов хранимых процедур ... 400
Вызов процедуры выбора ... 400
Вызов исполняемой процедуры ... 402
Механизм транзакций ... 404
Механизм кэшированных изменений ... 407
Компонент *UpdateSQL* ... 408
Компоненты *Database* и *Query* ... 412

Глава 16. Технология InterBase Express ... **417**
Общая характеристика ... 417
Установление соединения с сервером ... 418
Управление транзакциями ... 420
Компоненты доступа к данным ... 422
Генераторы для автоинкрементных полей ... 423
Доступ к таблицам ... 424
Выполнение запросов ... 424
Получение и редактирование данных ... 425
Компонент *IBSQL* ... 430
Пример приложения ... 430

Глава 17. Инструменты для работы с удаленными базами данных ... **435**
Программа IBConsole ... 435
Управление сервером ... 436

Подключение к серверу ... 436
Регистрация сервера ... 438
Просмотр протокола работы сервера ... 439
Операции с сертификатами ... 439
Управление пользователями ... 440
Управление БД ... 440
Регистрация базы данных ... 441
Подключение базы данных ... 442
Создание базы данных ... 442
Просмотр метаданных ... 442
Сбор мусора ... 443
Проверка состояния базы данных ... 443
Анализ статистики ... 444
Сохранение и восстановление базы данных ... 445
Интерактивное выполнение SQL-запросов ... 449
Программа SQL Monitor ... 452

Глава 18. Трехуровневые приложения ... 455

Принципы построения трехуровневых приложений ... 455
Сервер приложений ... 457
Приложение клиента ... 463

Предметный указатель ... 475

Предисловие

Система программирования Borland C++ Builder 6 завоевала достаточно прочные позиции среди профессиональных и начинающих программистов. Здесь можно отметить ряд причин: большую популярность языка программирования C++, удобство визуального конструирования приложений, развитые возможности доступных средств системы, эффективность генерируемого кода и др.

Несмотря на появление современных технологий типа .NET и соответствующих систем программирования, таких как Visual C++ .NET, система C++ Builder будет устойчиво занимать свою нишу. Это обусловлено меньшей требовательностью к аппаратным ресурсам при разработке приложений, большей легкостью в освоении и применении средств системы для разработки приложений различной степени сложности.

В книге рассматриваются технологии применения C++ Builder 6 для разработки приложений баз данных. C++ Builder 6 не является системой управления базами данных (СУБД), строго ориентированной на разработку приложений для работы с ними. Тем не менее ее возможности практически ни в чем не уступают возможностям таких СУБД, как Visual FoxPro или Access. Она позволяет создавать приложения с помощью инструментальных программных средств, визуально подготавливать, а также непосредственно писать SQL-запросы к базам данных. С ее помощью можно создавать приложения для работы с локальными и удаленными базами данных.

Для полноты представления работы с информационными системами в книге затрагиваются вопросы проектирования реляционных баз данных, для работы с которыми и используется система C++ Builder 6. Дается описание языка структурированных запросов SQL.

Описываются основные компоненты, свойства, методы и события для работы с локальными и распределенными базами данных. По традиции подробное описание и примеры приводятся для компонентов, используемых для доступа к данным с помощью механизма BDE и других механизмов (dbExpress, ADO

и InterBase Express). Приводятся отличительные особенности и даются примеры применения.

Дается описание технологии применения различных инструментальных средств, используемых для создания приложений по работе с локальными и удаленными базами данных. В частности, рассматриваются BDE Administrator, IBConsole и SQL Monitor. Приводятся примеры реально работающих программ, которые читатель может использовать в своих разработках.

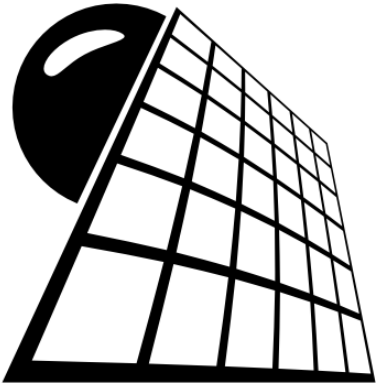
При написании книги использовалась версия Enterprise системы C++ Builder 6, как предоставляющая наибольшие возможности. В книге получили освещение сравнительно новые средства и технологии, появившиеся или получившие дальнейшее развитие в системе C++ Builder 6 (технологии dbExpress и ADO работы с базами данных, технологии InterBase Express работы с сервером баз данных InterBase и др.).

Несмотря на относительную простоту построения приложений в среде C++ Builder, имеются определенные трудности в правильном использовании свойств и методов компонентов системы. Приведенные в справочной помощи системы примеры, как выясняется, не всегда верны. Все это свидетельствует о необходимости создания методически отработанных материалов, где в структурированном виде с наглядными примерами дается описание технологии построения приложений по работе с базами данных. Именно этого и хотелось достичь при подготовке материалов книги. Насколько это удалось, судить читателю.

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

Выражаем признательность В. Э. Гофману, совместная работа с которым над книгами по Delphi во многом способствовала появлению этой книги.

Авторы



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

Система C++ Builder не является СУБД в буквальном смысле этого слова, тем не менее, она обладает вполне развитыми возможностями СУБД. Предоставляемые C++ Builder средства обеспечивают создание и ведение локальных и клиент-серверных БД, а также разработку приложений для работы практически с любыми БД. Назвать систему C++ Builder обычной СУБД мешает, наверное, только то, что 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 C++ Builder 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.

Механизмы доступа приложений

Одно- и двухуровневые приложения C++ Builder 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 Express реализует непосредственный доступ к базам данных сервера 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.

Трехуровневые приложения C++ Builder 6 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 приложения C++ Builder, осуществляющие работу с БД. Приложение через BDE передает запрос к базе данных, а обратно получает требуемые данные. Механизм BDE до 6-й версии системы C++ Builder получил самое широкое распространение ввиду широкого спектра предоставляемых им возможностей. Идеологи фирмы Borland планируют более широкое применение других механизмов как более эффективных. 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.

InterBase Express

Механизм доступа InterBase Express ориентирован строго на работу с сервером InterBase и основан на прямом применении функций API этого сервера. Отсюда следуют все достоинства и недостатки использования этого механизма доступа. Он обеспечивает высокую скорость работы компонентов

доступа к данным. Очевидным недостатком механизма доступа InterBase является невозможность применения его для серверов баз данных, отличных от сервера InterBase SQL Server.

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.

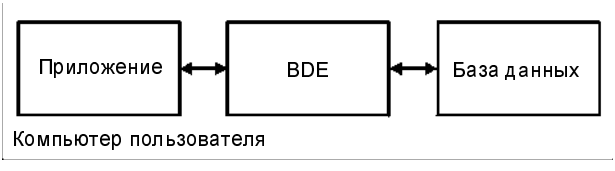


Рис. 1.1. Локальная архитектура

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).

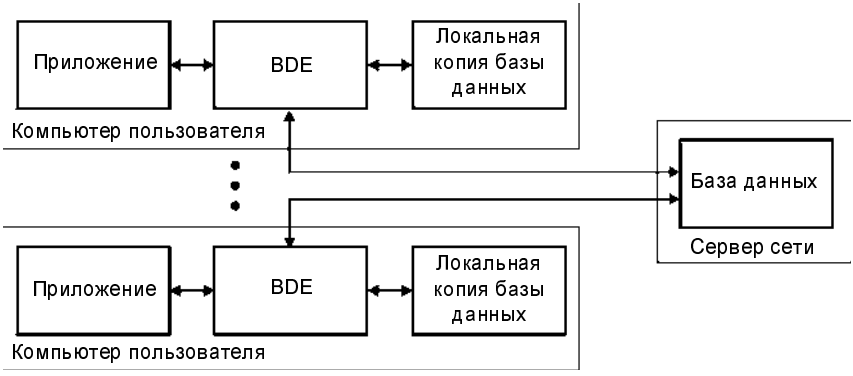


Рис. 1.2. Архитектура "файл-сервер"

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 называют также *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.

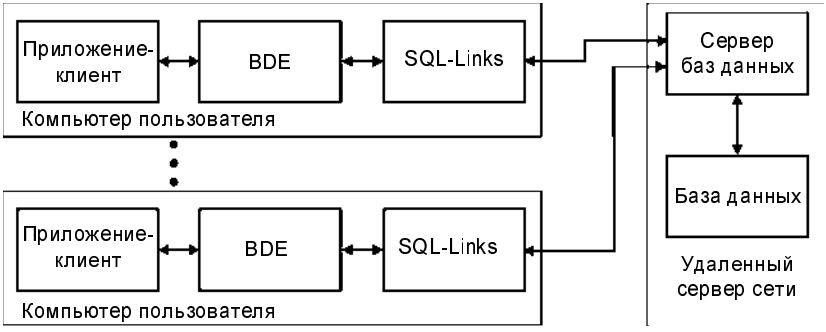


Рис. 1.3. Архитектура "клиент-сервер" ("толстый" клиент)

Для реализации архитектуры "клиент-сервер" обычно используются многопользовательские СУБД, например, Oracle или Microsoft SQL Server. Подобные СУБД называют также *промышленными*, так как 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 приложения C++ Builder к промышленным СУБД осуществляется через драйверы SQL-Links. Отметим, что при работе с "родной" для C++ Builder СУБД InterBase можно обойтись без драйверов SQL-Links.

Описанная архитектура является двухуровневой (уровень приложения-клиента и уровень сервера БД). Клиентское приложение называют также *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).

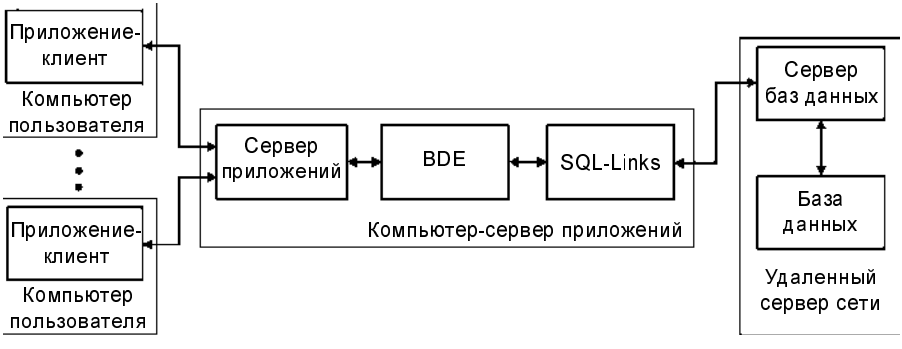


Рис. 1.4. Трехуровневая архитектура "клиент-сервер" ("тонкий" клиент)

В трехуровневой архитектуре" часть средств и кода, предназначенных для организации доступа к данным и их обработки, 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, его имя 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 — 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 приведены далее в *разд. "Таблицы форматов dBase и Paradox"* данной главы.

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.

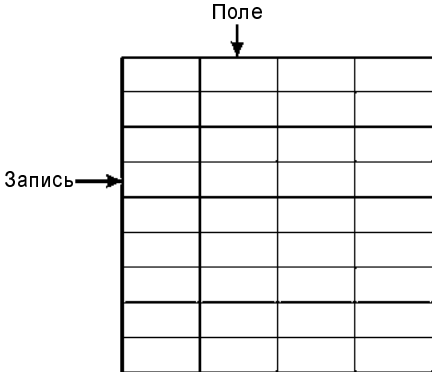


Рис. 2.1. Схема таблицы базы данных

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. Если они 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 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 является использование для него поля соответствующего типа, которое автоматически обеспечивает поддержку уникальности значений. Например, для таблиц Paradox таким является поле автоинкрементного типа, 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 автоинкрементных полей.

Отметим, что при создании и ведении БД правильным подходом считается задание в каждой таблице ключа даже в том 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.

Индекс, как и ключ, 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 C++ Builder 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 индексируются автоматически. 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 максимально 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` именно в таком порядке!

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

- навигационный;
- реляционный (SQL-ориентированный).

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 C++ Builder, реализацию навигационного способа доступа для механизма BDE можно осуществить с применением компонентов `Table` или `Query`, а реляционного — с помощью компонента `Query`. Для других механизмов доступа к данным также можно использовать навигационный и реляционный способы доступа, естественно с применением соответствующих аналогов компонентов `Table` и `Query` (например, `ADOTable` и `ADOQuery` для механизма ADO).

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

Связи 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 о некоторой системе, связанную не в буквальном (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 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.

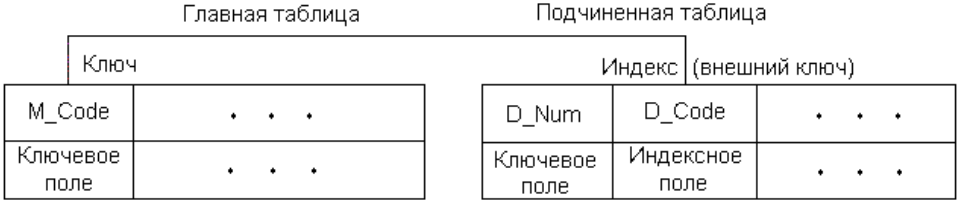


Рис. 2.2. Схема связи между таблицами базы данных Paradox

В главной таблице определен ключ, построенный по полю `M_Code` автоинкрементного типа. В подчиненной таблице определен ключ по полю `D_Num` 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 — с буквы `D` (Detail). Подобное именование полей облегчает ориентацию в их названиях, особенно при большом количестве таблиц.

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 записи. 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 направлением. 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 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 выполняться автоматически);
- контроль (запрет) редактирования полей связи;
- организацию (запрет) каскадного удаления записей.

Например, при удалении 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 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 помощью компонентов `Database`, `ADOConnection`, `TSQLConnection`, `IBTransaction` (для технологий BDE, ADO, DBExpress и InterBase Express соответственно) 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 из первой таблицы, но во вторую таблицу еще не попала. Если запись сначала заносится во вторую таблицу, а потом удаляется из первой таблицы, то при сбое возможна ситуация, когда запись будет находиться одновременно 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 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

нии, 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.

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 рассматриваемом 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 7*, 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 *(см. главу 12)*.

Таблицы форматов dBase и Paradox

C++ Builder 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 значения `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

Таблица 2.2 *(продолжение)*

Тип	Обозначение	Описание значения
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
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 автоинкрементного поля больше не будет использовано. Значение автоинкрементного поля доступно для чтения и обычно используется в качестве ключевого поля

Таблица 2.2 *(окончание)*

Тип	Обозначение	Описание значения
`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 C++ Builder 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 цифр и латинских букв.

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, из таблицы заказов. Такое 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 (исключительная ситуация), 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 запре-

щается вводить отрицательные значения. Максимальное значение поля `Salary` зависит от организации, страны и 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, требующихся 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 базами данных

Система C++ Builder не имеет своего формата таблиц БД, тем не менее она обеспечивает мощную поддержку большого количества различных СУБД — как локальных (например, dBase и Paradox), так и промышленных (напри-

мер, Informix, Sybase и InterBase). Средства C++ Builder, 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 C++ Builder 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 C++ Builder 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 C++ Builder-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 C++ Builder, доступ также можно организовать напрямую через BDE, не используя драйвер SQL Links. 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`.
- 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 C++ Builder предоставляет разработчику специальные объекты, например, объекты типа `TField`, представляющие поля наборов данных. Как и другие элементы управления C++ Builder, 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 называют также *элементами, чувствительными к данным*.

Компоненты, используемые для работы с БД, находятся на страницах **Data Access**, **Data Controls**, **dbExpress**, **DataSnap**, **BDE**, **ADO**, **InterBase**, **Decision Cube**, **QReport** и **InterBase Admin** Палитры компонентов. Некоторые из этих компонентов предназначены специально для работы с удаленными БД в архитектуре "клиент-сервер".

Замечание

Состав компонентов можно настроить в диалоговом окне **Palette Properties** (Свойства палитры), вызываемом командой **Properties** (Свойства) контекстного меню Палитры компонентов. Здесь приведен состав Палитры компонентов, который получается после установки C++ Builder 6.

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).

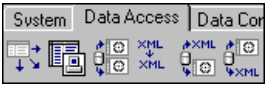


Рис. 2.3. Страница Data Access

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 для работы с базами данных с помощью dbExpress:

- `SQLConnection` (инкапсуляция dbExpress-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);
- `SQLClientDataSet` (клиентский набор данных).

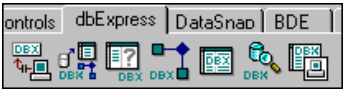


Рис. 2.5. Страница dbExpress

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** (см. рис. 2.7) 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);
- `BDEClienDataSet` (клиентский набор данных, использующий BDE).

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

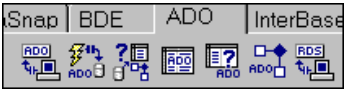


Рис. 2.8. Страница ADO

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).

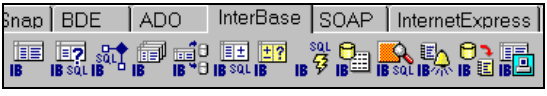


Рис. 2.9. Страница InterBase

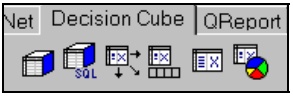


Рис. 2.10. Страница Decision Cube

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

- `QuickRep` (отчет);
- `QRSubDetail` (полоса отчета для таблиц, связанных отношением "главный-подчиненный");
- `QRStringsBand` (строковая полоса отчета);
- `QRBand` (полоса отчета);
- `QRChildBand` (дочерняя полоса отчета);
- `QRGroup` (группа);
- `QRLabel` (надпись);
- `QRDBText` (текстовое поле набора данных);
- `QRExpr` (выражение);
- `QRSysData` (системная информация);
- `QRMemo` (многострочный текст);
- `QRExprMemo` (многострочное выражение);
- `QRRichText` (форматированный текст);

- QRDBRichText (форматированный текст поля набора данных);
- QRShape (геометрическая фигура);
- QRImage (графический образ);
- QRDBImage (графический образ поля набора данных);
- QRCompositeReport (составной отчет);
- QRPreview (окно просмотра отчета);
- QRTextFilter (текстовый фильтр);
- QRCSVFilter (CSV-фильтр);
- QRHTMLFilter (HTML-фильтр);
- QRChart (диаграмма).

Рис. 2.11. Страница QReport

Имена многих компонентов, предназначенных для работы с данными, содержат префиксы, например, DB, IB или QR. 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 QR означает, что компонент используется для построения отчетов QuickReport. 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 C++ Builder предоставляет следующие дополнительные классы исключений:

- EDatabaseError — ошибка БД. Потомки этого класса:
  - EDBEngineError — ошибка BDE (для локальных и сетевых БД);
  - EDBClient — ошибка в приложении клиента (для сетевых БД архитектуры "клиент-сервер"); код ошибки ErrorCode 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 (объектом класса `TDataSet` и его потомков, 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 `int` — содержит количество возникших исключений;
- `__property TDBError* Errors[int Index] = {read=GetError};` — 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.

Объект класса `TDBError` содержит информацию об исключении, определяемую следующими свойствами:

- `Message` типа `AnsiString` (текст сообщения, характеризующего возникшую ошибку);
- `ErrorCode` типа `Word` (код ошибки);
- `Category` типа `Byte` (категория исключения);
- `SubCode` типа `Byte` (группа, или подкод, исключения);
- `NativeError` типа `int` (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 далее.

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 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 соответствующих 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*.

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

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
try
 {if (OpenDialog1->Execute())
 {Table1->Active = false;
  Table1->TableName = OpenDialog1->FileName;
  Table1->Active = true;
  }
  }
 catch (EDatabaseError&Err){ MessageDlg("Ошибка EDatabaseError!",
    mtError, TMsgDlgButtons() << mbOK, 0);
     }
 catch (EDBEngineError&Err){ MessageDlg("Ошибка EDBEngineError!",
    mtError, TMsgDlgButtons() << mbOK, 0);
    }
```

```
  catch (...) {MessageDlg("Неопознанная ошибка!", mtError,
TMsgDlgButtons() << mbOK, 0);}
}
```

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

Проблемы и подходы к проектированию

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

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

Проектирование структуры данных (структуры БД) называют также *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 последующая декомпозиция 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 средств автоматизации разработки программного обеспечения (CASE-средств) схемы БД или прикладной информационной системы;
- структурирование информации в процессе проведения системного анализа на основе совокупности правил и рекомендаций.

Рассмотрим первый подход к проектированию структур данных, как исторически более ранний и являющийся классическим. Проиллюстрируем его на примере проектирования БД, содержащей сведения о сотрудниках фирмы.

Естественным является сбор информации об объектах решаемой задачи (в нашем случае о людях, работающих в одном институте) и размещение ее в одной глобальной таблице. Делается это просто: берется самый общий объект (работник института), для него выделяются атрибуты (характеризующие его свойства с учетом задач, стоящих перед БД) и все это организуется в единую таблицу (табл. 3.1).

Таблица 3.1. *Сотрудники института*

Фамилия, инициалы	**Должность**	**Дата рождения**	**Оклад по должности**	**Ученая степень**	**Надбавка за степень**	**Телефон**
Петров И. И.	Директор	11.05.44	12 000	Д. т. н.	1500	990-42-90
Сидоров А. А.	Зам. директора	12.07.48	8000	К. т. н.	900	777-12-88
Столяров И. И.	Зав. лаб.	03.02.51	6000	Д. т. н.	1500	123-33-44
. . .	. . .	. . .	. . .	. . .	. . .	. . .
Орлов Г. Н.	Научный сотрудник	06.11.68	2500	—	—	333-99-00

Недостатками такой уникальной таблицы являются:

- жесткость (обязательная модификация самой таблицы при изменении постановки задачи, например, если потребуется хранить данные о количестве и именах детей сотрудников);
- избыточность данных (под избыточностью понимают дублирование данных, содержащихся в БД) и, как следствие, потенциальная противоречивость и повышенный расход ресурсов.

При выполнении операций с данными избыточность приводит к различным аномалиям — нарушению целостности БД. Выделяют следующие аномалии:

- удаления;
- обновления;
- ввода.

Различают простое (неизбыточное) дублирование и избыточное дублирование данных.

Примером *неизбыточного дублирования* может служить список внутренних телефонов сотрудников организации (табл. 3.2). Такое дублирование является естественным и допустимым.

Таблица 3.2. *Пример неизбыточного дублирования*

Сотрудник	Телефон
Иванов А. А.	12-36
Петров П. П.	12-36
Сидоров С. С.	45-62
Кузнецов К. К.	78-91
Васин В. В.	12-36

В приведенном примере три сотрудника имеют одинаковый номер телефона 12-36, это может быть, 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 фамилии сотрудника — Иванова А. А., Петрова П. П. или Васина В. В., что является *аномалией удаления.*

При смене номера телефона в комнате его нужно изменить для всех сотрудников, которые в ней находятся. Если для какого-либо сотрудника этого не сделать, например, для Петрова П. П., то возникает несоответствие данных, называемое *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).

Рассмотрим пример *избыточного дублирования* данных. Список телефонов (табл. 3.3) дополним номерами комнат, в которых находятся сотрудники. При этом номер телефона укажем только для одного из сотрудников — в примере для Иванова А. А., 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 3.3. *Пример избыточного дублирования*

Сотрудник	Комната	Телефон
Иванов А. А.	17	12-36
Петров П. П.	17	12-36
Сидоров С. С.	22	45-62

Таблица 3.3. *(окончание)*

Сотрудник	Комната	Телефон
Кузнецов К. К.	8	78-91
Васин В. В.	17	12-36

При таком построении таблицы появляются проблемы. При определении номера телефона сотрудника необходимо произвести его поиск в другой строке 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 (в примере — Иванова А. А.), будет утеряна информация о номере телефона для всех сотрудников 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. 3.4 и 3.5). *Разбиение* (декомпозиция) — это процесс деления таблицы на несколько 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 по номеру комнаты считывается номер телефона.

Таблица 3.4. *Список сотрудников и номеров комнат*

Сотрудник	Комната
Иванов А. А.	17
Петров П. П.	17
Сидоров С. С.	22
Кузнецов К. К.	8
Васин В. В.	17

Таблица 3.5. *Список номеров комнат и телефонов*

Комната	Телефон
17	12-36
22	45-62
8	78-91

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.

Отметим, что для сотрудников указывается номер комнаты, который приводит к неизбыточному дублированию данных. Как указано выше, такое дублирование является приемлемым для БД.

Функциональные зависимости атрибутов

Как отмечалось, метод нормальных форм является классическим методом проектирования реляционных БД. Он основан на фундаментальном в теории реляционных БД понятии — *зависимости между атрибутами* отношений.

Между атрибутами отношений существуют следующие основные виды зависимостей:

- функциональные;
- многозначные;
- транзитивные.

Понятие функциональной зависимости является базовым.

Определение. Атрибут *B функционально* зависит от атрибута *A*, если каждому значению *A* соответствует в точности одно значение *B*.

A и *B* могут состоять из нескольких атрибутов (быть составными).

Различают частичную и полную функциональные зависимости. Под *частичной* функциональной зависимостью понимают зависимость неключевого атрибута от части составного ключа.

Полная функциональная зависимость — зависимость неключевого атрибута от всего составного ключа.

Определение. В отношении *R* атрибут *B многозначно* зависит от атрибута *A*, если каждому значению *A* соответствует множество значений *B*, не связанных с другими атрибутами из *R*.

Многозначные зависимости могут быть:

- *"один-ко-многим" (1:M)* — одному значению атрибута *A* соответствует несколько значений атрибута *B*;
- *"многие-к-одному" (M:1)* — несколько значений атрибута *A* соответствует одному значению атрибута *B*;

- *"многие-ко-многим" (M:M)* — нескольким значениям атрибута *A* соответствует несколько значений атрибута *B*.

Определение. Атрибут *C* зависит от атрибута *A* *транзитивно*, если для атрибутов *A*, *B* и *C* выполняются следующие условия: атрибут *C* функционально зависит от атрибута *B*, а атрибут *B* функционально зависит от атрибута *A*, но обратная зависимость отсутствует.

Нормализация баз данных

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

Выделяют следующую последовательность нормальных форм:

- первая нормальная форма (1НФ);
- вторая нормальная форма (2НФ);
- третья нормальная форма (3НФ).

Имеются также нормальные формы высшего порядка, к которым относятся:

- нормальная форма Бойса—Кодда (БКНФ), являющаяся усиленной 3НФ;
- четвертая нормальная форма (4НФ);
- пятая нормальная форма (5НФ).

Первая нормальная форма. Отношение находится в первой нормальной форме, если все его атрибуты являются простыми (имеют единственное значение, а не массив, список, множество и т. д.).

Исходное отношение строится таким образом, чтобы оно было в 1НФ.

Перевод отношения в следующую нормальную форму осуществляется методом разбиения (декомпозиции) без потерь.

Вторая нормальная форма. Отношение имеет вторую нормальную форму, если оно имеет 1НФ и каждый атрибут отношения, не входящий ни в один ключ (т. е. неключевой атрибут), полностью зависит от любого возможного ключа целиком, а не от его подмножества.

Приведение отношения ко второй нормальной форме позволяет убрать зависимость неключевых атрибутов от части ключа.

Если все ключи в отношении состоят только из одного атрибута, то отношение *автоматически* имеет 2НФ.

Третья нормальная форма. Отношение находится в третьей нормальной форме, если оно находится в 2НФ и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

3НФ исключает транзитивную зависимость атрибутов.

Приведение отношения к 3НФ в большинстве случаев оказывается достаточным, чтобы избежать проблем, возникающих с ненормализованными отношениями. На практике, как правило, процесс проектирования реляционной БД на этом заканчивается. Поэтому рассмотрение нормальных форм высшего порядка нами опускается.

Сказанное проиллюстрируем примером нормализации БД о сотрудниках института, представленной в табл. 3.1.

Как отмечалось, 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.

Проверим исходную таблицу (см. табл. 3.1) на соответствия первой нормальной форме, для которой должны выполняться следующие условия:

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

Очевидно, что данная исходная таблица находится в 1НФ.

На следующем шаге определим первичный ключ отношения. Анализ существующих атрибутов позволяет предложить в качестве первичного ключа отношения атрибут "Фамилия, инициалы". Но в организации возможно наличие сотрудников с одинаковыми фамилиями и даже инициалами. Следовательно, в нашем примере первичный ключ должен быть составным. Наиболее удачным в данном случае будет использование в качестве первичного ключа атрибутов "Фамилия, инициалы" и "Должность".

Замечание

Строго говоря, возможна ситуация, когда и эти атрибуты перестанут удовлетворять требованиям, предъявляемым к первичному ключу, поэтому "правилом хорошего тона" является введение искусственного ключа.

Очередным шагом будет приведение нашего отношения к 2НФ. Анализ существующих функциональных зависимостей позволяет сделать вывод, что атрибут "Оклад по должности" находится в зависимости от части первичного ключа — атрибута "Должность". Для перехода к 2НФ в этом случае ис-

ходную таблицу необходимо разбить на две связанные таблицы: главную и подчиненную. Связь между этими таблицами осуществляется по полю "Должность", выполняющему функции внешнего ключа для подчиненной таблицы. Результат приведения исходного отношения к 2НФ представлен в виде табл. 3.6 и 3.7.

Таблица 3.6. *Главная таблица*

Должность	Оклад по должности
Директор	12 000
Зам. директора	8000
Зав. лаб.	6000
. . .	. . .
Научный сотрудник	2500

Таблица 3.7. *Подчиненная таблица*

Фамилия, инициалы	Должность	Дата рождения	Ученая степень	Надбавка за степень	Телефон
Петров И. И.	Директор	11.05.44	Д. т. н.	1500	990-42-90
Сидоров А. А.	Зам. директора	12.07.48	К. т. н.	900	777-12-88
Столяров И. И.	Зав. лаб.	03.02.51	Д. т. н.	1500	123-33-44
. . .	. . .	. . .	. . .	. . .	. . .
Орлов Г. Н.	Научный сотрудник	06.11.68	—	—	333-99-00

Проверим отношения на выполнение требований 3НФ. Очевидно, что исходная таблица содержит транзитивную зависимость атрибута "Надбавка за степень" от первичного ключа. Чтобы исключить эту зависимость, вновь воспользуемся методом декомпозиции. Для этого сформируем еще одну пару таблиц. В главную таблицу войдут находящиеся в функциональной зависимости атрибуты "Ученая степень" и "Надбавка за степень", а подчиненная будет содержать оставшиеся атрибуты исходной таблицы. Связь главной таблицы с подчиненной в этом случае будет осуществляться по полю " Ученая степень", которое выполняет функции внешнего ключа. Результат приведения отношений к 3НФ изображен в виде табл. 3.8 и 3.9.

Таблица 3.8. *Таблицы, являющиеся главными при организации связей*

Должность	Оклад по должности
Директор	12 000
Зам. директора	8000
Зав. лаб.	6000
. . .	. . .
Научный сотрудник	2500

Ученая степень	Надбавка за степень
Д.т.н.	1500
К.т.н.	900
-	-

Таблица 3.9. *Подчиненная таблица*

Фамилия, инициалы	Должность	Дата рождения	Ученая степень	Телефон
Петров И. И.	Директор	11.05.44	Д. т. н.	990-42-90
Сидоров А. А.	Зам. директора	12.07.48	К. т. н.	777-12-88
Столяров И. И.	Зав. лаб.	03.02.51	Д. т. н.	123-33-44
. . .	. . .	. . .	. . .	. . .
Орлов Г. Н.	Научный сотрудник	06.11.68	—	333-99-00

Анализ полученных отношений позволяет сделать вывод, что избыточное дублирование данных устранено и, следовательно, исключена возможность возникновения аномалий при работе с данными. Проектирование БД на этом можно считать завершенным.

В заключение дадим ряд рекомендаций для отдельных шагов проектирования реляционной БД, позволяющих облегчить этот процесс.

- **Шаг 1** — анализ предметной области. Необходимо четко выделить объекты (сущности), которые представляют интерес в данной задаче, классифицировать их, определить для них необходимые атрибуты. Если при описании какого-либо объекта (сущности) возникает необходимость задействовать другой объект, то описание этого объекта должно быть вынесено в отдельную таблицу.
- **Шаг 2** — поиск ключа. Атрибуты, входящие в ключ, должны быть тщательно проанализированы на возможность однозначной идентификации реального объекта. В противном случае необходимо ввести искусственный ключ.

- **Шаг 3** — детальная спецификация атрибутов для таблиц, в том числе определение типов данных, которые будут приписаны этим атрибутам.
- **Шаг 4** — анализ схемы БД на предмет зависимостей и неточностей. Для упрощения этого анализа следует рассматривать действия, которые будут совершаться с хранимыми данными.

Средства CASE

В предыдущем подразделе 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 разного 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 C++ Builder 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), C++ Builder (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 C++ Builder:

- 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 C++ Builder, 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.

Глава 4

Технология создания информационной системы

В состав информационной системы входят БД, приложение, вычислительная система и СУБД. Предположим, что компьютер или компьютерная сеть уже существуют и их характеристики удовлетворяют потребностям будущей информационной системы. В качестве СУБД выберем C++ Builder.

Продемонстрируем возможности C++ Builder по работе с БД на примере создания простой информационной системы. Эту информационную систему можно разработать 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 для работы с БД.

В простейшем случае БД состоит из одной таблицы. Если таблицы уже имеются, то первый этап не выполняется. Отметим, что совместно с C++ Builder поставляется много 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.

Варианты создания таблиц

Создание таблицы для базы данных можно выполнить *программно* (в процессе выполнения приложения) или *с помощью инструментального средства*, например, Database Desktop (перед началом работы с приложением). Приведем пример кода программного создания таблицы для базы данных:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
Table1->Active = false; // Компонент Table1 должен быть не активным
Table1->DatabaseName = "BCDEMOS";
Table1->TableType = ttParadox; // Задается тип таблицы Paradox
Table1->TableName = "CustInfo1"; // Имя таблицы
if (!Table1->Exists) // Проверка существования таблицы
{
 // Описание полей таблицы
 Table1->FieldDefs->Clear();
 TFieldDef *pNewDef = Table1->FieldDefs->AddFieldDef();
 pNewDef->Name = "Fld1"; // Имя 1-го поля
 pNewDef->DataType = ftInteger;
 pNewDef->Required = true;
 pNewDef = Table1->FieldDefs->AddFieldDef();
 pNewDef->Name = "Field2"; // Имя 2-го поля
 pNewDef->DataType = ftString;
 pNewDef->Size = 30;
 // Описание индексов
 Table1->IndexDefs->Clear();
 // 1-й индекс не имеет имени, как первичный ключ в таблице Paradox
 Table1->IndexDefs->Add("","Field1",  TIndexOptions()  <<ixPrimary  <<
ixUnique);
 Table1->IndexDefs->Add("Fld2Index","Field2",  TIndexOptions()  <<  ixCa-
seInsensitive);
 // Создание таблицы и ее активизация
 Table1->CreateTable();
 Table1->Active = true;
 }
}
```

В приведенном примере в обработчике события создания формы приложения выполняется описание полей, задание индексов, создание и активизация таблицы. Такой вариант создания таблиц, по-видимому, наиболее целесообразно использовать для создания временных таблиц с последующим их использованием и удалением. Более удобным вариантом создания таблиц является использование инструментальных средств.

Для свойства `TableTYpe`, определяющего тип таблицы, могут задаваться также следующие значения:

- `ttDefault` — таблица, тип которой назначается в зависимости от расширения имени файла (db или без расширения — таблица Paradox, dbf — таблица dBASE, txt — текстовый файл);
- `ttDBase` — таблица dBASE;
- `ttFoxPro` — таблица FoxPro;
- `ttASCI` — текстовый файл произвольной длины с ограниченными строками для каждого поля.

Создание таблицы с помощью Database Desktop

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, выполнение визуальных и SQL-запросов, операции с псевдонимами), которые будут рассматриваться в *главе 12*.

Замечание

Почти все рассматриваемые далее действия по управлению структурой таблицы можно выполнить также программно, как 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) 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.

После выбора формата таблицы появляется окно определения структуры таблицы (рис. 4.1), 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 необходимости. Часть действий, таких как задание ключа и паролей, производится только для таблиц определенных форматов, например, для таблиц Paradox.

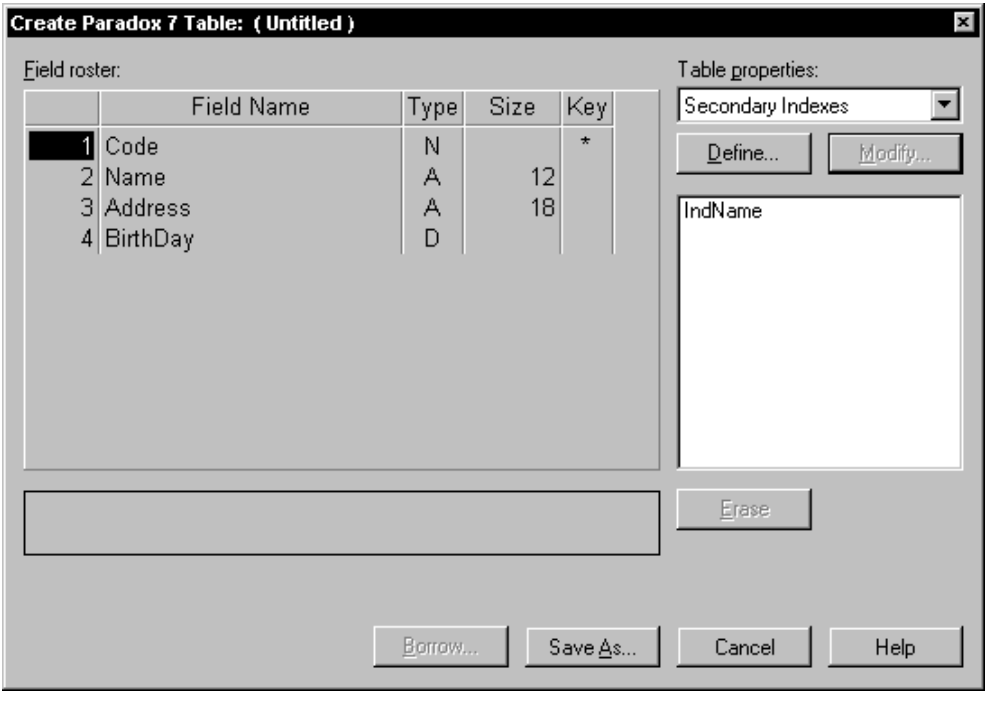


Рис. 4.1. Определение структуры таблицы

При создании новой таблицы сразу после выбора ее формата можно не создавать структуру таблицы, а скопировать ее из другой таблицы: при нажатии кнопки **Borrow** (Взаймы) открывается окно **Select Borrow Table** (Выбор таблицы для заимствования) (рис. 4.2).

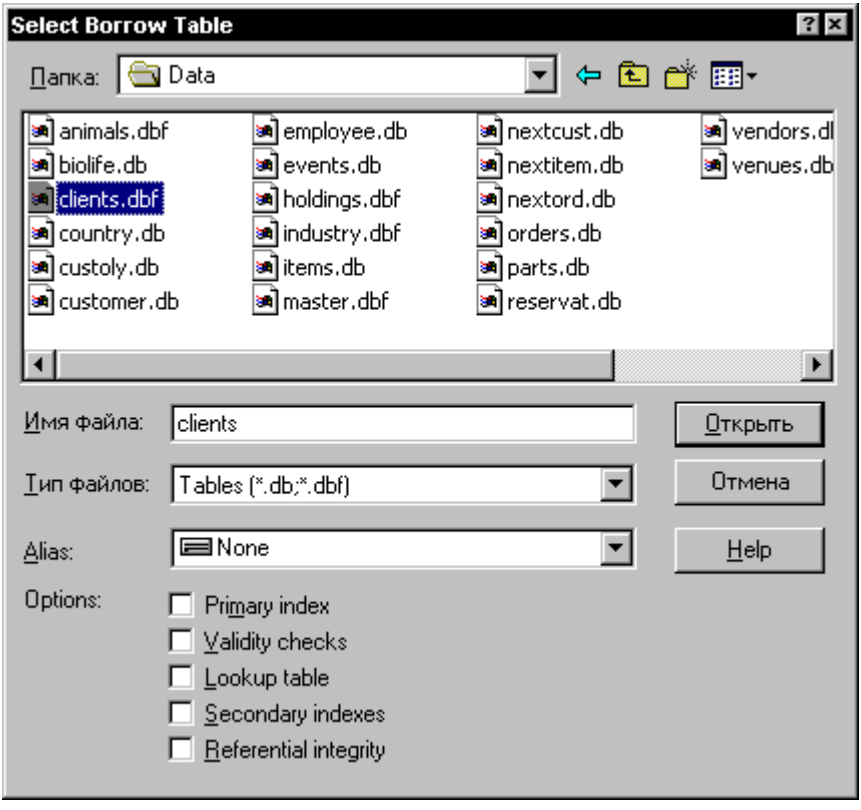


Рис. 4.2. Выбор таблицы для заимствования ее структуры

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 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 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. 4.3), раскрываемом нажатием клавиши <пробел> или щелчком правой кнопки мыши в столбце **Type** (рис. 4.1). 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>.

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. 4.1).

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.

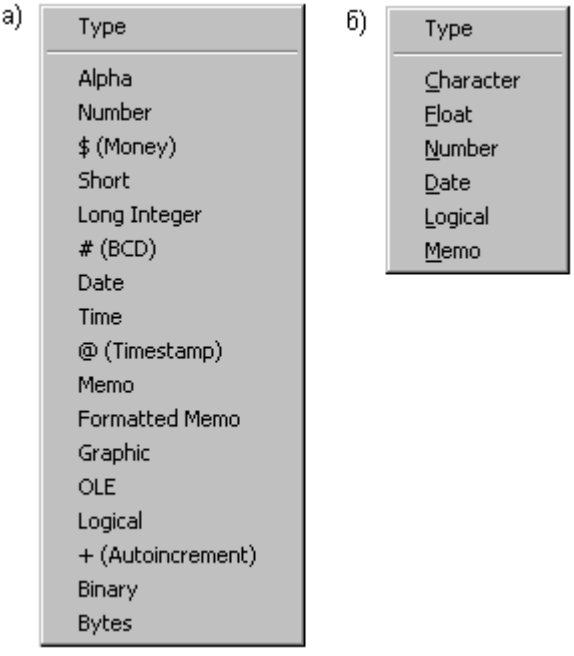


Рис. 4.3. Типы полей таблиц:
а) Paradox 7, б) dBASE IV

Для выполнения остальных действий по определению структуры таблицы используется комбинированный список **Table properties** (Свойства таблицы) (см. рис. 4.1), 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. Напомним, что для таблиц Paradox индекс называют не просто индексом, а 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 `indName`.

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. 4.4).

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.

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.

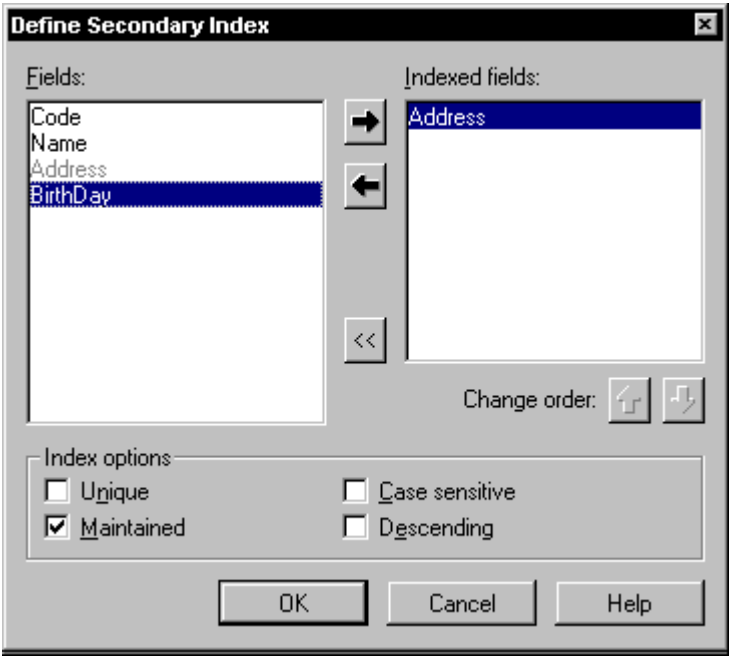


Рис. 4.4. Окно задания индекса

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. 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 целостности 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. 4.4). При нажатии кнопки **OK** появляется окно сохранения индекса, содержащее имя изменяемого индекса, которое можно исправить или оставить прежним.

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.

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 C++ Builder. 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. 4.1), при этом ниже списка появляются флажок **Required Field** (Обязательное поле), поля редактирования **Minimum Value**, **Maximum Value**, **Default Value**, **Picture** (Маска ввода) и кнопка **Assist** (Помощь). 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 возникнуть также 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. 4.1) 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) (рис. 4.5), вызываемое нажатием кнопки **Assist**.

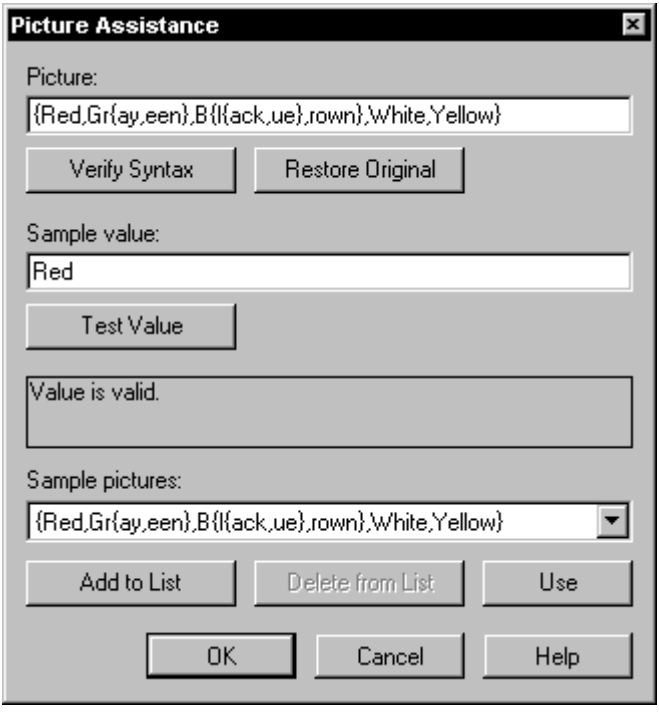


Рис. 4.5. Диалоговое окно формирования маски

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. 4.1). 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. 4.6).

Поле связи следует выбрать в списке **Fields** 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.

Главная таблица указывается в списке **Table**, имена таблиц выбираются в рабочем каталоге программы Database Desktop (каталог, определенный как Working Directory). 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

Задание рабочего каталога программы Database Desktop выполняется с помощью команды **File\Working Directory** (Файл\Рабочий каталог).

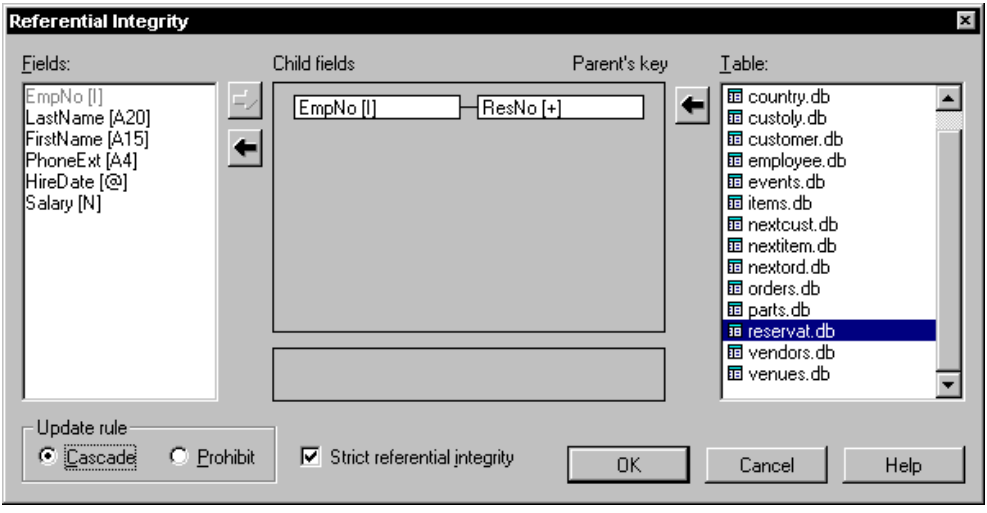


Рис. 4.6. Задание условия ссылочной целостности

Главная таблица должна быть закрыта и не может 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. 4.7). 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.

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 вы-

деления имени условия в списке и нажатия кнопки **Modify** открывается окно определения ссылочной целостности (см. рис. 4.6). При нажатии кнопки **OK** измененное условие ссылочной целостности сохраняется под тем же именем.

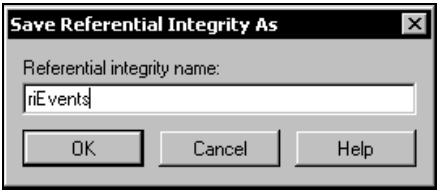


Рис. 4.7. Задание имени для условия ссылочной целостности

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 C++ Builder.

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 C++ Builder.

Для выполнения операций, связанных с заданием пароля, нужно выбрать строку **Password Security** в комбинированном списке **Table properties** окна определения структуры таблицы (см. рис. 4.1). При этом под списком становятся доступными кнопки **Define** и **Modify**. Нажатие кнопки **Define** открывает окно **Password Security** (рис. 4.8), в котором задается главный пароль.

Главный пароль таблицы вводится дважды — в полях **Master password** (Главный пароль) и **Verify master password** (Подтвердить главный пароль). При нажатии кнопки **OK** значения сверяются, и при их совпадении пароль принимается. Когда пароль определен, кнопка **Define** блокируется и стано-

вится доступной кнопка **Modify** изменения пароля. Ее нажатие снова вызывает окно задания пароля, в котором появляются кнопки **Change** и **Delete**, а поля ввода заблокированы.

Рис. 4.8. Задание главного пароля

При нажатии кнопки **Delete** главный пароль удаляется, после чего его можно ввести заново. Если в качестве значения главного 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 редактирования.

Рассмотренный нами главный пароль 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. 4.9).

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.

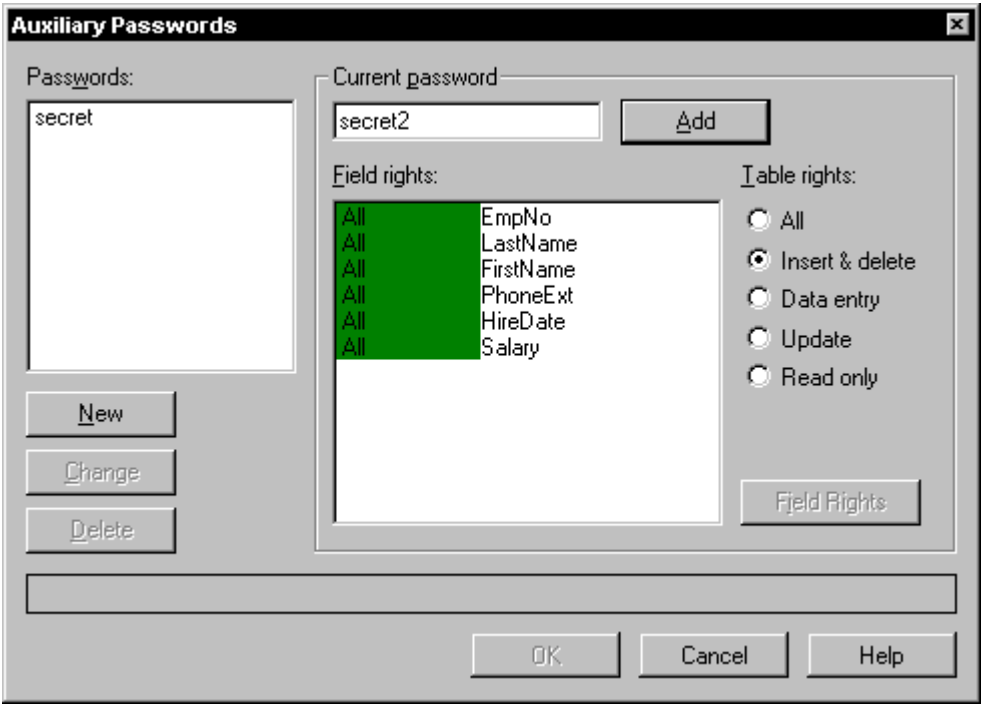


Рис. 4.9. Задание дополнительных паролей

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 редак-

тирования 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**.

Замечание

При выполнении приложения, использующего механизм доступа BDE, можно добавлять и удалять пароли с помощью методов компонента Session. Метод void __fastcall AddPassword(const AnsiString Password); добавляет новый пароль, заданный параметром Password; метод void __fastcall RemovePassword(const AnsiString Password); удаляет указанный пароль, а метод void __fastcall RemoveAllPasswords(void); удаляет все пароли. Например, добавление пароля задается следующим образом: Session->AddPassword("Word_secret");.

Задание языкового драйвера

Для задания языкового драйвера нужно выбрать пункт **Table Language** (Язык таблицы) комбинированного списка **Table properties** окна определения структуры таблицы (см. рис. 4.1). При этом под списком становится доступной кнопка **Modify**, открывающая окно **Table Language** (рис. 4.10). Под "языком" таблицы понимается языковой драйвер, используемый для этой таблицы. В поле списка **Language** отображается текущий драйвер.

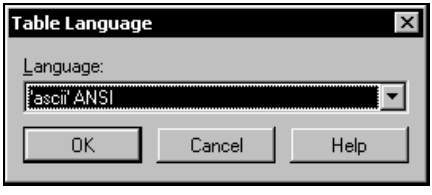


Рис. 4.10. Выбор языкового драйвера

В списке можно выбрать драйвер нужного языка, для русского языка — 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 BDE Administrator. Параметры BDE и их настройка рассматриваются в *главе 12*.

Задание таблицы для выбора значений

На практике возможны случаи, когда в поле таблицы должны заноситься значения из какого-либо набора, который может формироваться различны-

ми 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. 4.1). При этом под списком становится доступной кнопка **Define**, нажатие которой открывает окно **Table Lookup** (рис. 4.11).

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.

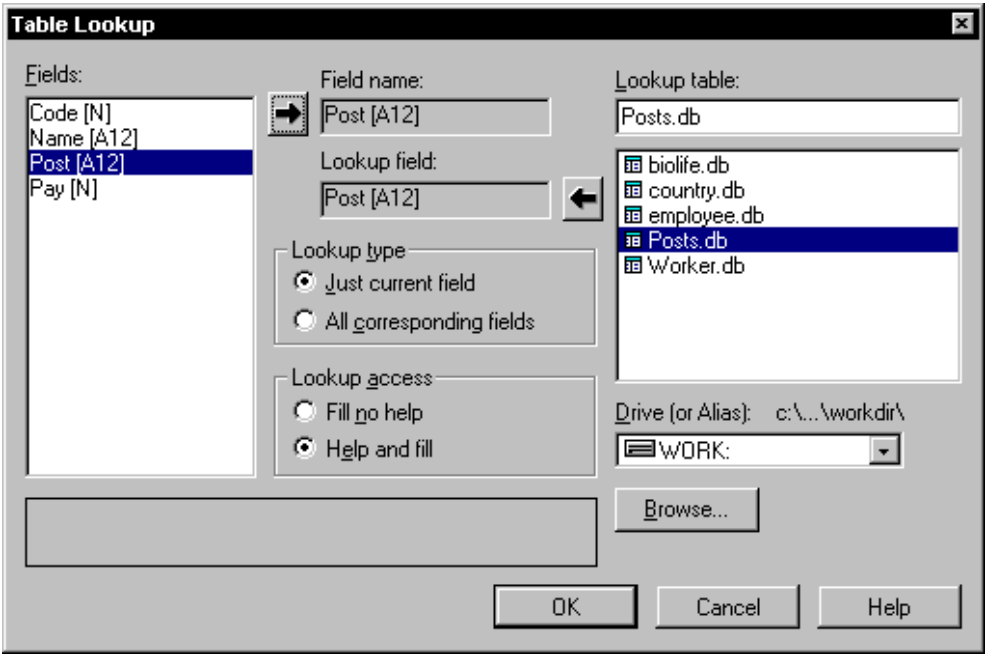


Рис. 4.11. Окно задания таблицы выбора

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.

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. 4.12), появляется при нажатии комбинации клавиш <Ctrl>+<пробел> в редактируемом поле. После выбора нужного значения и нажатия кнопки **OK** оно заносится в поле.

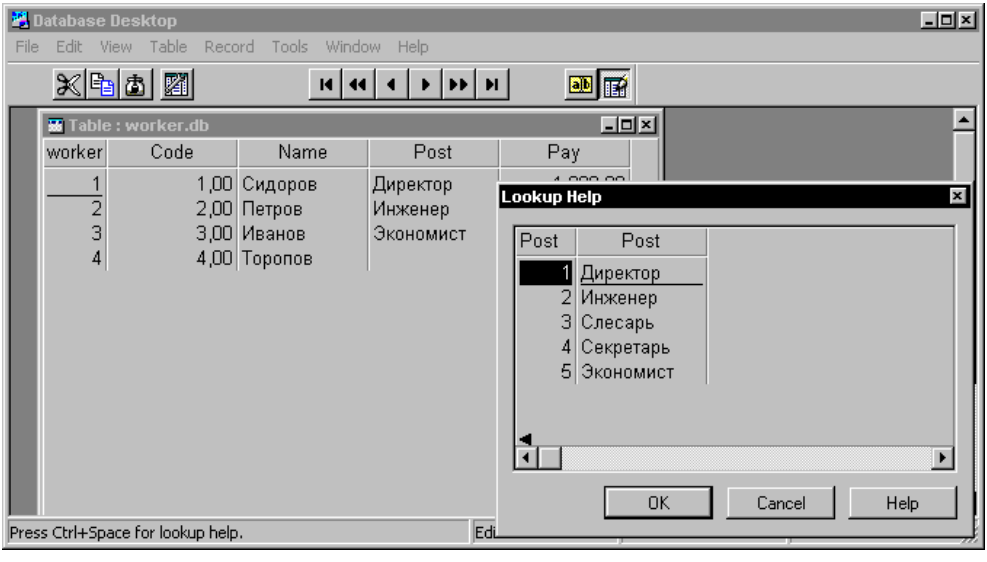


Рис. 4.12. Заполнение поля с использованием таблицы выбора

В приведенном на этом рисунке примере в окне программы Database Desktop таблица `Worker` содержит данные о сотрудниках организации. В частности, в ней определено строковое поле должности (`Post`), для которого задана таблица выбора `Posts`. В это поле можно вводить только допустимое название должности. Перечень всех должностей содержится в единственном поле `Post` таблицы `Posts`. Для предотвращения ввода в таблицу `Worker` неправильных значений ее полю `Post` назначена таблица выбора `Posts` с полем `Post`. В таблице `Worker` для первых трех сотрудников введены должности, имеющиеся в таблице `Posts`. Попытка задать для четвертого сотрудника любую должность, которая не входит в перечень должностей из таблицы `Posts`, будет блокироваться, т. к. эта должность не является разрешенной для ввода. При необходимости эту должность сначала нужно ввести в таблицу `Posts`, после чего она станет доступной для таблицы `Worker`.

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. 4.1) 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 *главах 5* и *6*.

Просмотр списка подчиненных таблиц

Таблица, связанная с другими таблицами, является либо главной, либо подчиненной. Ранее мы рассмотрели задание ссылочной целостности для подчиненной таблицы. Для главной таблицы можно просмотреть список подчиненных таблиц, отображаемый при выборе пункта **Dependent Table** (Подчиненная таблица) комбинированного списка **Table properties** (см. рис. 4.1). 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 C++ Builder. Поэтому предварительно необходимо закрыть C++ Builder или приложение, в котором компоненты Table связаны с перестраи-

ваемой таблицей. Другим вариантом является отключение активности компонентов Table, связанных с перестраиваемой таблицей, для чего свойству Active этих компонентов через Инспектор 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 (см. рис. 4.1) 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\Info Structure**. 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 записи. 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. 4.13, где в форме размещены компоненты `Table1`, `DataSource1`, `DBGrid1` и `DBNavigator1`.

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

В рассматриваемом приложении использована таблица клиентов, входящая в состав поставляемых с C++ Builder примеров, ее главный файл — Clients.dbf. Файлы этой и других таблиц примеров находятся в каталоге, путь к которому указывает псевдоним `BCDEMOS`. 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 взаи-

модействует с этой таблицей. На компонент `Table1`, с которым связан компонент `DataSource1`, указывает свойство `DataSet` последнего.

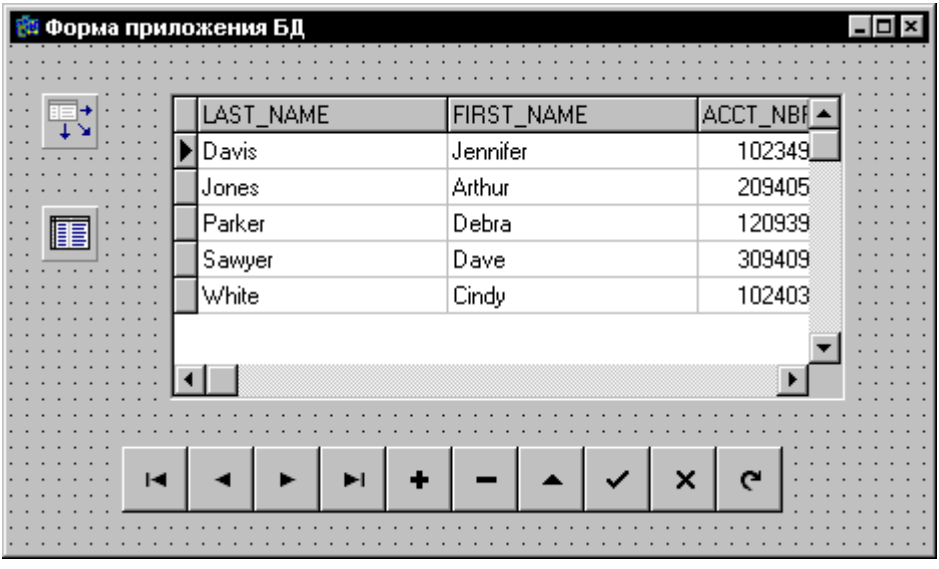


Рис. 4.13. Форма приложения для работы с БД

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.

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`.

Схема взаимосвязи компонентов приложения и таблицы БД и используемые при этом свойства компонентов показаны на рис. 4.14.

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.

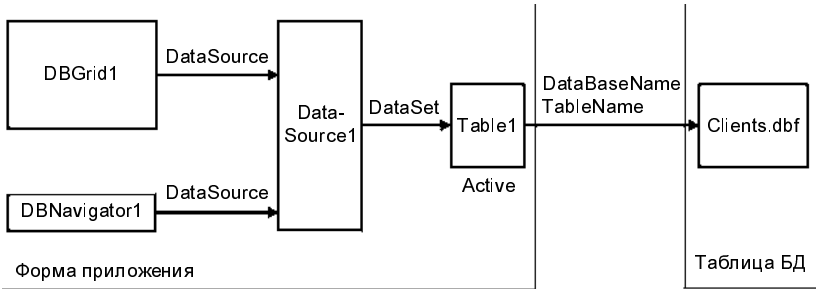


Рис. 4.14. Схема взаимосвязи компонентов приложения и таблицы БД

В последнем случае приложение создается визуально 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. 4.15. Этот Мастер расположен на странице **Business** Хранилища объектов, вызываемого с помощью команды меню **File\New\Other** (Файл\Создать\Другой).

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

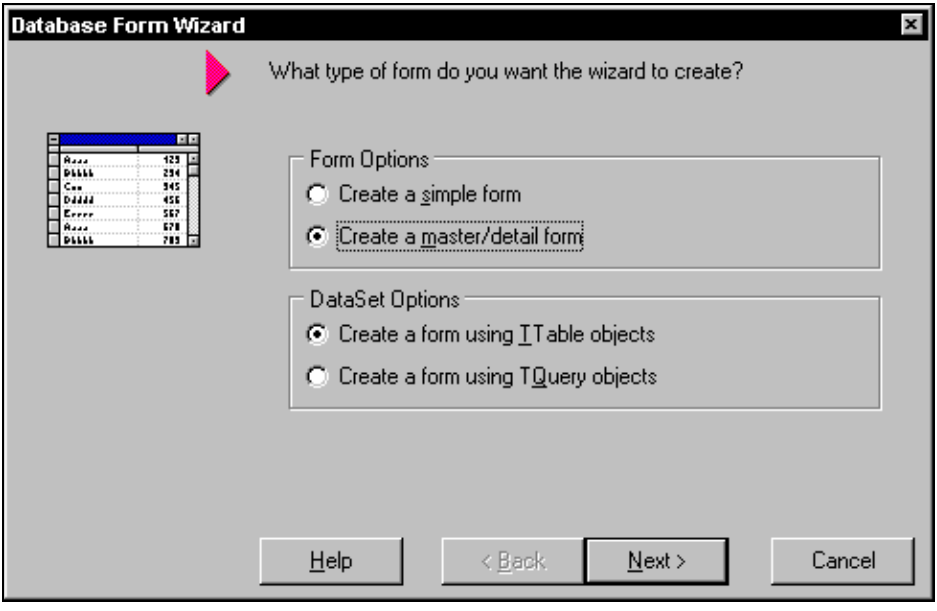


Рис. 4.15. Окно Мастера **Database Form Wizard**

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.

Далее рассматривается *простой модуль* данных, который представлен объектом `DataModule`. Использование *удаленного модуля* данных изучается при рассмотрении трехуровневых приложений *(см. главу 18)*.

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.

Если применяется простой модуль данных, то схема взаимосвязи компонентов приложения и таблицы БД имеет вид, показанный на рис. 4.16.

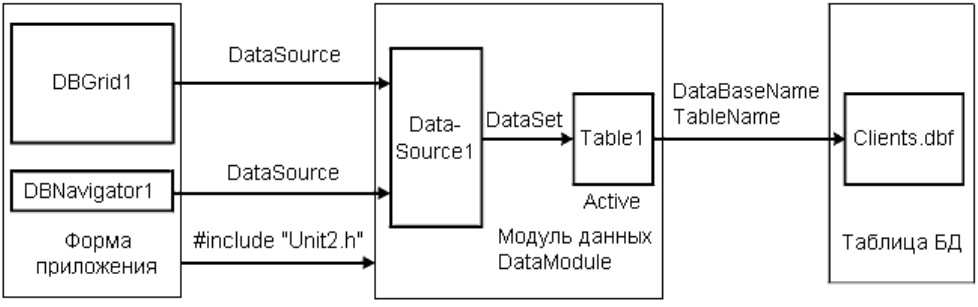


Рис. 4.16. Схема взаимосвязи компонентов и таблицы при использовании модуля данных

Модуль данных, как и форма, является контейнером для своих невизуальных компонентов, и для него создается модуль кода с расширением cpp. Добавление модуля данных к проекту выполняется командой **File\New\DataModule** главного меню C++ Builder. В окне модуля компоненты размещаются так же, как и в форме (рис. 4.17). При выборе объекта в окне Инспектора объектов отображаются его свойства, значения которых можно просматривать и изменять.

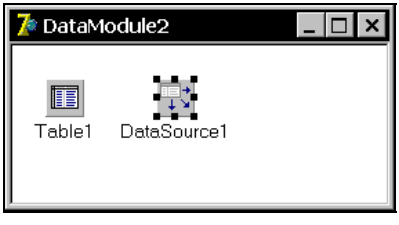


Рис. 4.17. Модуль данных

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>
```

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

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        DataModule2->Table1->DatabaseName = "BCDEMOS";
        DataModule2->Table1->TableName = "Clients.dbf";
        DataModule2->DataSource1->DataSet = DataModule2->Table1;
        DBGrid1->DataSource = DataModule2->DataSource1;
        DBNavigator1->DataSource = DataModule2->DataSource1;
        DataModule2->Table1->Active = true;
}
//---------------------------------------------------------------------------
```

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 нем необходимо указать предложение препроцессора, выполняющего подключение модуля данных:

```
#include "Unit2.h";
```

Ссылку на другой модуль можно написать самостоятельно, но C++ Builder позволяет вставить ее автоматически. При выборе команды **File\Include Unit Hdr** (Файл\Подключить заголовок модуля) появляется диалоговое окно **Use Unit**. После выбора нужного модуля и нажатия кнопки **OK** соответствующее предложение препроцессора добавляется в модуль формы.

Если предложение препроцессора для подключения требуемого модуля отсутствует, но в коде используется имя модуля данных, то при компиляции приложения будет выдано сообщение об ошибке.

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.

Глава 5

Компоненты доступа к данным с помощью BDE

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 освещены в *главах 13* и *14*.

Наборы данных

Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. В C++ Builder *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 C++ Builder, многие СУБД вместо термина *набор данных* используют термины *выборка* или *таблица*.

В C++ Builder 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 InterBase). 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 далее.

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 имеют схожие с базовыми классами характеристики и поведение, но у каждого из них есть 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 `AnsiString`. 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 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 свойству `Active` 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:\Program Files\Borland\CBuilder\Examples\Data";
```

Для набора данных `Table1` таблицы БД расположены в каталоге, на который указывает псевдоним `BDPlace`. Таблицы БД для набора данных `Table2` расположены в каталоге c:\Program Files\Borland\CBuilder\Examples\Data. 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 `int`. 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 при обработке события нажатия кнопки `Button1` производится перебор всех записей набора данных:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Table1->First();
for (int i = 1; i <= Table1->RecordCount; i++)
 {
  // Инструкции, выполняющие
  // обработку очередной записи:
  ListBox1->Items->Add( Table1->Fields->Fields[0]->AsString);
  Table1->Next();
 }
}
```

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`.

В качестве инструкции обработки очередной записи используется заполнение списка (компонент `ListBox1`) значениями первого поля таблицы (`Table1->Fields->Fields[0]`) (рис. 5.1).

Рис. 5.1. Вид формы приложения после обработки записей таблицы

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 `int`, 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.

Напомним *(см. главу 2)*, для выполнения операций с наборами данных используются два способа доступа к данным:

- навигационный;
- реляционный (SQL-ориентированный).

Состояния наборов данных

Наборы данных могут находиться в *открытом* или *закрытом* состояниях, на что указывает свойство `Active` типа `bool`. Если свойству `Active` установле-

но 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 свойству `Active` 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 свойству `Active` наборов данных автоматически устанавливается 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`:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        Query1->Active = false;
    Query1->DatabaseName="BCDEMOS";
        Query1->SQL->Clear();
        Query1->SQL->Add("select * from country.db");
        Query1->Active = true;
}
```

Если набор данных `Query1` связан соответствующим образом с компонентами `DataSource1` и `DBGrid1` *(см. главу 4)*, то при выполнении приложения

нажатие кнопки приведет к отбору всех записей из таблицы country.db и отображению их с помощью компонента DBGrid1.

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

Метод Open открывает набор данных, его вызов эквивалентен установке свойству Active значения true. При вызове метода Open генерируются события BeforeOpen и AfterOpen, а также вызываются процедуры-обработчики этих событий.

Метод Close закрывает набор данных, его вызов эквивалентен установке свойству Active значения 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:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        Query1->Close();
    Query1->DatabaseName="BCDEMOS";
        Query1->SQL->Clear();
        Query1->SQL->Add("select * from country.db");
        Query1->Open();
}
```

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

```
typedef void __fastcall (__closure *TDataSetNotifyEvent)(TDataSet* Data-
Set);
```

TDataSetNotifyEvent является указателем на метод, уведомляющий компонент набора данных о наступлении события. 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:

```
void __fastcall TForm1::Query1BeforeOpen(TDataSet *DataSet)
{
       if (!CheckBox1->Checked) Abort();
}
```

Если флажок `CheckBox1`, управляющий возможностью открытия набора данных, не установлен, то открытие набора данных `Query1` 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, например, как в приведенной процедуре:

```
void __fastcall TForm1::Query1BeforeOpen(TDataSet *DataSet)
{
  if (!CheckBox1->Checked)
   { MessageDlg("Данные запроса "+ Query1->Name + " недоступны!",
    mtError, TMsgDlgButtons() << mbOK, 0);
    Abort();
   }
}
```

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:

```
void __fastcall TForm1::Table1AfterOpen(TDataSet *DataSet)
{
if (CheckBox2->Checked)
   { MessageDlg("Данные таблицы "+ Table1->TableName + " доступны!",
    mtInformation, TMsgDlgButtons() << mbOK, 0);
    Abort();
   }
}
```

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`.

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:

```
void __fastcall TForm1::Table1BeforeClose(TDataSet *DataSet)
{
if (Table1->State == dsEdit || Table1->State == dsInsert) Table1->Post();
}
```

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 процедуре:

```
void __fastcall TForm1::Table1AfterClose(TDataSet *DataSet)
{
        if (CheckBox2->Checked)
    {
         Beep();
         MessageDlg("Таблица "+ Table1->TableName + " закрыта!",
            mtInformation, TMsgDlgButtons() << mbOK, 0);
    }
}
```

Если установлен флажок `CheckBox2`, управляющий режимом выдачи сообщений, то после закрытия набора данных `Table1` подается звуковой сигнал

и выдается сообщение о закрытии таблицы, связанной с этим набором данных.

Заметим, 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 при выполнении 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 режимов:

- `dsInactive` (неактивен) — набор данных закрыт и доступ к его данным невозможен. В этот режим набор данных переходит после своего закрытия, когда свойству `Active` установлено значение `false`;
- `dsBrowse` (навигация по записям набора данных и просмотр данных) — в этот режим набор данных переходит так:
  - из режима `dsInactive` — при установке свойству `Active` значения `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 типа `TTable`, т. к. для компонента типа `TQuery` 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 свойству `NewValue` компонента поля);
- `dsOldValue` (обращение к свойству `OldValue` компонента поля);
- `dsCurValue` (обращение к свойству `CurValue` компонента поля);
- `dsBlockRead` (запрет изменения элементов управления и генерации событий при вызове метода `Next`);
- `dsInternalCalc` (указание на необходимость вычислять значения полей, свойство `FieldKind` которых имеет значение `fkInternalCalc`);
- `dsOpening` (открытие набора данных).

Схема взаимосвязи между основными режимами наборов данных показана на рис. 5.2, где приведены также некоторые методы и свойства, с помощью которых набор данных переходит из одного режима в другой.

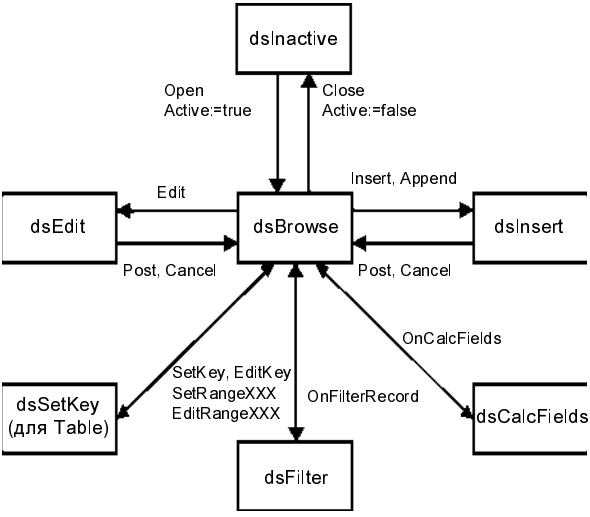


Рис. 5.2. Схема взаимосвязи режимов наборов данных

Иногда при описании операций, выполняемых с записями набора данных, под режимом *редактирования* подразумевается не только режим `dsEdit` из-

менения 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 одноименного свойства `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:

```
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
    switch (Table1->State)
    {
       case dsInactive:
        {
        Label1->Caption = "Набор данных закрыт";
        break;
        }
       case dsBrowse:
        {
        Label1->Caption = "Просмотр набора данных";
        break;
        }
       case dsEdit:
        {
        Label1->Caption = "Редактирование набора данных";
        break;
        }
       case dsInsert:
        {
        Label1->Caption = "Вставка записи в набор данных";
        break;
        }
       default:  Label1->Caption = "Режим не определен";
    }
}
```

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 набора данных (Table1), а не 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 C++ Builder 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 int 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 __property TFields* Fields = {read=FFields}; 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 int. 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.

Для примера рассмотрим чтение полей текущей записи:

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
  for(int i = 0; i < Table1->FieldCount; i++)
  ListBox1->Items->Add(Table1->Fields->Fields[i]->AsString);
}
```

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 TField* __fastcall FindField(const AnsiString FieldName); 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 NULL. 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

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
  Label1->Caption = Table1->FieldByName("Name")->AsString;
  int x = Table1->FieldByName("Size")->AsInteger;
  Edit4->Text=IntToStr(x);
}
```

строковое значение поля Name отображается в надписи Label1, а переменной x присваивается целочисленное значение поля Size и оно помещается в редактор Edit4. Если же поле Size 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:

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
        // Перевод набора данных в режим редактирования
        Table1->Edit();
        // Изменение значений полей
        Table1->FieldByName("Name")->AsString = Edit1->Text;
        Table1->FieldByName("Size")->AsInteger = 5;
        // Сохранение изменений
        Table1->Post();
}
```

выполняется присвоение новых значений полям `Name` и `Size` 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` типа `TTable` 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 `AnsiString`, которое задает имя той таблицы базы данных, которую этот компонент инкапсулирует (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 свойству `Active` набора данных автоматически устанавливается значение `false`. При задании имени таблицы программным способом набор данных предварительно необходимо закрыть, установив его свойству `Active` значение `false`. В противном случае генерируется исключение.

Приведем пример, иллюстрирующий, как задается имя таблицы БД:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
       Table1->Close();
    if (OpenDialog1->Execute())
    Table1->TableName=OpenDialog1->FileName ;
       Table1->Open();
}
```

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.

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

- `ttDefault` — тип таблицы определяется автоматически 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.

C++ Builder 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 `bool`. 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 `bool`. По умолчанию свойство имеет значение `false`, и для таблицы, с которой связан набор данных, действует многопользовательский режим доступа. Если установить свойству `Exclusive` значение `true`, то приложение получает монопольный доступ к соответствующей таблице, при этом другим приложениям доступ к таблице запрещается. Перед заданием значения свойству `Exclusive` набор данных должен быть закрыт, т. е. разорвана его связь с таблицей. Если какое-либо приложение или набор данных этого же приложения уже взаимодействует 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 C++ Builder 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 C++ Builder и/или Database Desktop. В приложении Database Desktop таблицу нужно закрыть, а в среде C++ Builder достаточно через Инспектор объектов установить значение `false` свойству `Active` набора данных, связанного с таблицей.

В наборе данных `Table` можно указать текущий индекс для сортировки, поиска записей и установления связей между таблицами.

Текущий индекс устанавливается с помощью свойства `IndexName` или `IndexFieldNames` типа `AnsiString`. 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;Size";
```

Здесь для таблицы Paradox, с которой связан компонент `Table1`, определен ключ, в который входят поля `Name` и `Size`. 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 `int` содержит число полей в текущем индексе и доступно для чтения при выполнении приложения.

Свойство `Db::TField* IndexFields[int Index]` 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. Программист может изменять определенные для таблицы индексы динамически, т. е. в процессе выполнения приложения, с помощью методов `AddIndex` и `DeleteIndex`.

Замечание

С помощью методов `AddIndex` и `DeleteIndex` допускается изменять индексы для таблицы, открытой в режиме *монопольного* доступа, когда свойство `Exclusive` имеет значение `true`.

Метод `void __fastcall AddIndex(const AnsiString Name, const AnsiString Fields, Db::TIndexOptions Options, const AnsiString DescFields = "");` добавляет к имеющейся 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 атрибуты 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).

Параметр `DescFields` представляет собой строку, содержащую список имен полей, разделенных точкой с запятой. Поля, определенные в `DescFields`, представляют собой поля в новом индексе, для которых упорядочение будет установлено в порядке *убывания*. Поля в определении индекса, но не в списке параметра `DescFields`, по умолчанию следуют в порядке возрастания. Таким образом, для одного индекса возможно совместное задание порядка возрастания и убывания полей.

Приведем пример добавления индекса к таблице:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
 // Перевод таблицы в режим монопольного доступа
       Table1->Close();
       Table1->Exclusive = true;
       Table1->Open();
       // Добавление индекса
       Table1->AddIndex("IndNC", "Name;Capital",
              TIndexOptions() << ixCaseInsensitive);
       // Закрытие режима монопольного доступа к таблице
       Table1->Close();
       Table1->Exclusive = false;
       Table1->Open();
}
```

Здесь к таблице, с которой связан набор данных `Table1`, добавляется индекс `IndNC`, построенный по полям `Name` и `Capital`. Для нового индекса определяется порядок сортировки по возрастанию значений (по умолчанию), 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("IndNC");
```

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 на этапах разработки и выполнения приложения. Класс `TIndexDefs` 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 `int` содержит количество индексов и доступно для чтения. Управление количеством индексов осуществляется косвенно, т. е. через другие свойства и методы.

Свойство `Items[int Index]` 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.

Например, чтобы считать в редактор Edit1 список полей, составляющих второй индекс (первый вторичный индекс), можно использовать следующий код:

```
Edit1->Text = Table1->IndexDefs->Items[1]->Fields;
```

Свойство Name типа AnsiString 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 AnsiString 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. 5.3). Так как компонент Table1 связан с таблицей Paradox, то имя главного индекса, построенного по полю Name, содержит пустую строку.

Обработчик события нажатия кнопки Button1 выглядит так:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int n = 0; n<Table1->IndexDefs->Count;n++)
  {
  ListBox1->Items->Add(Table1->IndexDefs->Items[n]->Name);
  ListBox2->Items->Add(Table1->IndexDefs->Items[n]->Fields);
  }
}
```

Для получения списка имен индексов можно использовать метод void __fastcall GetIndexNames(Classes::TStrings* List); класса TTable. Он возвращает список имен индексов через параметр List, в качестве которого можно использовать любой объект типа TStrings.

Например, получение списка имен индексов в программе может выглядеть так:

```
ListBox1->Items->Clear();
Table1->GetIndexNames(ListBox1->Items);
```

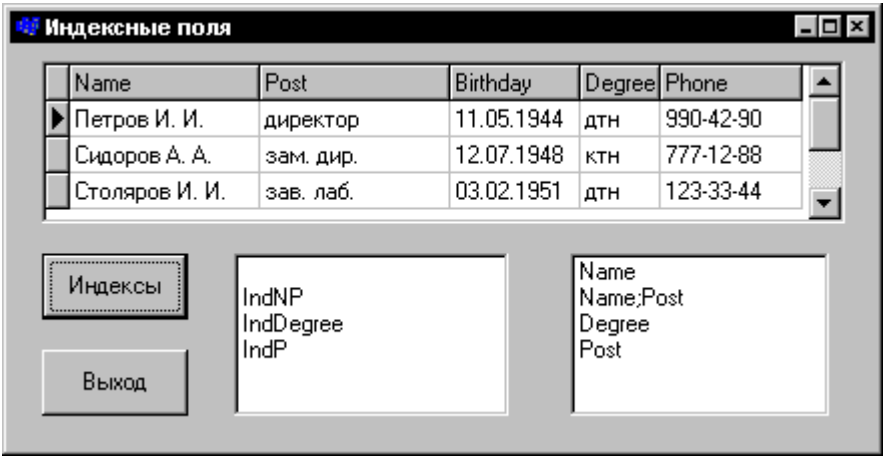


Рис. 5.3. Получение списка индексов и полей

Для добавления и удаления *динамических* индексов (задаваемых на этапе выполнения приложения) можно использовать методы `Add` и `Clear` класса `TIndexDefs`. Метод `HIDESBASE void __fastcall Add(const AnsiString N, const AnsiString F, TIndexOptions Options);` добавляет новый индекс. Параметры этого метода (кроме того, что число их на единицу меньше) не отличаются от соответствующих параметров метода `AddIndex` набора данных `Table`, однако для метода `Add` не требуется перевода таблицы в режим монопольного доступа, что делает его использование более удобным.

Замечание

Динамические индексы действуют только при выполнении приложения, в файле таблицы не сохраняются и поэтому не требуют перевода таблицы в режим монопольного доступа. Более того, при выполнении метода `Update` класса `TIndexDefs` происходит обновление текущего списка индексов. При этом динамические индексы заменяются индексами из файла таблицы.

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

Метод `Clear` удаляет все индексы, содержащиеся в списке.

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

```
Table1->IndexDefs->Clear();
Table1->IndexDefs->Add("", "Name", TIndexOptions()<<ixPrimary<<ixUnique);
Table1->IndexDefs->Add("indNP","Name;Post",
        TIndexOptions()<<ixDescending<<ixCaseInsensitive);
```

В этом примере из набора данных `Table1`, связанного с таблицей Paradox, удаляются все динамические индексы и создаются два новых: первичный индекс, построенный по полю `Name`, и вторичный индекс `indNP`, построенный по полям `Name` и `Post`.

Ряд методов класса `TIndexDefs` 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 может возникнуть исключение.

Функция `int __fastcall IndexOf(const AnsiString AName);` 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:

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{ int n=Table1->IndexDefs->IndexOf(Edit1->Text);
 if (n== -1)
  {
       MessageDlg("Индекс отсутствует!",
              mtInformation, TMsgDlgButtons() << mbOK, 0);
       if (Edit1->CanFocus()) Edit1->SetFocus();
  }
 Edit1->Text=IntToStr(n);
}
```

Здесь при нажатии кнопки `Button4` 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`. В любом случае в поле ввода `Edit1` отображается результат обращения к функции `IndexOf`.

Функция `TIndexDef* __fastcall FindIndexForFields(const AnsiString Fields);` осуществляет поиск индекса по списку полей, заданному параметром `Fields`. В случае успешного поиска функция возвращает информацию об индексе, если 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.

Отметим еще раз, что при запуске приложения информация об индексах таблицы считывается с диска из соответствующих индексных файлов. При изменении *динамических* индексов в процессе выполнения приложения, с помощью методов `Add` и `Clear` класса `TIndexDefs` содержимое индексных файлов остается прежним. Если же динамические индексы меняются при выполнении приложения с помощью методов `AddIndex` и `DeleteIndex`, предполагающих монопольное использование таблицы, то содержимое индексных файлов изменяется соответствующим образом.

Свойство `StoreDefs` типа `bool` класса `TTable` указывает, остаются ли определения полей таблицы и индексов с модулем данных или формы. Если это свойство имеет значение `true`, то определения полей таблицы и индексов хранятся вместе с модулем данных или формы. Установка свойству `StoreDefs` значения `true` означает, что метод `CreateTable` класса `TTable` реализуется как одношаговая процедура, которая создает поля, индексы и ограничения значений при выполнении приложения *(см. главу 4)*. По умолчанию свойство `StoreDefs` имеет значение `false`, оно принимает значение `true`, когда `FieldDefs` или `IndexDefs` обновляются или редактируются вручную. Чтобы предотвратить сохранение отредактированных или импортированных определений нужно установить свойству `StoreDefs` значение `false`.

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 способ доступа. В результате набор данных `Query` будет мало отличаться от набора данных `Table`.

Набор данных `Query`, в отличие от компонента `Table`, 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 с помощью вызова его методов `ExecSQL` или `Open`. Запрос SQL иногда называют SQL-программой.

При формировании запроса на этапе разработки приложения можно использовать текстовый редактор (рис. 5.4), вызываемый через Инспектор объектов двойным щелчком в области значения свойства `SQL`.

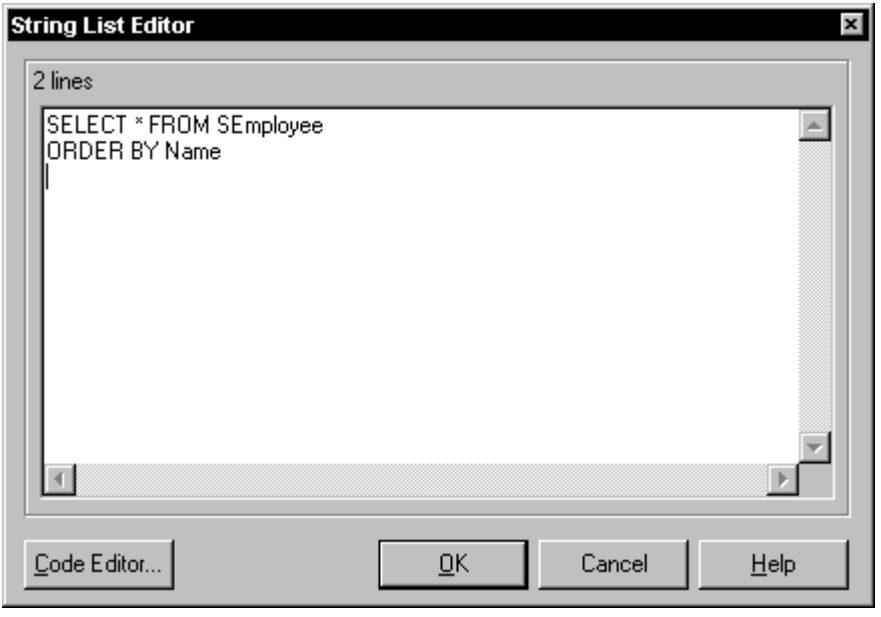


Рис. 5.4. Редактирование запроса SQL

SQL-запрос также можно формировать и изменять динамически, внося изменения в его текст (значение свойства `SQL` компонента `Query`) 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. 5.5 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.

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

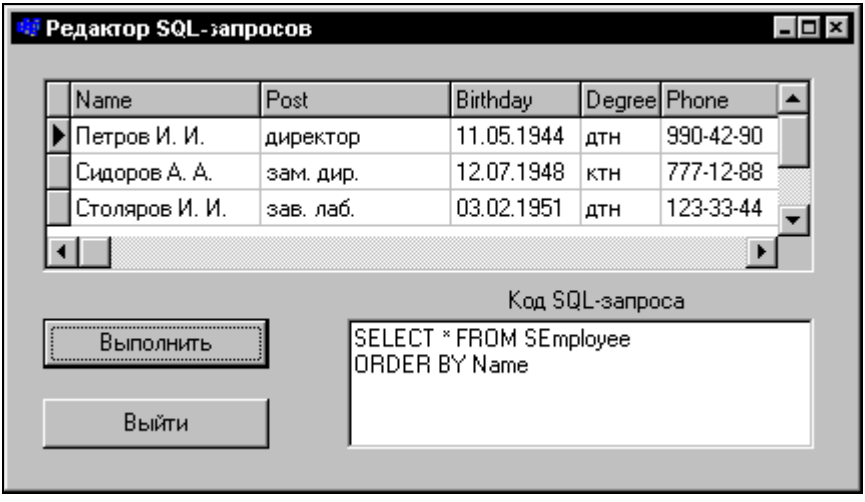


Рис. 5.5. Приложение-редактор SQL-запросов

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.

Приведем код заголовочного файла `Unit1Query.h` и файла `Unit1Query.cpp` формы `Form1` приложения:

```
#ifndef Unit1QueryH
#define Unit1QueryH
//---------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <DB.hpp>
#include <DBGrids.hpp>
#include <DBTables.hpp>
#include <Grids.hpp>
//---------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
```

```
    TQuery *Query1;
    TDataSource *DataSource1;
    TDBGrid *DBGrid1;
    TButton *Button1;
    TButton *Button2;
    TMemo *Memo1;
    TLabel *Label1;
    void __fastcall FormCreate(TObject *Sender);
    void __fastcall Button1Click(TObject *Sender);
private:      // User declarations
public:              // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif

//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1Query.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
       DataSource1->DataSet = Query1;
       DBGrid1->DataSource = DataSource1;
```

```
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        Query1->DatabaseName="BCDEMOS";
        Query1->Close();
        Query1->SQL->Assign(Memo1->Lines);
        Query1->Open();
}
```

Напомним, что с помощью свойства `DatabaseName` задается псевдоним, определяющий путь к каталогу, в котором находятся файлы таблиц базы данных.

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.

При наличии ошибки в тексте SQL-запроса генерируется исключение и выдается сообщение об ошибке (рис. 5.6), а результат запроса остается не определен. При этом набор данных `Query1` автоматически закрывается.

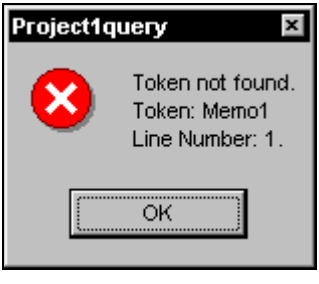


Рис. 5.6. Сообщение об ошибке в тексте SQL-запроса

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 свойству `Active` значения `true`. В ряде случаев при выполнении SQL-запроса не требуется возвращать набор данных, например, при удалении, вставке или изменении записей. В этом случае пред-

почтительнее 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 `bool`. По умолчанию свойство имеет значение `false`, и набор данных `Query` доступен только для чтения. Если пользователю или программисту требуется возможность редактирования записей, то свойству `RequestLive` нужно установить 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 свойству `RequestLive` значения `true` игнорируется.

Чтобы проверить результат установки значения свойству `RequestLive`, можно воспользоваться свойством `CanModify` типа `bool`. Если оно имеет значение `true`, то набор данных является редактируемым, если `false` — нередактируемым.

Чтобы получить в результате выполнения SQL-запроса редактируемый набор данных, кроме установки свойству `RequestLive` значения `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 свойству `SQL` значение `SELECT * FROM DBNameTable`, а свойству `RequestLive` — значение `true`, то набор данных `Query` будет аналогичен набору данных `Table`. (Здесь `DBNameTable` является именем таблицы БД, которое для компонента `Table` задается в свойстве `TableName`.) 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`.

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

Объекты поля

Объект поля `Field` имеет тип `TField` и служит полем набора данных (таблицы, запроса и т. п.). Тип `TField` является абстрактным классом и непосредственно не используется. Вместо него применяются производные классы (важнейшие из них приведены в 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
`TAutoincField`	Поле автоинкрементного значения (32 разряда)
`TBCDField`	Поле BCD-значения
`TBinaryField`	Поле двоичного значения
`TBLOBField`	Поле BLOB-значения (BLOB, Binary Large Object — большой двоичный объект)
`TBooleanField`	Поле логического значения
`TBytesField`	Поле байтового значения фиксированной длины
`TCurrencyField`	Поле значения денежной суммы
`TDateField`	Поле значения даты
`TDateTimeField`	Поле значения даты и времени
`TFloatField`	Поле вещественного значения
`TGraphicField`	Графическое поле
`TIntegerField`	Поле целочисленного значения (32 разряда)

Таблица 5.1 *(окончание)*

Тип объекта	Вид поля
`TLargeintField`	Поле целочисленного длинного значения (64 разряда)
`TMemoField`	Мемо-поле (поле комментария)
`TNumericField`	Поле числового значения
`TSmallintField`	Поле целочисленного короткого значения (16 разрядов)
`TStringField`	Поле строкового значения
`TTimeField`	Поле значения времени
`TVarBytesField`	Поле байтового значения переменной длины
`TWordField`	Поле целочисленного значения без знака (16 разрядов)

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:

- автоматически (динамические поля — по умолчанию);
- с помощью Редактора полей (статические поля).

По умолчанию для каждого поля набора *автоматически* создается свой объект типа `TField` при каждом открытии набора данных, как на этапе проектирования, так и на этапе выполнения приложения. 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 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.

Отметим, что при модификации структуры таблицы, например, удалении поля или изменении его типа, открытие набора данных, имеющего статические поля, может вызвать исключение.

Редактор полей

Как отмечалось, 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. 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 `bool`. Если его значение равно `true`, то набор данных по умолчанию имеет динамические поля, в противном случае для набора данных заданы статические поля.

Для запуска Редактора полей (рис. 5.7) следует сделать двойной щелчок на компоненте `Table` или `Query` или, вызвав для одного из них контекстное меню правой кнопкой мыши, выбрать пункт **Fields Editor**. В заголовке Редактора полей выводится составное имя набора данных, например, `Form1->Query1`. 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 можно 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. 5.7. Редактор полей

Рис. 5.8. 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. 5.8). 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 выделить его в списке и задать команду **Delete** контекстного меню или нажать клавишу <Delete>. После удаления статического поля оно становится недоступным для операций в программе, но при необходимости его снова можно сделать статическим, добавив в список Редактора полей. Отметим, что все свойства этого поля устанавливаются заново, а 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>+<↑> и <Ctrl>+<↓>.

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. 5.9).

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. C++ Builder 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).

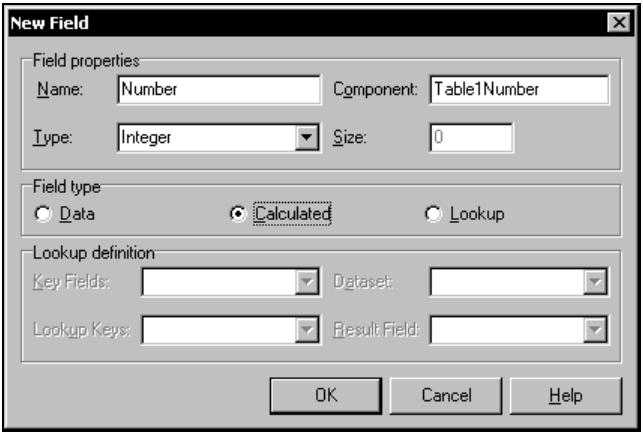


Рис. 5.9. Окно создания статического поля

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 с помощью Инспектора объектов и могут быть изменены. При этом каждому параметру, задаваемому с помощью поля ввода или переключателя окна **New Field** (рис. 5.9), соответствует определенное свойство объекта типа `TField` (табл. 5.2). Все свойства объекта доступны с помощью Инспектора объектов, за исключением свойства `FieldName`, которое доступно только во

время выполнения приложения, однако при разработке приложения значение этого свойства видно в окне Редактора полей и Инспектора объектов. 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 и числовых полей)	Поле ввода **Size**
`Precision` (для полей с плавающей точкой)	
`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` (физическое поле в базе данных);
- `fkCalculated` (вычисляемое поле);
- `fkLookup` (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 свойству `Calculated` типа `bool` устанавливается 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 свойству `AutoCalcFields` типа `bool` установлено 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.

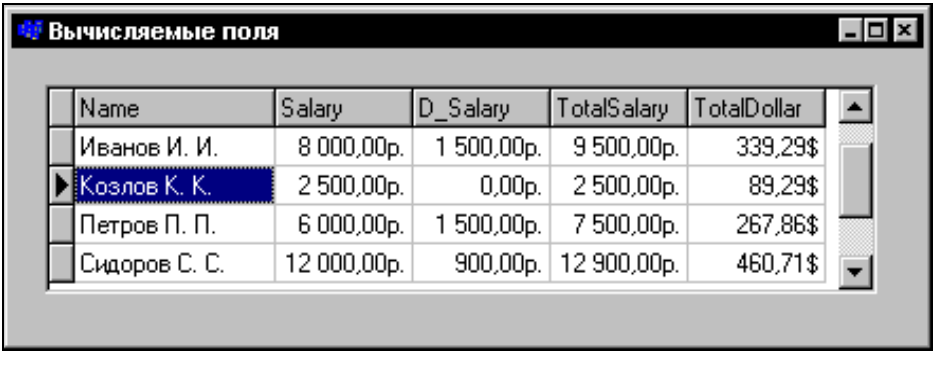


Рис. 5.10. Использование вычисляемых полей

Рассмотрим пример использования вычисляемых полей.

Пусть в состав таблицы входят три поля: `Name` (имя), `Salary` (оклад) и `D_Salary` (надбавка). Набор данных должен содержать для каждого сотрудника также его общую зарплату в рублях и в долларах. Общая зарплата со-

трудника складывается из оклада и надбавки, поэтому в набор данных добавляются два вычисляемых поля: `TotalSalary` и `TotalDollar` для рублей и долларов соответственно (рис. 5.10).

Для обновления значений вычисляемых полей используется обработчик события `OnCalcFields` компонента `Table1`, код которого приведен далее. Обращение к вычисляемым полям и полям данных выполнено разными способами: по имени компонента (для вычисляемых полей `TotalSalary` и `TotalDollar`) и по имени поля (для полей данных `Salary` и `D_Salary`). Следует иметь в виду, что поля `Salary` и `D_Salary` 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 28. В принципе, значение этого коэффициента должно быть переменным с возможностью его редактирования пользователем.

```
void __fastcall TForm1::Table1CalcFields(TDataSet *DataSet)
{
 Table1TotalSalary->AsFloat = Table1->FieldByName("Salary")->AsFloat+
                 Table1->FieldByName("D_Salary")->AsInteger;
 Table1TotalDollar->AsFloat = Table1TotalSalary->AsFloat/28;
}
```

Если в качестве обменного курса взять некоторое реальное значение, то значения поля `TotalDollar` будут содержать большое число дробных разрядов. Поэтому необходимо округлять получаемые значения или выводить полученные значения в требуемом формате. Для получения нужного формата чисел можно использовать свойство `DisplayFormat` этого поля, установив его значение с помощью Инспектора объектов или при создании формы, например, следующей инструкцией:

```
Table1TotalDollar->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, в частности, может возникнуть в случае, когда при назначении нового сотрудника на должность в научном учреждении на него задается такая, например, информация (в скобках приведены названия полей таблицы):

- уникальный номер сотрудника (R_Number);
- имя сотрудника (R_Name);
- дата приема на работу (R_Date);
- должность (R_Post);
- оклад по должности (R_Salary);
- примечание (R_Note).

Данные об окладе по каждой должности хранятся в другой таблице, имеющей поля:

- должность (G_Post);
- оклад по должности (G_Salary);
- примечание (G_Note).

В таблице данных о должностных окладах в поле оклада G_Salary значения могут корректироваться с учетом распоряжений вышестоящих органов, изменения штатного расписания. При приеме на работу нового сотрудника или изменении его должности в поле оклада по должности R_Post должно заноситься соответствующее значение. Для установления соответствия между данными о должностях таблицы связываются по этим полям. Однако при занесении данных о вновь назначенном сотруднике в отделе кадров нужно правильно задавать значение оклада, соответствующее должности.

Чтобы избежать возможных ошибок при занесении данных об окладе, в данном случае удобно сформировать поле выбора, содержащего требуемые значения, из которого будет выбираться нужный вариант. Тогда после выбора должности соответствующий оклад будет заноситься в поле R_Salary таблицы данных о сотрудниках автоматически. Окно создания поля выбора показано на рис. 5.11.

Поле выбора имеет имя R_Post и принадлежит набору данных Table1, содержащему записи таблицы данных о сотрудниках. Для формирования списка выбора используется поле G_Post набора данных Table2, содержащего записи таблицы данных о должностях и окладах. Связь между наборами данных Table1 и Table2 осуществляется через их поля должности R_Post и G_Post соответственно (рис. 5.12).

Использование поля выбора заключается в том, что пользователь выбирает значение в поле связи R_Post, содержащем список, который построен на основании значений поля связи G_Post. После выбора значения для поля связи R_Post из результирующего поля G_Salary автоматически заносится соответствующее значение в поле R_Salary. Таким образом, поле связи R_Post, содержащее список, используется для выбора, а поле R_Salary — для занесения в него значения.

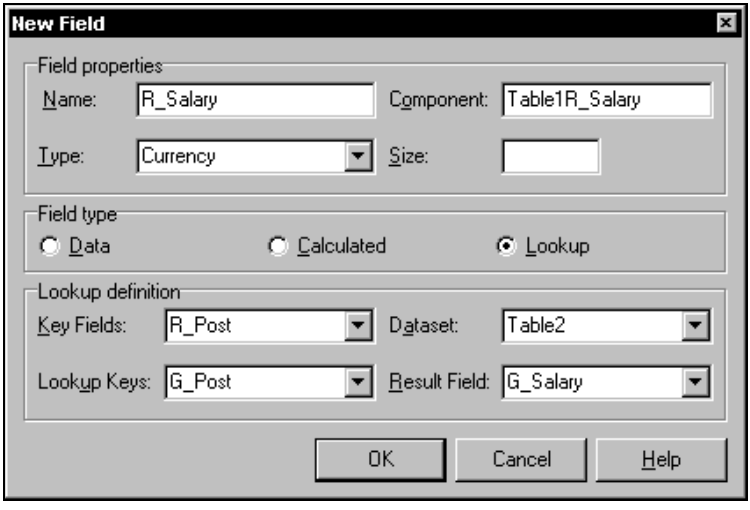


Рис. 5.11. Окно создания поля выбора

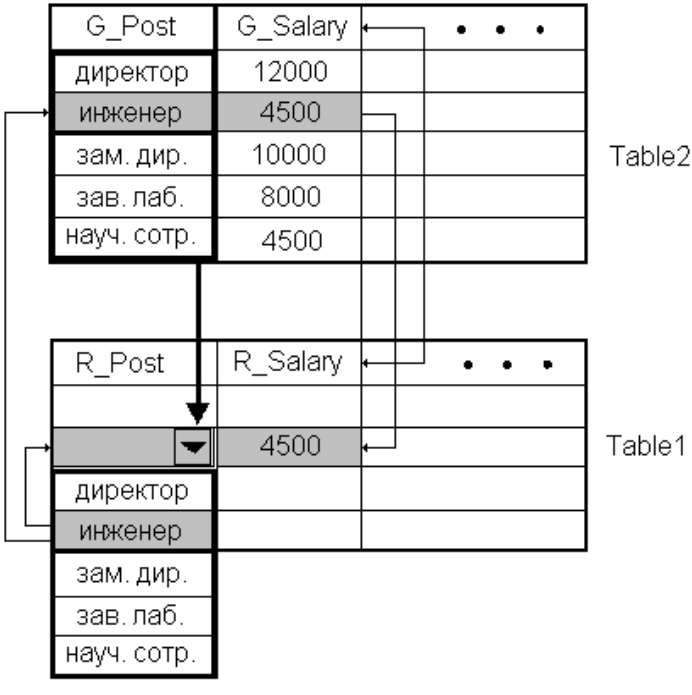


Рис. 5.12. Схема использования поля выбора

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 выбрать значение в списке.

На рис. 5.13 приведен вид формы приложения, в котором применяется поле выбора. В верхней части формы размещена таблица данных о сотрудниках, в нижней — таблица данных о должностях и окладах (приведена для наглядности). 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.

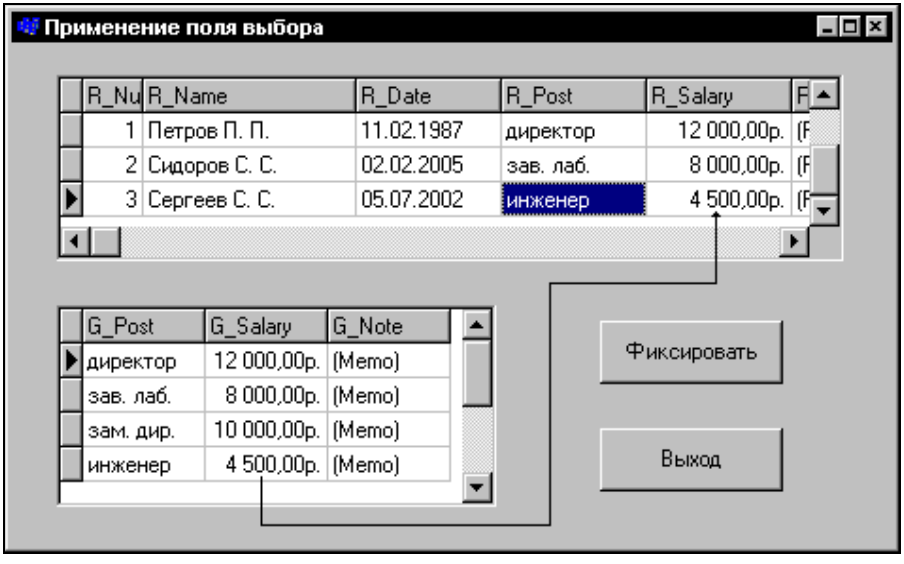


Рис. 5.13. Использование поля выбора

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` класса `TField`.

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 с помощью Инспектора объектов.

Доступ к значению поля

Объект поля, как и любой другой объект, имеет имя (название), определяемое его свойством `Name` типа `AnsiString`. 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`. Можно изменить это имя с помощью Инспектора объектов, когда соответствующее статическое поле выбрано в Редакторе полей.

В отличие от имени объекта `Field`, свойство `FieldName` типа `AnsiString` 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("R_Salary")->DisplayLabel = "Оклад";
Table1R_Salary->DisplayLabel = "Оклад";
```

Здесь для статического поля `R_Salary` приведены два возможных способа обращения: по имени поля в наборе данных и по имени объекта `Field` поля. Заметим, что свойство `DisplayLabel` определяет текст, отображаемый в качестве заголовка поля набора данных.

Для определения порядкового номера поля в наборе данных (нумерация начинается с единицы) можно использовать свойство `FieldNo` типа `int`, например, так:

```
    int x = Table1->FieldByName("R_Date")->FieldNo;
Edit1->Text=IntToStr(x);
```

Для доступа к значению поля служат свойства `Value` и `AsXXX`. Для объекта поля класса `TField` свойство `Value` имеет тип `Variant` и представляет со-

бой фактические данные в этом объекте. 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 ряда различных классов объектов, наследуемых от класса `TField`.

Таблица 5.3. *Ряд возможных типов свойства* `Value`

Тип объекта поля	**Тип свойства `Value`**
`TField`	`Variant`
`TStringField`, `TBLOBField`	`AnsiString`
`TIntegerField`, `TSmallIntField`, `TWordField`, `TAutoincField`	`int`
`TBCDField`	`Currency`
`TFloatField`, `TCurrencyField`	`double`
`TBooleanField`	`bool`
`TDateTimeField`, `TDateField`, `TTimeField`	`TDateTime`

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

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        AnsiString s;
        float  y;
        // Доступ по имени в наборе данных
        s = Table1->FieldByName("G_Salary")->Value;
        y = Table1->FieldByName("G_Salary")->Value;
        Label1->Caption = s;
        Label2->Caption = FloatToStr(y);
        // Доступ, как к отдельному компоненту
        y = Table1G_Salary->Value;
        Label3->Caption = FloatToStr(y);
        s = Table1G_Salary->Value;
    Edit1->Text=s;
}
```

Здесь чтение значения поля `G_Salary` текущей записи набора данных `Table1` выполняется несколькими способами. При доступе с помощью свойства `Value` к полю по имени в наборе данных значение поля `G_Salary` типа `$ (Money)` (и вещественного типа тоже) можно читать и использовать и как строковое, и как вещественное значение.

Замечание

Пояснение к приведенному примеру подразумевает автоматическое допустимое преобразование типов. К примеру, вещественный тип можно преобразовать к строковому типу, обратное преобразование в общем случае невыполнимо. То же справедливо для доступа с помощью свойства `Value` к полю как отдельному компоненту.

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`:

- `AsBCD` типа `Tbcd`;
- `AsBoolean` типа `bool`;
- `AsCurrency` типа `Currency`;
- `AsDateTime` типа `TdateTime`;
- `AsFloat` типа `double`;
- `AsInteger` типа `int`;
- `AsSQLTimeStamp` типа `TSQLTimeStamp`;
- `AsString` типа `AnsiString`;
- `AsVariant` типа `Variant`.

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`:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    // Доступ по имени поля в наборе данных
    AnsiString s = Table1->FieldByName("N_Number")->AsString;
    float x = Table1->FieldByName("Time")->AsFloat;
```

```
    Edit1->Text= s;
    Edit2->Text = FloatToStr(x);
    // Доступ, как к отдельному компоненту
    s = Table1Currency->AsString; // денежное поле
    x = Table1Date->AsFloat;    // Поле даты
    Edit3->Text = s;
    Edit4->Text = FloatToStr(x);
}
```

Как и в предыдущем примере, чтение значений полей вещественного типа, типа времени, денежного типа и типа даты осуществляется двумя способами. Доступ к полю выполняется по имени поля и по имени объекта поля, а значение поля интерпретируется как строковое или как вещественное с помощью свойств `AsString` и `AsFloat` соответственно.

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 `bool`. Возможность модификации данных в отдельном поле определяется значением свойства `CanModify` типа `bool`. 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` имеет значение `false`), но разрешено для редактирования (свойство `ReadOnly` имеет значение `false`), то значение этого поля можно изменить 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, если его свойству `ReadOnly` не установлено значение `true`.

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`, а для сохранения содержимого поля в файле — метод `SaveToFile`.

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 не определен);
- `ftString` (короткая строка длиной не более 255 символов);
- `ftSmallInt` (16-битное целое);
- `ftInteger` (32-битное целое);
- `ftWord` (целое без знака);
- `ftBoolean` (логическое значение);
- `ftFloat` (вещественное число);
- `ftCurrency` (денежное значение);
- `ftBCD` (число в формате BCD (двоично-кодированный 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` (типизированное 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:

```
typedef Set<char, 0, 255> TFieldChars;
```

Чтобы проверить, разрешен ли символ для ввода в качестве значения поля, удобно использовать метод `virtual bool __fastcall IsValidChar(char InputChar);`. 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 (!Table1->Fields->Fields[2]->IsValidChar(Edit1->Text[1]))
   { AnsiString s=Edit1->Text[1];
    MessageDlg("Символ "+s+" недопустим",
          mtError, TMsgDlgButtons() << mbOK, 0);
    }
```

Если первый символ текста, введенного в компонент `Edit1`, является недопустимым символом для третьего поля (нумерация полей в списке начинается с нуля) 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 динамических 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` наследуемого от него.

Например, ограничение вводимых значений в программе может быть установлено в обработчике создания формы следующим образом:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
Table1F_Number->MinValue = 0;
Table1F_Number->MaxValue = 2.6E+16;
Table1F_Int->MinValue = 0;
Table1F_Int->MaxValue = 1000;
}
```

Здесь для поля `F_Number` (вещественный тип) устанавливается допустимый диапазон `0 .. 2.6E+16`, а для поля `F_Int` (целый тип) — диапазон `0 .. 1000`.

Такие же ограничения для динамических полей могут быть заданы с помощью Инспектора объектов и Редактора полей. Для этого нужно в Редакторе

полей выбрать имя нужного поля и в Инспекторе объектов задать значения свойств `MinValue` и `MaxValue`.

Если при вводе значения в поле имеет место выход за пределы диапазона, то выдается сообщение об ошибке (рис. 5.14).

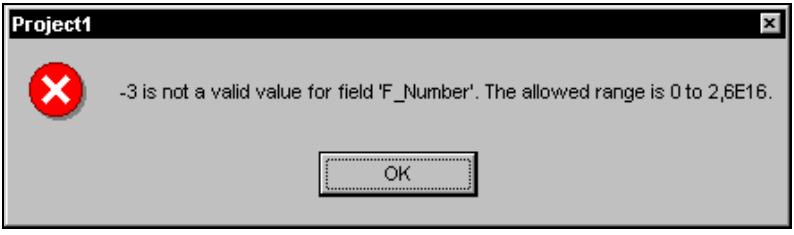


Рис. 5.14. Сообщение об ошибке

Замечание

При задании вещественных констант в программе в качестве разделителя указывается точка (например: `2.6E+16`). При задании таких же вещественных констант в качестве значений свойств `MinValue` и `MaxValue` с помощью Инспектора объектов в качестве разделителя следует указывать запятую (например: `2,6E+16`).

Ограничения на вводимые значения можно определить также с помощью свойства `CustomConstraint` типа `AnsiString`. 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.

Задание свойства `CustomConstraint` выполняется с помощью Инспектора объектов, при этом используется также Редактор ограничений (рис. 5.15), вызываемый двойным щелчком мыши в строке свойства `Constraints` набора данных (например, `Table1`). Для задания нового ограничения (или редактирования имеющегося) нужно на панели инструментов Редактора ограничений нажать кнопку **Add New** (или выбрать мышью строку с имеющимся ограничением) и в окне Инспектора объектов в строке `CustomConstraint` задать (или отредактировать) ограничение.

Здесь для полей `F_Int` (целый тип) и `F_Number` (вещественный тип) устанавливаются диапазоны допустимых значений (рис. 5.15).

В Инспекторе объектов при задании свойства `CustomConstraint` можно также в строке свойства `ErrorMessage` задать текст сообщения, которое будет выдаваться в случае нарушения заданных нами ограничений.

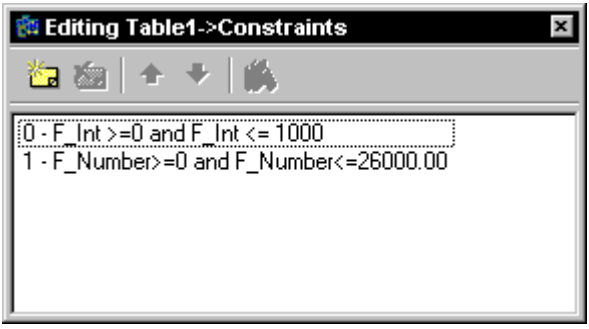


Рис. 5.15. Вид окна Редактора ограничений

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.

Замечание

При нарушении ограничений, установленных с помощью свойства `CustomConstraint`, исключение возникает не сразу после занесения неверного значения в поле, а после перехода к другой записи или обновления измененной записи в таблице, например методом `Post`.

Разработчик может с помощью Инспектора объектов задавать через свойство `DefaultExpression` типа `AnsiString` значение по умолчанию, которое автоматически заносится в поле при добавлении новой записи в набор данных. Это свойство определяет выражение SQL, значение которого помещается в поле, для которого пользователь не обеспечивает значение самостоятельно. Для задания этого свойства следует вызвать Редактор полей (рис. 5.7) набора данных (например, `Table1`), выбрать в нем нужное поле и в строке `DefaultExpression` Инспектора объектов задать выражение или значение.

Замечание

Чтобы проверить, действуют ли для поля ограничения, заданные в свойствах `CustomConstraint`, `DefaultExpression`, а также в `ImportedConstraint` (ограничения, налагаемые сервером), используется свойство `HasConstraints` типа `bool`. Это свойство, действующее во время выполнения приложения, согласно справочной информации к C++ Builder 6, должно принимать значение `true`, если установлено хотя бы одно из перечисленных ограничений. В действительности такое поведение свойства `HasConstraints` не имеет места.

Программно можно выполнить и более сложную проверку, используя обработчики событий `OnSetText` (типа `TFieldSetTextEvent`), `OnValidate` и `OnChange` (типа `TFieldNotifyEvent`). Названые события возникают в указанном здесь порядке при изменении значения поля. Типы этих событий описаны так:

```
typedef void __fastcall (__closure *TFieldSetTextEvent)(TField* Sender,
const AnsiString Text);
typedef void __fastcall (__closure *TFieldNotifyEvent)(TField* Sender);
```

Указанные события возникают до изменения значения поля, поэтому программист может при необходимости 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.

```
void __fastcall TForm1::Table1N_NumberValidate(TField *Sender)
{
if(Table1->FieldByName("N_Number")->AsString == '1')
{Abort();}
}
```

Здесь в поле `N_Number` запрещается ввод значения `1`, которое зарезервировано для специальных целей. При попытке ввода неправильных значений после соответствующей проверки генерируется "тихое" исключение (путем вызова метода `Abort`). В результате попытка ввода отвергается, при этом сообщения не выдаются.

При необходимости для выдачи сообщения о возникшем исключении можно использовать, например, функцию `MessageDlg`, выводящую диалоговое окно.

Замечание

Чтобы задать обработчик одного из рассматриваемых нами событий для некоторого поля (в примере это статическое поле `Table1N_Number`), удобно выполнить следующее. В окне Редактора полей выбрать нужное поле, затем на вкладке **Events** Инспектора объектов двойным щелчком в строке с именем нужного события открыть заготовку процедуры его обработки.

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*.

Пример проверки вводимых значений с помощью обработчика события `OnSetText`:

```
void __fastcall TForm1::Table1N_NumberSetText(TField *Sender,
   const AnsiString Text)
{
 if (!((Text == '1') || (Text == '5')))
   // занесение значения параметра Text в поле
  {Table1->FieldByName("N_Number")->AsString = Text;}
}
```

В этом примере с помощью обработчика предотвращение ввода в поле с именем `N_Number` значений `1` и `5`.

Ряд полей, например, индексное поле, не могут быть пустыми, в этом случае у поля свойство `Required` типа `bool` имеет значение `true`. Для проверки полей (содержат ли они отличное от пустого значение) используется свойство `IsNull` типа `bool`. 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`.

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 к отображению значений в визуальных компонентах и не затрагивает вид и размещение данных в таблице БД.

Свойство `DisplayFormat` типа `AnsiString` управляет отображением значений полей в визуальных компонентах, например, `DBGrid` и `DBEdit`. Это свойство действует 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 `AnsiString` 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 `AnsiString` 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 *(см. главу 3)*.

Названные 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:

```
typedef void __fastcall (__closure *TFieldGetTextEvent)(TField* Sender,
AnsiString &Text, bool DisplayText);
```

Параметр `Text` содержит значение, которое выводится в визуальном компоненте. Программист должен задать это значение. Логический параметр `DisplayText` (только для чтения) указывает, для каких целей выводится значение: для отображения (`true`) или для редактирования (`false`). Значение параметра `DisplayText` зависит от значения свойства `ReadOnly` визуального компонента (используемого для просмотра и редактирования значений полей наборов данных), такого как `DBGrid` или `DBEdit`.

Рассмотрим пример форматирования отображаемых и редактируемых значений:

```
void __fastcall TForm1::Table1A_fieldGetText(TField *Sender,
   AnsiString &Text, bool DisplayText)
{
if (DisplayText) {Text = Table1A_field->AsString;}
 else {Text = AnsiUpperCase(Table1A_field->AsString); }
}
//---------------------------------------------------------------------
void __fastcall TForm1::Table1G_SalaryGetText(TField *Sender,
   AnsiString &Text, bool DisplayText)
{
if (DisplayText) {Text = Table1G_Salary->AsString+'$';}
 else {Text = AnsiUpperCase(Table1G_Salary->AsString); }
}
```

Здесь обработчик `Table1A_fieldGetText` обеспечивает то, что при редактировании значения поля `A_field` оно выводится строчными буквами, а при просмотре — прописными. Если для перевода символов строки в верхний регистр вместо функции `AnsiUpperCase` использовать `UpperCase`, то символы русского алфавита будут преобразовываться некорректно.

Значение поля `G_Salary` отображается в визуальных компонентах со знаком `'$'`. При редактировании этого поля знак процента отсутствует.

Замечание

Если определен обработчик события `OnGetText`, то свойства `DisplayFormat` и `EditMask` не действуют.

Свойство `DisplayWidth` типа `int` определяет число символов, используемых 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 `AnsiString`. 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. Например, так:

```
Table1G_Salary->DisplayLabel = Edit1->Text;
```

Здесь заголовку поля `G_Salary` присваивается текст, введенный в компонент `Edit1`. Если это поле отображается в компоненте `DBGrid`, то соответствующий столбец сетки также изменяет свое значение на указанное в свойстве `Text`.

Источник данных

Источник данных используется как промежуточное звено между набором данных и визуальными компонентами, с помощью которых пользователь *управляет* этим набором данных. В C++ Builder 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, но 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:

```
DataSource1->DataSet=NULL;
DataSource1->DataSet = Table2;
DataSource1->DataSet->Open();
```

Здесь для компонента DataSource1 имеющаяся связь с некоторым набором данных (связанным с этим компонентом) закрывается, затем назначается и открывается набор данных Table2.

Для анализа состояния, в котором находится набор данных, можно использовать свойство State типа TDataSetState именно источника данных (компонента DataSource), а не самого набора данных (например, Table). 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 bool 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 свойству AutoEdit значения false.

Замечание

Значение свойства AutoEdit влияет на возможность редактирования набора данных со стороны пользователя. Программно можно изменять записи независимо от значения этого свойства.

Вне зависимости от 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:

```
typedef void __fastcall (__closure *TDataChangeEvent) (TObject* Sender,
TField* Field);
```

Параметр Field указывает на измененное поле; если данные изменены более чем в одном поле, то этот параметр имеет значение NULL. Отметим, 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, на-

пример, 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:

```
void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
   TField *Field)
{
Label1->Caption = "Запись номер " + IntToStr(Table1->RecNo);
}
```

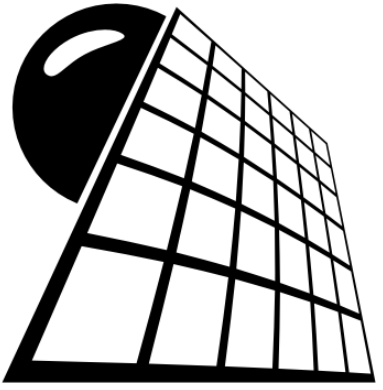
При модификации текущей записи, кроме события `OnDataChange`, генерируется еще событие `OnUpdateData` типа `TNotifyEvent`. Оно возникает непосредственно перед записью данных в БД, поэтому в его обработчике можно предусмотреть дополнительный контроль и обработку введенных в поля значений, а также некоторые 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 `bool` источника данных.

Рассмотрим пример, демонстрирующий, как производится такое управление:

```
void __fastcall TForm1::Button6Click(TObject *Sender)
{
// Отключение отображения записей
DataSource1->Enabled = false;
Table1->First();
for (int n = 1;n <= Table1->RecordCount; n++)
  {
   // Обработка записи набора данных Table1
   // ...
   Table1->Next();
  }
// Включение отображения записей
DataSource1->Enabled = true;
}
```

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.

Часть визуальных компонентов для работы с данными служит 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`.

Каждый из таких компонентов имеет два свойства, через которые обеспечивается его связь с полем выбранного источника данных. Свойство `DataSource` задает имя источника данных, который подключен к одному из наборов данных приложения. Свойство `DataField` задает связь элемента управления с конкретным полем источника данных, значение этого свойства выбирается путем выбора имени поля из раскрывающегося списка.

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 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*

Компоненты страниц *Standard*, *Additional* и *Win32*	**Компоненты страницы *Data Controls***	**Компоненты страницы *QReport***
`Label`	`DBText`	`QRLabel`
`Edit`	`DBEdit`	`QRRichEdit`
`Memo`	`DBMemo`	`QRMemo, QRExprMemo`
`RichEdit`	`DBRichEdit`	`QRRichEdit, QRDBRichEdit`
`ListBox`	`DBListBox`	
`ComboBox`	`DBComboBox`	
`CheckBox`	`DBCheckBox`	
`RadioGroup`	`DBRadioGroup`	
`Image`	`DBImage`	`QRImage, QRDBImage`

Таблица 6.1 *(окончание)*

Компоненты страниц *Standard*, *Additional* и *Win32*	**Компоненты страницы *Data Controls***	**Компоненты страницы *QReport***
`StringGrid`	`DBGrid`	
`Chart`	`DBChart`	`QRChart`

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, т. е. допустимы значения `true`, `True` и `TRUE`.

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 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 (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`.

Состояние флажка `DBCheckBox`, в отличие от компонента `CheckBox`, нельзя изменять с помощью мыши при выполнении приложения. Изменить состояние флажка `DBCheckBox` можно программно, например:

```
DBCheckBox1->Checked=true;
```

Важную роль в использовании флажка `DBCheckBox` играют свойства `ValueChecked` и `ValueUnChecked`. С их помощью флажок можно применять для отображения и редактирования строковых полей.

Свойство `ValueChecked` типа `AnsiString` содержит строковые значения, которые устанавливают связанный с этим полем флажок во включенное со-

стояние. 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 `YES` и `yes` считаются одинаковыми. Например:

```
DBCheckBox1->ValueChecked = "True;T;Yes;Y;Да;Д";
```

Свойство `ValueUnChecked` типа `AnsiString` 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.

Связь компонента флажка `DBCheckBox` с нужным полем набора данных выполняется с помощью его свойств `DataSource` и `FieldName`. Первое из свойств указывает имя компонента источника данных `DataSource`, через который осуществляется связь с набором данных. Свойство `FieldName` для связанного набора данных указывает имя поля, со значением которого происходит сравнение элементов значения свойств `ValueChecked` и `ValueUnChecked`, и в зависимости от его результата автоматически устанавливается значение свойства `Checked`.

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.

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`.

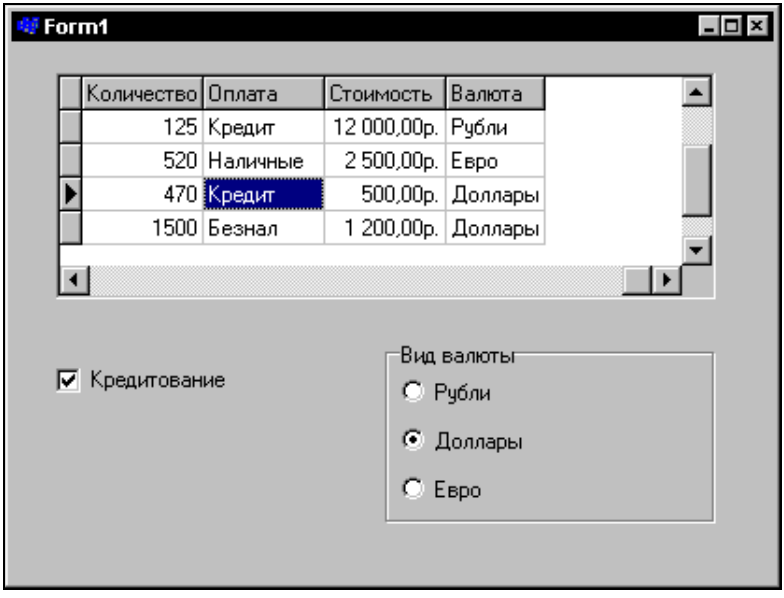


Рис. 6.1. Использование зависимых переключателей

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 **Вид валюты** и связана с полем `Валюта` набора данных. При перемещении на третью запись в группе автоматически выбирается второй переключатель, имеющий название (и такое же значение) **Доллары**. Если выбрать

другой переключатель, например третий, то в поле `Валюта` третьей записи автоматически занесется новое значение — **Евро**.

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 `int`, 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:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Table1->Edit();
DBRadioGroup1->ItemIndex = 1;
Table1->Post();
}
```

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 Оплата выбрано новое значение Наличные, в сетке DBGrid это не отображается. Закрепление в поле нового значения, а также отображение этого значения в других визуальных компонентах, связанных с данным полем, происходит при выборе этого поля с помощью указателя мыши или при переходе к другой записи.

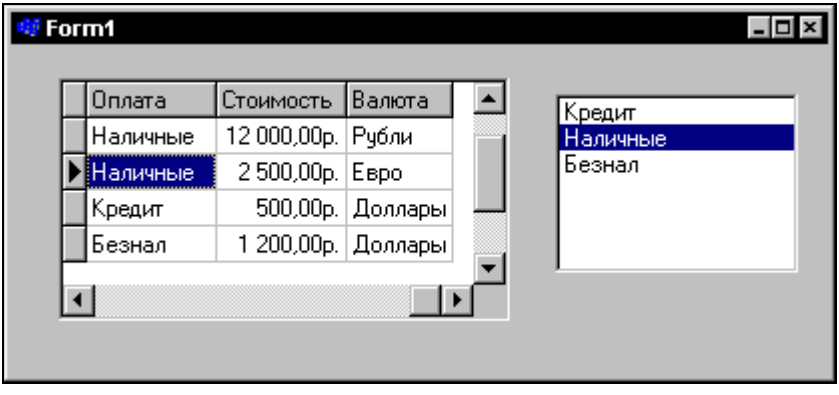


Рис. 6.2. Использование простого списка

Списки, сформированные по значениям поля набора данных

В рассмотренных нами простом и комбинированном списках перечень возможных значений поля формируется программно и в общем случае не зависит от значений записей наборов данных. В ряде случаев это может быть неудобно и привести к ошибкам ввода или выбора. В составе C++ Builder имеются компоненты DBLookupListBox и DBLookupComboBox, которые также предназначены для выбора значения в списке или для непосредственного ввода значения в поле редактирования (для DBLookupComboBox). Во многом эти компоненты похожи на простой список DBListBox и комбинированный

список DBComboBox соответственно и отличаются от них способом формирования списка возможных значений.

Техника работы с компонентами DBLookupComboBox и DBLookupListBox аналогична работе с полем выбора, рассмотренной 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, с помощью Инспектора объектов.

Как обычно, 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 AnsiString), указывающих на второй набор данных и его поле, используемое для заполнения списка.

В свойстве KeyField типа AnsiString указывается поле второго набора данных, значение которого заносится в поле, связанное с компонентом списка.

Представление записей в табличном виде с помощью сетки

Для вывода записей набора данных в табличном виде удобно использовать сетку, представленную 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

Сетка DBGrid отображает все записи, имеющиеся в наборе данных, т. е. числом строк в ней управлять нельзя.

Основным свойством сетки DBGrid является свойство Columns типа TDBGridColumns, которое представляет собой массив (коллекцию) объектов Column типа TColumn, описывающих отдельные столбцы сетки.

Свойство SelectedIndex типа int задает номер текущего столбца в массиве Columns, а свойство SelectedField указывает на объект типа TField, которому соответствует текущий столбец сетки.

Свойство FieldCount типа int, доступное во время выполнения программы, содержит число видимых столбцов сетки, а свойство Fields[int FieldIndex] 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 цвет 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 (пользователь может редактировать данные в ячейках);
- dgAlwaysShowEditor (сетка постоянно находится в режиме 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);
- dgColLines (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:

```
typedef void __fastcall (__closure *TDBGridClickEvent)(TColumn* Column);
```

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:

```
typedef void __fastcall (__closure *TMovedEvent)(System::TObject* Sender, int FromIndex, int ToIndex);
```

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, графическую информацию, например, рисунок. Для программной реализации отображения сетки используется обработчик события OnDrawColumnCell типа TDrawColumnCellEvent, которое возникает при прорисовке любой ячейки. Тип события OnDrawColumnCell описан так:

```
typedef void __fastcall (__closure *TDrawColumnCellEvent)(System::TObject* Sender, const Types::TRect &Rect, int DataCol, TColumn* Column, Grids::TGridDrawState State);
```

Здесь 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 `bool`. 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 иметь значение true
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
      const TRect &Rect, int DataCol, TColumn *Column,
      TGridDrawState State)
{
 AnsiString  s ;
 s = Table1->FieldByName(Column->FieldName)->AsString;
 if (Column->FieldName == "Area")
   {
     DBGrid1->Canvas->Brush->Color = clYellow;
     DBGrid1->Canvas->Font->Color = clRed;
     DBGrid1->Canvas->Font->Style = TFontStyles()<< fsBold << fsItalic;
     DBGrid1->Canvas->FillRect(Rect);
     DBGrid1->Canvas->TextOut(Rect.Left+2, Rect.Top , s);
   }
}
```

Здесь для столбца сетки, который соответствует полю `Area` (Площадь) набора данных, устанавливается желтый цвет фона, а данные с помощью строковой переменной `s` выводятся красным цветом и полужирным курсивом.

При прорисовке ячеек используется свойство `Canvas` элемента `DBGrid1`, а также параметр `Rect`.

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 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 свойства 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. В Редакторе статические столбцы перечисляются в порядке их создания (он 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.

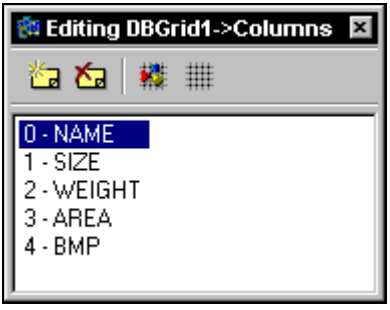


Рис. 6.3. Диалоговое окно Редактора столбцов

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 свойства 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 задать значение его свойству `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 `int` — указывает число столбцов сетки.
- `Field` типа `TField` — определяет объект поля набора данных, связанный со столбцом.
- `FieldName` типа `AnsiString` — 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:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
        // Установка параметров второго столбца
        DBGrid1->Columns->Items[1]->Title->Caption = "Количество";
        DBGrid1->Columns->Items[1]->Title->Alignment = taCenter;
        DBGrid1->Columns->Items[1]->Alignment = taCenter;
        // Установка параметров третьего столбца
    DBGrid1->Columns->Items[2]->Title->Alignment = taCenter;
    DBGrid1->Columns->Items[2]->Alignment = taLeftJustify;
    DBGrid1->Columns->Items[2]->PickList->Clear();
    DBGrid1->Columns->Items[2]->PickList->Add("Кредит");
    DBGrid1->Columns->Items[2]->PickList->Add("Наличными");
    DBGrid1->Columns->Items[2]->PickList->Add("Безнал");
        // Установка параметров четвертого столбца
        DBGrid1->Columns->Items[3]->Title->Alignment = taCenter;
        DBGrid1->Columns->Items[3]->Alignment = taRightJustify;
    // Скрытие пятого столбца
    DBGrid1->Columns->Items[4]->Visible = false;
}
```

Для второго столбца задается заголовок столбца, для второго, третьего и четвертого столбца задается выравнивание заголовка и выравнивание значений. Пятый столбец устанавливается невидимым. Кроме того, для третьего столбца создан список выбора. Установка свойств столбцов произведена с помощью обработки события создания формы, эти же действия можно выполнить с помощью Инспектора объектов. При выполнении приложения форма имеет вид, показанный на рис. 6.4.

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.

Рассмотрим пример использования компонента `DBGrid`, в котором осуществляется преобразование значений записей набора данных в текст.

Рис. 6.4. Задание свойств для столбцов сетки

В качестве набора данных используется компонент `Table`, записи которого отображаются в сетке `DBGrid1`. При нажатии кнопки **Преобразовать в текст** (`Button1`) происходит последовательный просмотр полей всех записей набора данных (сетки) и преобразование их в текст, который помещается в многострочное поле редактирования `Memo1` (рис. 6.5).

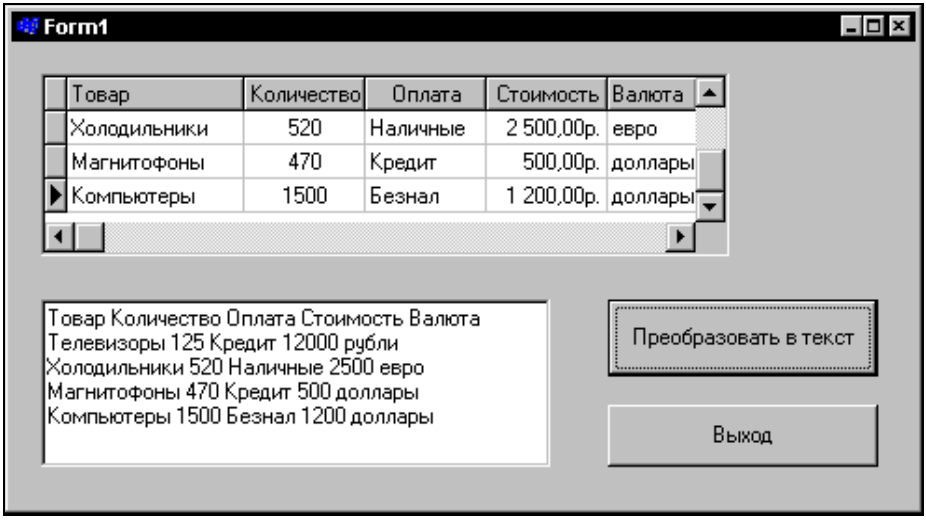


Рис. 6.5. Преобразование записей набора данных в текст

Ниже приведен код обработчика события нажатия кнопки **Преобразовать в текст** (`Button1`).

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        int i, n;
```

```
        AnsiString  s, rs;
        Memo1->Clear();
        Table1->First();
        // Перебор всех записей набора данных
        for (n = 1; n<= Table1->RecordCount; n++)
    {
          rs = ""; s = "";
          // Чтение названий столбцов сетки
          if (n == 1)
          { for (i = 0; i<= DBGrid1->Columns->Count - 1;i++)
      { s = DBGrid1->Columns->Items[i]->FieldName + ' ';
          rs = rs + s;
          }
        Memo1->Lines->Add(rs);
        rs = ""; s = "";
        }
        // Чтение значений полей текущей записи
        for (i = 0; i<= DBGrid1->Columns->Count - 1;i++)
        { s = DBGrid1->Columns->Items[i]->Field->AsString + ' ';
          rs = rs + s;
        }
        Memo1->Lines->Add(rs);
        Table1->Next();
    }
}
```

Для доступа к названиям и значениям полей набора данных использованы свойства `FieldName`, `Count` и `Field` столбцов сетки.

Компонент "модифицированная сетка"

В дополнение компонента `DBGrid` для управления записями таблицы предназначен компонент `DBCtrlGrid` — модифицированная сетка. Компонент `DBCtrlGrid` представляет собой несколько отдельных панелей, на которых располагаются обычные визуальные компоненты, например, такие как `DBEdit` и `DBImage` (рис. 6.6).

Как мы знаем, компонент `DBGrid` отображает информацию одновременно из нескольких записей набора данных. В отличие от него, компонент `DBCtrlGrid` позволяет отображать каждую запись набора данных на отдельной панели.

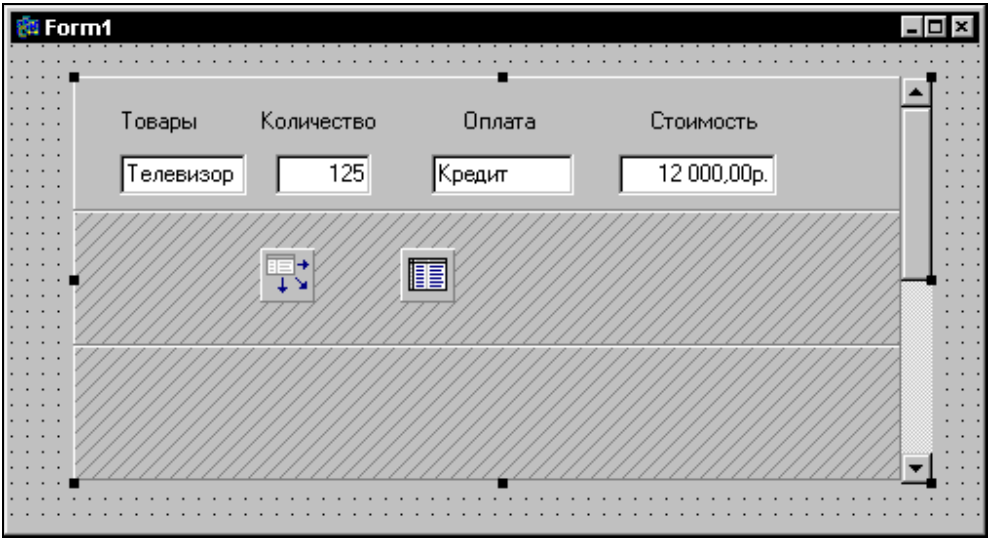


Рис. 6.6. Вид компонента `DBCtrlGrid` при проектировании приложения

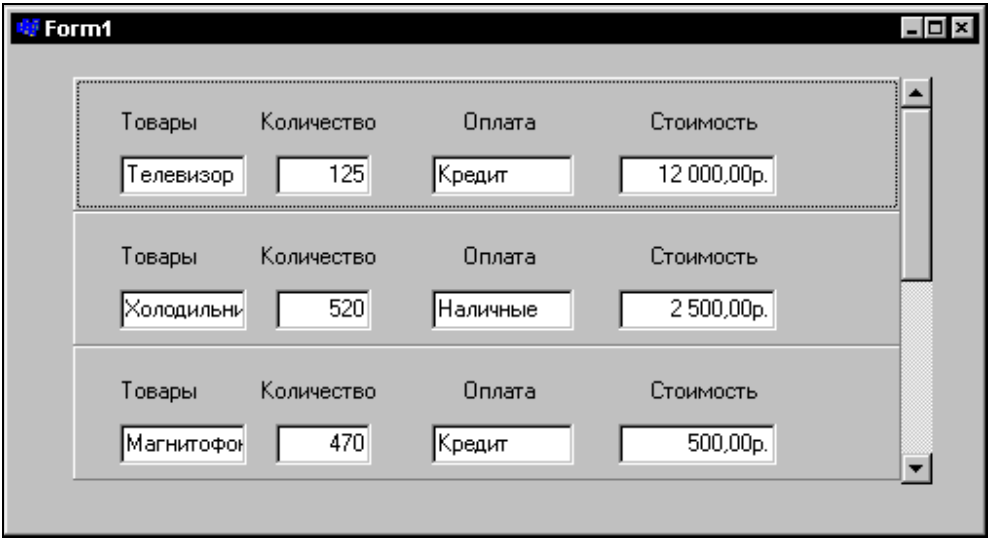


Рис. 6.7. Вид компонента `DBCtrlGrid` при выполнении приложения

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 помеща-

ются на панель модифицированной сетки, у них автоматически устанавливается нужное значение свойства `DataSource`, взятое из одноименного свойства модифицированной сетки. Каждому визуальному компоненту нужно задать значение свойства `DataField` для связи с требуемым полем набора данных (программно или выбрать из списка с помощью Инспектора объектов).

При выполнении приложения компоненты, размещенные на одной панели, дублируются на другие панели сетки. На приведенной (рис. 6.7) сетке расположены четыре компонента `Label` — для названия полей и четыре компонента `DBEdit` — отображения содержимого этих полей: Товары, Количество, Оплата и Стоимость.

Число панелей, одновременно видимых в модифицированной сетке, определяется свойством `PanelCount` типа `int`, доступным для чтения во время выполнения приложения. Свойство `PanelIndex` типа `int` указывает текущую панель, на которой находится просматриваемая запись набора данных. Установив этому свойству соответствующее значение, можно сделать текущей нужную панель.

Все панели сетки имеют одинаковые *размеры*, определяемые свойствами `PanelHeight` (высота) и `PanelWidth` (ширина) типа `int`. 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 `int`, значения которых по умолчанию равны трем и одному. Это соответствует трем панелям в одном столбце и наличию вертикальной полосы прокрутки.

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).

При задании более одного столбца и значении `goHorizontal` свойства `Orientation` у сетки появляется горизонтальная полоса прокрутки (рис. 6.8).

На панели может отображаться *прямоугольник фокуса* (прямоугольная рамка), указывающий на текущую запись. Его отображением управляет свойство `ShowFocus` типа `bool`, которое по умолчанию имеет значению `true`, что

соответствует отображению прямоугольника фокуса. Значение `false` свойства `ShowFocus` скрывает прямоугольник фокуса.

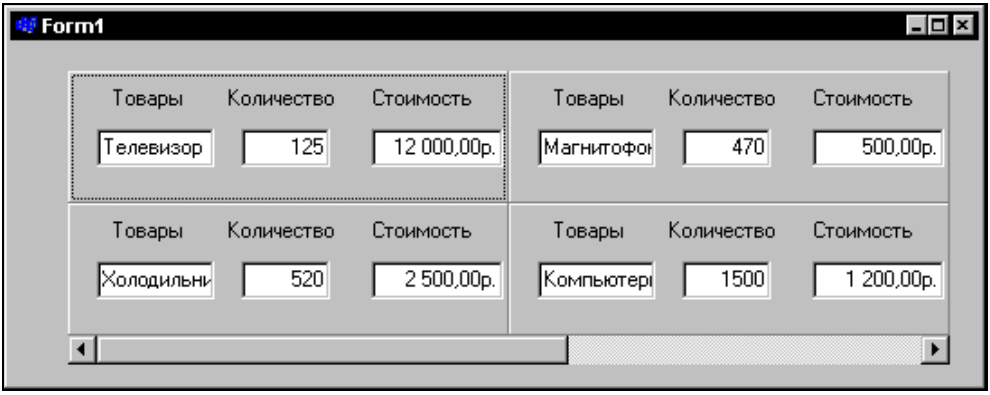


Рис. 6.8. Вид сетки с панелями в две строки и два столбца

Свойства `AllowDelete` и `AllowInsert` типа `bool` 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 `bool` 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`, как и в случае компонента `DBGrid`.

Программисту предоставляются дополнительные возможности управления набором данных, связанные с использованием метода `void __fastcall DoKey(TDBCtrlGridKey Key);`. Выполняемые с помощью этого метода операции задает параметр `Key`, принимающий следующие значения:

- `gkNull` (нет действий);
- `gkEditMode` (переключение значения свойства `EditMode`);
- `gkPriorTab` (передача фокуса ввода предыдущему элементу формы);
- `gkNextTab` (передача фокуса ввода следующему элементу формы);
- `gkLeft` (переход на одну панель влево);
- `gkRight` (переход на одну панель 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).

Например, переход на одну панель вправо обеспечивает следующий код:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
DBCtrlGrid1->DoKey(gkRight);
}
```

При необходимости можно выполнить *программную прорисовку* панелей, используя для этого событие OnPaintPanel типа TPaintPanelEvent, возникающее непосредственно перед отображением панелей. Тип этого события описан как

```
typedef void __fastcall (__closure *TPaintPanelEvent)(TDBCtrlGrid*
DBCtrlGrid, int Index);
```

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 C++ Builder навигатор представлен компонентом `DBNavigator` (рис. 6.9).

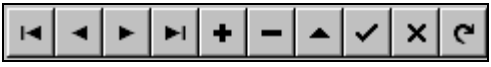


Рис. 6.9. Навигатор

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 `void __fastcall BtnClick(TNavigateBtn Index);` 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:

```
DBNavigator1->BtnClick(nbPrior);
```

В ней имитируется нажатие кнопки `nbPrior`, вызывающей переход к предыдущей 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 `bool`, по умолчанию имеющее значение `true`, т. е. окно подтверждения выводится. Если при отладке приложения установить этому свойству значение `false`, то запись будет удаляться без запроса подтверждения.

Свойство `Flat` типа `bool` 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 `AnsiString`. 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 процедуре:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 Table1->Next();
}
```

при нажатии кнопки `Button1` выполняется переход к следующей записи набора данных `Table1`.

Компонент "графическое изображение"

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 дополнительные характеристики компонента `DBImage`.

Свойство `Stretch` типа `bool` указывает, нужно ли приводить в соответствие размеры изображения размерам области компонента `DBImage1`. По умолчанию это свойство имеет значение `false`, и размеры изображения не подстраиваются под размеры области.

Свойство `AutoDisplay` типа `bool` 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 свойству `AutoDisplay` установить значение `false`, например, так:

```
DBImage1->AutoDisplay = 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:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        DBImage1->LoadPicture();
}
```

В этой процедуре при нажатии кнопки `Button1` в графическом компоненте `DBImage1` выводится содержимое графического поля текущей записи.

С помощью компонента `DBImage` можно работать с *буфером обмена* Windows, выполняя перемещение или копирование изображения в буфер и вставку изображения 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 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`.

Для примера рассмотрим программу для работы с электронным каталогом книг.

Пусть информация о книгах хранится в таблице Paradox, представляющей собой электронный каталог и включающей следующие поля:

- `Код` (автоинкрементное поле);
- `Авторы` (символьное поле);
- `Издательство` (символьное поле);
- `Название` (символьное поле);
- `Серия` (символьное поле);
- `Цена` (денежное поле);

- `Обложка` (графическое поле).

Для просмотра и редактирования каталога создадим приложение, форма которого показана на рис. 6.10.

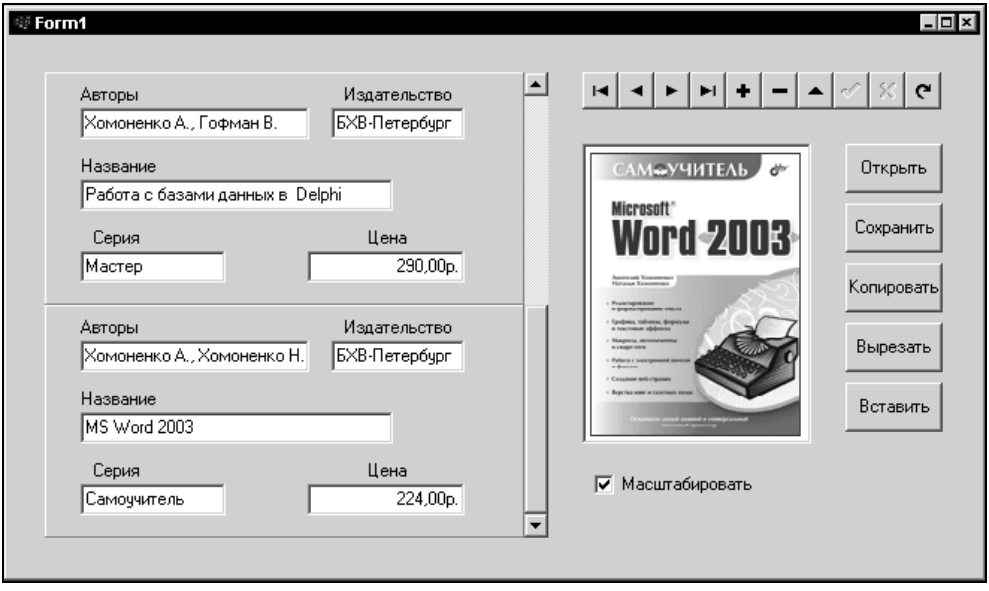


Рис. 6.10. Вид формы для работы с электронным каталогом книг

Содержимое каталога выводится в соответствующих визуальных компонентах, расположенных на панели компонента `DBGrid`. В верхней части расположены поля редактирования `DBEdit`, содержащие по одному из указанных полей. Фото обложки книги выводится в графическом компоненте `DBImage1`. Для навигации по электронному каталогу книг используется навигатор `DBNavigator1`. Флажок `CheckBox1` управляет масштабированием изображения по размеру компонента `DBImage1`. По умолчанию этот флажок снят, и изображение не подстраивается под размеры графического компонента.

Все символьные поля и денежное поле редактируются с помощью соответствующих компонентов `DBEdit`. 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.

Кнопки с названиями **Копировать**, **Вырезать** и **Вставить** выполняют соответствующий обмен между графическим компонентом `DBImage1` и буфером. Кнопка **Закрыть** прекращает работу приложения.

Ниже приведен код файла главного модуля приложения.

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1Albom.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
       if (OpenPictureDialog1->Execute())
    DBImage1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
       OpenPictureDialog1->Filter =
       "Все файлы (*.bmp;*.ico;*.emf;*.wmf)|*.bmp;*.ico;*.emf;*.wmf";
       SavePictureDialog1->Filter = "*.bmp|*.bmp";
}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 if (SavePictureDialog1->Execute())
   DBImage1->Picture->SaveToFile(SavePictureDialog1->FileName);
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
 DBImage1->CopyToClipboard();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
 DBImage1->CutToClipboard();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{
 DBImage1->PasteFromClipboard();
}
//---------------------------------------------------------------------
-------
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
 DBImage1->Stretch = CheckBox1->Checked;
}
```

Отметим, что в отличие от компонента `Image`, у графического компонента `DBImage` нет свойства `Canvas`, позволяющего рисовать на поверхности изображения.

Построение диаграмм

Компонент `DBChart` класса `TDBChart` предназначается для подготовки диаграмм на основании информации, содержащейся в наборе данных. Класс `TDBChart` является производным от класса `TChart` и наследует всю его функциональность (свойства и методы).

Компонент `DBChart` позволяет создавать диаграммы различных типов, в том числе объемные. Этот компонент является достаточно сложным и имеет

большое количество свойств, многие из которых являются объектами и также имеют свои свойства. Задание 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, например, `SeriesList` или `Title` (в этом случае активной становится страница, соответствующая выбранному свойству).

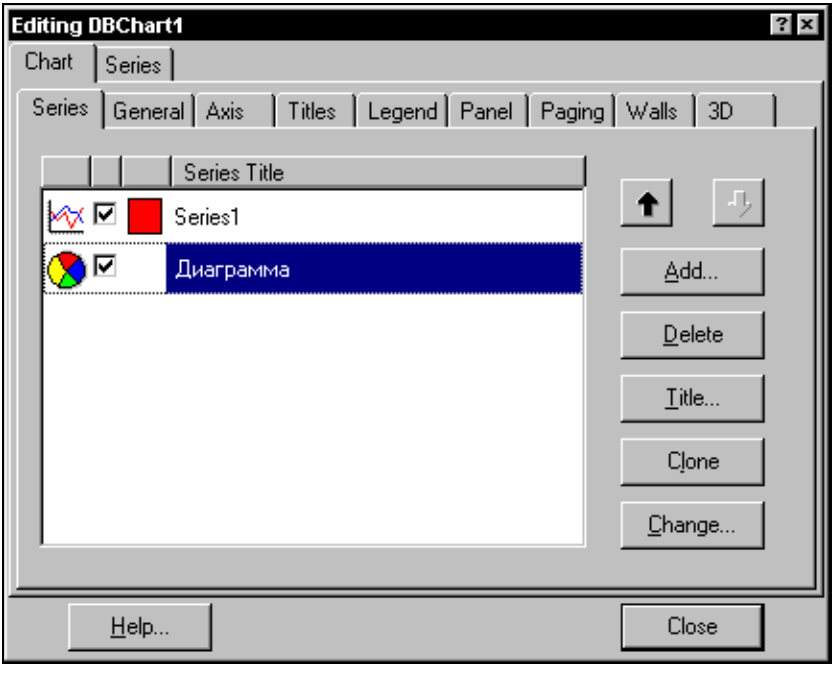


Рис. 6.11. Редактор диаграмм

Важнейшим свойством компонента `DBChart` является свойство `Series [Index:Longint]` типа `TChartSeries`, представляющее собой массив диаграмм, выводимых в области компонента (чаще всего компонент содержит одну диаграмму).

Для каждой диаграммы можно установить:

- тип;
- название;
- оси;
- описание;
- источник данных

и другие параметры.

Несмотря на сложность рассматриваемого компонента, работать с ним достаточно просто ввиду наличия установок по умолчанию. От разработчика требуется, по меньшей мере, 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.
- **Single Record** — значения, входящие в состав одной записи набора данных. При выполнении приложения отображаемая диаграмма соответствует данным текущей записи набора данных.

Если выбран программный способ (вариант **No Data**) ввода значений, то при выполнении приложения нужно вызывать соответствующие методы. Для *управления значениями*, по которым строится диаграмма (принадлежащими элементам массива — свойству `Series`), часто используются методы `Add`, `Delete` или `Clear`.

Функция `Add (Const AValue: Double; Const ALabel: AnsiString; 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 (например, `AddValue` и `AddNull`).

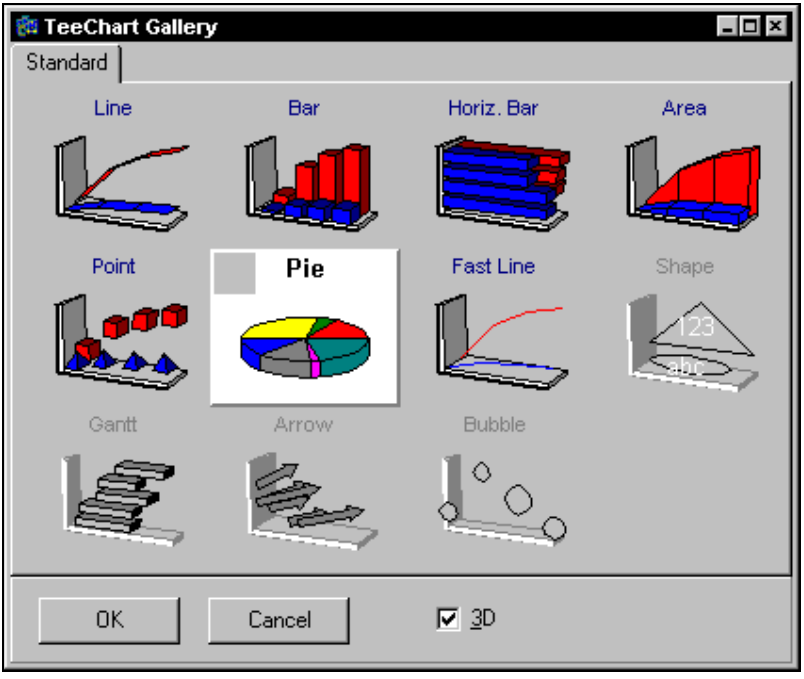


Рис. 6.12. Выбор типа диаграммы

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 двух графиков.

Тип диаграммы (каждого из графиков) задан при разработке приложения. Для нашего примера в окне Редактора диаграмм (рис. 6.11) нужно добавить две диаграммы. В качестве типа каждой из этих диаграмм следует выбрать вариант **Fast Line** (рис. 6.12). Вид графиков при выполнении приложения показан на рис. 6.13. Используемые при построении первого и второго графиков значения вводятся построчно в редакторах `Memo1` и `Memo2`.

Ниже приведены обработчики событий для кнопок формы приложения.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
DBChart1->Title->Text->Add("График");
for (int n = 0; n<= Memo1->Lines->Count - 1;n++)
  { Edit1->Text=IntToStr(n);
   DBChart1->Series[0]->Add(StrToFloat(Memo1->Lines->Strings[n]),
                IntToStr(n), clRed);
```

```
    DBChart1->Series[1]->Add(StrToFloat(Memo2->Lines->Strings[n]),
                 IntToStr(n), clRed);
   }
  }
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 DBChart1->Title->Text->Clear();
 DBChart1->Series[0]->Clear();
 DBChart1->Series[1]->Clear();
}
```

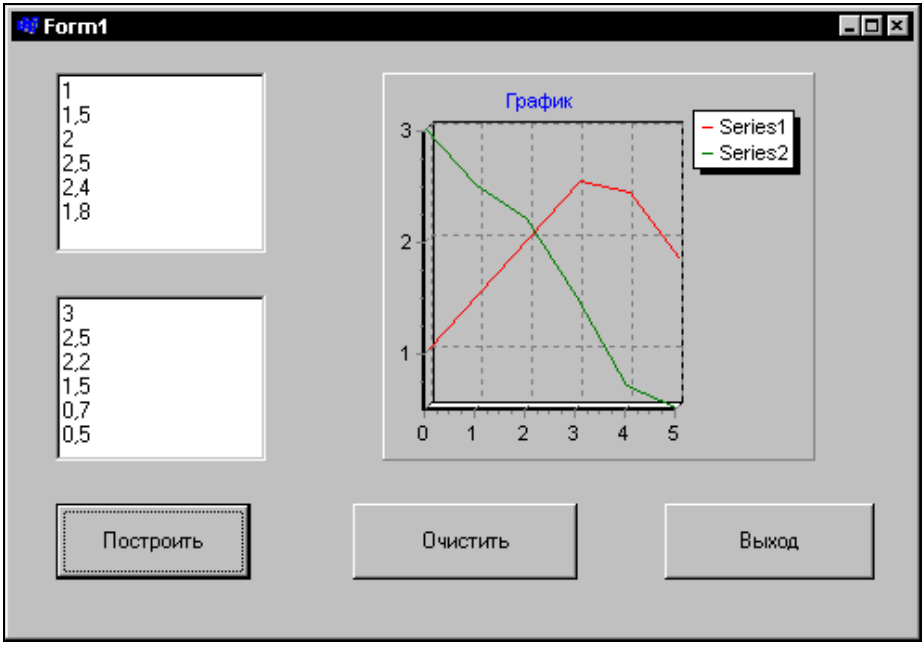


Рис. 6.13. Вывод графика

При нажатии кнопки **Построить** (`Button1`) значения из редакторов `Memo1` и `Memo2` заносятся в компонент `DBChart1`, отображающий первый и второй графики. Удаляются графики нажатием кнопки **Очистить** (`Button2`).

Задание случайных чисел (вариант **Random Values**) в качестве источника данных для диаграммы может быть 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.

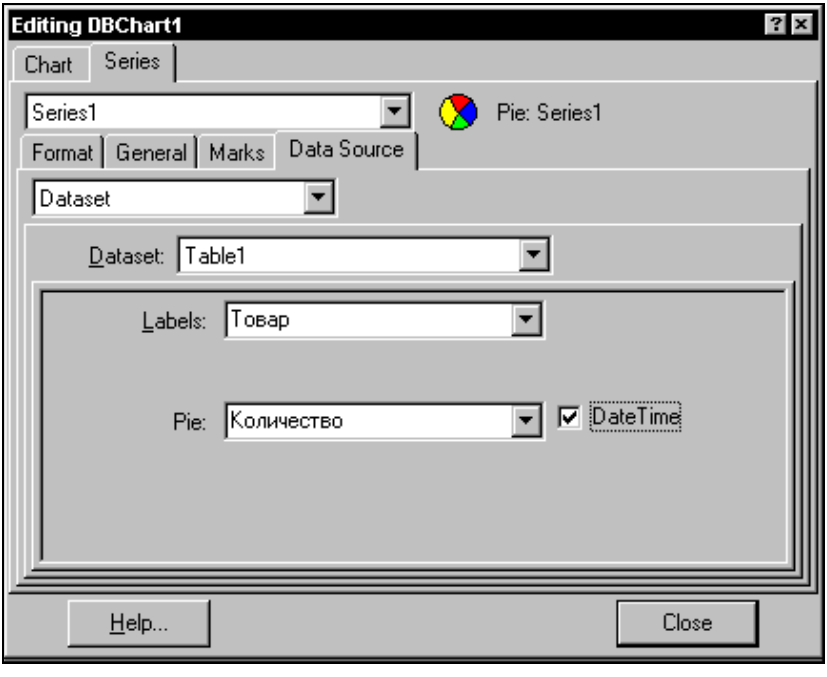


Рис. 6.14. Задание источника данных для диаграммы

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 `Товар`), данные из которого используются в качестве надписей для обозначения секторов диаграммы, а в списке **Pie** — имя поля (в примере `Количество`), из которого выбираются данные для построения секторов диаграммы.

После закрытия окна Редактора диаграмма автоматически строится системой C++ Builder на основании записей, составляющих набор данных (рис. 6.15).

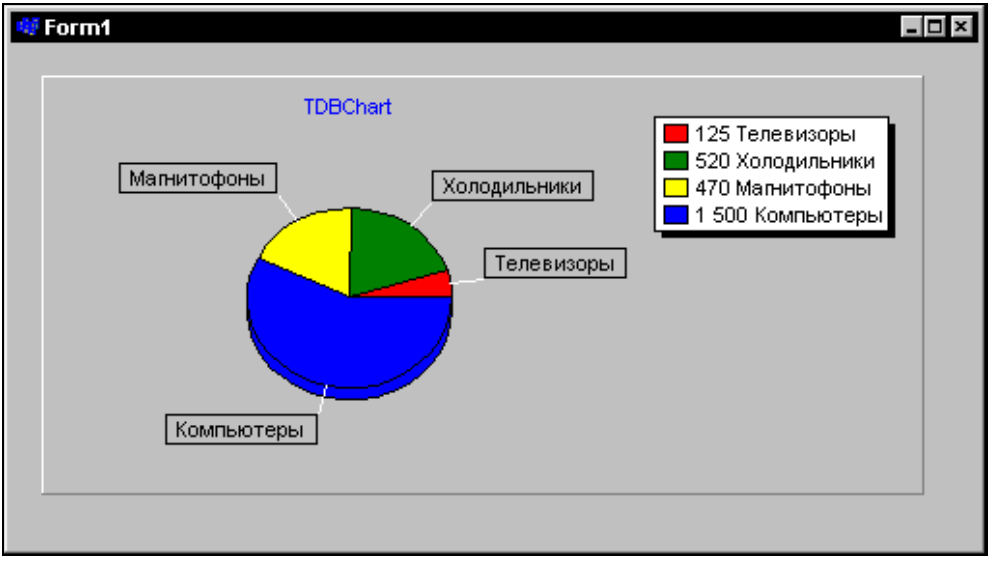


Рис. 6.15. Диаграмма, построенная по значениям набора данных

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.

Глава 7

Навигационный доступ к данным

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 этой 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 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.

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`).

Как отмечалось в *главе 4*, для свойства `TableTYpe`, определяющего тип таблицы, могут задаваться также следующие значения: `ttDefault` (формат таблицы по умолчанию, определяется на основании расширений имен файлов таблиц); `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`.

Метод `HIDESBASE void __fastcall Add(const AnsiString Name, TFieldType DataType, int Size = 0, bool Required = false);` 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` и т. п. 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.
Пример создания таблицы с помощью метода `Create` приводился нами в начале *главы 4*.

Для компонента `Query` действия, связанные с созданием таблицы, выполняются через запрос SQL.

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 `void __fastcall RenameTable(const AnsiString NewTableName);`, 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 `void __fastcall LockTable(TLockType LockType);` 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 `void __fastcall UnlockTable(TLockType LockType);` снимает установленную ранее блокировку.

Рассмотрим пример блокировки таблицы.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{ int   n, k; AnsiString s;
try
  {   Table1->LockTable(ltWriteLock);
      // Пересчет количества
      Table1->First();
      for ( n = 0;n< Table1->RecordCount;n++)
        {  Table1->Edit();
           Edit1->Text=IntToStr(n);
           Table1->FieldByName("Количество")->AsFloat =
                   Table1->FieldByName("Количество")->AsFloat * 1.5;
```

```
          Table1->Next();
         }

       Table1->Post();
  }
catch(...)
   {
         // Отмена блокировки таблицы
         Table1->UnlockTable(ltWriteLock);
   }
}
```

Здесь для каждой записи таблицы значение ее поля с заголовком `Количество` увеличивается в полтора раза. В начале каждой итерации в цикле таблица переводится в режим редактирования с помощью метода `Edit`. В противном случае изменения значений поля записи производиться не будет.

На время пересчета таблица блокируется, что защищает ее от внесения изменений другими приложениями. 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 разными 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 индекс.

Главный индекс (ключ) таблиц Paradox не имеет имени, поэтому выполнить сортировку по этому индексу можно только с помощью свойства `IndexFieldNames`.

Приведем пример сортировки с указанием имен индексов и индексных полей:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  switch  (RadioGroup1->ItemIndex)
       { case   0: Table1->IndexFieldNames = "Товар"; break;
         case 1: Table1->IndexName = "indОплата";   break;
         case 2:  Table1->IndexFieldNames = "Валюта";
       }
}
```

В качестве набора данных используется компонент `Table1`, а сортировка выполняется тремя способами: по имени индексного поля `Товар` (для связанной с набором данных таблицы поле `Товар` определено в качестве главного индекса), по индексу `indОплата`, созданному для поля `Оплата` и по имени индексного поля `Валюта` (индекс `indВалюта`).

Возможный вид формы приложения, обеспечивающего указанные варианты сортировки, приведен на рис. 7.1. В качестве набора данных используется компонент `Table1`. Пользователь может управлять сортировкой его записей с помощью группы переключателей, которыми определяется вид сортировки. Сортировка выполняется после нажатия кнопки **Задать сортировку** (`Button2`).

Замечание

При необходимости изменить порядок сортировки по некторому полю это удобно сделать либо при создании индексов программно *(см. главу 5)* путем задания значения `ixDescending` для параметра `Options`, либо при настройке свойств таблицы с помощью программы Database Desktop *(см. главу 4)*.

Рис. 7.1. Вид формы для сортировки набора данных

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 `int __fastcall MoveBy(int Distance);` — перемещение на число записей, определяемое параметром `Distance`. Если его значение больше нуля, то перемещение осуществляется вперед, если меньше — 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 далее в этой главе.

Замечание

При изменении порядка сортировки набора данных расположение его записей может измениться, что чаще всего и происходит, но указатель по-прежнему адресует на первоначальную запись, даже если она находится на другом месте и имеет новое значение свойства `RecNo`.

Рассмотрим следующий пример:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        float s; int  n;
        s = 0;
        // Установка текущего указателя на первую запись
        Table1->First();
        for (n = 1 ; n<=Table1->RecordCount;n++)
         {  s = s + Table1->FieldByName("Стоимость")->AsFloat;
            // Перемещение текущего указателя на следующую запись
            Table1->Next();
         }
        Edit1->Text = FloatToStr(s);
}
```

В приведенной процедуре перебираются все записи набора данных `Table1`, при этом в переменной `s` накапливается сумма значений, содержащихся в поле `Стоимость`. 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.

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:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        Table1->RecNo = StrToInt(Edit2->Text);
}
```

При нажатии кнопки `Button2` 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 соответственно свойства `Bof` и `Eof` типа `bool`. 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 значение `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, свойству `Eof` устанавливается 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:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{    float s; int  n;
```

```
        Table1->First();
        s  = 0;
        while (!Table1->Eof)
            {
                s = s + Table1->FieldByName("Стоимость")->AsFloat;
                Table1->Next();
            }
        Edit1->Text = FloatToStr(s);
}
```

Как и в предыдущем примере, здесь перебираются все записи набора данных `Table1` и подсчитывается сумма значений, содержащихся в поле `Стоимость`. 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`.

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 `Стоимость`, 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 ранее. 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:

```
int  RecordNumber;
. . .
void __fastcall TForm1::Button1Click(TObject *Sender)
{ // Запоминание номера текущей записи
  RecordNumber = Query1->RecNo;
  // Операция, изменяющая положение указателя
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ // Переход к записи с запомненным номером
  Query1->First();
  Query1->MoveBy(RecordNumber - 1);
}
```

Здесь в обработчике события нажатия кнопки `Button2` для перемещения текущего указателя используется номер записи, запоминаемый в переменной

RecordNumber. Объявление int RecordNumber; должно быть размещено в секции Public: заголовочного файла приложения.

В таблицах Paradox для возврата к записи можно установить свойству RecNo набора данных предварительно сохраненное значение. Например:

```
int  RecordNumber;
...
// Запоминание номера текущей записи
RecordNumber = Table1->RecNo;
// Операция, изменяющая положение указателя
// Переход к записи с запомненным номером
Table1->RecNo = RecordNumber;
```

Кроме описанных способов, для перехода на определенную запись можно использовать *закладки* — специальные пометки каких-либо записей. Закладка имеет тип TBookmark. С помощью закладки можно ссылаться на различные объекты (в нашем случае — на записи набора данных) с помощью методов GetBookmark, GotoBookmark и FreeBookmark.

Перед использованием закладку нужно *создать*, т. е. определить запись, с которой эта закладка связана. Для этого предназначена функция virtual void * __fastcall GetBookmark(void);, 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.

На установленную закладку можно перейти с помощью метода void __fastcall GotoBookmark(void * Bookmark);. Он 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 virtual void __fastcall FreeBookmark(void * Bookmark); Он освобождает закладку перед ее переназначением 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`.

Перед использованием закладки целесообразно уточнить, существует ли она. Функция `virtual bool __fastcall BookmarkValid(void * Bookmark);` 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.

Приведем пример использования закладок для перехода к нужным записям:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
       // Создание закладки bm
       bm = Table1->GetBookmark();
       Table1->First();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
        // Переход на закладку bm
       if (Table1->BookmarkValid(bm))
        {Table1->GotoBookmark(bm);}
       // Освобождение закладки bm
       if (Table1->BookmarkValid(bm))
         {Table1->FreeBookmark(bm); }
}
```

При обработке события нажатия кнопки `Button1` на текущей записи набора данных `Table1` устанавливается закладка `bm`. Далее в приложении могут выполняться операции с записями набора данных `Table1`. При обработке события нажатия кнопки `Button2` для восстановления прежнего положения текущего указателя выполняется переход на закладку, и закладка удаляется. Перед переходом на закладку проверяется ее существование. Объявление закладки `TBookmark bm;` должно быть размещено в секции `Public:` заголовочного файла приложения.

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 C++ Builder 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 набора данных используется его свойство `__property System::AnsiString Filter = {read=FFilterText, write=SetFilterText};`. 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 C++ 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 <= 1000
Post = 'Лаборант' OR Post = 'Инженер'
```

Первое выражение обеспечивает отбор всех записей, для которых значение поля оклада Salary не превышает 1000. Второе выражение обеспечивает отбор записей, поле должности Post которых содержит значение Лаборант или Инженер.

Замечание

В выражениях фильтрации имена полей заключаются в одиночные апострофы, а не в двойные. Для проверки условия равенства значений используется одиночный знак равенства =, а не двойной, как в операторе if.

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 Filtered типа bool. По умолчанию это свойство имеет значение false, и фильтрация выключена. При установке свойству Filtered значения 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.

В выражении фильтра нельзя использовать имена полей, в которых присутствуют буквы русского алфавита.
Если выражение фильтра содержит ошибки, то при попытке выполнить его генерируется исключение. Если фильтр не активен (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` представляет собой пустое множество.

Рассмотрим в качестве примера обработчики событий формы, используемой для фильтрации записей набора данных по выражению. Вид формы приведен на рис. 7.2.

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 **Без фильтра** (`btnNoFilter`) фильтр отключается, при этом показываются все записи.

Ниже приведены три обработчика событий главного модуля приложения.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{       Table1->Active=true;
        Table1->FilterOptions >>foCaseInsensitive;
        Table1->Filtered = false;
}
```

```
//---------------------------------------------------------------------------
void __fastcall TForm1::NotFilter1Click(TObject *Sender)
{
        Table1->Filtered = false;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FilterClick(TObject *Sender)
{
        Table1->Filtered = true;
        Table1->Filter = EditFilter->Text;
}
```

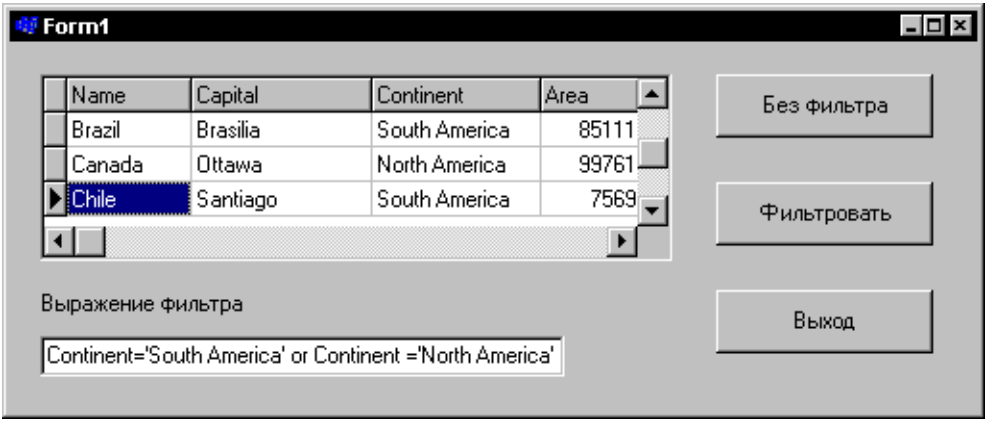


Рис. 7.2. Фильтрация по выражению

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.

Например, чтобы задать фильтрацию по значению поля типа даты, можно использовать следующий обработчик события:

```
void __fastcall TForm1::FilterClick(TObject *Sender)
{
        Table1->Filtered = true;
        Table1->Filter = "HireDate<'01.01.1990'" ;
}
```

Здесь при фильтрации обеспечивается отбор записей, в которых значение поля `HireDate` (дата приема на работу) содержит дату более ранюю, чем `01.01.1990`, т. е. отбираются данные о сотрудниках, принятых на работу до указанной даты.

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`).

Можно задавать 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 текстовом редакторе — компоненте типа `TEdit`.

При необходимости фильтр можно оставить активным, но отменить фильтрацию путем очистки выражения фильтра, например:

```
Table1->Filter = '';
```

При включении фильтрации путем установки свойству `Filtered` значения `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:

```
typedef void __fastcall (__closure *TFilterRecordEvent)(TDataSet* DataSet, bool &Accept);
```

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 C++.

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 далее*) 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 методы 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 bool, 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 и задается фильтрация по диапазону, в котором верхняя граница остается открытой:

```
void __fastcall TForm1::FilterClick(TObject *Sender)
{
Table1->IndexFieldNames="Стоимость";
Table1->SetRangeStart();
Table1->KeyExclusive = true;
Table1->FieldByName("Стоимость")->AsFloat = StrToFloat(EditFilter->Text);
Table1->ApplyRange();
}
```

Индекс по полю Стоимость устанавливается текущим. Если в свойстве EditFilter->Text задать некоторое значение, например 500, то записи, содержащие в поле Стоимость значение 500, не войдут 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 (как в примере), то диапазон открыт, т. е. нижняя граница становится равной минимальному возможному, а верхняя — 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 од-

ному полю. Предварительно в качестве текущего индекса должен быть установлен индекс, построенный по этим полям.

Метод `void __fastcall SetRange(const System::TVarRec* StartValues, const int StartValues_Size, const System::TVarRec* EndValues, const int EndValues_Size);` объединяет возможности методов `SetRangeStart`, `SetRangeEnd` и `ApplyRange`. Он 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 имени и дня рождения
   Table1->IndexFieldNames="Name;BirthDay";
   Table1->SetRange(ARRAYOFCONST(("Петров", "1.1.1945")),
                    ARRAYOFCONST(("Сидоров", "1.1.1948")));
```

В набор данных должны попадать записи о сотрудниках с фамилиями в диапазоне `Петров-Сидоров` и родившимися в диапазоне дат с по `1.1.1945` по `1.1.1948`. 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 (как и показано в примере) должен быть установлен до вызова метода `SetRange`.

Замечание

При задании фильтрации по нескольким полям (как в предыдущем примере) в действительности фильтрация происходит только первому из указанных полей.

При вызове метода `SetRange` ключевое слово `ARRAYOFCONST` задает макрос, используемый для формирования массива констант.

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 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 C++ Builder 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 свойству `Filtered` значения `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` с объявлением `virtual bool __fastcall Locate(const AnsiString KeyFields, const System::Variant &KeyValues, TLocateOptions Options);` 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` 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.

Замечание

При наличии у параметра `Options` значения `loPartialKey` к нему автоматически добавляется значение `loCaseInsensitive`.

Пример поиска по одному полю:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{   TLocateOptions   SOptions;
```

```
    Table1->Locate("Товар", "Магнитофоны", SOptions<<loCaseInsensitive);
}
```

Поиск выполняется по полю `Товар` и ищется первая запись, для которой значением этого поля является строка `Магнитофоны`. В качестве параметра поиска используется значение `loCaseInsensitive`.

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` с объявлением `extern PACKAGE Variant __fastcall VarArrayOf(constVariant * Values, const int Values_Size);` . Параметр `Values_Size` определяет индекс последнего элемента массива. Порядок следования значений должен соответствовать порядку полей параметра `KeyFields`. Например:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{     TLocateOptions   SOptions;
      Variant arrvalues[]={Edit1->Text,Edit2->Text};
      Table1->Locate("Оплата;Валюта", VarArrayOf(arrvalues,1),
              SOptions<<loPartialKey<<loCaseInsensitive);
}
```

Поиск выполняется по полям `Оплата` и `Валюта`, ищется первая запись, для которой значение поля оплаты содержится в `Edit1->Text`, а значение поля валюты содержится в `Edit2->Text`. Регистр букв значения не имеет.

Замечание

Значение `loPartialKey` параметра `Options` должно обеспечивать возможность частичного совпадения строковых значений в ключевых полях, по которым осуществляется поиск. В действительности поиск записи происходит только при строгом совпадении значений полей.

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.

Для поиска в наборе данных также используется метод `Lookup`, который работает аналогично методу `Locate`. Метод имеет объявление

```
virtual System::Variant __fastcall Lookup(const AnsiString KeyFields, const System::Variant &KeyValues,const AnsiString ResultFields);
```

и осуществляет поиск записи, удовлетворяющей определенным условиям. В отличие от метода `Locate`, она не перемещает указатель текущей записи на найденную запись, а считывает информацию из полей записи. 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.

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`.

При неудачном поиске метод `Lookup` возвращает вариантный массив, для которого функция `VarType` возвращает значение `varNull`.

Приведем пример использования метода `Lookup`.

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  int C; AnsiString A; Variant V;
  V = Table1->Lookup("Continent;Capital",
  VarArrayOf(OPENARRAY(Variant, ("South America","Brasilia"))),
                              "Name;Area");
  if (!(VarType(V) == varNull))
     { A = V.GetElement(0);
       C = V.GetElement(1);
       ShowMessage(A + " имеет площадь = "+IntToStr(C) +" кв. км");
```

```
    }
  else
    ShowMessage("Search unsuccessful!");
}
```

В примере поиск ведется по полям континента `Continent` и столицы `Capital`, а из найденной записи считываются значения полей названия государства `Name` и его площади `Area`. 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.

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 частичное совпадение заданных для поиска значений и значений полей записей.

Метод с объявлением `bool __fastcall FindKey(const System::TVarRec * KeyValues, const int KeyValues_Size);` 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:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Table1->IndexFieldNames = "Continent;Capital";
  if (!Table1->FindKey (ARRAYOFCONST((EditContinent->Text, EditCapital->Text))))
```

```
    { ShowMessage("Запись не найдена!");}
}
```

Здесь в наборе данных выполняется поиск первой записи, поля Continent и Capital которой содержат значения, введенные в редакторы EditContinent и EditCapital. В случае неудачного поиска пользователю выдается сообщение. Перед выполнением поиска текущим устанавливается индекс, построенный по поисковым полям.

Аргументы для метода FindKey задаются с помощью функции ARRAYOFCONST, которая в примере из двух констант формирует массив, имеющий тип TVarRec.

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 FindNearest имеет объявление void __fastcall FindNearest(const System::TVarRec * KeyValues, const int KeyValues_Size); и, 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 метода FindNearest всегда является успешным и перемещает указатель текущей записи на запись, в наибольшей степени отвечающую условиям поиска.

Например:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
        Table1->IndexFieldNames = "Capital";
        Table1->FindNearest (ARRAYOFCONST(("Br")));
}
```

В приведенной процедуре указатель текущей записи перемещается на запись, поле Capital которой содержит значение, начинающееся с букв Br. Если такой записи нет, то будет найдена запись, в которой поле Capital имеет значение, начинающееся со следующих за Br букв.

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.

При поиске по нескольким полям текущего индекса используется свойство KeyFieldCount типа bool, 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 `bool`, при установке которому значения `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 `bool`. По умолчанию это свойство имеет значение `false`, и набор данных `Query` доступен только для чтения. Чтобы получить разрешение визуально и программно редактировать записи, свойству `RequestLive` нужно установить значение `true`. 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 `bool`, 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 сам набор данных 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 свойству AutoEdit источника данных DateSource для визуальных компонентов должно быть установлено значение true (по умолчанию). В противном случае (при значении false) 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, 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 сотрудников, а также набор данных, предназначенный для выбора сотрудника при составлении графика отпусков. Компоненты Table обоих наборов могут находиться в разных формах. После редактирования списка сотрудников следует обновить оба набора данных. Приведем фрагмент кода, в котором проводится обновление набора данных:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
        Table1->Edit();
        Table1->FieldByName("Capital")->AsString=Edit1->Text;
        Table1->Post();
        Table1->Refresh();
        Form2->Table1->Refresh();
}
```

Здесь при программном изменении набора данных `Table1`, находящегося в форме `Form1`, обновляется он сам, а также набор данных `Table1`, 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` (предложение `#include "Unit2.h"`).

Редактирование записей

Редактирование записей заключается в изменении значений их полей. Отредактировать можно только текущую запись, поэтому перед редактированием обычно выполняются операции поиска и перемещения на требуемую запись. Если 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, то генерируется исключение, а выполнение метода `Edit` прекращается.
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) 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.

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 можно предложить подтвердить свои действия. Для отмены процесса редактирования в обработчике события `BeforeEdit` можно сгенерировать "тихое" исключение.

При переходе в режим редактирования обычно удобнее всего использовать обработчик события `BeforeEdit`, т. к. оно генерируется при переводе набора данных в режим редактирования любым способом. Например:

```
void __fastcall TForm1::Table1BeforeEdit(TDataSet *DataSet)
{
if (MessageDlg("Выполнить редактирование?",mtConfirmation,
                 TMsgDlgButtons() <<mbYes << mbNo, 0)== mrNo)
        Abort();
}
```

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("Name")->AsString = Edit1->Text;
Table1->FieldByName("Area")->AsInteger = StrToInt(Edit2->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 `bool`. 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.

Например: рассмотрим фрагмент кода, в котором осуществляется редактирование записей:

```
// Изменение площади в текущей записи
Table1->Edit();
Table1->FieldByName("Area")->AsInteger = StrToInt(Edit2->Text);
Table1->Post();
// Переход к следующей записи
Table1->Next();
```

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`. Конкретные 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`.

Метод имеет объявление `void __fastcall SetFields(const System::TvarRec * Values, const int Values_Size);` и 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. Если в качестве значения элемента массива задать `NULL`, то соответствующее поле примет значение `0`. Если число значений в массиве превышает число полей, то при выполнении метода `SetFields` генерируется исключение.

Замечание

Если для набора данных использовался Редактор полей, то при выполнении метода `SetFields` учитывается порядок полей, установленный с помощью это-

го 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 (ARRAYOFCONST(("Country", "Capital", NULL, NULL, 22222)));
```

Здесь в первое, второе и пятое поля текущей записи набора данных `Table1` заносятся страна, столица и численность населения соответственно. Третье и четвертое поля этой записи принимают значение `0`.

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 перед записью, 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:

```
Table1->Insert();
Table1->FieldByName("Name")->AsString = Edit1->Text;
Table1->Post();
```

Здесь в новой записи задаются значение поля наименования (`Name`), 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.

Метод `InsertRecord` объединяет функциональность методов `Insert` и `SetFields`, выполняя те же действия, что и их последовательный вызов. Метод имеет объявление `void __fastcall InsertRecord(const System::TvarRec * Values, const int Values_Size);` и вставляет в позицию указателя текущей записи новую, задавая значения всех или части ее полей. Например:

```
Table1->InsertRecord (ARRAYOFCONST(("Country", "Capital", NULL, NULL,22222)));
```

Здесь вставляется запись и затем, как и в примере с методом `SetFields`, в первое, второе и пятое поля вставленной записи набора данных `Table1` заносятся страна, столица и численность населения соответственно. Третье и четвертое поля этой записи принимают значение `0`.

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:

```
void __fastcall TForm1::Table1NewRecord(TDataSet *DataSet)
{
```

```
Table1->FieldByName("Name")->AsString = Edit1->Text;
Table1->FieldByName("Capital")->AsString = Edit2->Text;
Table1->FieldByName("Area")->AsInteger = StrToInt(Edit3->Text);
}
```

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 C++ Builder 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.

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.

Удаление нескольких последовательно расположенных записей имеет особенность, связанную с тем, что при вызове метода `Delete` после удаления текущей записи указатель автоматически перемещается на следующую запись.

Приведем пример удаления записей набора данных с текущей по первую:

```
void __fastcall TForm1::Button8Click(TObject *Sender)
{   int    n = Table1->RecNo;
           while (n>=1)
          {Table1->Delete();Table1->RecNo=Table1->RecNo-1;n--;}
 }
```

В примере перебор записей выполняется с текущей записи набора данных. После удаления текущей записи указатель снова оказывается на записи с номером на единицу меньше вплоть до первой записи набора данных.

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.

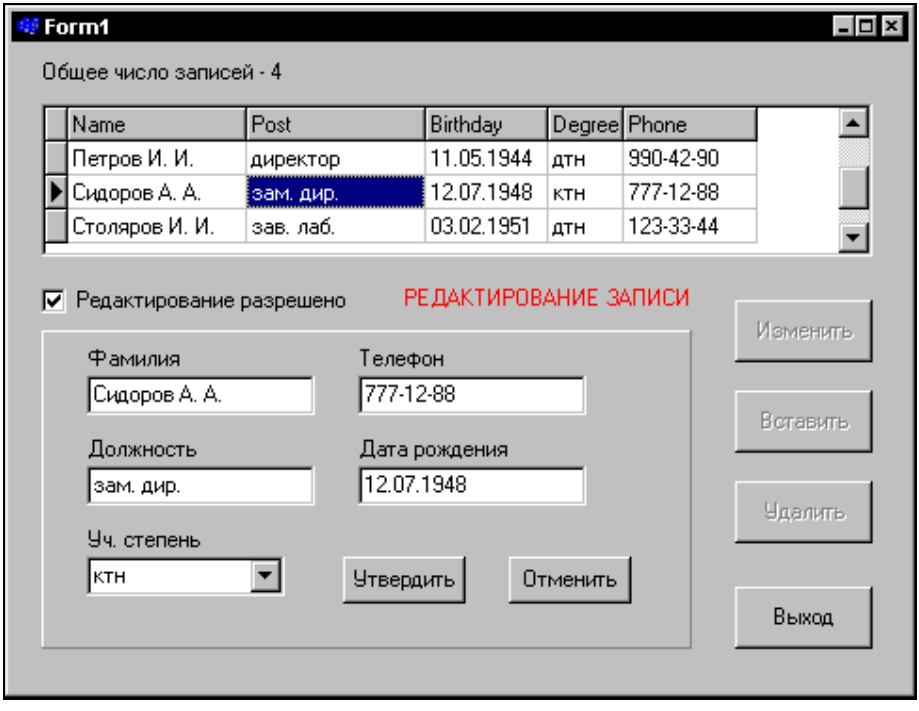


Рис. 7.3. Форма приложения для работы с даными о сотрудниках

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. 7.3), с помощью которой можно изменять записи таблицы, содержащей сведения о сотрудниках. Данные о сотрудниках включают фамилию (поле `Name`), должность (поле `Post`), дату рождения (поле `Birtday`), ученую степень (поле `Degree`) и телефон (поле `Pfone`). Поле фамилии и поле должности обязательно должны быть заполнены.

Далее приводится код главного модуля приложения.

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit17Example.h"
//---------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TForm1::SBrowse(TObject *Sender)
{ // Процедура настройки режима просмотра записей
  Label1RecordCount->Caption = "Общее число записей - " + IntToStr(Table1-
>RecordCount);
  // Вызов процедуры обработки события щелчка флажка CheckBox1CanEdit
  CheckBox1CanEditClick(Sender);
  Button4ChangeOK->Enabled = false;
  Button5ChangeCancel->Enabled = false;
  Label1Regime->Font->Color = clBlack;
  Label1Regime->Caption = "ПРОСМОТР ЗАПИСИ";
}
//---------------------------------------------------------------------
void __fastcall TForm1::SChange(TObject *Sender)
{ // Процедура настройки режима изменения записей
  Button1Edit->Enabled = false;
  Button2Insert->Enabled = false;
  Button3Delete->Enabled = false;
  Button4ChangeOK->Enabled = true;
  Button5ChangeCancel->Enabled = true;
}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Table1->Active = true;
  // Исходное состояние элементов управления
  SBrowse(Sender);
  // Изменение записей запрещено
  CheckBox1CanEdit->Checked = false;
  // Запрет автоматического перехода в режим редактирования
  DataSource1->AutoEdit = false;
  // Формирование списка ученых степеней
  DBComboBox1Degree->Items->Clear();
  DBComboBox1Degree->Items->Add("нет");
  DBComboBox1Degree->Items->Add("ктн");
  DBComboBox1Degree->Items->Add("дтн");
}
//---------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
 Table1->Active = false;
}
//---------------------------------------------------------------------
void __fastcall TForm1::CheckBox1CanEditClick(TObject *Sender)
{
  TBookmark bm1;
  // Запоминание положения текущей записи
  bm1 = Table1->GetBookmark();
  // Отключение отображения изменений данных в визуальных компонентах
  Table1->DisableControls();
  if (!CheckBox1CanEdit->Checked)
    {
      Table1->Active = false;
      Table1->ReadOnly = true;
      Table1->Active = true;
      // Блокирование элементов перехода
      // в режимы изменения записей
      Button1Edit->Enabled = false;
      Button2Insert->Enabled = false;
      Button3Delete->Enabled = false;
    }
   else
    {
      Table1->Active = false;
```

```
      Table1->ReadOnly = false;
      Table1->Active = true;
      // Разблокирование элементов перехода
      // в режимы изменения записей
      Button1Edit->Enabled = true;
      Button2Insert->Enabled = true;
      // Запрет удаления записей для пустого набора данных
      if (Table1->RecordCount > 0)
          Button3Delete->Enabled = true;
           else Button3Delete->Enabled = false;
     }
   // Возврат к текущей записи
   if (Table1->BookmarkValid(bm1))
       Table1->GotoBookmark(bm1);
   if (Table1->BookmarkValid(bm1))
       Table1->FreeBookmark(bm1);
   // Включение отображения изменений данных в визуальных компонентах
   Table1->EnableControls();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Table1BeforeEdit(TDataSet *DataSet)
{
  // Запрос на подтверждение редактирования
  if (MessageDlg("Выполнить редактирование?",mtConfirmation,
               TMsgDlgButtons() <<mbYes << mbNo, 0)== mrNo)
     Abort();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button1EditClick(TObject *Sender)
{ // Переход в режим редактирования
  Table1->Edit();
  Label1Regime->Font->Color = clRed;
  Label1Regime->Caption = "РЕДАКТИРОВАНИЕ ЗАПИСИ";
  SChange(Sender);
  if (DBEdit1Name->CanFocus()) DBEdit1Name->SetFocus();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Table1BeforeInsert(TDataSet *DataSet)
{
   if (MessageDlg("Выполнить вставку?",mtConfirmation,
               TMsgDlgButtons() <<mbYes << mbNo, 0)== mrNo)
      Abort();
}
```

```
//---------------------------------------------------------------------
void __fastcall TForm1::Button2InsertClick(TObject *Sender)
{ // Переход в режим вставки
  Table1->Insert();
  Label1Regime->Font->Color = clGreen;
  Label1Regime->Caption  = "ВСТАВКА ЗАПИСИ";
  SChange(Sender);
  if (DBEdit1Name->CanFocus()) DBEdit1Name->SetFocus();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button3DeleteClick(TObject *Sender)
{
  // Подтверждение перехода в режим просмотра удаляемой записи
   if (MessageDlg("Выполнить вставку?",mtConfirmation,
                TMsgDlgButtons() <<mbYes << mbNo, 0)== mrNo)
         Abort();
  Label1Regime->Font->Color = clRed;
  Label1Regime->Caption = "УДАЛЕНИЕ ЗАПИСИ";
  SChange(Sender);
  if (Button5ChangeCancel->CanFocus())
      Button5ChangeCancel->SetFocus();
}
//---------------------------------------------------------------------
void __fastcall TForm1::Table1BeforePost(TDataSet *DataSet)
{
  // Проверка, задана ли фамилия
  if (DBEdit1Name->Text == "")
     {Beep();
      MessageDlg("Фамилия не задана",
               mtInformation, TMsgDlgButtons() << mbOK, 0);
      if (DBEdit1Name->CanFocus()) DBEdit1Name->SetFocus();
      Abort();
     }
  // Проверка, задана ли должность
  if (DBEdit2Post->Text == "")
     {Beep();
      MessageDlg("Должность не задана",
               mtInformation, TMsgDlgButtons() << mbOK, 0);
      if (DBEdit2Post->CanFocus()) DBEdit2Post->SetFocus();
      Abort();
     }
}
//---------------------------------------------------------------------
```

```
void __fastcall TForm1::Button4ChangeOKClick(TObject *Sender)
{
  // Утверждение изменений в текущей записи (редактируемой или новой)
  // или удаления текущей записи (просматриваемой)
  if ((Table1->State==dsEdit) || (Table1->State==dsInsert))
      Table1->Post() ;
  else if (Label1Regime->Caption == "УДАЛЕНИЕ ЗАПИСИ")
         Table1->Delete();
  SBrowse(Sender);
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button5ChangeCancelClick(TObject *Sender)
{
  // Если набор данных был в режиме просмотра (при удалении записи),
  // то никаких действий метод Cancel не выполняет
  Table1->Cancel();
  SBrowse(Sender);
}
//---------------------------------------------------------------------
void __fastcall TForm1::Button6Click(TObject *Sender)
{
Close();
}
```

Файл заголовка приложения приведен далее.

```
//---------------------------------------------------------------------------
#ifndef Unit17ExampleH
#define Unit17ExampleH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <DB.hpp>
#include <DBGrids.hpp>
#include <DBTables.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
#include <DBCtrls.hpp>
#include <Mask.hpp>
//---------------------------------------------------------------------------
```

```
class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TTable *Table1;
        TDBGrid *DBGrid1;
        TDataSource *DataSource1;
        TButton *Button1Edit;
        TButton *Button2Insert;
        TButton *Button3Delete;
        TCheckBox *CheckBox1CanEdit;
        TPanel *Panel1;
        TLabel *Label1Regime;
        TLabel *Label2;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
        TLabel *Label6;
        TButton *Button4ChangeOK;
        TButton *Button5ChangeCancel;
        TButton *Button6;
        TLabel *Label1RecordCount;
        TDBEdit *DBEdit1Name;
        TDBEdit *DBEdit2Post;
        TDBEdit *DBEdit1Phone;
        TDBEdit *DBEdit1Birthday;
        TDBComboBox *DBComboBox1Degree;
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall SBrowse(TObject *Sender);
        void __fastcall SChange(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
        void __fastcall CheckBox1CanEditClick(TObject *Sender);
        void __fastcall Table1BeforeEdit(TDataSet *DataSet);
        void __fastcall Button1EditClick(TObject *Sender);
        void __fastcall Table1BeforeInsert(TDataSet *DataSet);
        void __fastcall Button2InsertClick(TObject *Sender);
        void __fastcall Button3DeleteClick(TObject *Sender);
        void __fastcall Table1BeforePost(TDataSet *DataSet);
        void __fastcall Button4ChangeOKClick(TObject *Sender);
        void __fastcall Button5ChangeCancelClick(TObject *Sender);
        void __fastcall Button6Click(TObject *Sender);
private:       // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
```

```
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

В качестве набора данных используется компонент `Table1`, записи которого отображает сетка `DBGrid1`. Значения полей текущей записи отображаются в компонентах `DBEdit1Name`, `DBEdit2Post`, `DBEdit1Phone`, `DBEdit1Birthday` и `DBComboBox1Degree`. Установка свойств, определяющих взаимные отношения компонентов, связанных с набором данных, выполнена при разработке приложения с помощью Инспектора объектов.

Пользователь не может переводить набор данных в режимы изменения с помощью визуальных компонентов, т. к. свойству `AutoEdit` источника данных `DataSource1` установлено значение `false`.

Для перевода набора данных в режимы редактирования и вставки служат кнопки **Изменить** (`Button1Edit`) и **Вставить** (`Button2Insert`) соответственно. Переход в эти режимы осуществляется вызовом методов `Edit` и `Insert`, после чего название режима отображается в надписи `Label1Regime`, блокируются кнопки, связанные с переходом в режимы изменения, и разблокируются кнопки `Button4ChangeOK` и `Button5ChangeCancel`, 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 `DBEdit1Name`, `DBEdit2Post`, `DBEdit1Phone`, `DBEdit1Birthday` и `DBComboBox1Degree`.

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** (`Button3Delete`) набор данных остается в режиме просмотра текущей записи. При этом `Label1Regime` указывает на переход в режим удаления, и если пользователь подтвердит удаление, нажав кнопку `Button4ChangeOK`, то для удаления текущей записи вызывается метод `Delete`. Запрос на подтверждение перехода в режим удаления кодируется в обработчике события нажатия кнопки `Button3DeleteClick`.

Флажок `CheckBox1CanEdit` 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 (в обработчике события смены состояния флажка `CheckBox1CanEdit`) проверяется число его записей, и если набор данных пуст, то кнопка `Button3Delete` блокируется.

Связывание таблиц

В общем случае приложение может иметь доступ к нескольким связанным таблицам. Как отмечалось *(см. главу 2)*, связь между таблицами устанавливается через поля связи, которые обязательно должны быть индексированными. При задании связи между двумя таблицами имеет место отношение подчиненности: 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.

Для организации связи между таблицами в подчиненной таблице (компонент `Table` в случае механизма BDE или его аналоги для других механизмов доступа) используются следующие свойства:

- `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.

При работе со связанными таблицами нужно учитывать 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 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 ссылочной целостности (по изменению полей связи и каскадному удалению записей) могут быть наложены на таблицы при их создании, например, в среде программы Database Desktop, или устанавливаться программно.

Рассмотрим следующий пример. Предположим, что требуется установить связь между тремя таблицами, Две из них являются подчиненными (таблицы `PSalary` — оклад по должности и `DSarary` — надбавка за степень) и одна — главной (`SEmployee` — данные о сотрудниках) (рис. 7.4).

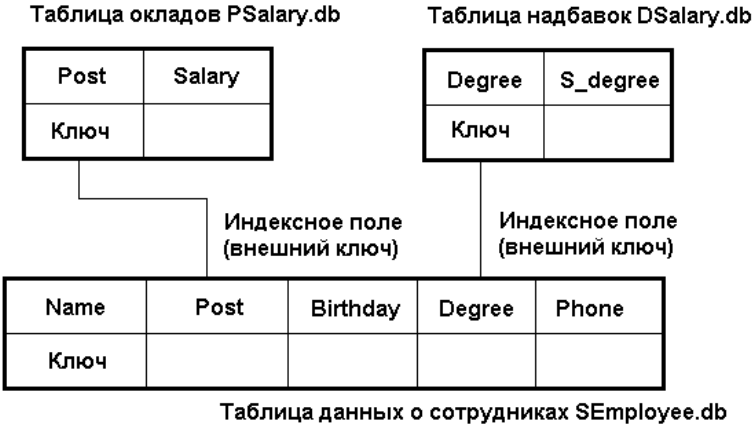


Рис. 7.4. Связи между таблицами

Для каждой из трех таблиц в приложении зададим по два компонента `Table` и `DataSource`. А именно `Table1` и `DataSource1` — для таблицы данных о сотрудниках, `Table2` и `DataSource2` — для таблицы окладов, `Table3` и `DataSource3` — для таблицы надбавок.

Для компонентов `Table1-Table3` в качестве значения свойства `DatabaseName` установим `BCDEMOS`, а свойству `TableName` зададим значения `SEmployee`, `PSalary` и `DSarary` соответственно.

Для компонентов `DataSource1-DataSource3` свойству `DataSet` выберем значения `Table1-Table3` соответственно.

Для каждого из компонентов подчиненных таблиц (`Table2` и `Table3`) свойству `MasterSource` путем выбора из списка зададим одинаковое значение

`DataSource1`. Затем щелчком кнопки мыши в строке свойства `MasterFields` откроем диалоговое окно **Field Link Designer** (Конструктор полей связи) (рис. 7.5). Здесь 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. 7.5). После нажатия кнопки **Add** выбранные поля связываются между собой, что отображается в списке **Joined Fields** (Связанные поля), например, так: `P_Post -> Post`. При этом оба поля пропадают из своих списков. Заполнение свойств `IndexName` и `MasterFields` происходит после закрытия окна при нажатии кнопки **OK**.

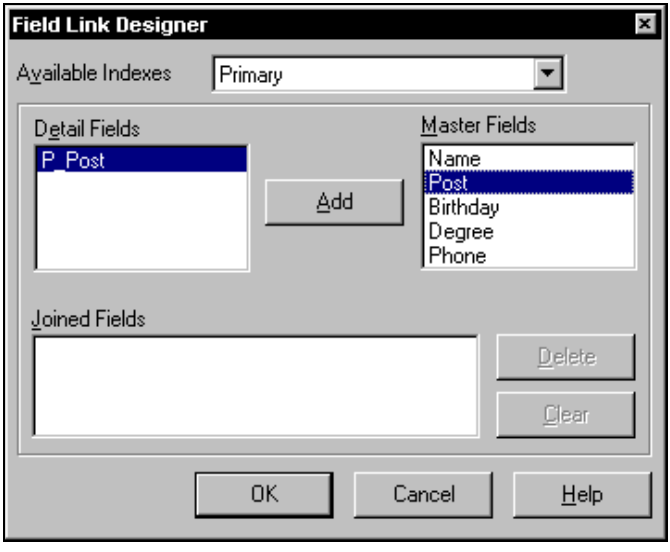


Рис. 7.5. Диалоговое окно Конструктора полей связи

Установление связей между главной и подчиненными таблицами приводит к тому, что при выполнении приложения выбор некоторой записи в главной таблице будет приводить к выбору соответствующих записей в каждой из подчиненных таблиц (рис. 7.6).

В частности, при выборе в главной таблице записи со значением `вед. нс` поля `Post` в подчиненной таблице автоматически отображается строка с тем же значением в поле связи `P_Post`.

В рассмотренном нами примере использовалась связь типа "многие-к-одному". Чаще применяется связь типа "один-ко-многим". Однако в этом случае описанным способом можно установить связь только между двумя таблицами (одной главной таблицей и одной подчиненной таблицей), в то

время как при таком типе связи требуется наличие нескольких главных и одной подчиненной таблиц.

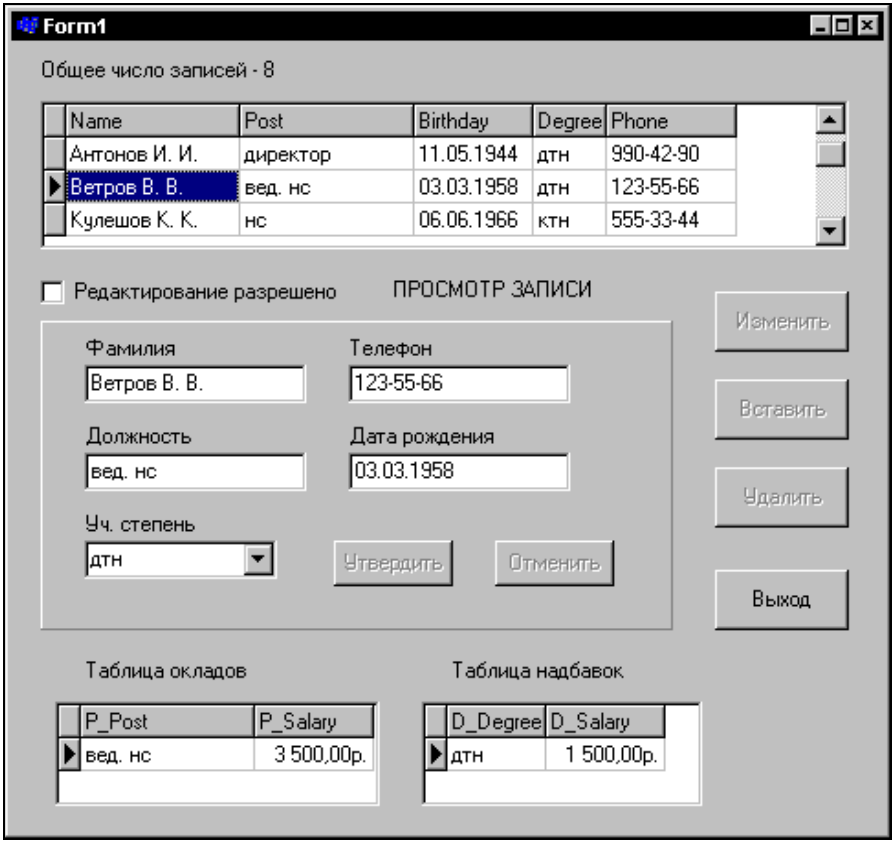


Рис. 7.6. Вид приложения с несколькими таблицами при выполнении

Глава 8

Доступ к данным с помощью запросов

Доступ к данным с помощью запросов (реляционный способ) 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 C++ Builder при использовании механизма BDE в качестве набора данных должны применяться такие компоненты, как `Query` или `StoredProc`, позволяющие выполнить SQL-запрос. При использовании других механизмов доступа к данным также можно использовать реляционный способ доступа. Так, для механизма ADO должны применяться компоненты `ADOQuery` и `ADOStoredProc`.

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 операторов 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 C++ Builder. При этом для работы с командами SQL предлагаются соответствующие средства и компоненты. В C++ Builder 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. При этом имеет место SQL-запрос *с параметрами*.

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.

Рассмотрим основные возможности *локального SQL* (Local SQL), используемого в C++ Builder. Локальный SQL представляет собой подмножество SQL-92 и предназначен для доступа к таблицам локальных баз данных dBASE, Paradox и FoxPro. С одной стороны, по сравнению со стандартом языка локальный SQL включает ряд дополнений, учитывающих особенности указанных баз данных. С другой — в нем имеется ряд ограничений, например, в нем нельзя работать с просмотрами и управлять привилегиями.

Особенности языка SQL, используемого для работы с удаленными БД (промышленными СУБД), будут рассмотрены в *главах 14* и *15*.

В приложениях C++ Builder для выполнения операторов 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 свойству `Active` значения `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.

Набрать и выполнить в интерактивном режиме текст SQL-запроса позволяют также инструментальные программы, поставляемые вместе с C++ Builder, 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` или его аналога), производятся на этапе выполнения приложения. При наличии синтаксических ошибок в тексте SQL-запроса генерируется исключение. 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 операторов 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 операторов языка. Точка с запятой в конце SQL-операторов 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 программы. Комментарий ограничивается символами `/*` и `*/`.

Операторы локального языка SQL разделяют на два подъязыка:

- язык определения данных (Data Definition Language — DDL);
- язык манипулирования данными (Data Manipulation Language — DML).

Язык определения данных включает операторы SQL, используемые для создания, изменения и удаления таблиц, а также для создания и удаления индексов. Например, оператор `CREATE TABLE` предназначен для создания таблицы.

Язык манипулирования данными включает операторы SQL, используемые для выборки, вставки, обновления и удаления данных таблиц. Например, оператор `SELECT` служит для выборки данных из таблиц.

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 операторов определения данных языка SQL.

Создание и удаление таблицы

Для создания таблицы служит оператор `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 оператора.

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` — общее количество цифр в представлении данных рассматриваемого типа, `Y` — количество 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
NUMERIC(X,Y)	Number (X,Y)	Number
FLOAT(X,Y)	Number	Float (X,Y)
CHARACTER(N)	Character	Alpha

Таблица 8.1 *(окончание)*

SQL	dBase	Paradox
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 оператора 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 Employee.db
   (Code  AUTOINC,
    Name   CHAR(20),
```

```
    Birthday DATE,
    Salary  NUMERIC(10,2),
    PRIMARY KEY(Code));
```

Новая таблица `Employee` имеет формат Paradox, и для нее определены автоинкрементное поле кода `Code`, символьные поля фамилии `Name`, поле даты `BirthDay`, а также числовое поле оклада `Salary`. По полю кода построен главный ключ.

Для удаления таблицы предназначена оператор:

```
DROP TABLE <Имя таблицы>;
```

Например:

```
DROP TABLE Table1.dbf;
```

В результате выполнения этого оператора с диска удаляются все файлы, относящиеся к таблице с именем `Table1`. 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 операторе `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 Employee.db
      ADD  Section SMALLINT,
      ADD  Note  CHAR(30),
      DROP Post;
```

К таблице `Employee` добавляются целочисленное поле номера отдела `Section` и символьное поле примечаний `Note`, поле `Post` 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 оператора `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 оператора `CREATE INDEX` является единственным способом определения индекса для таблиц dBase.

Так можно создать индекс по одному полю:

```
CREATE INDEX
   indName ON Employee.db (Name)
```

А так по двум:

```
CREATE INDEX
   indNamePosition ON Employee.db (Name, Position)
```

Для удаления индекса используется оператор `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 оператора `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 операторе

```
DROP INDEX
   "Employee.db".indNamePost
```

из таблицы Employee удаляется индекс indNamePost, созданный по полям Name и Post.

Первичный ключ удаляется так:

```
DROP INDEX
    "Employee.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 оператора SELECT.

Описание оператора *SELECT*

Оператор SELECT является важнейшим оператором языка SQL. Он предназначен 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  <Вложенный оператор SELECT>]
```

Результат выполнения SQL-запроса, заданного инструкцией SELECT, представляет собой выборку записей, отвечающих заданным условиям. При рассмотрении оператора SELECT будем предполагать, что SQL-запрос набран и выполнен с помощью компонента Query. При этом результатом выполнения запроса является соответствующий этому компоненту набор данных.

В полученном 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 за-

дано 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 операторов используется операнд `UNION`, в котором располагается вложенный оператор `SELECT`, называемый также *подзапросом*. Результирующий набор данных представляют записи, отобранные с учетом выполнения условий отбора, заданных операндами `WHERE` обоих операторов.

Оператор `SELECT` используется также внутри других операторов, например, операторов 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 ранее, при отборе всех записей таблицы условия отбора 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 Employee.db
```

В результате выполнения этого запроса из таблицы `Employee` в набор данных попадают все поля и все записи, и набор данных имеет вид:

```
Code  Name      Post      Salary
1   Иванов Р.О.  Директор   6700
2   Сидоров С.С. Зам. дир.   5200
3   Семенова И.И. Бухгалтер   5200
4   Кузнецов П.А. Зав. лаб.   3600
5   Попов А.Л.  Инженер    2400
6   Васин Н.Е.  Мл. нс    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, Post
   FROM Employee.db
```

В набор данных, формируемый в результате выполнения этого SQL-запроса, включаются поля `Name` и `Post` всех записей из таблицы `Employee`. Набор данных получает вид:

```
Name      Salary
Иванов Р.О.  Директор
Сидоров С.С. Зам. дир.
Семенова И.И. Бухгалтер
Кузнецов П.А. Зав. лаб.
Попов А.Л.  Инженер
Васин Н.Е. Мл. нс
```

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 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 настройкой SQL-запроса при разработке или при выполнении приложения.

В набор данных, кроме физических полей таблиц, можно включать также *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 Employee;
```

Для сотрудников организации (таблица `Employee`) 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 Post
   FROM Employee
```

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 `Post`):

```
Post
бухгалтер
вед. нс
директор
зав. лаб.
зам. дир.
мл. нс
нс
```

А так задается выборка записей с уникальными значениями двух полей:

```
SELECT DISTINCT Post, Degree
   FROM Employee
```

Сформированный набор данных содержит список занимаемых штатных должностей и окладов. В набор данных попадают записи с уникальной комбинацией значений двух полей (`Post` и `Degree`):

```
Post     Degree
Бухгалтер   -
вед. нс    дтн
директор   дтн
зав. лаб.   дтн
зам. дир.   ктн
мл. нс     -
мл. нс    ктн
нс      ктн
```

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 Employee, Info
```

или

```
SELECT Employee.*, Info.* FROM Employee, Info
```

Результирующий набор данных состоит из всех полей и всех записей таблиц `Employee` 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 Employee.Name, Info.I_Code
   FROM Employee, 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, `" Employee.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 " Employee.db" As E, "Info.db" As I
```

Для таблицы `Employee` определен псевдоним `E`, 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 операторе 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 Employee, Info
```

Имена полей `Name` и `I_Code` уникальны для таблиц `Employee` и `Info`, поэтому имена таблиц в обозначении полей можно опустить.

Простое условие отбора 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 далее.

- 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, пусть таблица с данными о сотрудниках имеет вид, приведенный в табл. 8.2.

Таблица 8.2. *Данные о сотрудниках*

Name	Post	Birthday	Degree	Phone
Антонов И. И.	директор	11.05.1944	дтн	990-42-90
Ветров В. В.	вед. нс	03.03.1958	дтн	123-55-66
Кулешов К. К.	нс	06.06.1966	ктн	555-33-44
Сергеев С. С.	бухгалтер	03.05.1966	—	111-11-11
Сидоров А. А.	зам. дир.	12.07.1948	ктн	777-12-88
Столяров И. И.	зав. лаб.	03.02.1951	дтн	123-33-44
Федоров Я. Я.	мл. нс	13.06.1976	—	123-45-67
Якушев Я. Я.	мл. нс	12.02.1975	ктн	123-45-67

Тогда оператор

```
SELECT Name
   FROM Employee
   WHERE degree = 'ктн';
```

задает получение следующего списка сотрудников, имеющих ученую степень кандидата технических наук:

```
Name
Кулешов К. К.
Сидоров А. А.
Якушев Я. Я.
```

Пример отбора записей по значениям символьного поля:

```
SELECT Name
   FROM Employee
   WHERE Post = 'Мл. нс';
```

В результате получим выборку с фамилиями младших научных сотрудников:

```
Name
Федоров Ф. Ф.
Якушев Я. Я.
```

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 должности `Post`.

В последнем примере в операции сравнения учитывается регистр символов, и слова `Мл. нс` и `мл. нс` 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 LOWER(TRIM(Post)) = 'Мл. нс'
```

Функция `TRIM` удаляет из строкового значения ведущие и конечные пробелы, а функция `LOWER` приводит символы полученной строки к нижнему регистру. В результате значение должности `водитель` независимо от наличия ведущих и конечных пробелов, а также регистра букв будет приведено к значению `мл. нс`.

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 Employee
   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 Employee
   WHERE Name LIKE "%" || "ро" || "%"
```

В приведенном запросе происходит отбор всех сотрудников, в фамилии которых входят символы `ро`. Набор данных, полученный при таком отборе, может иметь вид:

```
Name
Ветров В. В.
Сидоров А. А.
Столяров С. С.
Федоров Ф. Ф.
```

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 Employee
   WHERE Degree IS NULL
```

выполняется отбор всех полей записей таблицы сотрудников (`Employee`), для которых не определена ученая степень (`Degree`).

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, Degree
   FROM Employee
   WHERE LOWER(Post) IN ("нс", "мл. нс")
```

В результате его выполнения получим выборку фамилий и ученых степеней всех научных и младших научных сотрудников:

```
Name              Degree
Кулешов К. К.     ктн
Федоров Ф. Ф.     -
Якушев Я. Я.      ктн
```

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 Name, Post, Birthday
   FROM Employee
   WHERE Birthday BETWEEN "1.7.1950" AND "31.7.1967"
```

В результате его выполнения получим набор записей о сотрудниках, у которых дата рождения (поле Birthday) находится в диапазоне с 1 июля 1950 г. по 31 июля 1967 г.:

```
Name              Post        Birthday
Ветров В. В.      вед. нс     03.03.1958
Кулешов К. К.     нс          03.03.1966
Сергеев С. С.     бухгалтер   03.05.1966
Столяров И. И.    зав. лаб.   03.12.1951
```

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 (1.7.05), так и четырьмя (1.7.2005) 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 Post, Count (Post) Birthday
   FROM Employee
   WHERE Birthday BETWEEN "1.7.1950" AND "31.7.1967"
   AND Degree = 'ктн'
```

В результате его выполнения получим набор записей о сотрудниках с ученой степенью кандидата технических наук, у которых дата рождения (поле Birthday) находится в диапазоне с 1 июля 1950 г. по 31 июля 1967 г.:

```
Name              Post        Birthday
Кулешов К. К.     нс          03.03.1966
```

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 Post, COUNT(Post)
   FROM Employee
   WHERE Birthday BETWEEN "1.7.1950" AND "31.7.1967"
   GROUP BY Post
```

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 будет иметь следующий вид:

```
Post          COUNT(Post)
бухгалтер     1
вед. нс       1
зав. лаб.     1
мл. нс        2
нс            1
```

Совместно с операндом GROUP BY можно использовать операнд HAVING, с помощью которого задаются дополнительные условия группирования записей.

Рассмотрим пример запроса:

```
SELECT Post, COUNT(Post)
   FROM Employee
   WHERE Birthday BETWEEN "1.7.1950" AND "31.7.1967"
   GROUP BY Post
   HAVING COUNT(Post) > 1
```

Полученный набор данных будет иметь следующий вид:

```
Post          COUNT(Post)
мл. нс        2
```

Здесь отбираются данные для должностей, занимаемых сотрудниками с датами рождения, находящимися в заданном диапазоне, причем на этих должностях находится более одного сотрудника.

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 Employee.db
   ORDER BY Post
```

Сортировка записей задана по полю `Post`. Полученный набор данных будет иметь вид:

Name	Post	Birthday	Degree	Phone
Сергеев С. С.	бухгалтер	03.05.1966	—	111-11-11
Ветров В. В.	вед. нс	03.03.1958	дтн	123-55-66
Антонов И. И.	директор	11.05.1944	дтн	990-42-90
Столяров И. И.	зав. лаб.	03.02.1951	дтн	123-33-44
Сидоров А. А.	зам. дир.	12.07.1948	ктн	777-12-88
Федоров Я. Я.	мл. нс	13.06.1976	—	123-45-67
Якушев Я. Я.	мл. нс	12.02.1975	ктн	123-45-67
Кулешов К. К.	нс	06.06.1966	ктн	555-33-44

Еще один пример запроса на сортировку — на этот раз по двум полям:

```
SELECT Name, Post, Degree
   FROM Employee.db
   ORDER BY Post, Degree DESC
```

В набор данных входят поля `Name`, `Post` и `Degree` всех записей. Записи отсортированы по полям `Post` и `Degree`, при этом по полю `Degree` упорядочивание выполняется в порядке убывания значений. Полученный набор данных будет таким:

Name	Post	Birthday	Degree	Phone
Сергеев С. С.	бухгалтер	03.05.1966	—	111-11-11
Ветров В. В.	вед. нс	03.03.1958	дтн	123-55-66
Антонов И. И.	директор	11.05.1944	дтн	990-42-90
Столяров И. И.	зав. лаб.	03.02.1951	дтн	123-33-44
Сидоров А. А.	зам. дир.	12.07.1948	ктн	777-12-88
Якушев Я. Я.	мл. нс	12.02.1975	ктн	123-45-67
Федоров Я. Я.	мл. нс	13.06.1976	—	123-45-67
Кулешов К. К.	нс	06.06.1966	ктн	555-33-44

Если по полям `Post` и `Degree` построен индекс, то операции с набором данных будут выполняться быстрее.

При разработке приложения управление сортировкой осуществляется посредством различных элементов формы, например, кнопок и переключателей.

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 Employee.db P, Advanced.db A
```

Таблицы `Employee` 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  Birthday   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 кото-

рых 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 операторы 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 оператора 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 Employee
   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 операторе 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 одном операторе 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 оператора INSERT, который позволяет добавлять к таблицам одну или несколько записей. При добавлении одной записи оператор INSERT имеет формат:

```
INSERT INTO <Имя таблицы>
   [(<Список полей>)]
   VALUES (<Список значений>);
```

В результате выполнения этого оператора 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 оператор INSERT имеет формат:

```
INSERT INTO <Имя таблицы>
    (<Список полей>)
   Оператор SELECT;
```

В данном случае значения полей новых записей определяются через значения полей записей, отобранных с помощью оператора SELECT. Число добавленных записей равно числу отобранных записей. Список значений полей, возвращаемых инструкцией SELECT, должен соответствовать списку оператора 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 2005 г., 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 оператора `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 оператор `DELETE`, имеющий формат:

```
DELETE FROM <Имя таблицы>
    [WHERE <Условия отбора>];
```

В результате выполнения этого оператора 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 пример процедуры, в которой осуществляется формирование динамического запроса. Пользователь управляет сортировкой с помощью двух групп переключателей: в первой задается вид, во второй — направление сортировки. Сортировка выполняется после нажатия кнопки **Сортировать** (`Button1`). На рис. 8.1 показан вид формы при выполнении приложения.

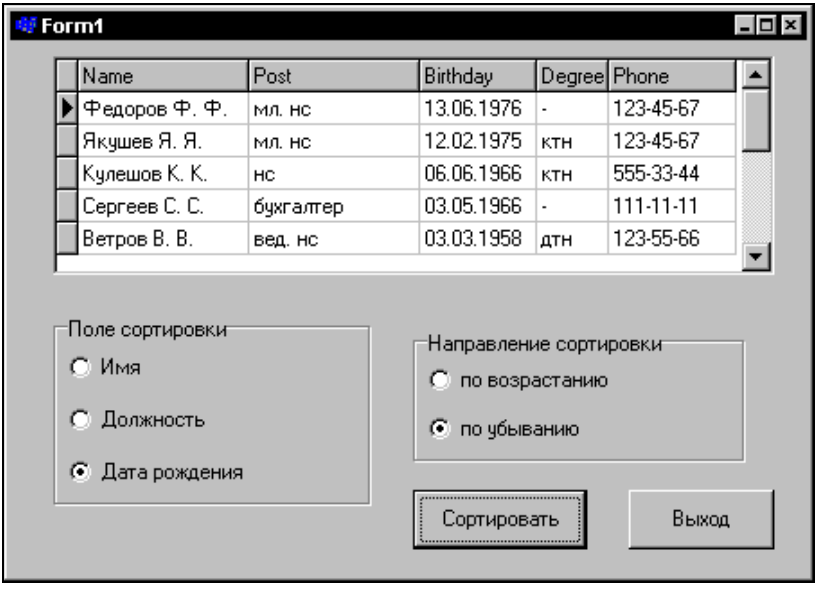


Рис. 8.1. Диалоговое окно сортировки набора данных

Ниже приводится код обработчика события нажатия кнопки `Button1`.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{  AnsiString d;
switch (RadioGroup2->ItemIndex)
 { case 0: d = ""; break;
  case 1: d = "DESC";
  }
  Query1->Close();
  Query1->SQL->Clear();
  Query1->SQL->Add("SELECT * FROM SEmployee.db");
 switch (RadioGroup1->ItemIndex)
 { case 0: d = "ORDER BY Name " + d; break;
  case 1: d = "ORDER BY Post " + d; break;
  case 2: d = "ORDER BY Birthday " +d;
 }
 Query1->SQL->Add(d);
 Query1->Active= true;
}
```

При нажатии кнопки `Button1` 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.

Запросы с параметрами

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 и должности сотрудников с датой рождения более поздней, чем указанная в редакторе `Edit1`, то организовать формирование и выполнение динамического запроса можно так:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
Query1->Close();
  Query1->SQL->Clear();
  Query1->SQL->Add("SELECT Name, Post FROM SEmployee.db ");
Query1->SQL->Add(" WHERE Birthday > " + Edit1->Text);
Query1->Open();
}
```

Здесь дата рождения в редакторе `Edit1` должна быть указана в одиночных кавычках.

С помощью параметров эту задачу можно решить проще, например, через включение в текст запроса параметра `prmSalary`:

```
SELECT Name, Post, Birthday
FROM SEmployee.db
WHERE Birthday >= :Birthday
```

Система C++ Builder 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 или обратиться по имени параметра. 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 целесообразно указать в свойстве `DataType` типа `TFieldType`.

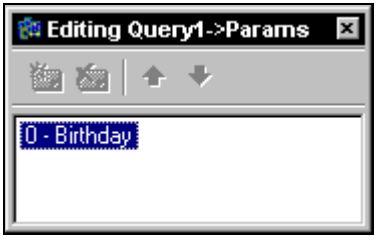


Рис. 8.2. Окно Редактора параметров

В последующем перед выполнением запроса вместо параметра необходимо подставить его значение, в данном случае из редактора `Edit1`. Далее приведен код обработчика события нажатия кнопки `Button1`, выполняющий указанное действие.

```
void __fastcall TForm1::Button2Click(TObject *Sender)

{

 Query1->Close();

 //Query1->Params->Items[0]->AsDate = Edit1->Text; - доступ к параметру
по индексу

 Query1->ParamByName("Birthday")->AsDate = (Edit1->Text);

 Query1->Open();

}
```

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 использовать такой подготовленный к выполнению запрос. Это позволяет ускорить обработку статических запросов, в том числе имеющих параметры. Например:

```
if(!Query1->Prepared)

{ Query1->Close();

 Query1->Prepare();

 Query1->Open();

}
```

Если текст подготовленного к выполнению запроса изменился (к изменению значений параметров это не относится), то автоматически вызывается метод `UnPrepare`, и свойству `Prepared` устанавливается значение `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).

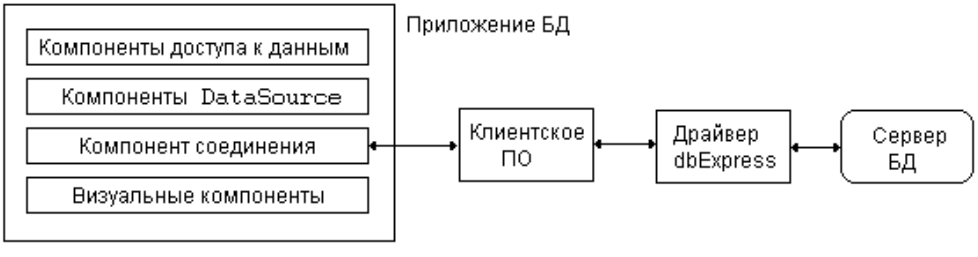


Рис. 9.1. Схема доступа к данным с помощью dbExpress

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 C++ Builder размещаются в папке ..\Borland\CBuilder6\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 C++ Builder находятся компоненты, используемые в технологии dbExpress: `SQLConnection` (`Database`), `SQLDataSet`, `SQLQuery` (`Query`), `SQLStoredProc` (`StoredProc`), `SQLTable` (`Table`), `SQLMonitor` (утилита SQL Monitor) и `SQLClientDataSet` (`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 механизма BDE не имеет аналогов для технологии dbExpress.

Определенным недостатком технологии dbExpress является то, что несколько 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 `SQLClientDataSet` большинство 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. Для наглядности приведем фрагмент кода из этого файла:

```
[OracleConnection]
DriverName=Oracle
DataBase=Database Name
User_Name=user
Password=password
RowsetSize=20
BlobSize=-1
ErrorResourceFile=
LocaleCode=0000
Oracle TransIsolation=ReadCommited
OS Authentication=False
Multiple Transaction=False
Trim Char=false
```

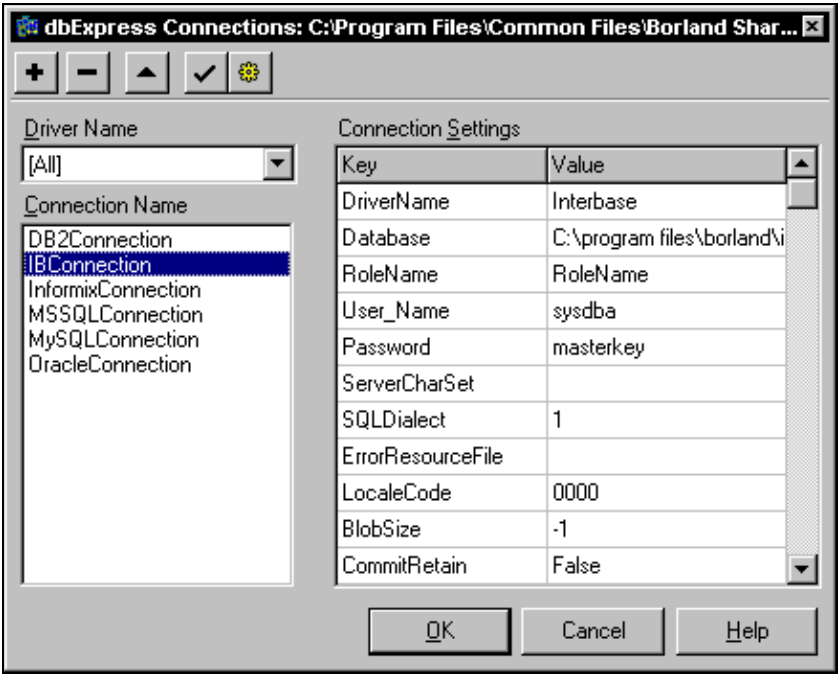


Рис. 9.2. Диалоговое окно Редактора соединений

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.

Редактор соединений позволяет настроить существующее соединение dbExpress или создать новое соединение, а также проверить правильность настроек с помощью кнопки **Test Connection**. Параметры соединения также можно настраивать 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 основных серверов БД*

Сервер БД	Параметр	Значение
Все серверы	`BlobSize`	Ограничение на объем пакета для данных типа BLOB
	`DriverName`	Имя драйвера
	`ErrorResource-File`	Файл сообщений об ошибках
	`LocaleCode`	Код локализации, определяющий влияние национальных символов на сортировку данных
	`User_Name`	Имя пользователя
	`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)

Таблица 9.2 *(окончание)*

Сервер БД	Параметр	Значение
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 `bool` 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:

```
void __fastcall Open(void);
void __fastcall Close(void);
```

С помощью свойства `LoginPrompt` типа `bool` 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.

Замечание

При установлении соединения с сервером `InterBase` в качестве имени пользователя и пароля можно использовать строки `sysdba` и `masterkey`.

При открытии и закрытии соединения можно воспользоваться любым из доступных событий, например:

```
__property Classes::TNotifyEvent AfterConnect = {read=FAfterConnect,
write=FAfterConnect};
__property Classes::TNotifyEvent AfterDisconnect =
{read=FAfterDisconnect, write
=FAfterDisconnect};
__property Classes::TNotifyEvent BeforeConnect = {read=FBeforeConnect,
write
=FBeforeConnect};
__property Classes::TNotifyEvent BeforeDisconnect =
{read=FBeforeDisconnect, write
=FBeforeDisconnect};
property OnLogin: TSQLConnectionLoginEvent;
```

Например, событие `BeforeConnect` можно использовать для вызова функции, выполняющей дешифрование пароля доступа к базе данных. Соответствующий обработчик события может быть задан следующим образом:

```
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
```

```
      Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

Отметим, что событие `OnLogin` 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.

Узнать о *текущем состоянии соединения* позволяет свойство `__property TConnectionState ConnectionState = {read=FConnectionState, write=FConnectionState};`, тип которого описан так:

```
enum TConnectionState {csStateClosed, csStateOpen, csStateConnecting, csStateExecuting, csStateFetching, csStateDisconnecting};
```

Указанные в определении типа значения соответствуют следующим состояниям соединения:

- `csStateClosed` — закрыто;
- `csStateOpen` — открыто;
- `csStateConnecting` — установление соединения;
- `csStateExecuting` — ожидание исполнения переданного SQL-запроса;
- `csStateFetching` — получение данных с сервера;
- `csStateDisconnecting` — завершение соединения.

Компонент `SQLConnection` позволяет выполнять SQL-запросы с помощью следующих функций.

Функция `int __fastcall Execute(const AnsiString SQL, TParams Params, void *ResultSet = NULL);` выполняет запрос, определенный значением константы `SQL`, с параметрами `Params`, используемыми в SQL-запросе. Используемые в запросе параметры должны иметь тот же порядок следования, что в `Params` и в инструкции SQL. Если SQL-запрос не содержит параметров, то свойству `Params` требуется задать значение `NULL`. 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 возвращения записей рекомендуется использовать функцию `int __fastcall Execute(const AnsiString SQL, void *ResultSet);`, 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, некоторые компоненты доступа к данным по технологии dbExpress (`SQLDataSet`, `SQLQuery`, `SQLStoredProc` и `SQLTable`) являются однонаправленными наборами 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, так как 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 и перехода на последнюю запись. Остальные компоненты можно использовать, как обычно.

Для компонента `SQLClientDataSet` 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 SQL-запроса – **CommandText Editor** (рис. 9.3). В окне редактора 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 формируемый SQL-запрос. После нажатия **OK** в окне редактора мы получаем нужное значение свойства `CommandText`.

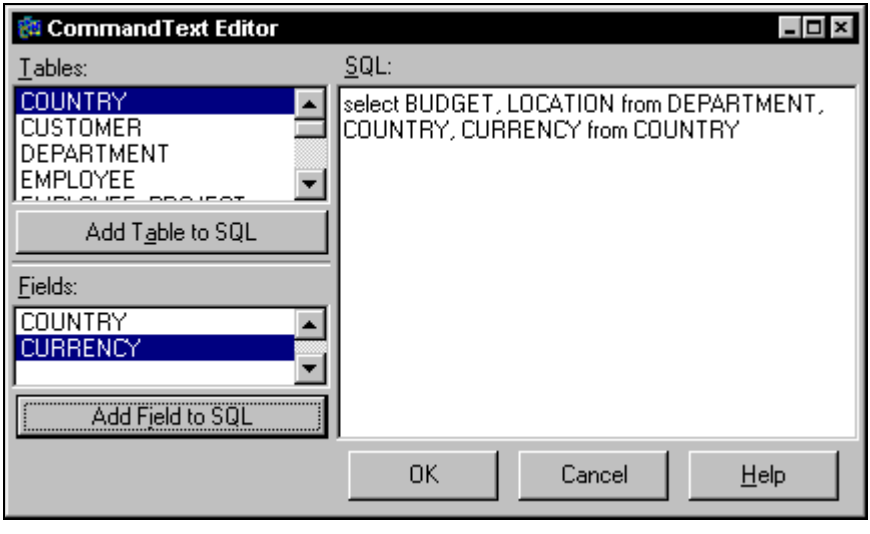


Рис. 9.3. Редактор построения SQL-запроса

При выполнении приложения аналогичные действия для выполнения SQL-запроса можно задать так:

```
SQLDataSet1->CommandType = ctQuery;
```

```
SQLDataSet1->CommandText = "select COUNTRY, CURRENCY from DEPARTMENT,
DEPT_NO,
                           DEPARTMENT from COUNTRY";
SQLDataSet1->ExecSQL();
```

При выборе значения `ctTable` в свойстве `CommandText` указывается 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

```
virtual int __fastcall ExecSQL(bool ExecDirect = false);
```

Параметр `ExecDirect` определяет необходимость подготовки параметров перед выполнением инструкции. Если параметры запроса или процедуры существуют, параметр `ExecDirect` должен иметь значение `false`. Это означает, что их нужно подготовить.

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. главы 8, 14)*. 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 7)*.

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.

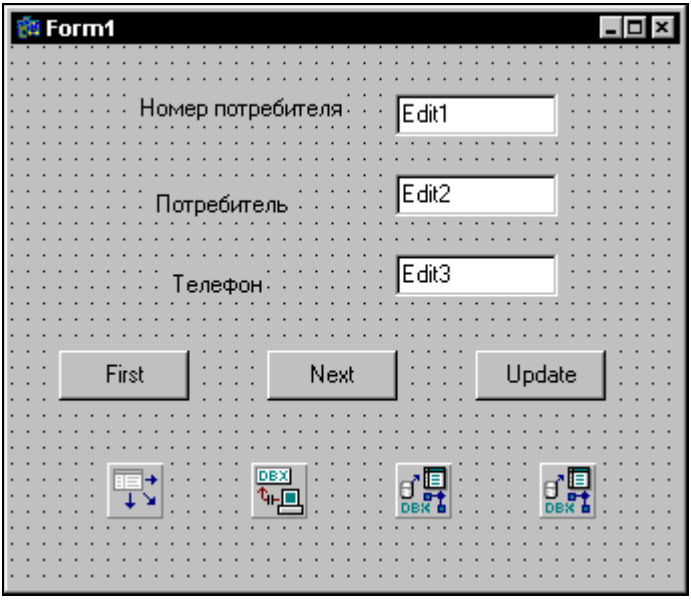


Рис. 9.4. Вид формы при разработке

Код главного модуля приложения для решения поставленной задачи имеет вид:

```
#pragma hdrstop
#include "Unit1SQLDataSet.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
SQLDataSet1->First();
}
```

```
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
SQLDataSet1->Next();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
SQLDataSet1->Active=true;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
SQLDataSet1->Active=false;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
 try
  {
    SQLDataSet2->CommandText = "UPDATE CUSTOMER SET CUST_NO =
:Cust_no,
       CUSTOMER = :Customer, PHONE_NO = :Phone_no WHERE CUST_NO =
:Cust_no";
    SQLDataSet2->Params->Items[0]->AsString = Edit1->Text;
    SQLDataSet2->Params->Items[1]->AsString = Edit2->Text;
    SQLDataSet2->Params->Items[2]->AsString = Edit3->Text;
    SQLDataSet2->ExecSQL();
  }
 catch (...)
  {MessageDlg("Ошибка!", mtError, TMsgDlgButtons() << mbOK, 0);}
}
//---------------------------------------------------------------------------
void __fastcall TForm1::SQLDataSet1AfterScroll(TDataSet *DataSet)
{
  Edit1->Text=SQLDataSet1->FieldByName("CUST_NO")->AsString;
  Edit2->Text=SQLDataSet1->FieldByName("CUSTOMER")->AsString;
  Edit3->Text=SQLDataSet1->FieldByName("PHONE_NO")->AsString;
}
```

Код заголовочного файла приложения имеет вид:

```
#ifndef Unit1SQLDataSetH
#define Unit1SQLDataSetH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <DB.hpp>
#include <DBXpress.hpp>
#include <FMTBcd.hpp>
#include <SqlExpr.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TSQLConnection *SQLConnection1;
        TSQLDataSet *SQLDataSet1;
        TDataSource *DataSource1;
        TSQLDataSet *SQLDataSet2;
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TLabel *Label1;
        TLabel *Label2;
        TLabel *Label3;
        TEdit *Edit1;
        TEdit *Edit2;
        TEdit *Edit3;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall SQLDataSet1AfterScroll(TDataSet *DataSet);
private:      // User declarations
public:              // User declarations
        __fastcall TForm1(TComponent* Owner);
```

```
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

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`. Именно с помощью этого компонента обеспечивается подсоединение к нужной нам базе данных employee.gdb. 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).

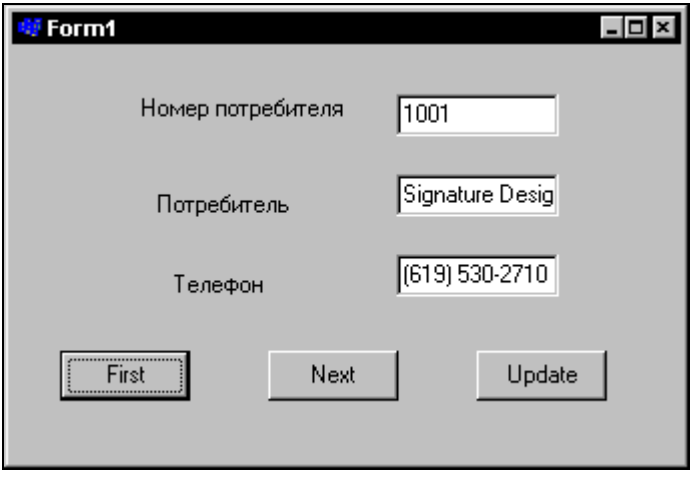


Рис. 9.5. Вид формы при выполнении приложения

Как видно из приведенного примера, компонент `SQLDataSet2` содержит SQL-запрос с параметрами *(см. главу 14)*. Как отмечалось, параметры задаются с помощью свойства `Params`, которое представляет коллекцию объектов типа `TParam`. Задание свойств каждого из параметров запроса выполняется с помощью Инспектора объектов и Редактора параметров SQL-запроса. Вид диалога настройки параметров запроса для компонента `SQLDataSet2` из рассматриваемого примера приведен на рис. 9.6.

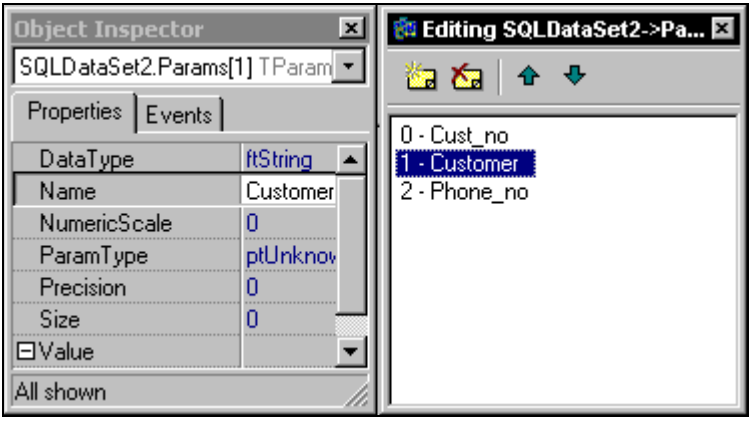


Рис. 9.6. Диалог настройки параметров SQL-запроса

Отметим, что необходимость использования компонента `SQLDataSet2` для фиксации внесенных изменений в таблице на сервере вызвана 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`. Он 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 `virtual void __fastcall PrepareStatement(void);` генерирует запрос на сервер БД.

Порядок следования данных в наборе определяется свойством IndexFieldNames типа AnsiString или IndexName типа AnsiString.

Список индексов таблицы можно получить с помощью метода void __fastcall GetIndexNames(Classes::TStrings* List); 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 редактора 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 (!(Edit3->GetTextLen()== 0))
        { SQLQuery1->SQL->Add("ORDER BY " + Edit3->Text); }
```

Свойство Text типа AnsiString 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 bool определяет, открыт набор данных или нет. Это свойство обычно используют, чтобы определить или установить заполнение набора данными.

Открыть набор данных SQL-запроса можно также с помощью метода Open либо с помощью метода virtual int __fastcall ExecSQL(bool ExecDirect

= false);. Напомним, что значение true 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.

Для примера приведем две процедуры обработки событий, в которых компонент SQLQuery1 используется для выполнения динамического SQL-запроса по отбору записей из таблицы CUSTOMER с целью просмотра содержимого полей 'CUST_NO' и 'PHONE_NO'. Как и в предыдущем примере, речь идет о таблице, принадлежащей базе данных employee.gdb сервера InterBase.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Edit1->Text="CUST_NO, PHONE_NO";
Edit2->Text="CUSTOMER";
Edit3->Text="CUST_NO";
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{  try
    {
      SQLQuery1->Close();
      SQLQuery1->SQL->Clear();
      SQLQuery1->SQL->Add("SELECT  "  +  Edit1->Text  +  "  FROM  "  +  Edit2-
>Text);
      if (!(Edit3->GetTextLen()== 0))
        { SQLQuery1->SQL->Add("ORDER BY " + Edit3->Text); }
      SQLQuery1->Open();
    }
   catch (...)
       {MessageDlg("Ошибка!", mtError, TMsgDlgButtons() << mbOK, 0);}
        Edit4->Text=SQLQuery1->FieldByName("CUST_NO")->AsString;
        Edit5->Text=SQLQuery1->FieldByName("PHONE_NO")->AsString;
}
```

При обработке события Button1Click нажатия кнопки с заголовком **Задать имена** в компоненты Edit1, Edit2 и Edit3 заносятся имена отбираемых полей, имя таблицы и имя поля для сортировки. При обработке события Button2Click нажатия кнопки с заголовком **Выполнить запрос** указанные име-

на используются при формировании и выполнении SQL-запроса с помощью компонента `SQLQuery1`.

Вид формы соответствующего приложения приведен на рис. 9.7.

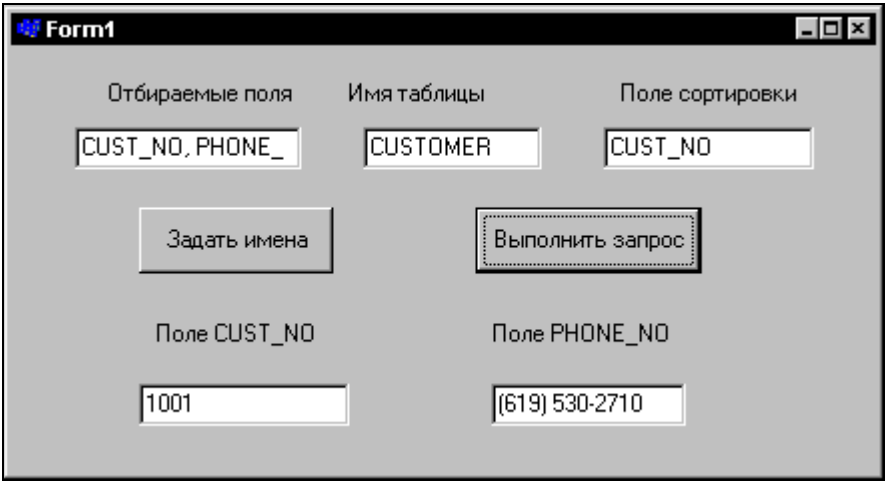


Рис. 9.7. Вид формы приложения при выполнении

Для просмотра значений отобранных полей с помощью SQL-запроса используются компоненты `Edit4` и `Edit5`. Приведенный вариант кода обеспечивает просмотр полей первой записи набора данных. При необходимости перемещения по записям при просмотре можно воспользоваться методами `First` и `Next`, аналогично тому, как это сделано в предыдущем примере.

Выполнение хранимых процедур

Для выполнения хранимых процедур, размещенных на сервере БД, служит компонент `SQLStoredProc`.

Имя хранимой процедуры задает свойство `StoredProcName` типа `AnsiString`. 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 некоторыми серверами порядок следования параметров до и после выполнения процедуры может меняться.

Для подготовки хранимой процедуры к выполнению на сервере служит метод `TCustomSQLDataSet __fastcall PrepareStatement(int &RecordsAffected);`. При его вызове сервером БД выделяются ресурсы и связываются их парамет-

ры. Поименованные параметры временно преобразуются к непоименованным, поскольку dbExpress поименованные параметры не поддерживает.

Если хранимая процедура не возвращает набор данных, то ее запускают с помощью метода `int __fastcall ExecProc(void);` .

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 `SQLClientDataSet`. Этот компонент использует двунаправленный курсор и позволяет редактировать данные, но в режиме редактирования. Тем самым он исправляет основные недостатки рассматриваемой технологии.

Для подготовки компонента `SQLClientDataSet` 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

```
virtual int __fastcall ApplyUpdates(int MaxErrors);
```

Здесь параметр `MaxErrors` определяет число ошибок, допустимых при передаче данных.

Локальный кэш компонента после сохранения изменений на сервере можно очистить от данных с помощью метода

```
bool __fastcall Reconcile(const System::OleVariant &Results);
```

Отмена локальных изменений данных может быть выполнена с помощью метода `void __fastcall CancelUpdates(void);`.

Пересылка данных между сервером и рассматриваемым компонентом осуществляется с помощью пакетов. Размер пакета (по числу записей) можно задать с помощью свойства `PacketRecords` типа `int`. 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.

```
int __fastcall GetNextPacket(void);
```

Если значение свойства `PacketRecords` равно `0`, то в пакете передаются только метаданные.

Свойство `DataSize` типа `int` устанавливает размер (в байтах) текущего пакета, доступного с помощью свойства `Data` типа `OleVariant`.

На общее число записей в источнике данных указывает свойство `RecordCount` типа `int`, номер текущей записи определяет свойство `RecNo` типа `int`.

Изменения в текущей записи можно отменить с помощью метода

```
void __fastcall RevertRecord(void);
```

Отменить последнюю операцию по изменению записи клиентского набора данных можно с помощью метода

```
bool __fastcall UndoLastChange(bool FollowChange);
```

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

```
void __fastcall RefreshRecord(void);.
```

Рассмотрим пример формы приложения БД, в котором с помощью компонента `SQLClientDataSet1` выполняется доступ к данным по технологии dbExpress. Пусть требуется обеспечить возможность просмотра и редактирования записей таблицы `DEPARTMENT`, принадлежащей базе данных employee.gdb сервера InterBase. Вид формы приложения на этапе разработки приведен на рис. 9.8.

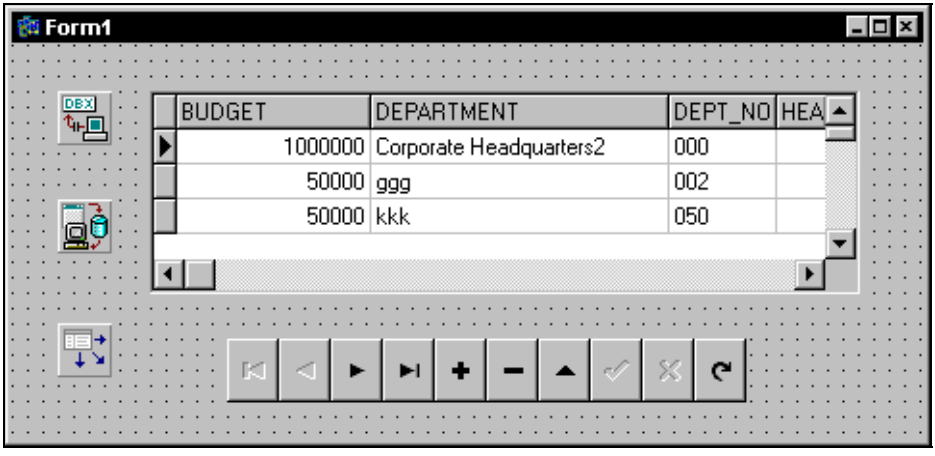


Рис. 9.8. Вид формы приложения на этапе разработки

Код главного модуля приложения для решения поставленной задачи имеет вид:

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1ClientDataSet.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::SQLClientDataSet1AfterPost(TDataSet *DataSet)
{
SQLClientDataSet1->ApplyUpdates(-1) ;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
SQLClientDataSet1->Open();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
SQLClientDataSet1->Close();
}
```

Код заголовочного файла приложения имеет вид:

```
#ifndef Unit1ClientDataSetH
#define Unit1ClientDataSetH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
```

```
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <DB.hpp>
#include <DBClient.hpp>
#include <DBCtrls.hpp>
#include <DBGrids.hpp>
#include <DBLocal.hpp>
#include <DBLocalS.hpp>
#include <DBXpress.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
#include <Provider.hpp>
#include <SqlExpr.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TSQLClientDataSet *SQLClientDataSet1;
        TSQLConnection *SQLConnection1;
        TDBGrid *DBGrid1;
        TDBNavigator *DBNavigator1;
        TDataSource *DataSource1;
        void __fastcall SQLClientDataSet1AfterPost(TDataSet *DataSet);
        void __fastcall FormCreate(TObject *Sender);
        void __fastcall FormDestroy(TObject *Sender);
private:       // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Компонент `SQLClientDataSet1` работает в табличном режиме. При разработке формы его свойству `CommandType` задано значение `ctTable`, с помощью свойства `CommandText` указано имя таблицы `DEPARTMENT`.

Для отображения записей редактируемого набора данных и навигации по нему используются компоненты `DBGrid1` и `DBNavigator1` соответственно. Для

установления связи названных компонентов с компонентом SQLClientDataSet1 используется компонент DataSource1. Для компонента DataSource1 его свойству DataSet установлено значение SQLClientDataSet1. Соответственно для каждого из компонентов DBGrid1 и DBNavigator их свойству DataSource задано значение DataSource1.

Фиксация внесенных при редактировании изменений и отправка данных на сервер осуществляется 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, тип которого описан так

```
typedef void __fastcall (__closure *TTraceEvent)(System::TObject* Sender,
pSQLTRACEDesc CBInfo, bool &LogTrace);
```

а сразу после записи в список возникает событие OnLogTrace типа

```
typedef void __fastcall (__closure *TTraceLogEvent)(System::TObject* Sender,
pSQLTRACEDesc CBInfo);
```

Содержимое списка можно сохранить в файле на жестком диске с помощью метода SaveToFile.

При задании свойству AutoSave типа bool значения true данные о проходящих командах автоматически заносятся в текстовый файл с именем, заданным значением свойства FileName типа AnsiString.

Свойство MaxTraceCount типа int 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 int.

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:

```
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender,
      pSQLTRACEDesc CBInfo)
{
  Memo1->Lines->Clear();
  Memo1->Lines=SQLMonitor1->TraceList;
  SQLMonitor1->TraceList->Clear(); // Очистка списка сообщений}
}
```

Как видно из приведенного кода, мы использовали обработчик события `OnLogin`.

В качестве возможного варианта сообщения, отслеживаемого при работе компонента монитора, возможна строка вида

```
INTERBASE - isc_dsql_free_statement
```

Как видно из приведенного текста, перехвачено сообщение от сервера INERBASE.

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 `SQLClientDataSet`. Обе библиотеки могут быть откомпилированы вместе с проектом и включены в exe-файл распространяемого приложения.

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 использовании технологии ADO (рис. 10.1) приложение БД может использовать данные из электронных таблиц, таблиц локальных и серверных баз данных, XML-файлов и т. д.

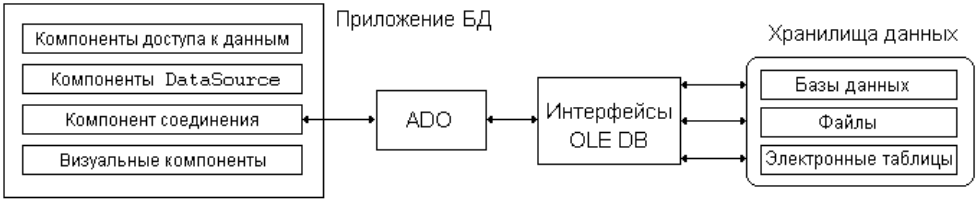


Рис. 10.1. Схема доступа к данным по технологии ADO

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, ин-

терпретацию 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 C++ Builder некоторые из этих объектов получили реализацию, и при необходимости мы будем рассматривать более подробно отдельные понятия.

В системе программирования C++ Builder 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, используются 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 компонент `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 путем установления соединения с источником данных напрямую. Для этого 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` также с помощью его свойства `ConnectionString` нужно предварительно 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 диалоговом окне (рис. 10.2), открываемом двойным щелчком в строке `ConnectionString` свойства соответствующего компонента доступа к данным в окне Инспектора объектов.

При установке переключателя **Use Data Link File** можно выбрать из списка или найти (после нажатия кнопки **Browse**) файл связи с данными с расшире-

нием 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.

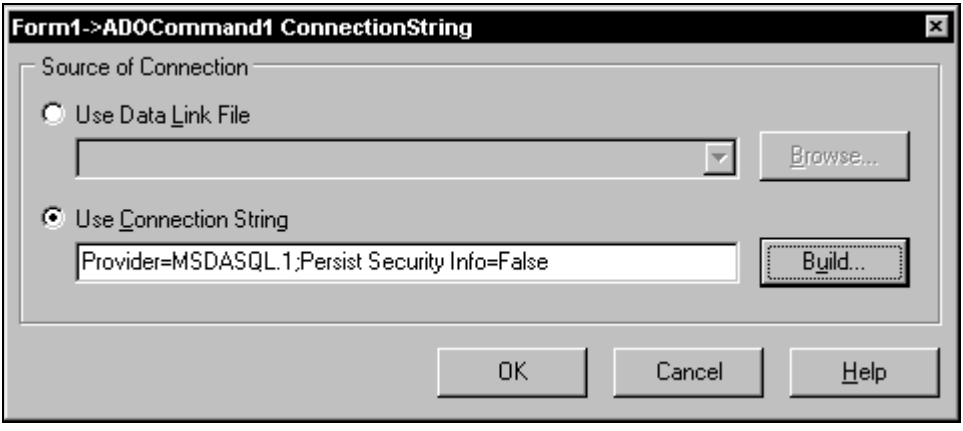


Рис. 10.2. Первое окно настройки строки соединения

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 четыре 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**.

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.

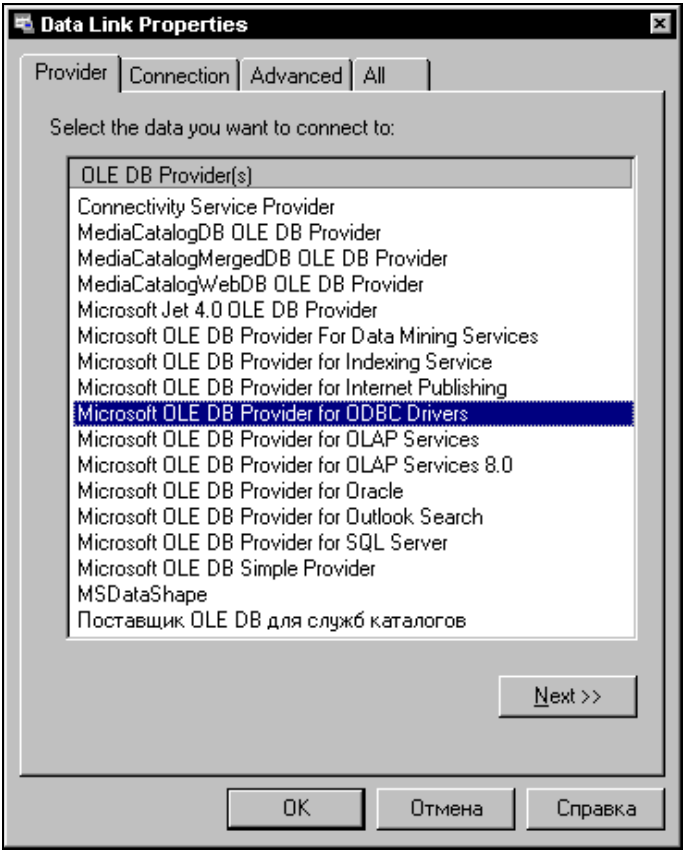


Рис. 10.3. Вкладка **Provider** окна настройки соединения

При нажатии на кнопку **Next** (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.

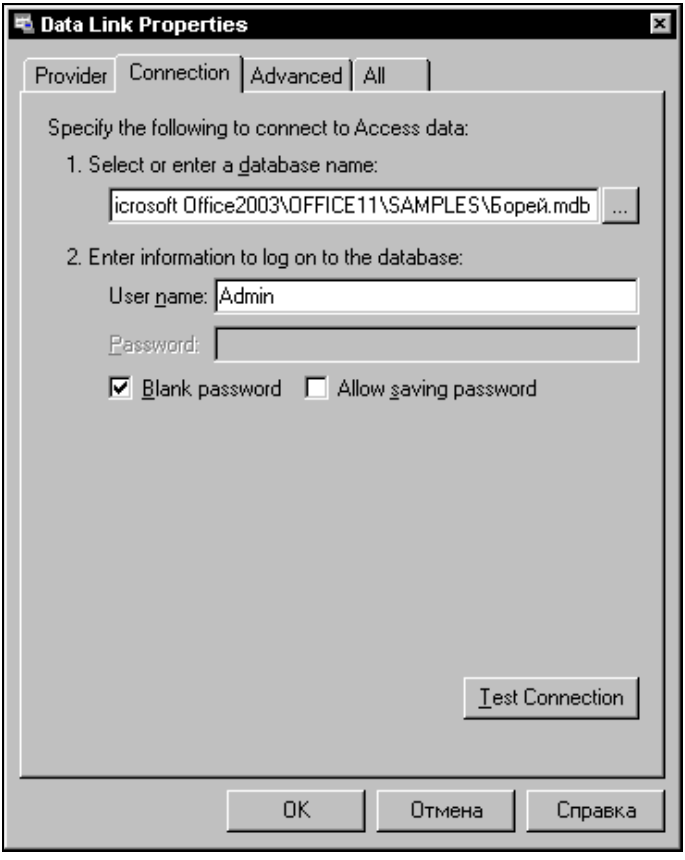


Рис. 10.4. Вкладка **Connection** окна настройки соединения

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 две другие 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.

При использовании компонента `ADOConnection` для активизации соединения после настройки достаточно установить свойству `Connected` этого компонента значение `true` или при выполнении приложения вызвать метод `Open`.

В случае использовании 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:

```
enum 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`, `UserID`, `Password` 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:

```
enum TEventStatus {esOK,esErrorsOccured, esCantDeny, esCancel, esUnwant-
edEvent};
```

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:

```
typedef void __fastcall (__closure *TConnectErrorEvent)(TADOConnection*
Connection, const _di_Error Error, TEventStatus &EventStatus);
```

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.

При разрыве соединения возникает событие `OnDisconnect` типа `TDisconnectEvent`, который описан так:

```
typedef void __fastcall (__closure *TDisconnectEvent)(TADOConnection*
Connection, TEventStatus &EventStatus);
```

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, так как 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`

Таблица 10.1 *(окончание)*

Компонент ADO	Компонент dbExpress	Компонент BDE
ADOConnection	SQLConnection	Database
ADODataSet	SQLDataSet, SQLSimpleDataSet	Table, Query, StoredProc
ADOCommand	-	-
RDSConnection	-	-

Стандартные компоненты доступа к данным в ADO (ADODataSet, ADOTable, ADOQuery, ADOStoredProc) 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:

```
enum TADOLockType { ltUnspecified, ltReadOnly, ltPessimistic, ltOptimis-
tic, 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. разд. "Управление соединением и транзакциями")*. При работе с клиентским курсором важную роль играет следующее свойство.

Передаваемые серверу данные определяет свойство MarshalOptions типа TMarshalOption , который описан так:

```
enum 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:

```
enum 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:

```
enum TRecordStatus { rsOK, rsNew, rsModified, rsDeleted, rsUnmodified,
rsInvalid, rsMultipleChanges, rsPendingChanges, rsCanceled, rsCantRe-
lease, rsConcurrencyViolation, rsIntegrityViolation, rsMaxChangesEx-
ceeded, rsObjectOpen, rsOutOfMemory, rsPermissionDenied, rsSchemaViola-
tion, rsDBDeleted };

typedef Set<TRecordStatus, rsOK, rsDBDeleted> TRecordStatusSet;
```

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 `bool`. 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.

При программировании действий по работе с хранилищем данных с помощью рассматриваемого компонента используются аналогичные средства, что 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 `bool` 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:

```
enum TCommandType { cmdUnknown, cmdText, cmdTable, cmdStoredProc,
cmdFile, cmdTableDirect };
```

Здесь:

- `cmdUnknown` — тип неизвестен;
- `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 C++ Builder. 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 C++ Builder эквивалентно 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 `TObjectStates`, который описан так:

```
enum TObjectState { stClosed, stOpen, stConnecting, stExecuting, stFetching };

typedef Set<TObjectState, stClosed, stFetching>  TObjectStates;
```

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:

```
enum TExecuteOption { eoAsyncExecute, eoAsyncFetch, eoAsyncFetchNonBlock-
ing, eoExecuteNoRecords };

typedef Set<TExecuteOption, eoAsyncExecute, eoExecuteNoRecords>
TExecuteOptions;
```

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:

- `_di__Recordset __fastcall Execute();` — такое объявление метода используется при выполнении команды без параметров;
- `_di__Recordset __fastcall Execute(const OleVariant &Parameters);` — в этом объявлении параметр `Parameters` определяет параметры команды;

- `_di__Recordset __fastcall Execute(int &RecordsAffected, const OleVariant &Parameters);` — здесь параметр `RecordsAffected` возвращает общее число обработанных запросом записей.

Отметим, что тип `_di__Recordset` описан так:

```
typedef System::DelphiInterface< _Recordset > _di__Recordset;
```

При выполнении команды ADO создается набор данных, метод Execute возвращает этот набор данных. Причем для доступа к этому набору данных нужно использовать компонент `ADODataSet`. Например:

```
ADODataSet1->Recordset = ADOCommand1->Execute();
```

Здесь возвращаемый при вызове метода `Execute` набор данных назначается свойству `Recordset` названого компонента.

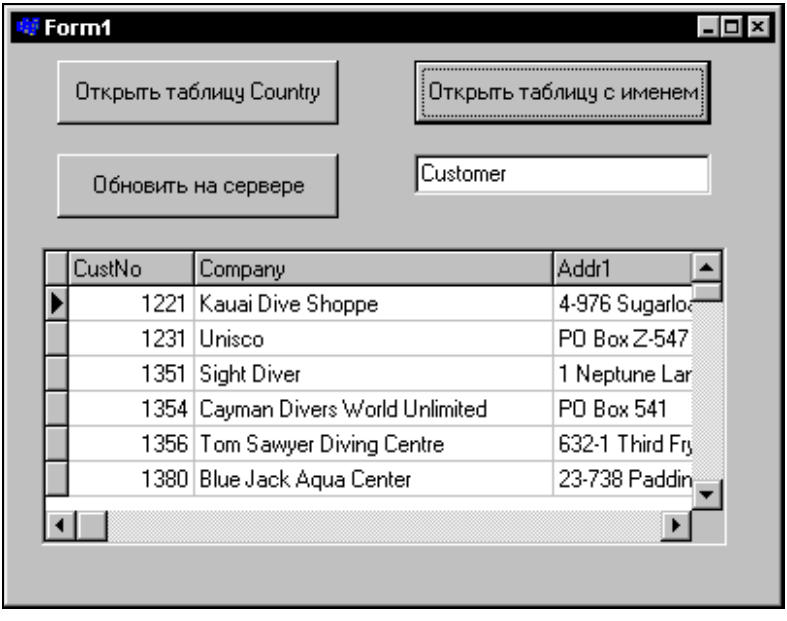


Рис. 10.5. Вид формы при выполнении приложения

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.

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:

```
#pragma hdrstop

#include "UnitCADO.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

void __fastcall TForm1::FormCreate(TObject *Sender)
{
  ADOConnection1->ConnectionString ="FILE NAME=" + DataLinkDir() +
"\\BCDEMOS.UDL";
  ADODataSet1->Open();
}
```

```
//---------------------------------------------------------------------------

void __fastcall TForm1::Open_CuntClick(TObject *Sender)
// Открытие табличного набора данных с именем "Country"
{
  ADODataSet1->Close();
  ADODataSet1->CommandType = cmdTable;
  ADODataSet1->CommandText = "Country";
  ADODataSet1->Open();

}
//---------------------------------------------------------------------------
void __fastcall TForm1::UpdateClick(TObject *Sender)
{
ADODataSet1->UpdateBatch();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Open_NamClick(TObject *Sender)
// Открытие табличного набора данных с заданным именем
{
  ADODataSet1->Close();
  ADODataSet1->CommandType = cmdTable;
  ADODataSet1->CommandText = Edit1->Text;
  try
     { ADODataSet1->Open();
     }
  catch(Exception &E)
       {
       MessageDlg("Исправьте имя таблицы",
       mtInformation,TMsgDlgButtons() << mbOK, 0);
       ADODataSet1->Close();
       }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCloseQuery(TObject *Sender, bool &CanClose)
```

```
// Обновление данных на сервере при закрытии формы
{
if (ADODataSet1->Active)
 {
   try
     { ADODataSet1->UpdateBatch();
     }
   catch(Exception &E)
     {
      if (MessageDlg("Данные не обновлены, выйти?", mtConfirmation,
TMsgDlgButtons() << mbOK << mbCancel,0) == mrCancel)
      CanClose = false;

     }
  }
}
//---------------------------------------------------------------------
```

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 **Открыть таблицу с именем**.

Обновление внесенных в открытую таблицу изменений происходит при нажатии на кнопку **Сохранить на сервере** или автоматически при закрытии формы приложения. Как видно из приведенного кода, для этого используется метод `UpdateBatch()` компонента `ADODataSet1`.

Глава 11

Создание и просмотр отчетов с помощью QuickReport

Отчет представляет собой печатный документ, содержащий такие же данные, как и полученные в результате выполнения запроса к БД. Можно выделить следующие виды отчетов:

- простой;
- с группированием данных;
- для таблиц, связанных отношением "главный-подчиненный";
- составной, объединяющий несколько разных отчетов.

В C++ Builder для создания отчетов служит генератор отчетов `QuickReport`, содержащий большой набор компонентов. В этой главе мы рассмотрим в целом средства создания отчета и приведем пример создания простого отчета.

Компоненты отчета

Компоненты, предназначенные для создания отчетов, находятся на странице **QReport** Палитры компонентов. Большинство компонентов отчета являются визуальными. Многие из них мало отличаются от аналогичных компонентов страниц **Standard**, **Additional** и **Data Controls**. Например, компоненту `QRImage` соответствуют компоненты `Image` и `DBImage`.

Компонент-отчет

Главным элементом отчета является компонент-отчет `QuickRep`, представляющий собой основу, на которой размещаются другие компоненты. Ком-

понент QuickRep обычно размещается на отдельной форме, предназначенной для создания отчета. По умолчанию он имеет имя QuickRep1. Если на форме размещается другой отчет (обычно так не делается), он получает имя QuickRep2 и т. д.

Компонент QuickRep при помещении его на форму имеет вид страницы формата А4, первоначально отображаемой в натуральную величину. При разработке приложения можно изменить масштаб страницы и размещенных на ней компонентов с помощью свойства Zoom типа int. Значение этого свойства устанавливается в процентах, по умолчанию 100%.

Отчет можно поместить на любую форму приложения, например, на главную. В этом случае отчет (страница) создает своего рода фон, на котором расположены управляющие элементы формы.

Компонент QuickRep связывается с набором данных Table или Query, для которого создается отчет, с помощью свойства DataSet. При этом набор данных Query может содержать записи, выбранные из разных таблиц. При печати отчета в процессе выполнения приложения набор данных должен быть открыт. Во время построения отчета можно использовать специально создаваемый набор данных и размещать его на форме, при этом источник данных DataSource не требуется. На практике компонент QuickRep обычно связывается с набором данных, записи которого отображаются на форме в визуальных компонентах. В этом случае в отчет попадают записи, удовлетворяющие, например, критерию фильтрации и/или сортировки, задаваемому пользователем.

Приведем пример связывания отчета с набором данных.

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
QuickRep1->DataSet=Form1->Table1;
}
```

Отчет QuickRep1, находящийся на форме Form2, при обработке события OnCreate этой формы связывается с набором данных Table1, расположенным на форме Form1. В файле реализации (исходного кода) модуля формы Form2 следует поместить директиву препроцессора #include "Unit1.h" для подключения заголовочного файла модуля формы Form1.

Отчет состоит из отдельных полос — составных частей отчета, которые определяют содержание и вид созданного документа. Полоса представляет собой элемент отчета. Каждый элемент размещается на своем месте и предназначен для отображения соответствующих компонентов отчета и вывода данных. Возможный вид отчета при разработке показан на рис. 11.1.

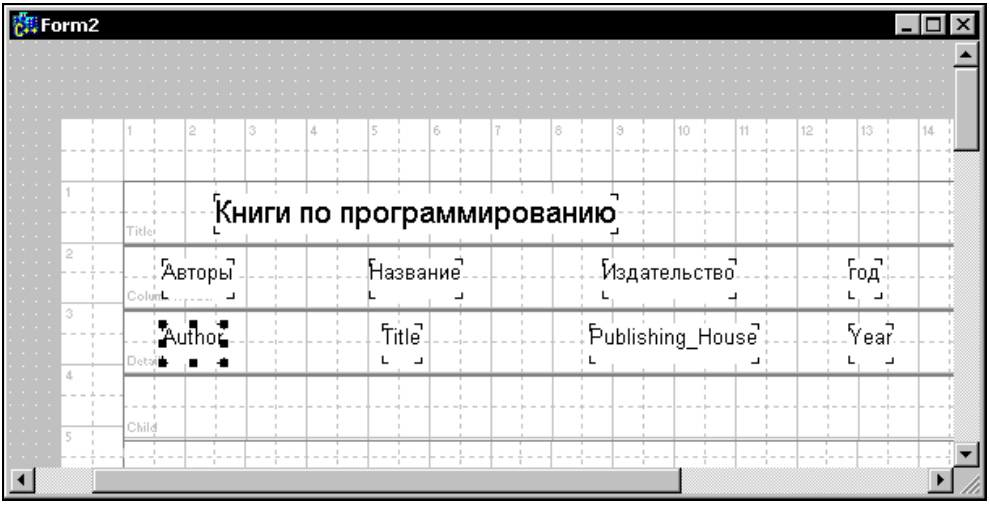


Рис. 11.1. Вид отчета при разработке

Конструирование отчета в основном аналогично конструированию формы. Управлять наличием полос в отчете можно с помощью свойства `Bands` множественного типа `TQuickRepBands`. При разработке приложения включение/отключение полосы выполняется установкой логического значения соответствующего подсвойства свойства `Bands`, например, для полосы области данных отчета этим подсвойством является `HasDetail`. С помощью этого свойства в простой отчет можно включать следующие полосы:

- `HasPageHeader` — верхний колонтитул;
- `HasTitle` — заголовок отчета;
- `HasColumnHeader` — заголовки столбцов;
- `HasDetail` — область данных;
- `HasSummary` — итог отчета;
- `HasPageFooter` — нижний колонтитул.

Перечисленные полосы можно также вставлять в отчет с помощью компонента полосы `QRBand`, при этом тип вставляемой полосы устанавливается через свойство `BandType` этого компонента.

Параметры страницы отчета определяются свойством `Page` типа `TQRPage`, через подсвойства которого можно настраивать следующие характеристики:

- `PaperSize` — формат страницы (по умолчанию A4);
- `Orientation` — ориентация страницы:
  - `poPortrait` — книжная (по умолчанию);
  - `poLandScape` — альбомная.

- `Length` и `Width` — высота и ширина страницы;
- `TopMargin`, `BottomMargin`, `LeftMargin` и `RightMargin` — размер верхнего, нижнего, левого и правого полей соответственно;
- `Ruler` — признак отображения сетки при разработке отчета, по умолчанию имеет значение `true`, при этом сетка видна.

Значения свойств `Length` и `Width` можно устанавливать в том случае, если свойство `PaperSize` имеет значение `Custom`, иначе высота и ширина страницы устанавливаются автоматически по формату и ориентации страницы, и разработчик не может их изменить.

Единицы измерения страниц, полос, полей, а также других элементов отчета определяет свойство `Unit` типа `TQRUnits`, принимающее следующие значения:

- `Inches` — дюймы;
- `MM` — миллиметры (по умолчанию);
- `Pixels` — пикселы;
- `Native` — десятые доли миллиметра;
- `Characters` — автоматическая настройка в соответствии с размером символов, установленным свойством `QuickRep->Font->Size`.

При необходимости разработчик может изменить параметры страницы, а также многие другие параметры отчета (например, шрифт по умолчанию) с помощью Инспектора объектов или в диалоговом окне **Report Setting** (Установки отчета) установки параметров отчета (рис. 11.2). Оно вызывается командой **Report Setting** контекстного меню страницы отчета или двойным щелчком на странице отчета.

Страница отчета может иметь рамку, параметры которой задает свойство `Frame` типа `TQRFrame`:

- `Color` — цвет (по умолчанию черный);
- `Width` — ширина в пикселах (по умолчанию 1);
- `Style` — стиль (по умолчанию сплошная линия);
- `DrawTop`, `DrawBottom`, `DrawLeft`, `DrawRight` — наличие линии сверху, снизу, слева и справа соответственно (по умолчанию все линии отсутствуют, и рамка не рисуется).

С помощью свойства `PrinterSetting` типа `TQuickRepPrinterSettings` устанавливаются параметры *принтера*:

- `FirstPage` и `LastPage` — номер первой и последней печатаемой страницы;
- `Copeis` — число копий;

- `Duplex` — включение режима двусторонней печати;
- `OutputBin` — способ подачи бумаги.

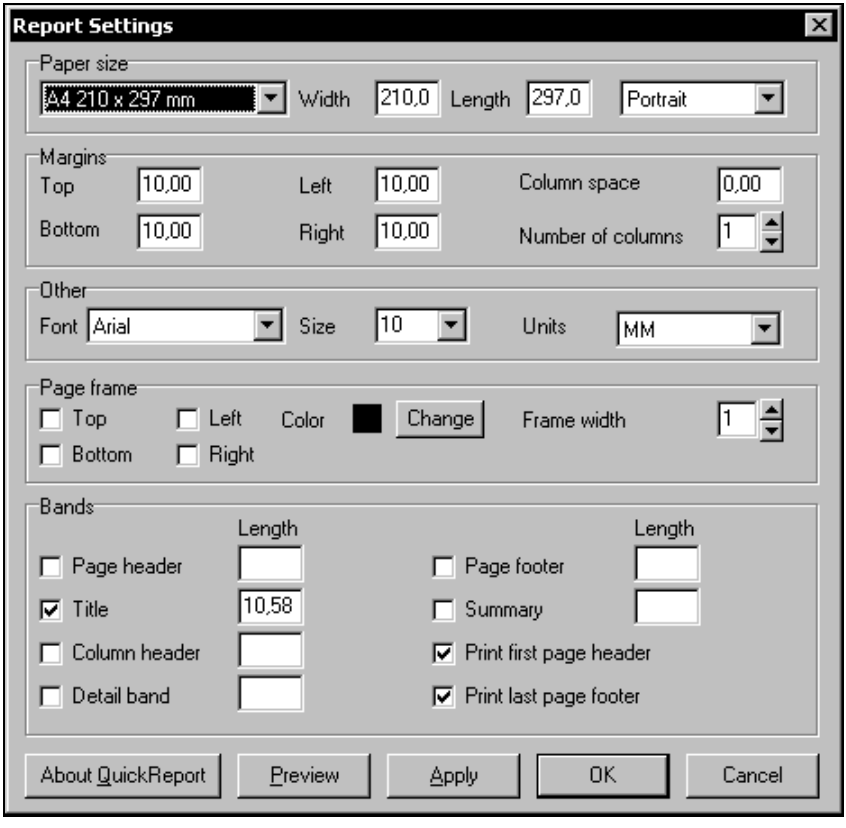


Рис. 11.2. Параметры страницы отчета

Параметры принтера можно устанавливать также с помощью стандартных диалогов `PrintDialog` и `PrinterSetupDialog` или метода `PrinterSetup`.

Отчет характеризуется тремя параметрами, которые задаются в свойстве `Options` множественного типа `TQuickReportOptions`:

- `FirstPageHeader` — печать верхнего колонтитула на первой странице отчета;
- `LastPageFooter` — печать нижнего колонтитула на последней странице отчета;
- `Compression` — отчет сохраняется в сжатом формате, при этом уменьшается объем занимаемой памяти, но снижается быстродействие.

По умолчанию свойство `Options` имеет значение

`[FirstPageHeader,LastPageFooter]`.

Свойство `PrintIfEmpty` типа `bool` определяет, выводить ли данные отчета для пустого набора данных. По умолчанию свойство имеет значение `true`, и отчет печатается, даже если в наборе данных нет ни одной записи. Это удобно, например, при печати бланков. Если свойству `PrintIfEmpty` установить значение `false`, то для пустого набора данных отчет не выводится, точнее, выводится пустая страница. То есть при отсутствии записей отсутствует не только область данных, но и ряд других полос, например, заголовок отчета.

Способ выравнивания элементов отчета определяет свойство `SnapToGrid` типа `bool`. Если оно имеет значение `true` (по умолчанию), то помещаемые в отчет компоненты автоматически выравниваются по линиям сетки. Шаг сетки задается в окне параметров среды C++ Builder и по умолчанию равен 8 пикселам. Если привязка элементов отчета к линиям сетки не требуется, то свойству `SnapToGrid` устанавливают значение `false`.

Процесс подготовки отчета к печати или просмотру может отображаться в отдельном окне. Наличие окна отображения для процесса подготовки документа определяет свойство `ShowProgress` типа `bool`. По умолчанию оно имеет значение `true`, и процесс подготовки отображается в окне на экране. Этот процесс может быть прерван нажатием клавиши <Esc>.

Как правило, свойства отчета и его отдельных элементов устанавливаются при создании отчета на этапе разработки приложения.

Для печати отчета предназначен метод `Print`, сразу после вызова которого отчет подготавливается к печати и направляется на установленный в системе принтер. Дополнительных подтверждений от пользователя не требуется. При необходимости текущий принтер и его параметры устанавливаются до начала печати, например, с помощью вызова метода `PrinterSetup`. Метод `Print` может вызываться, например, при нажатии кнопки с заголовком **Печать**, расположенной на форме, с которой пользователь работает.

Если компонент-отчет `QuickRep` связан с набором данных, записи которого выводятся в сетке `DBGrid` формы, то порядок записей отчета соответствует порядку записей, видимых пользователем на форме (рис. 11.3). После отбора (фильтрации) записей и/или сортировки при нажатии кнопки **Печать** происходит вывод отчета, причем учитывается новый состав и порядок следования записей.

При формировании отчета изменяется положение указателя текущей записи, поэтому при необходимости разработчик должен предусмотреть запоминание и восстановление положения указателя, например, с помощью закладки.

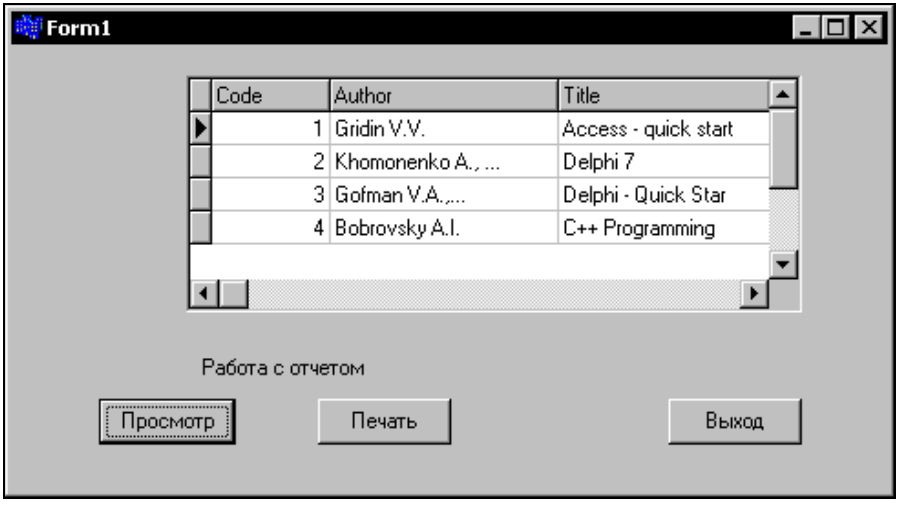


Рис. 11.3. Вид формы с сеткой `DBGrid`

Приведем пример процедуры предварительного просмотра и печати отчета.

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
TBookmark bm;
bm=Table1->GetBookmark();
// Просмотр отчета
Form2->QuickRep1->Preview();
// Печать отчета
Form2->QuickRep1->Print();
Table1->GotoBookmark(bm) ;
Table1->FreeBookmark(bm) ;
}
```

Закладка `bm` используется для запоминания и восстановления положения указателя текущей записи. В файле исходного кода модуля формы `Form1`, из которой выполняется печать отчета, следует указать директиву препроцессора `#include "Unit2.h"` для подключения заголовочного файла модуля формы `Form2`, в которой размещен компонент отчета.

При печати отчета генерируются события `BeforePrint` и `AfterPrint` типа `TQRBeforePrintEvent`. С помощью обработчика первого события можно задать действия, выполняемые непосредственно перед печатью отчета, а с помощью обработчика второго события — действия, выполняемые сразу после окончания печати.

Эти события возникают только при печати, а не при просмотре отчета. Поэтому если создан обработчик события `BeforePrint`, изменяющий вид и содержание отчета, то возможна ситуация, когда пользователь просматривает один вариант отчета, а напечатан он будет по-другому.

Для предварительного просмотра отчета перед печатью служит метод `Preview`, вызывающий окно просмотра (рис. 11.4). В этом окне можно сделать следующее:

- просмотреть отчет в разных масштабах;
- сохранить отчет в файле;
- загрузить предварительно сохраненный отчет;
- направить отчет на печать.

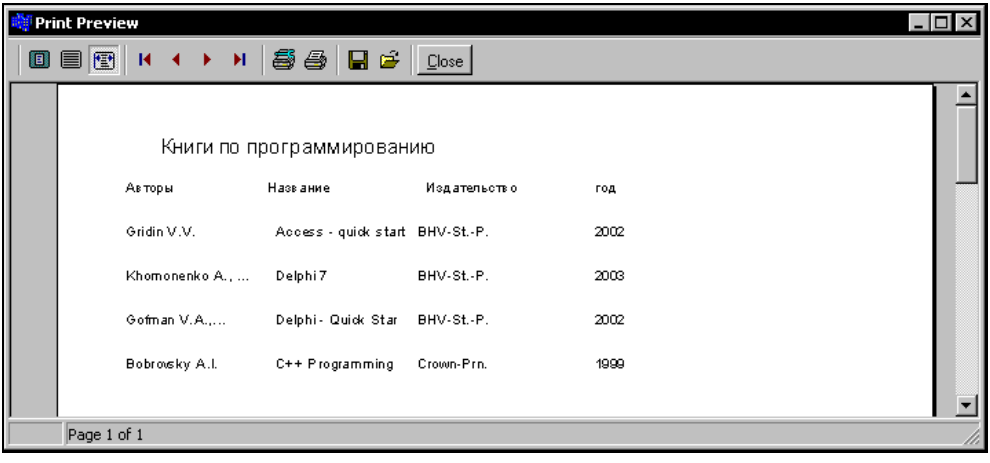


Рис. 11.4. Вид отчета в окне предварительного просмотра

Возможности метода `Preview` превосходят возможности метода `Print`, поэтому чаще выполняют именно предварительный просмотр документа, а не печать, что удобно делать и при отладке приложения. Печатать отчет можно непосредственно из окна предварительного просмотра.

На этапе разработки приложения также можно просмотреть отчет, выполнив команду **Preview** (Просмотр) контекстного меню отчета. Внешний вид отчета будет таким же, как при печати или в окне просмотра при выполнении приложения за исключением отсутствия значений вычисляемых полей.

При предварительном просмотре отчета в процессе выполнения приложения генерируются события `OnPreview` типа `TNotifyEvent` и `AfterPreview` типа `TQRAfterPreviewtEvent`. В обработчике первого события кодируются действия, выполняемые непосредственно перед предварительным просмот-

ром отчета, а в обработчике второго события — действия, выполняемые сразу после окончания предварительного просмотра.

Разработчик имеет возможность создать свое окно предварительного просмотра отчета, используя компонент `QRPreview`.

Полоса отчета

Полоса отчета (компонент `QRBand`) является основной составной частью (элементом) отчета, на которой размещаются другие его компоненты.

Тип полосы определяется свойством `BandType` типа `TQRBandType`, которое может принимать следующие значения:

- `rbTitle` — заголовок отчета (печатается в начале отчета под верхним колонтитулом);
- `rbPageHeader` — верхний колонтитул, который печатается сверху на каждой странице, в том числе на первой, если включен (по умолчанию) параметр `FirstPageFooter` свойства `Options` компонента отчета; если этот параметр выключен, то верхний колонтитул на первом листе не печатается;
- `rbDetail` — данные записей набора данных; выводятся для каждой записи набора данных;
- `rbPageFooter` — нижний колонтитул, который печатается внизу на каждой странице, в том числе на последней, если включен (по умолчанию) параметр `LastPageFooter` свойства `Options` компонента отчета; если этот параметр выключен, то нижний колонтитул на последней странице не печатается;
- `rbSummary` — итог отчета; выводится в конце отчета под всеми другими сведениями отчета, но выше нижнего колонтитула;
- `rbGroupHeader` — заголовок группы, который выводится для каждой группы;
- `rbGroupFooter` — примечание группы, которое выводится для каждой группы; может использоваться в качестве нижнего колонтитула для полосы `TQRSubDetail`;
- `rbSubDetail` — данные подчиненного набора данных для отчета "главный-подчиненный";
- `rbColumnHeader` — заголовки столбцов, печатаемые один раз на каждой странице над данными;
- `rbOverlay` — служит для совместимости с генератором отчетов предыдущей версии;
- `rbChild` — дочерняя полоса, печатаемая после полосы, с которой она связана.

При установке типа полосы она автоматически размещается на своем месте в отчете и выравнивается по ширине страницы отчета с учетом левого и правого полей. Разработчик не имеет возможности переместить полосу на другое место с помощью мыши. Изменить ширину полосы можно косвенно, изменяя размеры страницы и полей. Высота полосы меняется путем передвижения мышью верхней или нижней рамки полосы или через установку значения свойства `Height` в Инспекторе объектов.

Добавить новую полосу к отчету можно двумя способами:

- поместить компонент `QRBand` в отчет и присвоить требуемое значение свойству `BandType`;
- установить значение `true` соответствующему подсвойству свойства `Bands` компонента `QuickRep`, при этом к отчету добавляется полоса, а ее свойству `BandType` автоматически устанавливается нужное значение.

При создании отчета в него нужно включать не более одной полосы каждого вида, так как при печати отчета "лишние" полосы одного и того же вида учитываться не будут. Например, если в отчет включены две полосы заголовка отчета (`rbTitle`), то ошибки компиляции не возникает, но в качестве заголовка используется первая из этих полос. Полосы определенного вида, например, полоса `rbDetail`, при формировании отчета создаются автоматически для каждой записи набора данных.

При печати отчета полосы выводятся на странице целиком и не переносятся. Если, например, итоговая полоса `rbSummary` содержит много данных, то велика вероятность того, что она будет перенесена на следующую страницу, а на предыдущей странице может остаться достаточно много свободного места.

Каждая полоса может иметь отдельную рамку, которой управляет свойство `Frame`, не отличающееся от аналогичного свойства компонента отчета `QuickRep`.

При печати каждой полосы генерируются события `BeforePrint` типа `TQRBeforePrintEvent` и `AfterPrint` типа `TQRAfterPrintEvent`. Событие `BeforePrint` генерируется непосредственно перед печатью полосы, и его можно использовать для блокирования печати. Тип этого события описан как

```
type TQRBandBeforePrintEvent = procedure (Sender : TQRCustomBand;
                                          var PrintBand : Boolean)
```

Параметр `PrintBand` определяет, печатать ли полосу. Если вывод полосы нежелателен, то параметру следует присвоить значение `false`. Обычно обработчик события `BeforePrint` кодируется для полос данных, когда выполняется проверка данных полей текущей записи, и решение о печати полосы принимается в зависимости от выполнения определенных условий.

В процессе выполнения приложения при вызове метода `Preview` также генерируются события `BeforePrint` и `AfterPrint`.

Приведем пример обработчика, в котором осуществляется управление печатью полосы:

```
void __fastcall TForm2::DetailBand1BeforePrint(TQRCustomBand *Sender,
      bool &PrintBand)
{
PrintBand = Form1->Table1->FieldByName("Year")->AsInteger == 2002;
}
```

В отчете печатаются или отображаются при предварительном просмотре записи, имеющие в поле `Year` значение `2002`. Для рассмотренного примера отчета с данными о книгах это означает печать или отображение данных только о книгах 2002 г. выпуска.

Компоненты, размещаемые в полосе

После создания полосы определенного типа в ней размещаются соответствующие компоненты, при этом необходимо использовать только компоненты страницы `QReport`. Размещение на полосе других компонентов, например, `Label` или `Edit`, не вызывает ошибки трансляции, но в сформированный отчет такие компоненты не попадают.

Можно разместить компонент отчета и вне полосы — непосредственно на компоненте-отчете `QuickRep`. В этом случае он будет выводиться на каждой странице отчета (например компонент `QRImage`, содержащий логотип фирмы).

Обычно используются следующие компоненты отчета:

- `QRLabel` — надпись, содержащая текст (аналог надписи `Label`), может размещаться на любой полосе, но для полос данных обычно не используется;
- `QRDBText` — значение поля записи (аналог компонента `DBText`); обычно размещается в полосе данных;
- `QRExpr` — значение, формируемое на основе выражения, в котором могут использоваться значения полей записей; обычно используется для полос данных и нижних колонтитулов;
- `QRSysData` — системная информация, обычно используется для итоговых полос и полос колонтитулов; ее вид определяется свойством `Data` типа `TQRSysDataType`, принимающим значения:
  - `qrsColumnNo` — номер текущего столбца (для одноколоночного отчета равен единице);
  - `qrsDate` — текущая дата;

- qrsDateTime — текущие дата и время;
- qrsDetailCount — число записей в наборе данных (для набора данных Query, например, при запросе к нескольким таблицам, может быть не определено);
- qrsDetailNo — номер текущей записи (если отчет выводится для связанных таблиц, то в качестве номера берется номер текущей записи главной таблицы);
- qrsPageNumber — номер текущей страницы;
- qrsPageCount — общее число страниц в отчете (включение этого значения требует для формирования отчета двух проходов по набору данных, что увеличивает время подготовки отчета и может быть неэффективно при большом количестве записей);
- qrsTime — текущее время (по умолчанию).

❒ QRImage — графический образ, загружаемый аналогично графическому образу Image (может быть использован в любой полосе, но обычно не размещается в полосах данных); с помощью компонента QRImage выводится, например, логотип организации; кроме того, следует отличать этот компонент и от компонента QRDBImage, который обычно размещается в полосе данных и отображает рисунок из поля таблицы;

❒ QRShape — геометрическая фигура, размещаемая в любой полосе.

Перед компонентом QRSysData может находиться надпись, которая указывается в свойстве Text типа AnsiString. Например, для значения qrsPageNumber в качестве надписи подойдет текст Страница. По умолчанию надпись отсутствует.

Компонент QRShape позволяет оформить сетку вокруг выводимых данных. Для этого в полосу помещается компонент QRShape, имеющий форму прямоугольника, а сверху него располагается текстовый элемент. Прямоугольники будут образовывать рамки сетки, для чего соседние по вертикали и горизонтали прямоугольники выравниваются так, чтобы их стороны совпадали. Если для образования сетки использовать рамку самих текстовых элементов, то необходимо учитывать, что эта рамка выводится непосредственно вокруг текста, и при стыковке соседних элементов будет отсутствовать межстрочный интервал. Другой вариант создания сетки — образование ее из горизонтальных линий рамки полос и вертикальных линий рамки. Однако в этом случае вертикальные линии не будут сплошными.

Выравниванием текста внутри компонентов, содержащих текст, управляет свойство Alignment. К таким компонентам относятся, например, QRLabel и QRDBText. По умолчанию текст выравнивается по левому краю. При необходимости это значение можно изменить, установив, например, для поля даты

выравнивание по центру (taCenter), а для поля цены — выравнивание по правому краю (taRightJustify).

Возможностью автоматического изменения размеров содержащего текст компонента по размеру текста управляет свойство AutoSize. По умолчанию размер текстового компонента автоматически подстраивается под содержащийся в нем текст, так как свойство AutoSize имеет значение true. Это может привести к нарушению выравнивания столбцов отчета. Если компоненты отчета, в первую очередь элементы полосы данных, имеют рамку, то размеры этих компонентов должны быть постоянными. В таком случае свойству AutoSize устанавливается значение false.

Параметры шрифта устанавливаются через свойство Font отдельных компонентов. Для изменения шрифта сразу у всех компонентов удобно использовать свойство Font компонента QuickRep и свойство ParentFont компонентов отчета, управляющее возможностью наследования параметров шрифта от родительского компонента. В нашем случае родительским компонентом является отчет QuickRep. По умолчанию свойство ParentFont каждого компонента отчета имеет значение true, и при изменении шрифта отчета автоматически изменяется шрифт всех его элементов.

Если для какого-либо компонента изменяются параметры шрифта, то его свойству ParentFont присваивается значение false.

Для оформления рамки вокруг отдельного элемента отчета используется свойство Frame, управляющее наличием рамки, цветом, стилем и толщиной линий.

Многие компоненты отчета имеют событие OnPrint типа TQRLabelOnPrint-Event, возникающее непосредственно перед печатью компонента. Действие обработчика события OnPrint распространяется только на отчет и не затрагивает значения записей, находящихся в наборе данных.

Например, форматирование значений при печати:

```
void __fastcall TForm2::QRDBText4Print(TObject *sender, AnsiString
&Value)
{
Value=Value + " год";
}
```

Параметр Value обработчика события содержит значение, которое должно быть напечатано. Это значение можно изменить или отформатировать. Компонент QRDBText4 связан с символьным полем. При печати значения этого поля выводятся со строкой год.

Обработчик события OnPrint можно использовать, например, для улучшения выравнивания компонентов при использовании вокруг них вертикальных рамок. Так, при выравнивании текста компонента QRDBText вправо (его

свойство `Alignment` имеет значение `taRightJustify`) возможно наползание текста на правую линию рамки. В этом случае для улучшения читаемости текст передвигается на несколько пробелов влево путем добавления в обработчик события строки кода:

```
Value:=Value + "    ";
```

При необходимости в обработчике события `OnPrint` выполняется и более сложное форматирование данных.

Событие `OnPrint` генерируется также при вызове метода `Preview`, но не возникает при просмотре отчета на этапе разработки приложения.

Простой отчет

Простой отчет представляет собой отчет на основе данных из одного набора данных и содержит сведения, которые выводятся в табличном виде без дополнительных условий, например, группирования данных. Размещение и вид печатаемых в отчете данных аналогичны размещению и виду данных, отображаемых в сетке `DBGrid`. Отличием является то, что данные отчета размещаются не на форме, а на бумажном документе, и их нельзя редактировать.

Рассмотрим действия, выполняемые при подготовке простого отчета, на примере перечня книг по программированию (см. рис. 11.1, 11.3 и 11.4). Многие из этих действий используются и при разработке отчетов других видов. Для создания простого отчета требуется выполнить следующее.

1. Разместить на форме компонент `QuickRep`.
2. Создать для компонента `QuickRep` требуемые полосы отчета.
3. Разместить в полосах нужные компоненты отчета, чаще всего `QRLabel`, `QRDBText` и `QRExpr`.
4. Создать для событий, например, нажатия кнопок с заголовками **Печать** и **Просмотр**, обработчики, в которых вызываются методы печати и предварительного просмотра отчета соответственно.

Размещение на форме компонента отчета, установка параметров отчета, а также создание обработчика события нажатия кнопки печати рассмотрены ранее при описании компонента `QuickRep`.

Простой отчет может содержать следующие полосы, перечисляемые в порядке их размещения на странице:

- верхний колонтитул (`rbPageHeader`);
- заголовок отчета (`rbTitle`);
- заголовки столбцов (`rbColumnHeader`);
- данные (`rbDetail`);

- итог отчета (`rbSummary`);
- нижний колонтитул (`rbPageFooter`).

На этапе разработки название каждой полосы выводится серым цветом в ее левом нижнем углу.

Можно добавить к проекту шаблон простого отчета, вызвав командой меню **File/New/Other... (Файл/Новый/Другой)** Хранилища объектов и выбрав на странице **Forms** (Формы) объект **QuickReport List** (Лист отчета). Шаблон этого отчета содержит полосы заголовка отчета, заголовков столбцов и данных, нижний колонтитул и расположен на отдельной форме `QRListForm`. На форме отчета также расположен набор данных `Table`.

Заголовок отчета

Заголовок отчета выводится один раз на первой странице сразу под верхним колонтитулом, если он есть. В полосе заголовка обычно размещаются надписи `QRLabel`, содержащие требуемый текст (как правило, в качестве заголовка выводится название всего отчета). При необходимости в заголовке можно разместить, например, сведения о названии, адресе и телефоне организации, а также логотип. В приведенном на рис. 11.1 примере в полосе заголовка находится название отчета. Для вывода названия отчета используется компонент `QRlabel`, в котором набран текст `Книги по программированию`. Для названия установлен шрифт Arial размером 14 пт. Логотип загружается в компонент `QRImage`. Для шрифта других компонентов рассматриваемого отчета с помощью свойства `Font` компонента `QuickRep` установлен размер 10 пт.

Заголовки столбцов и данные

Полосы заголовков столбцов и данных являются основными полосами, в которых размещаются компоненты, обеспечивающие табличный вывод содержимого набора данных. Заголовки столбцов выводятся на каждом листе отчета.

Для заголовков столбцов данных в полосу заголовков обычно помещаются компоненты `QRlabel`, в которые заносится текст, соответствующий полям данных.

Для вывода значений полей записей в полосу данных обычно помещаются компоненты `QRDBText` и `QRExpr`. Более простым является использование компонентов `QRDBText`, каждый из которых отображает значение связанного с ним поля. Имя набора данных указывается в свойстве `DataSet`, а имя поля задается в свойстве `DataFileld`. Для наглядности схематично состав простого отчета и его связи с набором данных приведены на рис. 11.5.

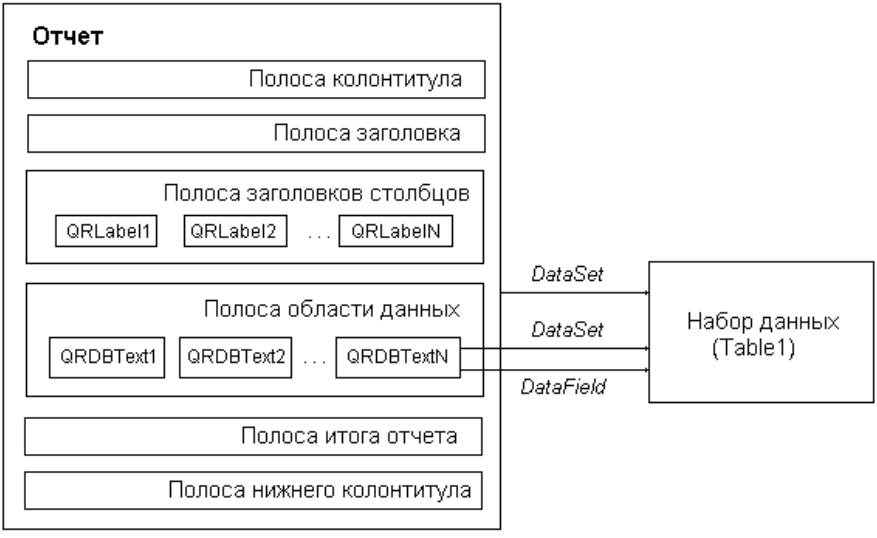


Рис. 11.5. Состав простого отчета и его связи с набором данных

На этапе разработки в отчете присутствует только одна полоса данных, но при формировании отчета отдельная полоса данных будет выведена для каждой записи отчета. Напомним, что если набор данных является пустым и не содержит записей, то область данных не выводится. Чтобы для пустого набора данных были выведены остальные полосы (кроме полос данных), свойству `PrintIfEmpty` компонента `QuickRep` устанавливается значение `true` (по умолчанию).

Компонент `QRExpr` позволяет вставлять в отчет значение выражения, рассчитываемого обычно с участием различных полей записей. Выражение заносится в свойство `Expression` типа `String`, для формирования которого удобно использовать окно **Expression Wizard** (Мастер выражений), вызываемое через Инспектор объектов.

Для вставки в выражение имени поля нужно нажать кнопку **Database Field** (Поле БД) и в открывшемся окне выбрать набор данных и имя поля.

В выражении можно использовать функции, которые разбиты по категориям и выбираются в специальном окне, вызываемом нажатием кнопки **Function** (Функция).

Названия полей и функций можно набирать и вручную, однако это увеличивает вероятность ошибки.

Чтобы протестировать введенное выражение, следует нажать кнопку **Validate**, при этом выполняется проверка выражения, а разработчику выдается сообщение о корректности выражения или об ошибке.

В выражении можно использовать только поля наборов данных, которые размещены на форме отчета. В противном случае набор данных оказывается недоступен для компонента `QRExpr`.

При разработке приложения такие компоненты отчета, как `QRlabel`, `QRDBText` и `QRExpr`, имеют одинаковый внешний вид.

Итоговая полоса

Итоговая полоса отчета выводится один раз в конце отчета сразу после полосы данных. В этой полосе обычно стоят либо итоговые сведения отчета, например, средние и максимальные значения по данным какого-либо поля, либо должность и фамилия лица, утверждающего отчет. В итоговой полосе обычно размещаются компоненты `QRlabel` для вывода надписей и `QRExpr` для вывода значений выражений.

Если итоговая полоса не помещается на текущей странице целиком, то она отделяется от последней полосы данных и переносится на другую страницу, что не соответствует правилам делопроизводства.

В выражении для итогового компонента `QRExpr` часто используются следующие статистические функции:

- `SUM` — сумма значений;
- `MIN` — минимальное значение;
- `MAX` — максимальное значение;
- `AVERAGE` — среднее значение;
- `COUNT` — число записей набора данных.

Для всех функций, кроме `COUNT`, в скобках нужно указывать имя поля, для которого выполняется расчет. Для вывода числа записей набора данных также можно использовать компонент `QRSysData` со значением `qrsDetailCount` свойства `Data`.

В рассматриваемом нами примере итоговая полоса не используется. Отметим, что в качестве итога может выводиться, к примеру, число записей в таблице или заверительная подпись.

Колонтитулы

Колонтитулы печатаются в начале и конце каждой страницы, в них обычно выводятся сведения о дате, времени печати, а также номер страницы. Для этого в полосах колонтитулов чаще всего размещаются компоненты `QRSysData`, которым устанавливается требуемое значение свойства `Data`.

В полосе колонтитула можно также разместить и другие компоненты, например, `QRLabel` для вывода на каждой странице названия организации.

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 C++ Builder поставляется большое число инструментов, применимых для работы как с локальными, так и с удаленными БД.

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 позволяет достаточно просто и удобно выполнить связывание (соединение) таблиц.

С помощью программы Data Pump 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.

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.

В этой главе рассматриваются важнейшие инструменты для работы с локальными базами данных — программы BDE Administrator и Database Desktop. 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 Адми-

нистратор). 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 C++ Builder 6\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 можно также 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 следующие:

- 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.

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.

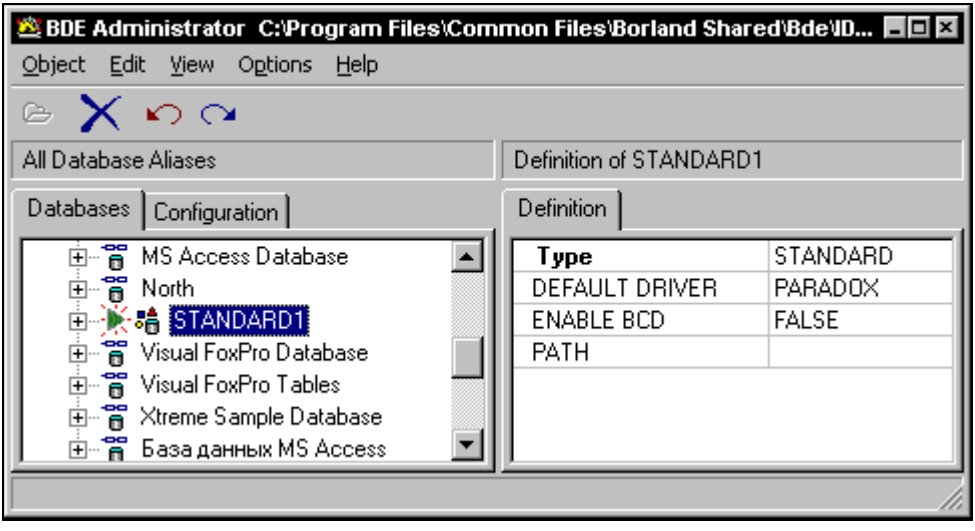


Рис. 12.1. Установка параметров псевдонима

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).

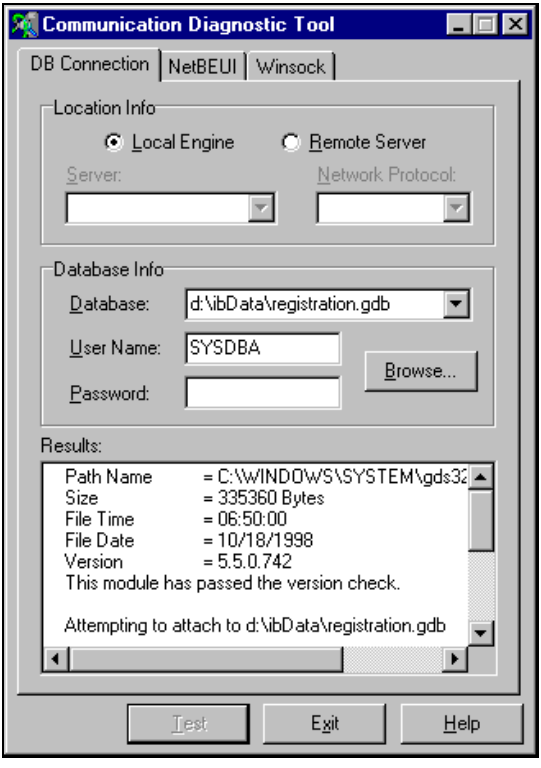


Рис. 12.2. Диалоговое окно проверки соединения с удаленной БД

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.

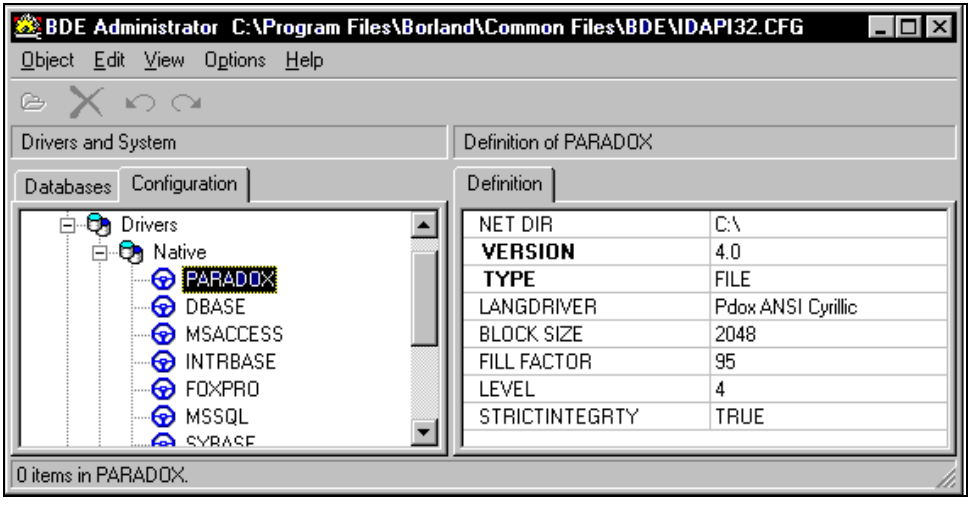


Рис. 12.3. Установка параметров драйвера базы данных Paradox

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.

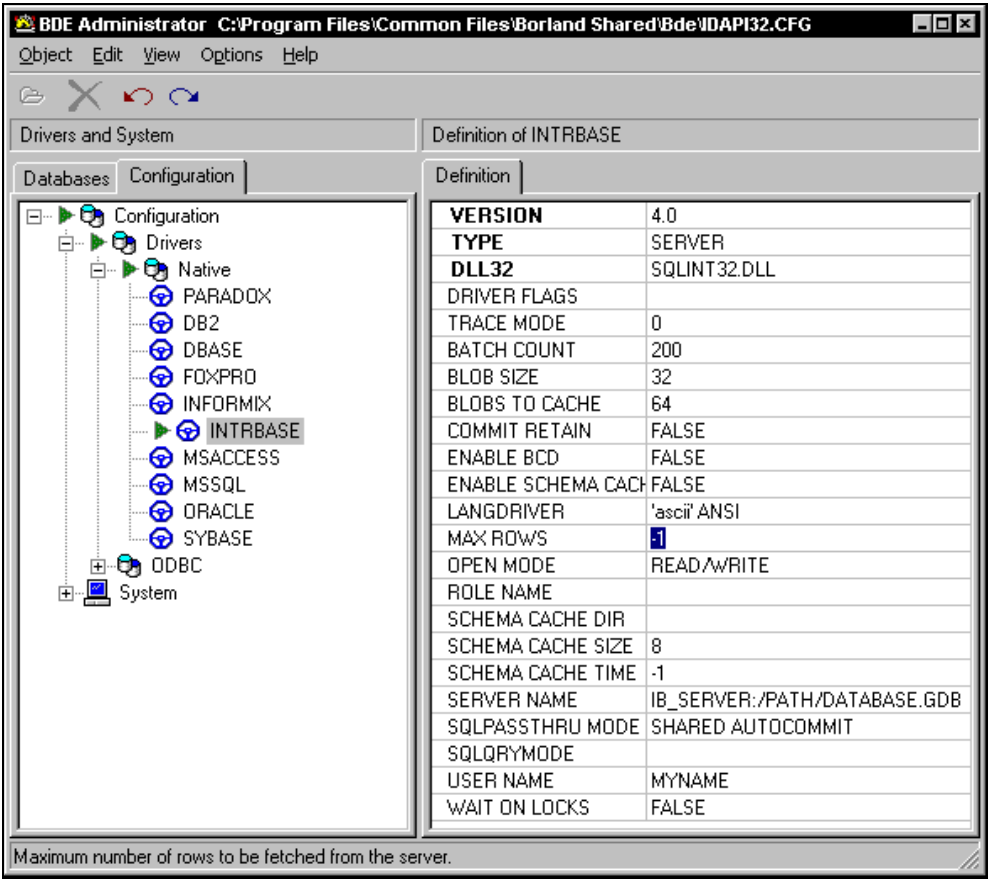


Рис. 12.4. Установка параметров драйвера InterBase

Наибольший интерес представляют следующие параметры:

- `DLL32` — указывает драйвер SQL-Links, используемый для доступа к БД (sqlint32.dll);
- `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 предоставляют 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);
- 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 ранее, 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 C++ Builder 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 C++ Builder 6\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 Directory** 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.

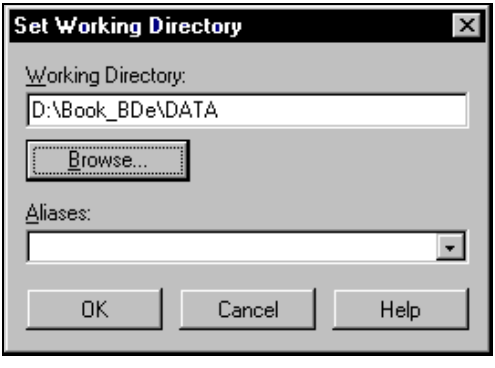


Рис. 12.5. Установка рабочего каталога

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.

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.

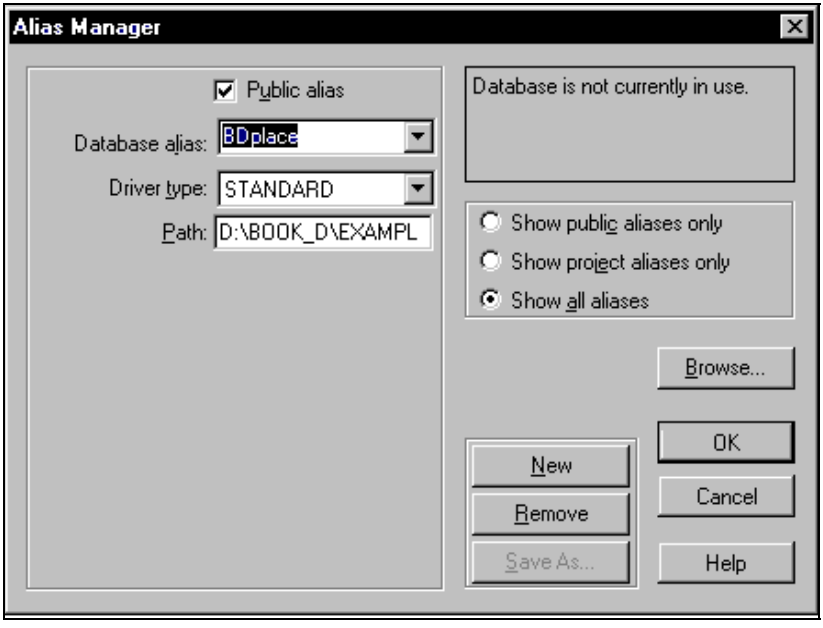


Рис. 12.6. Диалоговое окно Менеджера псевдонимов

Работа с SQL-запросами

Работа с SQL-запросами включает:

- создание запроса;
- редактирование запроса;
- выполнение запроса.

Для создания SQL-запроса нужно, выполнив команду **File\New\SQL File** программы Database Desktop, открыть окно редактора **SQL Editor** (рис. 12.7) и набрать в нем текст SQL-запроса.

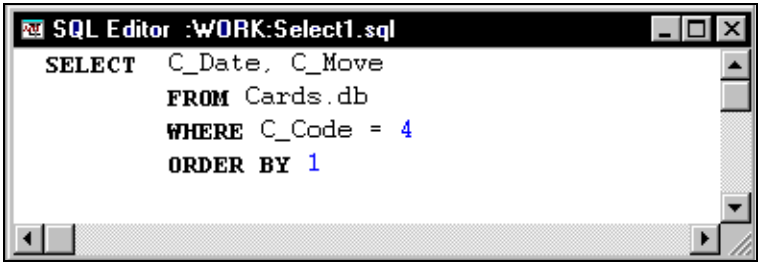


Рис. 12.7. Окно редактора SQL-запроса

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\Run 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.

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 **File\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. 12.8).

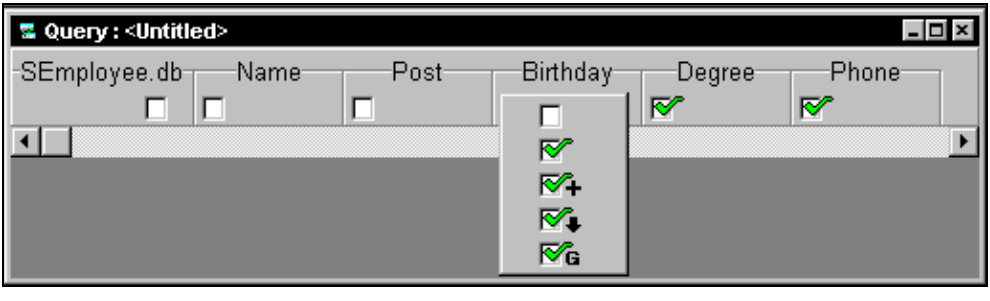


Рис. 12.8. Диалоговое окно визуального конструктора запросов SQL

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. 12.8), 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. 12.9, из таблицы `SEmployee` отбираются записи, для которых значение ученой степени (поле `Degree`) равно `ктн`. Кроме этого поля, в результат запроса включаются поля имени `Name` и должности `Post`.

Рис. 12.9. Отбор записей для сотрудников — кандидатов технических наук

В приведенном примере поле `Degree` 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.

Связывание таблиц

Можно построить запрос по образцу не только для одиночных таблиц, как это рассмотрено ранее, 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 и выбором в открывшемся окне **Select File** главного файла таблицы.

Для связывания таблицы нужно нажать кнопку **Join Tables** панели инструментов, 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 (рис. 12.10). Так же можно связать между собой три и более таблиц.

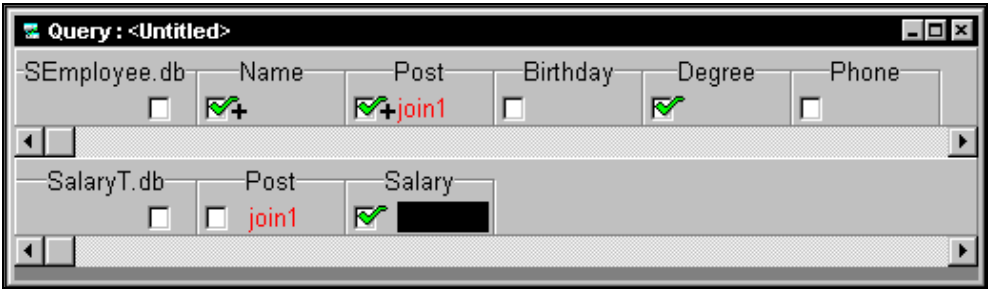
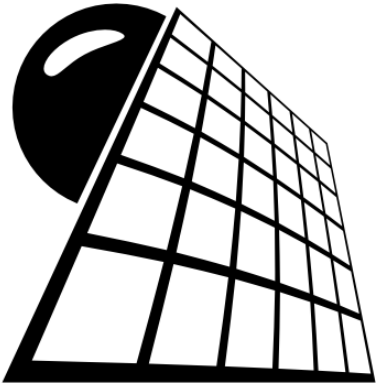


Рис. 12.10. Связывание таблиц в программе Database Desktop

Для связанных таблиц операции отбора, редактирования, вставки и удаления записей выполняются так же, как и для одиночных таблиц.

В качестве примера отметим следующее. Запрос, показанный на рис. 12.10, выводит из подчиненной таблицы `Salary` записи с окладом для должности, которая указана в главной таблице `SEmployee` данных о сотрудниках. Отбор записей осуществляется по полю `Post` должности, в результирующий набор данных включаются поля `Name` фамилии сотрудника, `Post` должности и `Degree` ученой степени.

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**.



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, затрудняется 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 единственном 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 далее.

Средства работы с удаленными БД

Система C++ Builder 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 запросов (реляционного способа) для 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 SEmployee WHERE Post = 'нс'");
Query1->ExecSQL();
```

Для работы с таблицами и запросами по-прежнему можно использовать такие программы, как Database Desktop и SQL Explorer.

Средства C++ Builder, 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 при навигационном способе доступа к данным. 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 `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, который является "родным" для C++ Builder. Совместно с C++ Builder поставляются две части сервера InterBase: серверная и клиентская. Несмотря на то, что сервер InterBase поставляется совместно с C++ Builder, устанавливается он отдельно: после установки C++ Builder 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, поставляемая вместе с C++ Builder, 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 сделать следущее:

- 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 C++ Builder.

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. Далее мы рассмотрим особенности языка SQL для удаленных БД.

При рассмотрении операторов языка будем опускать несущественные операнды и элементы. При описании формата операторов языка SQL используются следующие правила:

- символы < и > обозначают отдельные элементы формата операторов, например, имена таблиц и столбцов, и при записи операторов 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 операторов 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). Ограничение, заданное на *физическом* уровне, в дальнейшем нельзя нарушить или обойти.

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*. Достоинства такого подхода следующие:

- вычислительная нагрузка по управлению БД целиком ложится на сервер, что снижает нагрузку на приложение и сеть;
- 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. Например, программирование бизнес-правил в приложении может состоять в каскадном удалении 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 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 сервера 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 $3{,}4 \times 10^{-38}$ .. $3{,}4 \times 10^{38}$. Точность 7 цифр мантиссы
`DOUBLE PRECISION`	Число с плавающей точкой Диапазон по модулю $1{,}7 \times 10^{-308}$ .. $1{,}7 \times 10^{308}$. 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 C++ Builder реализует программа IB-Console.

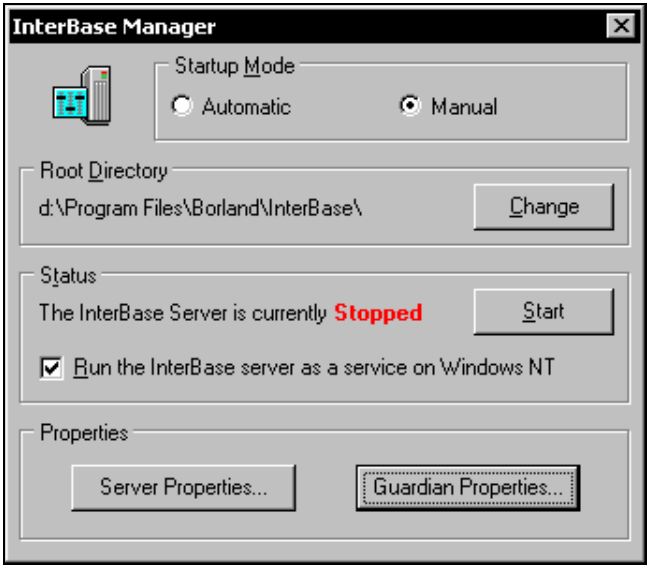


Рис. 13.1. Диалоговое окно **InterBase Manager**

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.

Если сервер InterBase запущен не как служба Windows NT, то в правом нижнем углу панели задач появляется значок . При этом завершить работу сервера также можно, открыв контекстное меню щелчком мыши на значке 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 C++ Builder 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 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 C++ Builder предоставляет соответствующие компоненты, в первую очередь, компонент `Database`.

Рассмотрим, как установить соединение с удаленной БД с помощью программы IBConsole. Перед установлением соединения с БД нужно с помощью команд **Server\Register** и **Server\Login** выполнить регистрацию сервера InterBase, а также задать имя и пароль пользователя.

Замечание

Заметим, что первоначально на сервере InterBase 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`.

Глава 14

Работа с удаленными базами данных InterBase

В данной главе мы рассмотрим основные операции, осуществляемые при работе с удаленной БД InterBase: 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). Перед созданием базы данных локальный сервер должен быть зарегистрирован с помощью команды **Register** меню **Server** и установлена связь с ним путем задания команды **Login** меню **Server** с последующим указанием параметров **User Name** (SYSDBA) и **Password** (masterkey).

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). В нем достаточно указать один файл. Для файла необходимо задать его точное расположение; для локального сервера местом расположения является 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.

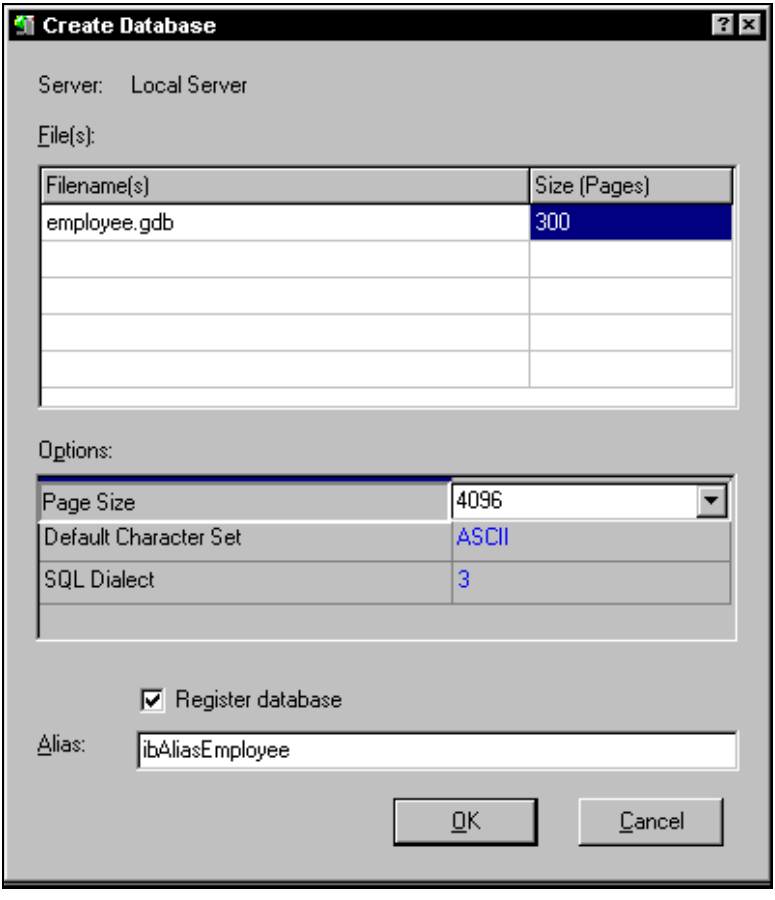


Рис. 14.1. Окно создания удаленной БД

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 с расширением 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`.

Оператор 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, оператор

```
CREATE DATABASE "D:\IBDATA\employee.gdb";
```

создает однофайловую БД с именем employee.gdb, расположенную в каталоге D:\IBDATA.

В свою очередь, оператор

```
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 оператора

```
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.

Оператор 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 операторов 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 оператора `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 — 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 — оператором `DROP TABLE`, имеющим 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 оператором `ALTER TABLE`, который позволяет добавлять и удалять отдельные столбцы, а также ограничения. В отличие от оператора `ALTER TABLE`, другие операторы 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 операторов. 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 тремя способами:

- задать тип столбца, например, `DATE` или `INTEGER`, при этом создается обычный столбец указанного типа. Типы столбцов InterBase описаны далее в данной главе;
- 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 этой главе.

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 вещественный 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 вычисляемого столбца автоматически будет определен как `FLOAT`.

Ограничения столбца

Ограничения столбца задавать не обязательно, 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 операторов 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 идентичным условиям, задаваемым при отборе записей в операторе SELECT, и здесь подробно не рассматриваются. Более того, при задании условий в операнде CHECK можно использовать оператор SELECT. Как и в операторе 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 уни-

кальное 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 оператора `SELECT`. Условие отбора записей, заданное в этом операторе, для элемента `Отбор1` должно возвращать одно значение или ни одного, а для элемента `ОтборМ` — несколько значений, в частном случае, ни одного. Оператор `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 оператора `SELECT`. В последнем случае состав набора зависит от результата работы оператора `SELECT`, который выполняется каждый раз при модификации записей таблицы.

Например, оператор

```
CREATE TABLE Test2
      (Name       VARCHAR(20),
       Post       VARCHAR(20)
            CHECK (Post IN ("Директор", "Инженер", "Бухгалтер")),
       Attrib     VARCHAR(20)
            CHECK (Attrib IN (SELECT Attrib From Positions)));
```

для полей таблицы `Test2` устанавливает список возможных значений: значение поля `Post` 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.

Добавить ограничение к уже существующей таблице можно с помощью оператора вида

```
ALTER TABLE <Имя таблицы>
      ADD CHECK ([CONSTRAINT <Имя ограничения >] <Условия>);
```

Для удаления ограничения предназначается оператор

```
ALTER TABLE <Имя таблицы> DROP <Имя ограничения>;
```

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, оператор

```
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 далее.

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, оператор:

```
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`.

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 операторы 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 следующим оператором:

```
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. Оператор создания индекса `CREATE INDEX` имеет формат:

```
CREATE [UNIQUE] {ASCENDING] [DESCENDING] INDEX
      <Имя индекса> ON <Имя таблицы > (<Имя столбца>, ..., [<Имя столб-
ца>]);
```

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, оператор

```
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 оператором DROP INDEX: для этого нужно выполнить реструктуризацию таблицы оператором 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 оператора

```
SHOW INDEX [<Имя таблицы>];
```

Этот оператор 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 операторов:

```
ALTER INDEX <Имя индекса> INACTIVE;
ALTER INDEX <Имя индекса> ACTIVE;
```

Первый оператор деактивизирует (отключает) указанный индекс, а второй — вновь активизирует (включает) его. Перестроить индекс можно и более кардинальным способом, удалив его и создав заново. Отметим, что вновь созданный индекс является активным.

Домены

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 программирования, например C++, то аналогом домена является тип данных. Таким образом, домен можно рассматривать как *шаблон* описания столбца.

Перед использованием домена его нужно создать с помощью оператора

```
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.

Представления

Представление, или просмотр (view), является логической (виртуальной) таблицей, записи в которую отобраны с помощью оператора `SELECT`. Преимущество просмотра заключается в том, что можно один раз отобрать записи и использовать их в дальнейшем без повторного выполнения оператора `SELECT`. Это выгодно при частом выполнении одинаковых запросов, особенно тех, в которых реализованы сложные условия отбора.

Просмотр создается следующим оператором:

```
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 операторе `SELECT`. Оператор `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 операторе `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 операторе `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 операторе `SELECT` просмотр `vStore` используется, как обычная таблица.

Удалить просмотр можно следующим оператором:

```
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 они реализуют единые для всех приложений-клиентов правила работы с БД.

Для выполнения процедуры, хранимой на сервере, в приложении используется компонент `StoredProc`.

Создание и изменение

Хранимая процедура создается оператором

```
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>,]
       ...
       [<ИмяпараметраN> <ТиппараметраN>]
```

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 префикса `ip` (от Input Parameter), а имя выходного параметра — с префикса `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
      <Оператор>
      ...
      [<Оператор>]
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 операторы, выполняющие необходимые действия. Операторы располагаются между ключевыми словами `BEGIN` и `END`. В теле процедуры должен содержаться как минимум один оператор.

Созданную хранимую процедуру можно удалить или изменить. Удаляется процедура оператором

```
DROP PROCEDURE <Имя процедуры>
```

Изменение процедуры выполняется оператором `ALTER PROCEDURE`, имеющим такой же формат, как и оператор `CREATE PROCEDURE`.

Виды хранимых процедур

По числу строк, возвращаемых в качестве результата, можно выделить следующие виды хранимых процедур:

- *процедуры выбора*, которые возвращают несколько строк — передают набор данных, записями в котором являются строки результатов;
- *исполняемые процедуры*, которые возвращают одну строку и обеспечивают тем самым возврат значений выходных параметров.

В теле процедуры выбора размещаются совместно используемые операторы 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.

Язык хранимых процедур

Для написания хранимых процедур и триггеров используется язык хранимых процедур. Язык хранимых процедур сервера InterBase представляет собой процедурный алгоритмический язык. Он включает операторы, позволяющие управлять ходом вычислительного процесса (условный оператор и оператор цикла). Кроме того, язык хранимых процедур обладает рядом возможностей языка SQL.

Рассмотрим кратко основные составляющие языка хранимых процедур.

Для задания комментариев в тексте процедуры или триггера используются символы `/*` и `*/`.

В язык хранимых процедур включены операторы:

- объявления переменных;
- присваивания;
- условные;
- составные;
- цикла;
- выбора записи;
- выбора нескольких записей (набора данных);
- возврата значений;
- выхода из процедуры;
- вызова процедуры;
- посылки сообщения.

Операторы (кроме составного) должны заканчиваться точкой с запятой.

Оператор 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.

Оператор присваивания служит для задания значения переменной и имеет вид

```
<Имя переменной> = <Выражение>;
```

При выполнении этого оператора вычисляется значение выражения и присваивается переменной, имя которой задано в левой части оператора. 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 i INTEGER;
DECLARE VARIABLE s VARCHAR(20);
...
i = 1;
s = UPPER(s);
```

целочисленной переменной `i` присваивается значение `1`, 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.

Условный оператор позволяет организовать ветвление вычислительного процесса и имеет следующий формат:

```
IF (<Условие>) THEN <Оператор1> [ELSE <Оператор2>];
```

`Условие` представляет собой выражение логического типа. Условный оператор выполняется так: если условие истинно (имеет значение `true`), то выполняется `Оператор1`, в противном случае — `Оператор2`. Оба указанных оператора могут быть составными.

Условный оператор может быть записан в сокращенной форме, когда слово `ELSE` и оператор после него отсутствуют. В этом случае при невыполнении условия управление сразу передается на оператор, следующий за условным.

Составной оператор представляет собой группу из произвольного числа любых операторов, ограниченную операторными скобками `BEGIN` и `END`. Составной оператор имеет следующий формат:

```
BEGIN <Оператор1> ... <ОператорN> END
```

Независимо от числа входящих в него операторов составной оператор воспринимается как единое целое и может располагаться в любом месте, где допускается наличие оператора. Наиболее часто составной оператор используется в условных инструкциях и инструкциях цикла. В конце составного оператора точка с запятой не ставится. Составной оператор объединяет также все операторы тела процедуры или триггера.

Пример использования условного и составного операторов:

```
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 операторы присваивания объединены в составной оператор. Если условие не выполняется, то числовой

переменной n присваивается новое значение, а значение строковой переменной s остается без изменений.

Оператор цикла (повтора) обеспечивает повтор группы операторов при соблюдении некоторых условий. Оператор цикла имеет следующий формат:

```
WHILE (<Условие>) DO <Оператор>;
```

Оператор, расположенный после слова DO, будет выполняться до тех пор, пока соблюдается указанное условие. Если в цикле требуется выполнить несколько операторов, то их нужно объединить в составной оператор. 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 C++ Builder. Этот и некоторые следующие примеры приводятся в упрощенном виде с целью показать отдельные операторы языка хранимых процедур.

Оператор выбора записи представляет собой оператор SELECT, который возвращает одну строку, поэтому ее называют также оператором выбора записи. Значения столбцов возвращаемой строки присваиваются указанным переменным или параметрам. При своем выполнении оператор выбора записи (как и оператор SELECT) отбирает несколько строк, но возвращает только одну.

Формат оператора выбора записи по сравнению с форматом оператора SELECT языка SQL, дополнен операндом:

```
INTO :<Имя1>, ... [:<ИмяN>]
```

Параметр Имя после двоеточия указывает переменную или выходной параметр, которым должны быть присвоены значения столбцов записи, полученной в результате выполнения оператора SELECT. Этот оператор часто используется совместно со статистическими функциями, такими как SUM (сумма) или AVG (среднее значение).

Рассмотрим использование оператора SELECT в хранимой процедуре на примере:

```
CREATE PROCEDURE pSalary
      RETURNS (opSum FLOAT, opAvg FLOAT)
      AS
      BEGIN
            SELECT SUM(Salary), AVG(Salary)
            FROM Employee
            INTO :opSum, :opAvg;
      END
```

Создается хранимая процедура `pSalary`, в которой для сотрудников организации (таблица `Employee`) 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.

Оператор выбора нескольких записей также представляет собой оператор `SELECT`, способный возвращать несколько записей (в частности, ни одной). Оператор выбора нескольких записей имеет формат:

```
FOR <Оператор выбора записи> DO <Оператор>;
```

Оператор выбора нескольких записей может задавать не только отбор, но и обработку записей. После отбора записей согласно оператору `SELECT` для каждой из них выполняется оператор, указанный после слова `DO`.

Пример хранимой процедуры с оператором выбора:

```
CREATE PROCEDURE pAvgSal
      RETURNS (opAvg INTEGER)
      AS
      DECLARE VARIABLE n INTEGER;
      DECLARE VARIABLE s INTEGER;
      BEGIN
            s = 0; i = 0;
            FOR SELECT Sal FROM Employee
                  INTO n
                  DO BEGIN s = s + n; i = i +1;  END;
            opAvg = s/i;
      END
```

Создается хранимая процедура `pAvgSal`, которая вычисляет среднее значений целочисленного столбца `Sal` таблицы `Employee`. Оператор `SELECT` обеспечивает отбор записей таблицы, а оператор присваивания увеличивает значение суммы `s` на значение столбца `Sal` очередной записи. После перебора всех записей вычисляется среднее, и результат возвращается через выходной параметр `opAvg`.

Часто после слова DO указывается оператор SUSPEND возврата значений, который передает в вызывающее приложение или хранимую процедуру значения выходных параметров, он имеет вид:

```
SUSPEND;
```

Обычно оператор SUSPEND используется в операторе SELECT выбора нескольких записей хранимой процедуры. Совместное использование этих операторов фактически обеспечивает возврат в приложение совокупности записей, представляющих собой набор данных. Поэтому оператор выбора нескольких записей называют также *оператором, возвращающим набор данных.*

Рассмотрим пример:

```
CREATE PROCEDURE pCurrency(ipCurrency VARCHAR(15))
      RETURNS (opCountry VARCHAR(20), opCurrency VARCHAR(20))
      AS
      BEGIN
            FOR SELECT Country, Currency
            FROM Country
            WHERE Currency=:ipCurrency
            INTO opCountry, opCurrency
      DO SUSPEND;
      END
```

Создается хранимая процедура pCurrency, которая возвращает набор данных. Совокупность записей этого набора образуют записи таблицы Country, соответствующие тем странам, в которых используется заданная валюта. Вид валюты задает входной параметр ipCurrency. Входной набор данных (таблица Country) содержит столбцы Country и Currency. Выходной набор данных включает столбцы с именами opCountry и Currency.

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 оператор 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.

Оператор выхода из процедуры EXIT прекращает выполнение хранимой процедуры и осуществляет передачу управления в вызывающую программу или процедуру. Если оператор выхода отсутствует, то завершение процедуры и передача управления в вызывающую программу происходят после выполнения последнего оператора процедуры. Например, оператор

```
CREATE PROCEDURE pDivide (ipA FLOAT, ipB FLOAT)
```

```
      RETURNS (opRes FLOAT)
      AS
      BEGIN
            IF (ipB = 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.

Оператор EXECUTE предназначен для вызова из процедуры другой хранимой процедуры и имеет формат:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
       [RETURNING_VALUES (<Список выходных параметров>)]
```

Этот оператор вызывает хранимую процедуру с указанным именем, входными и выходными параметрами. У вызываемой процедуры параметры могут отсутствовать. Так, оператор:

```
CREATE PROCEDURE pTest
      RETURNS (opRes FLOAT)
      AS
      BEGIN
            EXECUTE PROCEDURE pDivide (10, 30)
                  RETURNING_VALUES (opRes);
      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 10 и 30. Результат, возвращаемый процедурой pDivide, присваивается выходному параметру opRes процедуры pTest.

Оператор POST_EVENT предназначен для отправки сообщения приложениям, связанным с сервером. Сообщение посылается всем приложениям при возбуждении в сервере определенного события. Оператор отправки сообщения имеет следующий формат:

```
POST_EVENT "<имя события>";
```

Например, оператором

```
POST_EVENT "evInsertStore";
```

приложениям, связанным с сервером, посылается сообщение evInsertStore.

Отметим, что для получения соответствующего сообщения приложение должно зарегистрироваться на сервере. Для регистрации приложения и

обеспечения посылки им сообщений можно использовать компонент `IBEvents` со страницы **InterBase** Палитры компонентов.

Триггеры

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 транзакцию, 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 целостности, 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

Создание триггера выполняется оператором `CREATE TRIGGER`, имеющим 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 оператором

```
DROP TRIGGER <Имя триггера>
```

Изменение триггера выполняется оператором `ALTER TRIGGER`, формат которого не отличается от формата оператора создания триггера. После выполнения оператора `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

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 следующим образом:

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:

- 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. Можно выполнить создание таблицы заново с помощью оператора `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 следующим оператором:

```
CREATE GENERATOR <Имя генератора>;
```

Начальное значение задается оператором:

```
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)`.

Механизм 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 оператор `POST_EVENT` отправки сообщения, имеющий 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 приложения на предмет получения нужных сообщений можно воспользоваться специальными функциями API сервера InterBase. Одной из библиотек, является библиотека FIBPlus, компонент SuperIBAlerter которой содержит реализацию таких функций.

В случае доступа к удаленной БД с помощью механизма InterBase Express для регистрации сообщений в приложении можно использовать компонент `IBEvents`, свойство `Database` которого указывает на компонент `IBDatabase`, используемый для соединения с БД.

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 оператор

```
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] <Имя пользователяN>]
       [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 предоставленные 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 Employee TO Manager;
GRANT SELECT, UPDATE (Name, BirthDay) ON Employee TO TopManager;
```

Устанавливаются права доступа к таблице `Employee`. 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.

Удаление привилегий заключается в отмене заданного ранее права доступа. Отмену привилегии выполняет оператор REVOKE, формат которого аналогичен формату оператора GRANT установки привилегии. Отличие заключается в том, что оператор 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 Employee TO Chief;
```

пользователь с именем Chief лишается права доступа к таблице Employee.

Манипулирование данными

Операторы манипулирования данными предназначены для отбора и изменения данных. Отбор записей выполняется с помощью оператора SELECT. Изменение данных заключается в редактировании, удалении и вставке записей, для чего используются операторы UPDATE, INSERT и DELETE соответственно. Применительно к локальным БД указанные операторы рассмотрены в *главе 8,* посвященной доступу к данным с помощью запросов для механизма BDE. В этом разделе мы познакомимся с особенностями отбора записей в удаленных БД с помощью оператора SELECT, который 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 <Вложенный оператор 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 быть 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} (<От-
борM>) |
      EXISTS (<ОтборM>) |
      SINGULAR (<ОтборM>) |
      <Выражение> [NOT] CONTAINING <Выражение1> |
      <Выражение> [NOT] STARTING [WITH] <Выражение1>
```

Элементы, обозначенные как `Отбор`, представляют собой совокупность значений, отобранных с помощью вложенного оператора `SELECT`. Заданное в этом операторе условие отбора записей для элемента `Отбор1` возвращает одно значение или ни одного, а для элемента `ОтборM` — 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 оператора

```
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, оператором

```
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 г. 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 оператора 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`.

Операторы `UPDATE`, `INSERT` и `DELETE` языка SQL сервера InterBase не имеют существенных отличий от своих аналогов, реализованных для локальной версии.

Глава 15

Доступ к удаленным БД с помощью BDE

В *главе 14* нами были рассмотрены общие способы работы с удаленными базами данных на примере сервера InterBase. В этой главе мы рассмотрим особенности работы с удаленными базами данных с помощью механизма BDE.

Управление соединениями с базой данных

В настоящем разделе рассматриваются компоненты, используемые для установления соединения с базой данных (`Database` и `Session`), и техника установления соединения с базой данных из приложения.

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 `AnsiString` 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 `AnsiString` 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 `AnsiString`, 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 `AnsiString` 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.

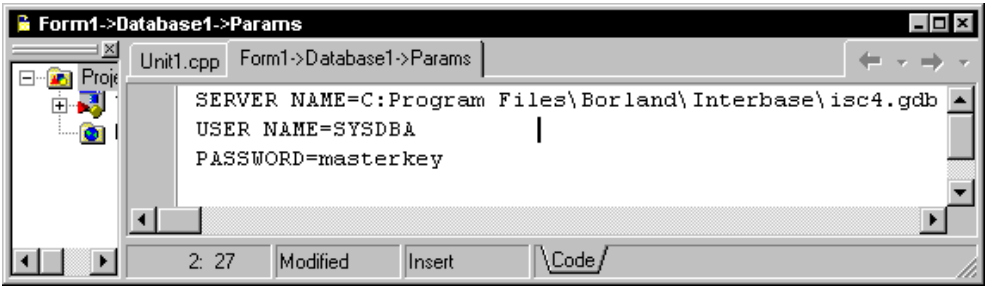


Рис. 15.1. Параметры соединения с БД в окне редактора кода

Нажатие кнопки **Code Editor** преобразует список к текстовому виду и отображает его на отдельной вкладке в окне редактора кода (рис. 15.1). 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**.

В приведенном на рис. 15.1 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);
{

 Edit1->Text = Database1->Params->Strings[1];

}
```

Здесь при нажатии кнопки в поле редактирования (компонент `Edit`) выводится второй параметр соединения.

В частности, для параметров, приведенных на рис. 15.1, 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 установок. Поэтому с помощью свойства `Params` нужно переустанавливать те параметры, значения которых не устраивают разработчика.

Свойство `LoginPrompt` типа `bool` управляет режимом отображения окна ввода имени пользователя и пароля **Database Login** (рис. 15.2). По умолчанию свойство имеет значение `true`, и окно появляется при первом соединении с БД. Если установить свойству `LoginPrompt` 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 `bool` 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 системных затрат на его поддержание. Постоянное соединение разрывается при окончании работы приложения или при программной установке свойству `Connected` значения `false`.

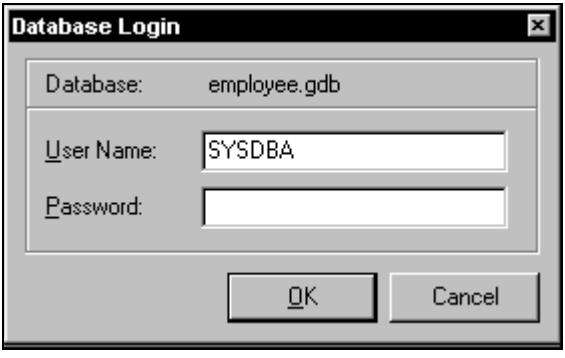


Рис. 15.2. Окно ввода имени пользователя и пароля

На этапе разработки приложения управлять значениями названных свойств можно в окне Редактора параметров соединения (рис. 15.3). Оно вызывается двойным щелчком на компоненте `Database` или вызовом команды **Database editor** контекстного меню компонента.

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. 15.2 и 15.3 в качестве имени базы данных выводится именно значение свойства `DatabaseName`, а не имя (значение свойства `Name`) компонента `Database`.

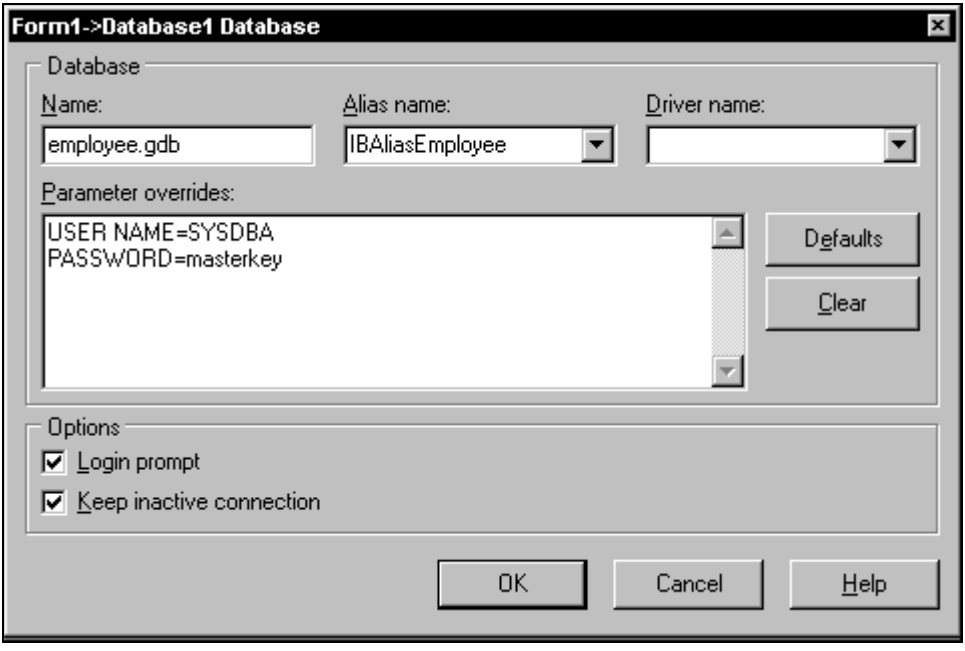


Рис. 15.3. Редактор параметров соединения с БД

Свойство `Connected` типа `bool` определяет, установлено ли соединение с БД. Соединение с БД можно установить при разработке приложения, задав этому свойству 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 `int` определяет число таких наборов, а свойство `TDBDataSet* DataSets[int Index]` 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`).

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 `AnsiString` 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 `int` содержит число таких БД, а свойство `Databases[int Index]` 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 `bool` 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 `AnsiString` 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 `bool` 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. Метод `GetAliasNames(Classes::TStrings* List)` 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:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
Session1->GetAliasNames(ListBox1->Items);
}
```

Здесь нажатие кнопки `Button1` приводит к загрузке в компонент `ListBox1` списка псевдонимов BDE.

Метод `GetAliasParams(const AnsiString AliasName, Classes::TStrings* List)` возвращает параметры псевдонима, имя которого задано параметром процедуры `AliasName`. Если имя псевдонима указано неправильно, то генерируется исключение. Например:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Session1->GetAliasParams("BCDEMOS", ListBox1->Items);
}
```

При нажатии кнопки `Button1` в компонент `ListBox1` заносятся параметры псевдонима `"BCDEMOS"`.

Для добавления и удаления псевдонимов предназначены методы `AddAlias` и `DeleteAlias` соответственно. Метод `AddAlias(const AnsiString Name, const AnsiString Driver, Classes::TStrings* List)` 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:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 Session1->AddAlias(Edit1->Text, "STANDARD", Memo1->Lines);
}
```

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 = C:\data
```

Имя нового псевдонима должно быть уникальным: если указанный псевдоним уже существует, то генерируется исключение.

В качестве примера рассмотрим следующую процедуру:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{  TStringList* Params = new TStringList();
try
{
  Params->Add("SERVER NAME=IB_SERVER:/PATH/DATABASE.GDB");
  Params->Add("USER NAME=MYNAME");
  Session1->AddAlias("IBAlias", "INTRBASE", Params);
}
__finally
{
  delete Params;
}
```

В ней создается псевдоним IBAlias 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 исключений. При их возникновении в любом случае будет удален динамически созданный список Params.

Метод DeleteAlias(const AnsiString Name) удаляет псевдоним, указанный параметром Name. Например:

```
Session1->DeleteAlias(Edit1->Text);
```

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

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 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`).

Собственно соединение с БД выполняется с помощью компонента `Database`, рассмотренного в *разд. "Компонент `Database`"*. Разместим в форме компонент `Database1`. Для соединения с БД, обозначенной псевдонимом `ibAliasEmployee`, свойству `AliasName` компонента `Database1` нужно задать значение `ibAliasEmployee` и свойству `DatabaseName` этого же компонента нужно задать в качестве значения имя базы данных, например Employee.gdb, с которой будет устанавливаться соединение.

Свойству `DatabaseName` компонента `Query1` также следует задать в качестве значения имя той же базы данных Employee.gdb, с которой будет устанавливаться соединение. В этом случае за установление соединения с базой данных будет отвечать компонент `Database1`, а не компонент `Query1`. При этом соединение с БД компонента `Query1` устанавливается следующими способами:

- через открытие набора данных `Query1`;
- путем установки свойству `Connected` компонента `Database1` значения `true`.

Свойству `DatabaseName` компонента `Query1` можно путем выбора из списка задать в качестве значения имя псевдонима `ibAliasEmployee`. Но при этом за установление соединения с базой данных компонент `Query1` будет отвечать сам, а не с помощью компонента `Database1`.

Замечание

Подчеркнем еще раз, что первоначально на сервере InterBase для пользователей есть стандартное имя SYSDBA и пароль "masterkey", которые могут изменяться администратором.

Соединение с БД можно устанавливать при разработке и при выполнении приложения.

Вызов хранимых процедур

Как отмечалось в *главе 14*, различают две разновидности хранимых процедур: процедуры выбора и исполняемые процедуры. Рассмотрим технологию их вызова при использовании доступа к удаленным данным с помощью BDE.

Вызов процедуры выбора

Хранимая процедура выбора возвращает набор данных, состоящий в общем случае из нескольких записей. При ее вызове нужно обеспечить возможность приема совокупности записей. В связи с этим хранимая процедура выбора, как правило, вызывается с помощью оператора выбора `SELECT`, внутри которого размещается вызов процедуры. Например:

```
SELECT * FROM pCurrency('Dollar')
```

Здесь вызывается хранимая процедура `pCurrency` *(см. главу 14)*, возвращающая наименования стран и вид валюты, у которых в качестве валюты используется доллар. Результат выполнения оператора `SELECT` приведен на рис. 15.4.

OPCOUNTRY	OPCURRENCY
USA	Dollar
Greece	Dollar
Fiji	Dollar
Taiti	Dollar

Рис. 15.4. Результат выполнения оператора SELECT

Замечание

При отладке хранимую процедуру выбора в программе IBConsole также можно вызвать оператором `EXECUTE PROCEDURE`, например, так:

```
EXECUTE PROCEDURE pCurrency('Dollar')
```

В этом случае будет получена только первая запись набора данных.

В приложении оператор `SELECT`, вызывающий хранимую процедуру выбора, выполняется с помощью набора данных `Query`, для чего используется его свойство `SQL`.

Рассмотрим пример:

```
void __fastcall TForm1::CallProcClick(TObject *Sender)
{       Query1->Close();
        Query1->SQL->Clear();
        Query1->SQL->Add("SELECT * FROM pCurrency('dollar')");
        Query1->Open();
}
```

При нажатии кнопки `CallProc` свойству `SQL` компонента `Query1` присваивается код вызова хранимой процедуры `pCurrency`, 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 `opCountry` и `opCurrency`.

Замечание

Для соединения с удаленной БД в форме, кроме набора данных `Query`, следует разместить компонент `Database` *(см. главу 13)*.

В приведенном примере SQL-запрос является статическим, т. к. входной параметр, определяющий вид валюты, передается процедуре через явно указанное значение `'dollar'`.

При необходимости значения параметров можно считывать, например, из компонентов `Edit`. Например:

```
Query1->SQL->Add("SELECT * FROM pCurrency('"+Edit1->Text+"')");
```

Вид валюты для страны вводится в поле редактирования `Edit1`.

Значения входных параметров процедуры также можно передавать через параметры компонента `Query1` (свойство `Params`). При этом запрос компонента `Query1` будет иметь вид:

```
SELECT * FROM pCurrency(:ipCurrency)
```

Здесь вид валюты задается одним параметром `ipCurrency`. Этот параметр является входным и предназначен для передачи сроковых значений. Поэтому для него нужно задать следующие значения свойств:

- `DataType` (тип данных) — `ftString`;
- `ParamType` (тип параметра) — `ptInput`.

Управление свойствами параметров осуществляется с помощью Инспектора объектов и Редактора параметров SQL-запроса (рис. 15.5), который вызывается щелчком в области значения свойства `Params` в окне Инспектора объектов. При необходимости для параметров можно задать значения (свойство `Value`).

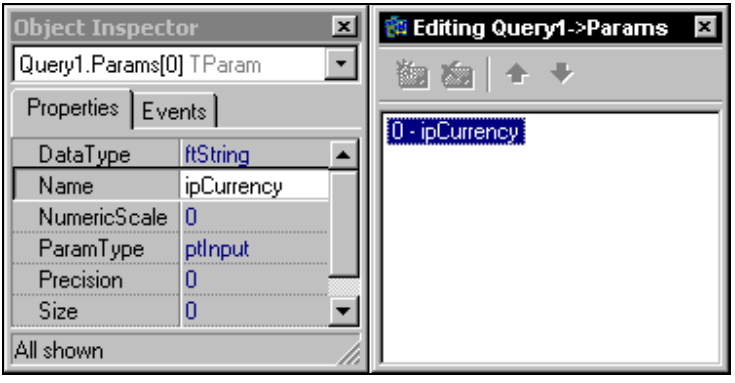


Рис. 15.5. Редактирование параметров SQL-запроса

При выполнении приложения значение параметра можно установить так:

```
Query1->ParamByName("ipCurrency")->Value = "Dollar";
```

или

```
Query1->ParamByName("ipCurrency")->Value = Edit1->Text;
```

Более подробно использование параметров SQL-запроса было рассмотрено в *главе 8*, посвященной доступу к данным с помощью запросов.

Вызов исполняемой процедуры

Исполняемая процедура вызывается оператором следующего формата:

```
EXECUTE PROCEDURE <Имя процедуры> [(<Список входных параметров>)]
```

Оператор вызывает указанную процедуру, передавая ей значения входных параметров. Этот оператор используется в программе IBConsole, при этом результаты работы процедуры (значения выходных параметров) выводятся в окне результатов выполнения запроса.

Например:

```
EXECUTE PROCEDURE pCurrency('Dollar')
```

Как отмечалось, при выполнении приведенного оператора в среде программы IBConsole в качестве результата возвращается только первая запись набора данных.

Для вызова исполняемой процедуры 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`, рассмотренного в *главе 13*.

Свойство `StoredProcName` типа `AnsiString` 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` получает значение `true`.

После задания имени хранимой процедуры 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`.

Приведем пример вызова исполняемой процедуры:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
  StoredProc1->StoredProcName = "pCurrency";
  StoredProc1->Prepare();
  StoredProc1->ParamByName("ipCurrency")->Value = Edit1->Text;
  StoredProc1->ExecProc();
  Edit1->Text = StoredProc1->ParamByName("opCountry")->Value;
  Edit2->Text = StoredProc1->ParamByName("opCurrency")->Value;
}
```

Здесь вызывается исполняемая процедура `pCurrency`. Значение входного параметра `ipCurrency` задается с помощью компонента редактирования `Edit1`. Ее результаты — значения выходных параметров `opCountry` и `opCurrency` — выводятся в компонентах `Edit1` и `Edit2` соответственно. Для вызова процедуры используется компонент `StoredProc1`.

Механизм транзакций

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. Явное управление транзакциями для локальных БД выполняется с помощью методов и свойств компонента `Database`. Средства компонента `Database` можно использовать и для удаленных баз данных. Кроме того, для удаленных БД имеются дополнительные возможности по управлению транзакциями.

Для управления транзакциями служат методы `StartTransaction` (инициирует начало транзакции), `Commit` (подтверждает текущую транзакцию) и `Rollback` (отменяет ее).

Свойство `InTransaction` типа `bool` 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) 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. Далее 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.

Разные серверы баз данных поддерживают разные уровни изоляции транзакций. Если приложение устанавливает свойство `TransIsolation` к неподдерживаемому уровню для удаленного сервера SQL, то BDE использует очередной наиболее высокий уровень, поддерживаемый этим сервером. Для примера в табл. 15.1 приведены уровни изоляции для ряда серверов, распознаваемых BDE.

Таблица 15.1. *Уровни изоляции серверов*

Сервер	Заданный уровень	Фактический уровень
InterBase	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Paradox, dBASE, Access, FoxPro	tiDirtyRead	tiDirtyRead
	tiReadCommitted	Не поддерживается
	tiRepeatableRead	Не поддерживается

Для локальных БД dBase и Paradox допустимым является только значение `tiDirtyRead`. Для сервера InterBase значение `tiDirtyRead` интерпретируется как `tiReadCommitted`.

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 оператор `COMMIT`; разные транзакции могут использовать общее соединение с БД;
- `NOT SHARED` — транзакции должны использовать различные соединения с БД и подтверждаться выполнением оператора `COMMIT`.

Возможность явного управления транзакциями предоставляет язык SQL сервера, в составе которого есть следующие операторы:

- `SET TRANSACTION` (начать транзакцию);
- `COMMIT` (подтвердить транзакцию);
- `ROLLBACK` (откатить транзакцию).

Оператор 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 этого оператора являются необязательными и позволяют управлять перечисленными далее 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 операторов `COMMIT` и `ROLLBACK` по подтверждению и откату транзакции аналогично действию одноименных методов компонента `Database`. Примеры использования транзакций приводятся в следующем подразделе.

Механизм 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. 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 мо-

гут 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 кэшированных 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.

Компонент *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 значение `true` свойствам `CachedUpdates` и `RequestLive`.

Роль компонента `UpdateSQL` заключается в выполнении SQL-запросов, обновляющих необходимые записи связанного с ним набора данных `Query`. Этот компонент позволяет 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`. На этапе разработки приложения удобно использовать для этих целей Редактор свойств компонента `UpdateSQL`, окно которого показано на рис. 15.6. 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.

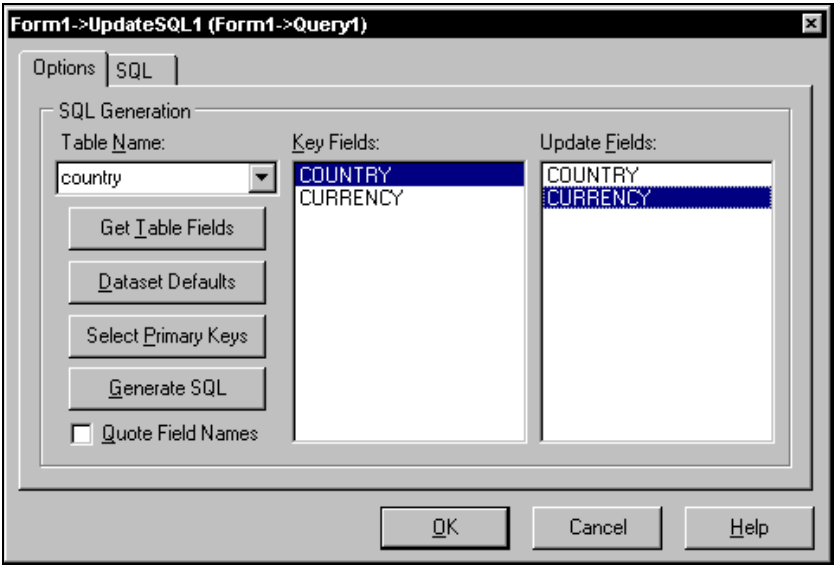


Рис. 15.6. Редактирование свойств компонента `UpdateSQL`

Замечание

При работе с локальными таблицами Paradox нельзя выбирать поля автоинкрементного типа.

Нажатие кнопки **Select Primary Keys** вызывает выделение ключевых полей 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. 15.7), 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.

Напомним, что набор данных `Query` должен быть связан с компонентами `Database` и `UpdateSQL` через свои свойства `DatabaseName` и `UpdateObject` соответственно.

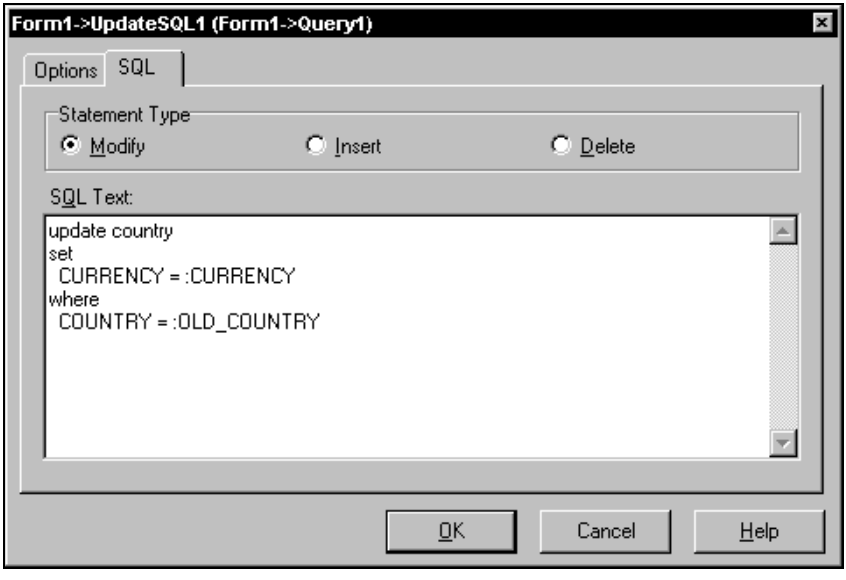


Рис. 15.7. SQL-запросы компонента `UpdateSQL`

Подтверждение кэшированных изменений также можно выполнить последовательным вызовом методов `SetParams` и `ExecSQL`. Метод `SetParams (Up-`

dateKind: 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:

```
typedef void __fastcall (__closure *TUpdateRecordEvent)
   (Db::TDataSet* DataSet, Db::TUpdateKind UpdateKind,
    TUpdateAction &UpdateAction);
```

Параметры этого события отличаются от параметров рассмотренного ранее события OnUpdateError отсутствием параметра E, содержащего объект-исключение.

Пример использования компонента UpdateSQL:

```
void __fastcall TForm1::Query1UpdateRecord(TDataSet *DataSet,
      TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  UpdateSQL1->SetParams(UpdateKind);
  UpdateSQL1->ExecSQL(UpdateKind);
  UpdateAction = uaApplied;
  MessageDlg("Запись обновлена",mtInformation,
                TMsgDlgButtons() << mbOK, 0);
}
```

Параметр UpdateAction принимает значение uaApplied, что соответствует подтверждению изменений в записях.

Последовательный вызов методов SetParams и ExecSQL можно заменить вызовом метода Apply(Db::TUpdateKind UpdateKind), объединяющего их функциональность.

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.

Компоненты *Database* и *Query*

Включением для набора данных режима кэшированных изменений управляет свойство `CachedUpdates` типа `bool`: значение `true` этого свойства включает (активизирует) режим, значение `false` — выключает его (по умолчанию). Для 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...catch` 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

```
void __fastcall TForm1::Button2Click(TObject *Sender)
```

```
{
  Database1->StartTransaction();
  try
    {  // Попытка записи обновлений в БД
       Query1->ApplyUpdates();
       // В случае успеха, подтверждение изменений;
       Database1->Commit();
       MessageDlg("Изменения внесены",mtInformation,
                  TMsgDlgButtons() << mbOK, 0);
    }
  catch (...)
    {
      Database1->Rollback(); // при неудаче отмена изменений
      throw; // Запуск исключения для запрета вызова CommitUpdates
    }
    Query1->CommitUpdates(); // В случае успеха, очистка кэша
}
```

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 TDBDataSet* const * DataSets, const int DataSets_Size)` 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.

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:

```
typedef void __fastcall (__closure *TUpdateErrorEvent)
        (Db::TDataSet* DataSet, Db::EDatabaseError* E,
         Db::TUpdateKind UpdateKind, TUpdateAction &UpdateAction);
```

Список параметров, передаваемых обработчику:

- `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:

```
void __fastcall TForm1::Query1UpdateError(TDataSet *DataSet,
      EDatabaseError *E, TUpdateKind UpdateKind,
      TUpdateAction &UpdateAction)
{
   UpdateAction = uaFail;
}
```

при подтверждении кэшированных изменений набора данных `Query1` отменяются изменения, сделанные во всех записях, которые приводят к ошибке.

При исправлении ошибочных записей нужно иметь в виду, что при доступе к значениям столбцов (полей) набора данных существуют две версии записи: старая и неподтвержденная новая. Для доступа к старому и новому значениям столбца текущей записи предназначены соответственно свойства `NewValue` и `OldValue`.

Приведем следующий пример:

```
void __fastcall TForm1::Query1UpdateError(TDataSet *DataSet,
      EDatabaseError *E, TUpdateKind UpdateKind,
      TUpdateAction &UpdateAction)
{
       Query1->FieldByName("Currency")->NewValue =
             Query1->FieldByName("Currency")->OldValue;
       UpdateAction = uaRetry;
}
```

При редактировании записей набора данных вносились изменения в столбец `Currency`. При возникновении ошибки подтверждения изменений в столбце текущей записи восстанавливается его предыдущее значение. После этого операция сохранения изменений подтверждается. Если ошибочная ситуация в обработчике события `OnUpdateError` не устранена, то при повторных попытках сохранения записи происходит зацикливание программы.

Глава 16

Технология InterBase Express

Общая характеристика

Технология InterBase Express строго ориентирована 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. Является аналогом компонента `Table` для BDE, совместим с визуальными компонентами. Полученный с помощью этого компонента набор данных является редактируемым;
- `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. 16.1), 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** (рис. 16.1) 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.

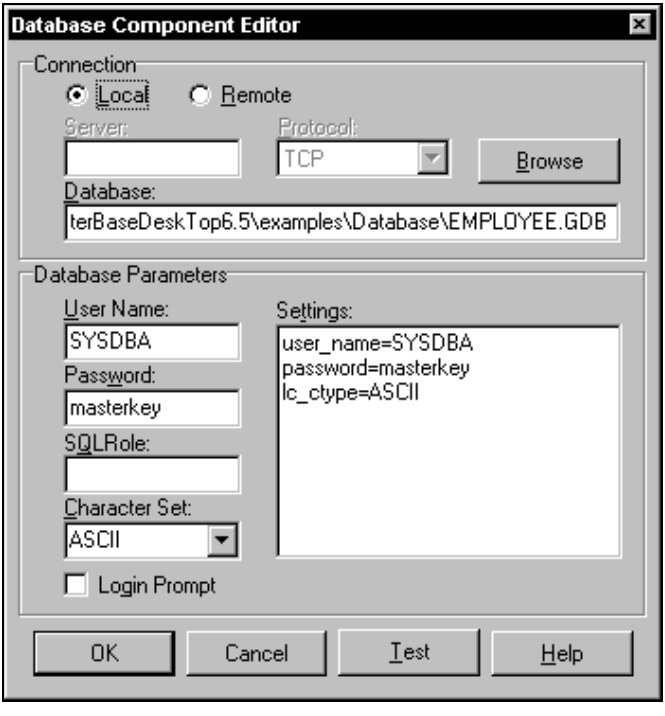


Рис. 16.1. Диалоговое окно **Database Component Editor**

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 соответственно с помощью методов `AddDatabase` и `RemoveDatabase`.

Для запуска, фиксации и отката транзакции соответственно служат методы: `StartTransaction`, `Commit` и `Rollback`. Начиная с 6-й версии сервер InterBase поддерживает два новых метода `CommitRetaining` и `RollbackRetaining`. Они оставляют транзакцию открытой после фиксации и после отката текущей транзакции соответственно.

Перед запуском транзакции для компонента `IBTransaction` 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;
- `taRollbackRetaining` — откат транзакции с сохранением ее контекста (начиная с 6-й версии InterBase);
- `taCommitRetaining` — фиксация транзакции с сохранением ее контекста (начиная с 6-й версии InterBase).

При управлении транзакциями целесообразно установить один из четырех возможных вариантов так называемого *уровня изоляции транзакции*. Сделать это можно с помощью редактора **Transaction Editor** (рис. 16.2). Вызов его осуществляется с помощью одноименной команды контекстного меню компонента `IBTransaction`.

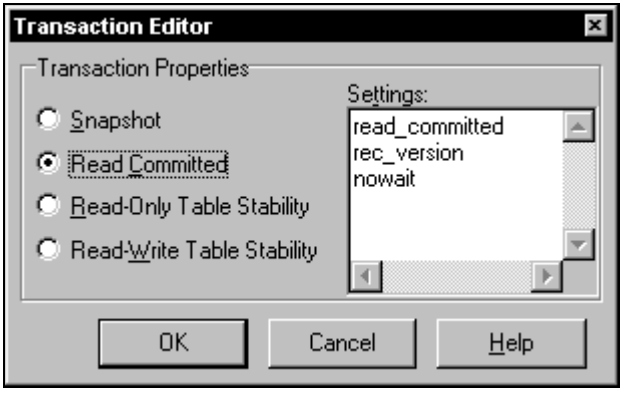


Рис. 16.2. Окно **Transaction Editor**

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 наиболее позднего 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 стандартных компонентов `UpdateRecordTypes` типа `TIBUpdateRecordTypes`, который описан так:

```
type  TIBUpdateRecordTypes = set of (cusModified, cusInserted, cus-
Deleted,
   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. 16.3), вызываемый двойным щелчком мыши в строке свойства окна Инспектор объектов.

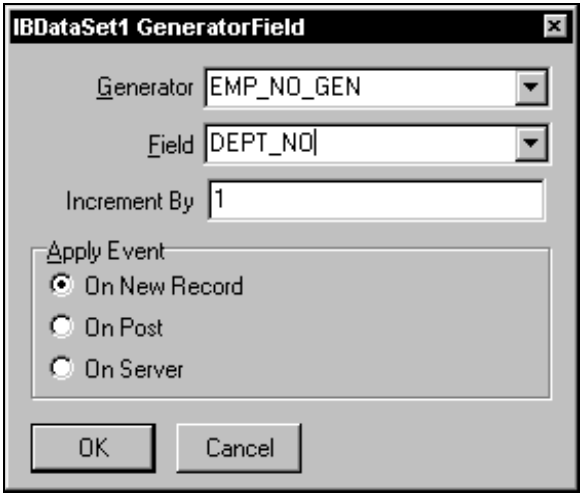


Рис. 16.3. Окно редактора свойства `GeneratorField`

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).

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.

Свойство `Exists` типа `boolean` определяет, существует ли в БД таблица, указанная с помощью свойства `TableName`. 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 ранее компонентов доступа к данным по рассматриваемой технологии, именно его 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`.

Для выполнения запроса на выборку достаточно выполнить следующее. В Инспекторе объектов выполнить двойной щелчок мышью в строке свойства `SelectSQL`. В открывшемся окне редактора запросов **CommandText Editor** (см. рис. 9.3) сформировать нужный оператор SELECT и нажать кнопку **OK**. Непосредственно выполнение запроса на выборку данных осуществляется путем задания свойству `Active` значения `true` или путем вызова метода `Open`. Например, так

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  IBDataSet1->Active=false;
  IBDataSet1->Active=true;
}
```

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 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 со-

ответствующий 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 некоторым 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:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   IBDataSet1->Edit();
   IBDataSet1->FieldByName("COUNTRY")->AsString = Edit1->Text;
   IBDataSet1->FieldByName("CURRENCY")->AsString = Trim(IBDataSet1-
>FieldByName("CURRENCY")->AsString);
   IBDataSet1->Post();
}
```

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. Функция Trim выполняет удаление ведущих и концевых пробелов.

Замечание

Для компонента IBDataSet должны быть обязательно заданы свойства DataBase и Transaction.

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. 16.4) 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**.

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.

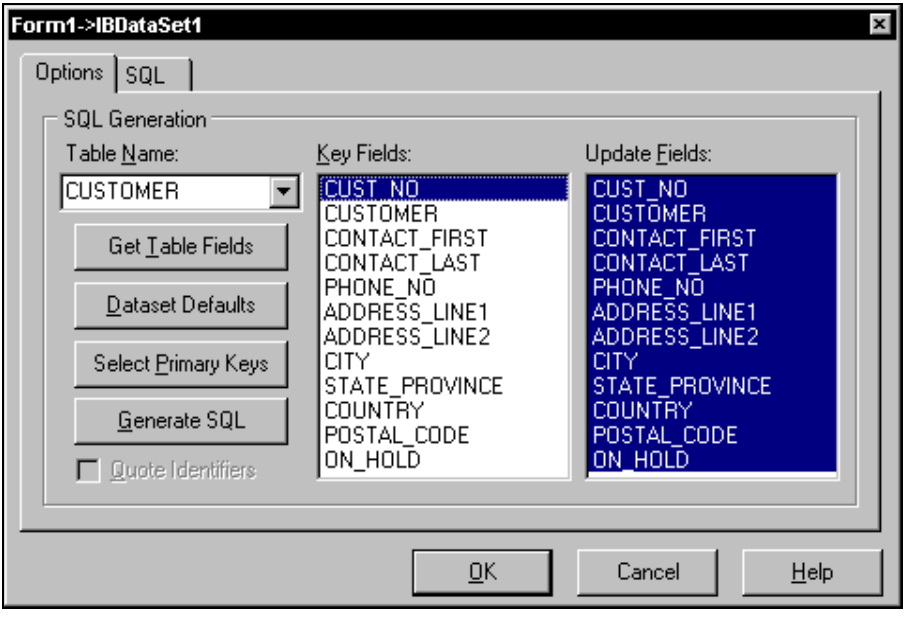


Рис. 16.4. Окно редактора модифицирующих запросов

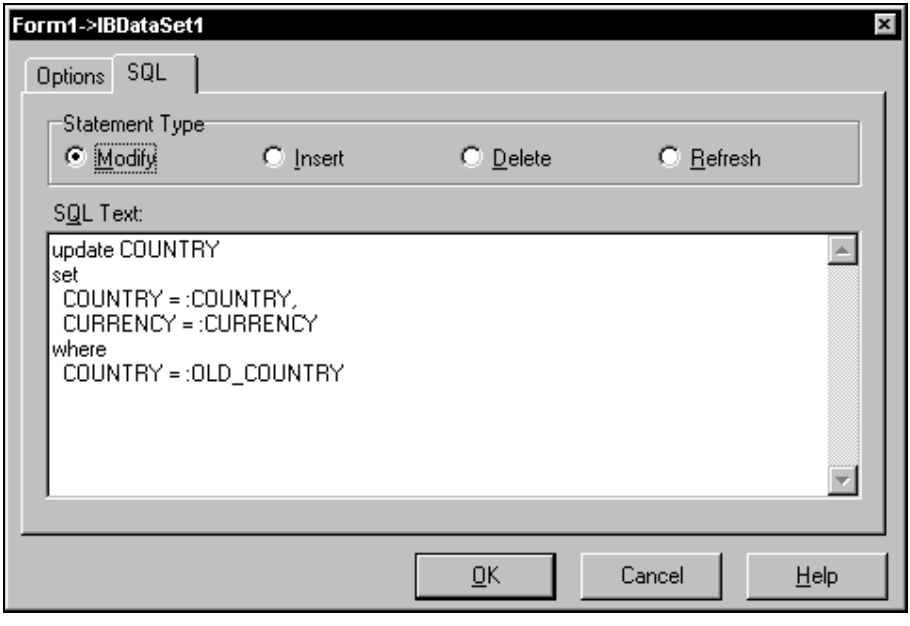


Рис. 16.5. Вкладка SQL окна редактора модифицирующих запросов

Нажатие кнопки **Generate SQL** вызывает генерацию запросов. На вкладке **SQL** окна редактора (рис. 16.5) отображаются тексты автоматически сгенерированных модифицирующих запросов, т. е. значения свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`.

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 приложения, 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 в отличие от `IBDataSet` не производит буферизацию записи. Тем самым мы увеличиваем скорость выполнения приложения.

Пример приложения

Рассмотрим пример приложения (рис. 16.6), построенного с использованием компонентов InterBase Express: `IBDataSet`, `IBDataBase`, `IBTransaction`,

DataSource и нескольких вспомогательных компонентов Edit, DBGrid и Button.

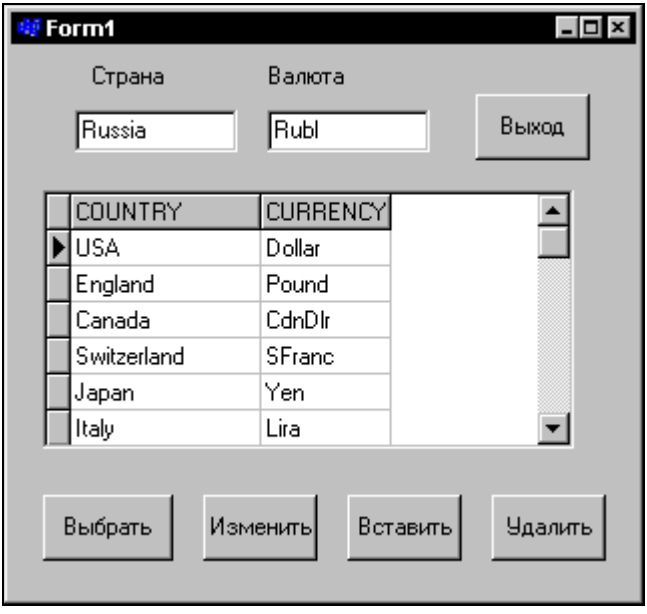


Рис. 16.6. Вид формы при выполнении приложения

В примере демонстрируется возможность выборки, изменения, вставки и удаления записей БД, отображаемой с помощью компонента DBGrid.

Код главного модуля приложения для решения поставленной задачи имеет вид:

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
```

```
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{ // Выборка записей
   IBDataSet1->Active=false;
   IBDataSet1->Active=true;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{ // Изменение записей
  IBDataSet1->Edit();
  IBDataSet1->FieldByName("COUNTRY")->AsString = Edit1->Text;
  IBDataSet1->FieldByName("CURRENCY")->AsString = Trim(IBDataSet1->Field-
ByName("CURRENCY")->AsString);
  IBDataSet1->Post();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{ // Вставка записи
 IBDataSet1->Insert();
 IBDataSet1->FieldByName("COUNTRY")->AsString = Edit1->Text;
 IBDataSet1->FieldByName("CURRENCY")->AsString = Edit2->Text;
 IBDataSet1->Post();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender){ // Удаление записи
  IBDataSet1->Delete();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{ // Выход
  IBDataSet1->Active = false;
  Form1->Close();
}
```

В обработчике события нажатия кнопки `Button1` (с заголовком **Выбрать**) компонент `IBDataSet1` устанавливается активным, тем самым выполняется запрос на выборку всех записей (свойство `SelectSQL`).

В обработчике события нажатия кнопки `Button2` (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 `Button3` (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 `Button4` (с заголовком **Удалить**) с помощью метода `Delete` выполняется запрос из свойства `DeleteSQL` на удаление текущей записи из набора данных.

В рассматриваемом примере подразумевается задание с помощью редактора модифицирующих запросов соответствующих значений свойств `ModifySQL`, `InsertSQL`, `DeleteSQL` и `RefreshSQL` компонента `IBDataSet`, как это было описано при рассмотрении указанного компонента. Естественно, что предварительно должно быть задано свойство `SelectSQL`.

Глава 17

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 этой главе мы познакомимся с программами, предназначенными 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.

Программа IBConsole обеспечивает:

- управление локальными и удаленными серверами;
- управление БД;
- интерактивное выполнение SQL-запросов.

Основную часть окна **IBConsole** занимают две панели (рис. 17.1). 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.

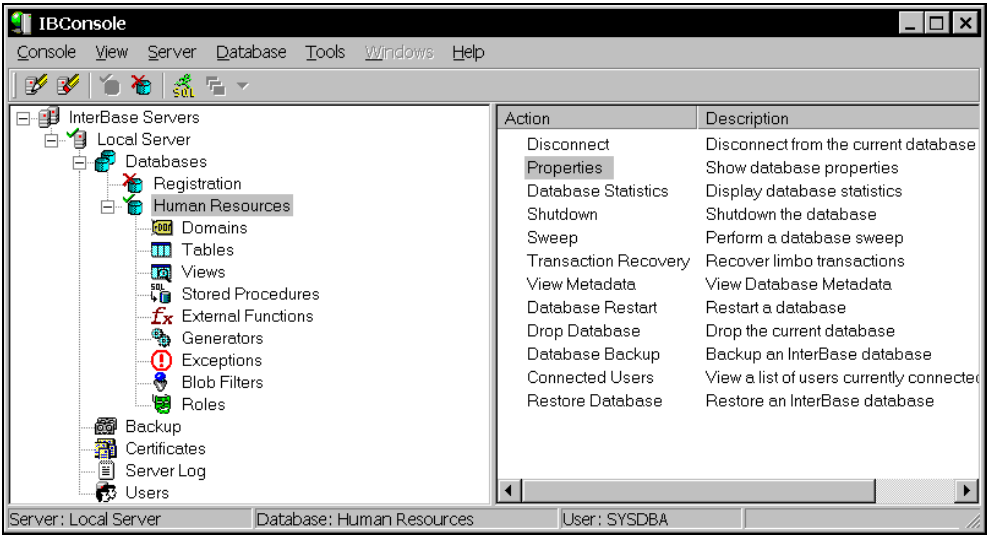


Рис. 17.1. Главное окно программы IBConsole

Управление сервером

Управление сервером включает:

- регистрацию сервера;
- подключение сервера;
- просмотр протокола работы;
- управление сертификатами;
- определение 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. 17.2), в котором необхо-

димо 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. 17.2. Подключение к серверу

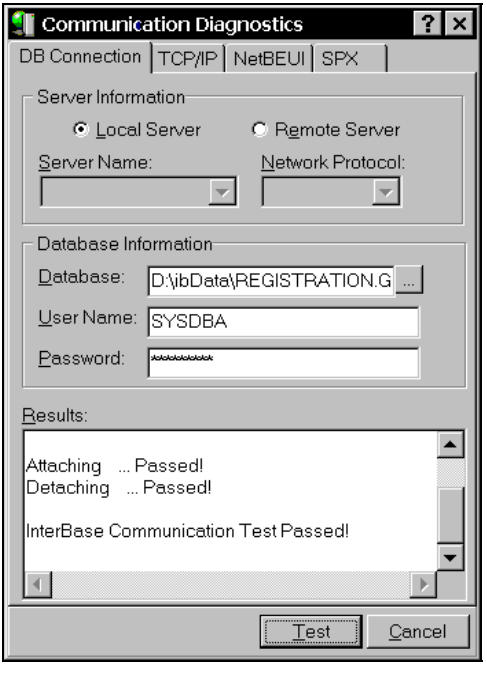


Рис. 17.3. Проверка подключения сервера к БД

После подключения к серверу можно выполнить *проверку подключения* к одной из его БД. Командой **Diagnose Connection** открывается окно проверки соединения **Communication Diagnostics** (рис. 17.3), в котором указывается информация о сервере. В качестве БД задается ее файл, который можно выбрать в окне **Open**, открываемом нажатием кнопки с тремя точками.

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. 17.4).

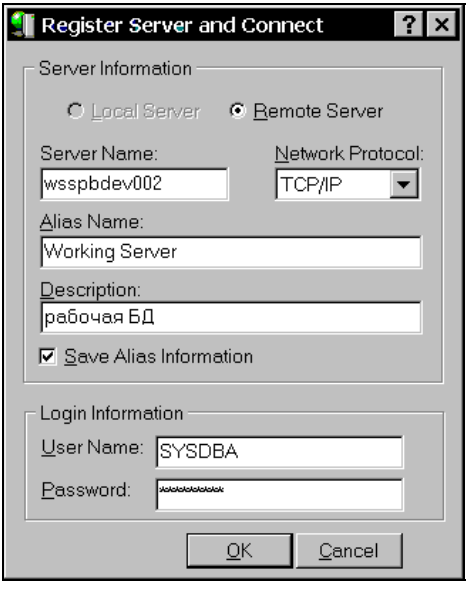


Рис. 17.4. Регистрация сервера

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. 17.5), в котором выводится краткий отчет о работе сервера.

Протокол можно распечатать или сохранить в текстовом файле.

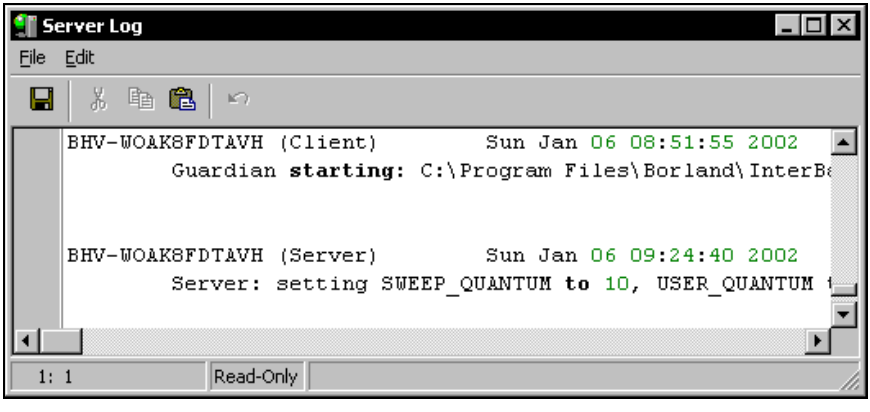


Рис. 17.5. Просмотр протокола работы сервера

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** 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.

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. 17.6), открываемом командой **User Security** или командами контекстного меню списка пользователей.

Рис. 17.6. 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. 17.1 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 включает:

- регистрацию БД;
- подключение БД;
- создание и удаление БД;
- просмотр метаданных;
- проверку состояния БД;
- анализ статистики;
- сохранение и восстановление 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. 17.7). 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.

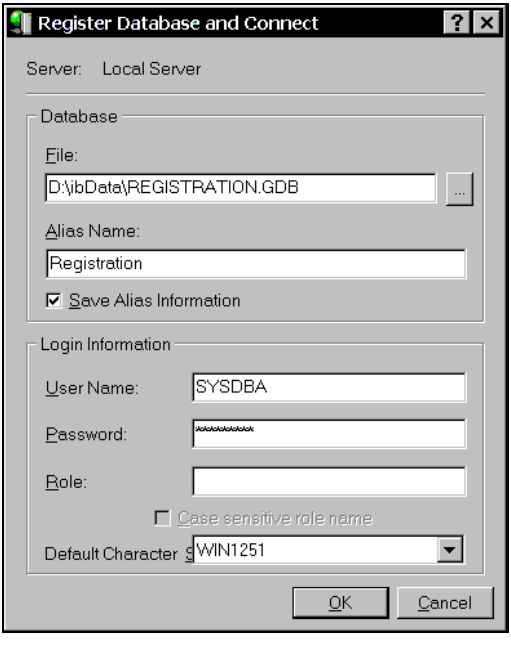


Рис. 17.7. Регистрация БД на сервере

Остальные данные не являются обязательными. 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.

Создание базы данных

Консоль позволяет достаточно удобно и быстро *создавать* БД, в том числе многофайловые. Создание БД, а также ее удаление, для чего 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. 17.8 показаны метаданные базы данных `emloyee.gdb`, которая поставляется вместе с сервером InterBase в качестве примера.

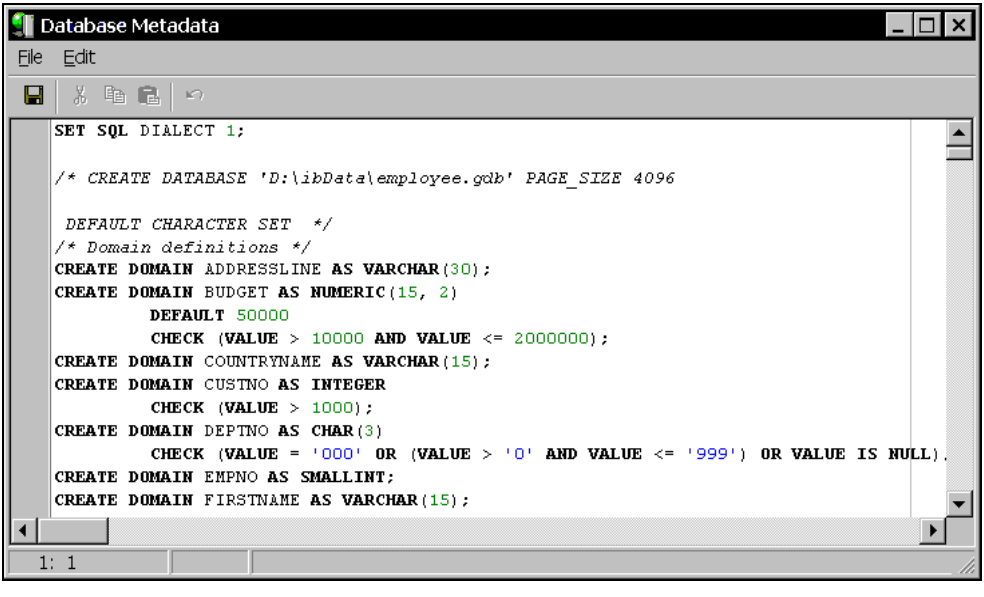


Рис. 17.8. Просмотр метаданных

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" (проводить "сбор мусора").

Удаление мусора можно выполнять в ручном или автоматическом режимах. В ручном режиме удаление мусора начинается командой **Maintenance\Sweep** (Обслуживание\Чистка). В автоматическом режиме удаление мусора начинается, 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 во время чистки 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. 17.9).

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**.

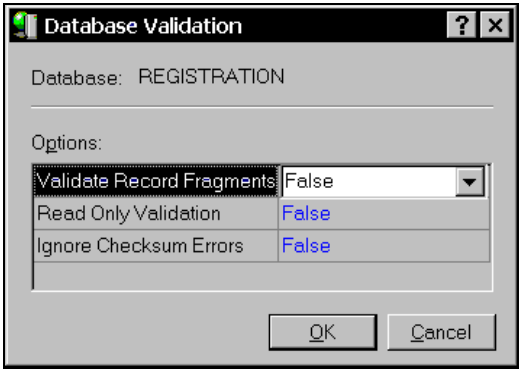


Рис. 17.9. Проверка состояния БД

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. 17.10).

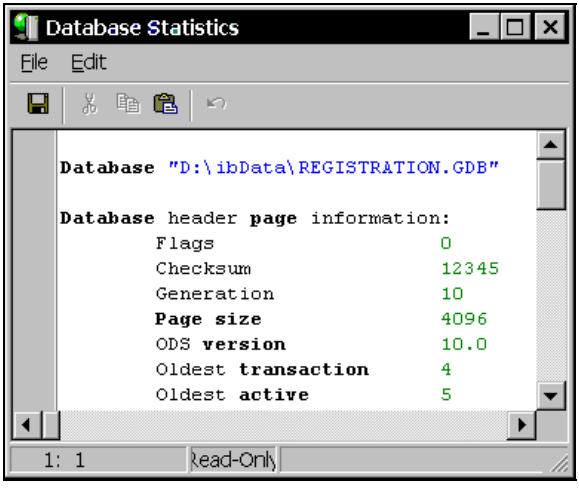


Рис. 17.10. Статистика БД

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. 17.11).

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. 17.11 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;

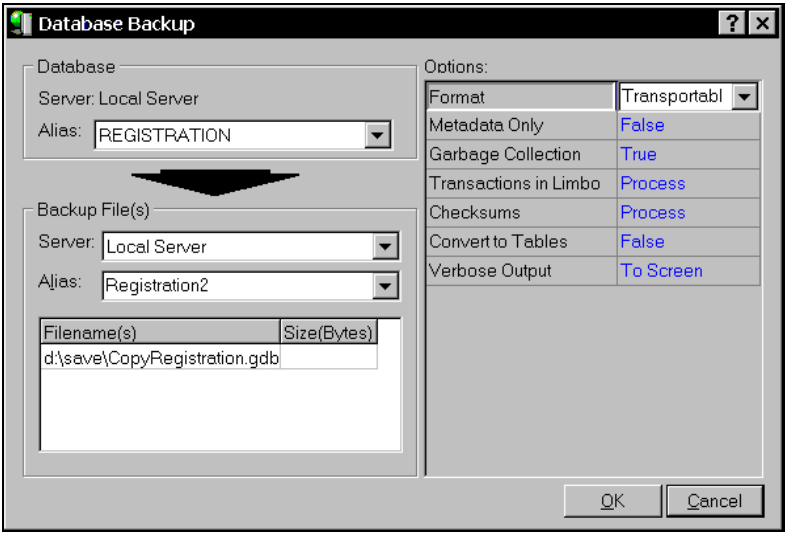


Рис. 17.11. Создание резервной копии БД

- **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 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. 17.12).

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;

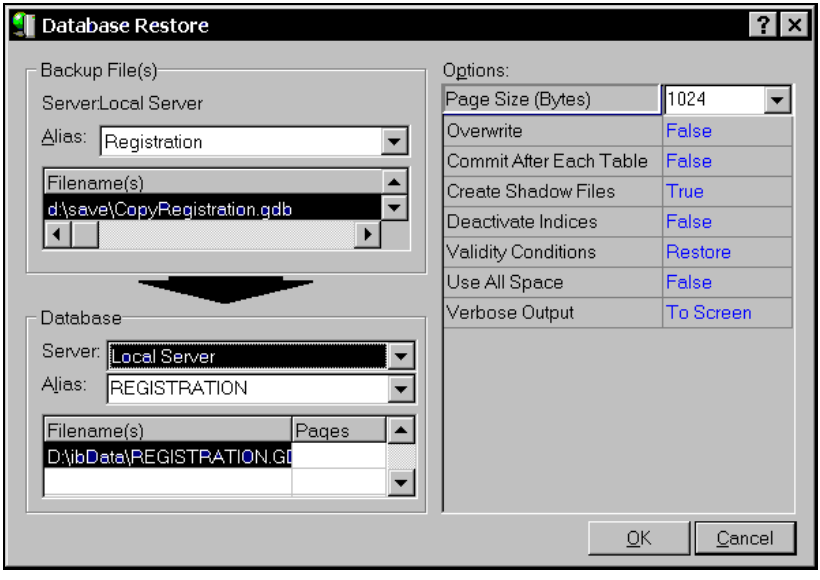


Рис. 17.12. Восстановление БД

- **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 в окне **Database Restore** выводится отчет о результатах.

Интерактивное выполнение SQL-запросов

Консоль IBConsole позволяет в интерактивном режиме выполнять операторы, заданные на языке SQL. Выполнение SQL-запросов и получение их результатов выполняются в окне **Interactive SQL** (рис. 17.13), 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.

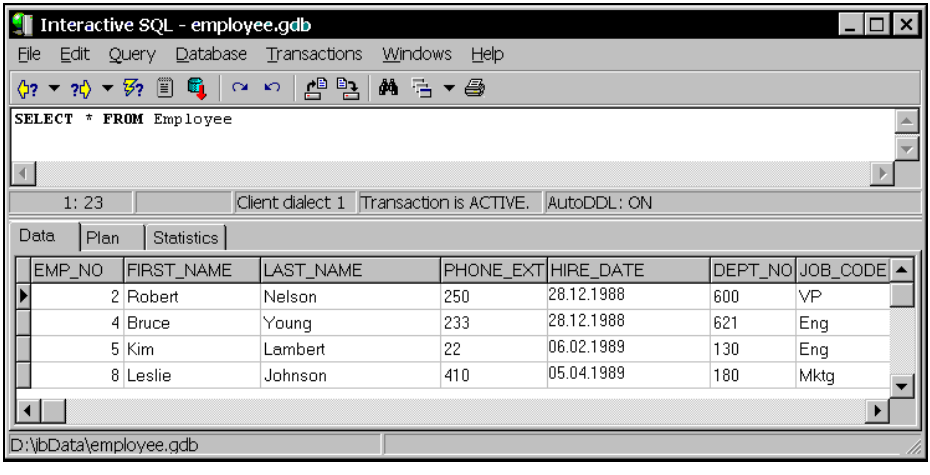


Рис. 17.13. Окно интерактивного выполнения SQL-запросов

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 операторов языка SQL — SQL-запросов. Эти операторы формируются и выполняются автоматически при выборе определенных ко-

манд меню. Так, при создании БД на основании указанных параметров формируется оператор `CREATE DATABASE`.

Кроме того, можно набирать и выполнять операторы вручную. Операторы вводятся в верхней части окна, а в нижней его части выдаются результаты их выполнения.

Операторы можно набирать и выполнять поочередно. Перемещение между отдельными операторами осуществляется нажатием кнопок с изображением желтых стрелок. Кнопка со стрелкой влево выводит в окне предыдущий, а со стрелкой вправо — следующий оператор. Выполнение оператора осуществляется при нажатии кнопки с изображением желтой молнии.

Результаты работы оператора 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 операторов SQL представляет собой сценарий, или скрипт, его удобно использовать для автоматизации операций с БД. Сохранение сценария выполняется командой **Query\Save Script**. Сценарий 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.

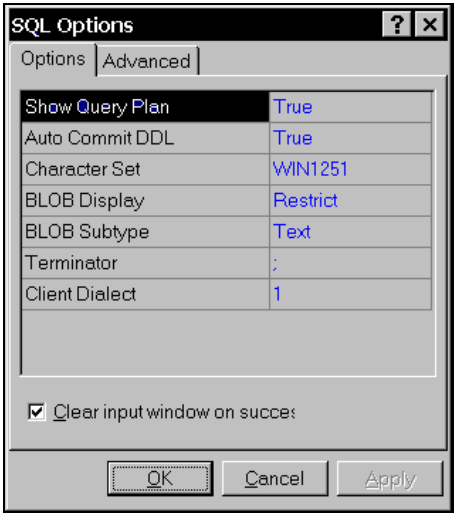


Рис. 17.14. Установка параметров выполнения SQL-запросов

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. 17.14).

На странице **Options** можно задать следующие параметры:

- **Show Query Plan** (Показывать план выполнения запроса);
- **Auto Commit DDL** (Автоматическое подтверждение операций DDL) — операторы определения данных, например, создания таблиц, автоматически подтверждаются, т. е. связанная с операторами 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**. 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 во вводе и выполнении операторов 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, чтобы показать особенности работы с удаленными БД, а также продемонстрировать язык SQL для удаленного сервера.

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 C++ Builder. Программу также можно запустить через меню **Пуск** Windows командой **Программы\Borland C++ Builder 6\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 операторов), ведя журнал работы с удаленной БД. Монитор отслеживает запросы для указанного в команде **Clients** клиента, при отладке C++ Builder-приложения таким клиентом является C++ Builder 6. Строки журнала выводятся в верхней части окна программы (рис. 17.15). 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**.

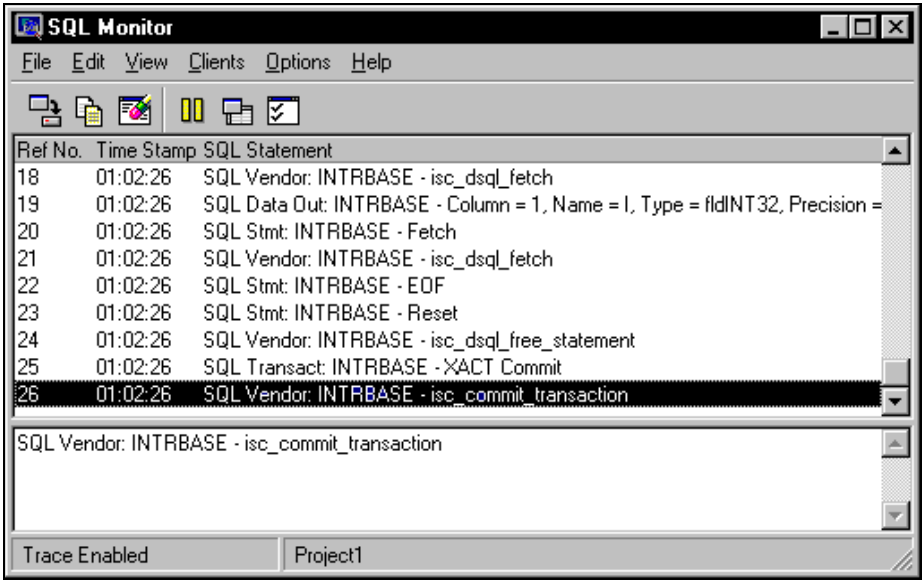


Рис. 17.15. Окно программы SQL Monitor

С помощью параметров Монитора можно регулировать степень подробности информации, заносимой в журнал. Параметры Монитора устанавливаются в окне параметров трассировки **Trace Options**, открываемом командой **Options\Trace Options**. 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 скорее затрудняет проверку выполнения запроса. Поэтому в этом случае можно ограничиться 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 18

Трехуровневые приложения

В трехуровневой архитектуре "клиент-сервер" (в отличие от двухуровневой), 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*, англ. — 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 C++ Builder 6 поддерживается создание многоуровневых приложений, основанных на перечисленных далее 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. 18.1. 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 *провайдером*. Этот интерфейс обеспечивает передачу информации в виде *пакетов данных*. Физически пакеты данных представляют собой совокупности дво-

ичных 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.

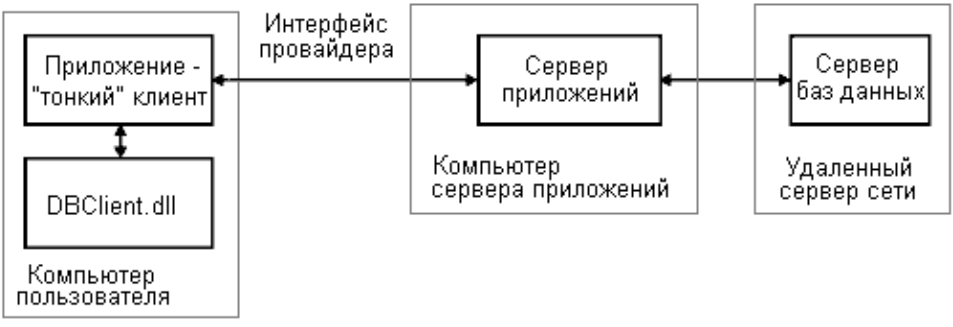


Рис. 18.1. Трехуровневая архитектура типа "клиент-сервер"

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/2000/2003. 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. 18.2).

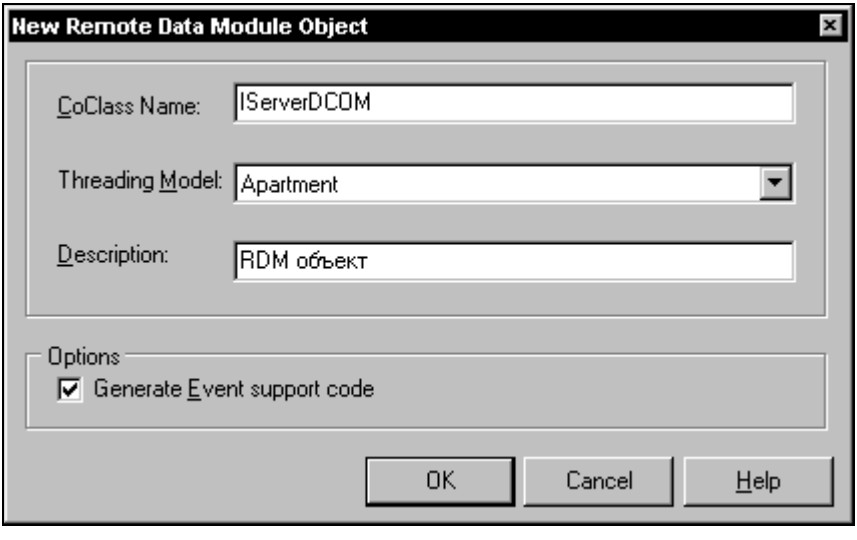


Рис. 18.2. Добавление удаленного модуля данных

В поле **CoClass Name** вводится имя модуля данных и интерфейса объекта автоматизации. Это имя используется в качестве значения свойства `ServerName` компонента соединения приложения клиента, чтобы обеспечить доступ приложения клиента к интерфейсу объекта автоматизации.

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).

В поле **Description** вводится текст, используемый как строка помощи для интерфейса в библиотеке типов.

Флажок **Generate Event Support Code** (Генерировать код поддержки событий) устанавливают, чтобы указать на необходимость реализации интерфейса управления событиями.

После нажатия кнопки **OK** модуль данных с установленными параметрами добавляется к проекту. В приведенном на рис. 18.2 примере модулю присвоено имя `IServerDCOM`.

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 `Query`, `Database`, `Session`, а также провайдер `DataSetProvider`. В самом простом случае достаточно разместить в модуле только набор данных и провайдер. Например, разместим в удаленном модуле набор данных `Query1` и зададим для него значения свойств `DataBaseName` и `SQL` так, чтобы включить в набор все поля всех записей таблицы `Country`. Указанным свойствам присвоим значения:

- свойству `DataBaseName` — значение `IBEmployee` (псевдоним);
- свойству `SQL` — значение `SELECT * FROM Country`;

Кроме того, поместим в модуль компонент `DataSetProvider1` и его свойству `DataSet` зададим значение `Query1`.

На этом создание простейшего сервера DCOM закончено. Перечислим еще раз действия, которые были при этом выполнены:

- к проекту добавлен удаленный модуль данных;
- в модуле размещены компоненты набора данных и провайдера, заданы значения их свойств.

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 C++ Builder. Отметим, что для сервера приложений основная функциональная нагрузка приходится на удаленный модуль 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.

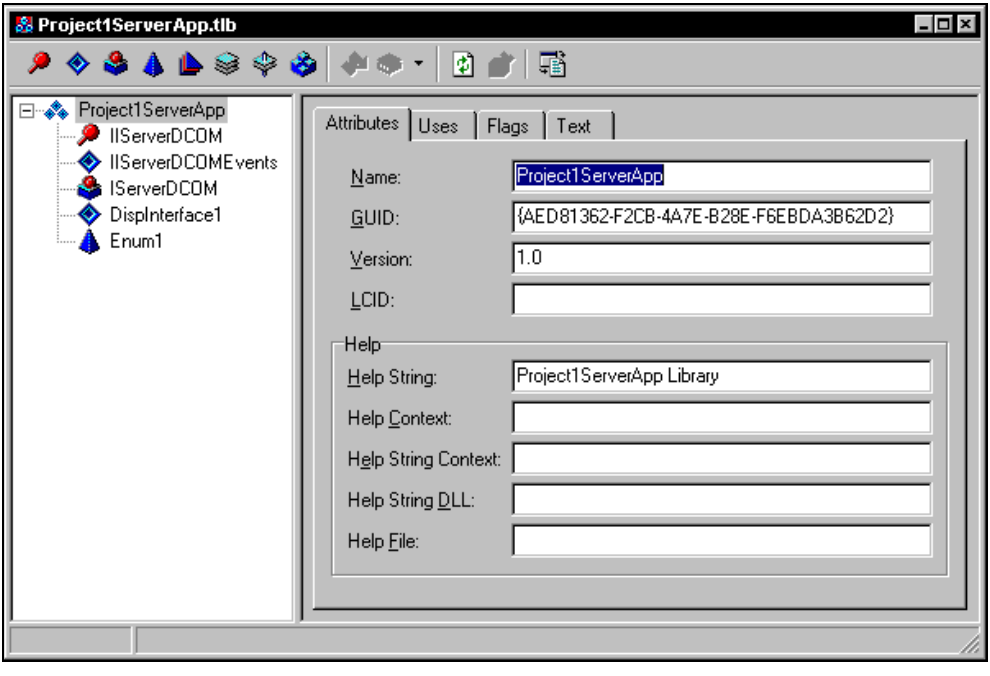


Рис. 18.3. Вид окна библиотеки типов

Библиотека типов (рис. 18.3) создается автоматически, а ее модуль сохраняется на диске при сохранении других файлов проекта. Библиотека занимает два файла с расширениями: tlb (содержит информацию о типах для сервера COM) и res (файл ресурсов).

После создания сервера DCOM его нужно зарегистрировать как сервер автоматизации. Регистрация сервера выполняется системой Windows автоматически при запуске приложения сервера.

Замечание

При необходимости нужно выполнить добавление объекта COM+. Для этого следует задать команду Run/Install COM+ Objects. В одноименном диалоговом окне (рис. 18.4) нужно выбрать имя приложения из списка. При этом сервер приложений также регистрируется автоматически.

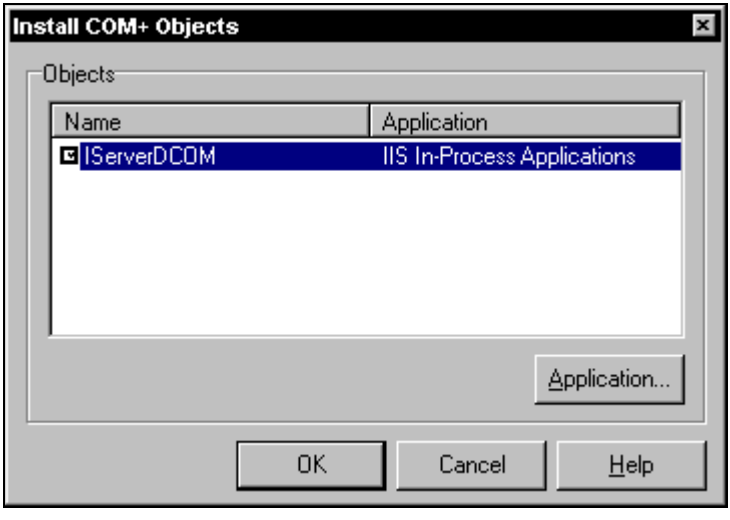


Рис. 18.4. Диалог добавления объекта COM+

По умолчанию интерфейс провайдера обеспечивает набор данных, в нашем случае это `Query1`. Кроме того, C++ Builder включает в свой состав компонент `DataSetProvider` (со страницы **Data Access**), 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 `AnsiString` — код SQL, описывающий ограничение;
- `ErrorMessage` типа `AnsiString` — текст, выдаваемый пользователю при нарушении данного ограничения;
- `FromDictionary` типа `bool` — признак, значение `true` которого указывает, что ограничение выбирается из словаря данных; по умолчанию свойство имеет значение `false`, и словарь данных не используется;
- `ImportedConstraint` типа `AnsiString` — 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. 18.5, справа. Центральную часть окна занимает список ограничений, 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 ограничения оно является "пустым", и в списке выводится название его типа `TCheckConstraint` (рис. 18.5). 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.

В приведенном на рис. 18.5 примере для данных о странах (таблица `Country`) установлено ограничение на значения поля `Country`: оно не может

быть пустым. При нарушении этих ограничений пользователю выдаются соответствующие сообщения: например, если не задано значение поля `Country`, то выдается сообщение `Нет названия`.

Рис. 18.5. Определение ограничений для набора данных `Query1`

Напомним, что 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` (со страницы **Data Access** Палитры компонентов), 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 серверу DCOM, для чего разместим в главной форме компонент `DCOMConnection` со страницы **DataSnap** Палитры компонентов. Основные свойства этого компонента:

- `ComputerName` типа `AnsiString` (имя компьютера, на котором расположен сервер приложений);
- `ServerName` типа `AnsiString` (имя сервера приложений);
- `ServerGUID` типа `AnsiString` (универсальный уникальный идентификатор GUID сервера приложений);
- `Connected` типа `bool` (признак, управляющий активностью соединения).

Для указания компьютера, на котором расположен сервер приложений, удобно использовать окно **Browse for Computer** (Обзор компьютеров) (рис. 18.6), 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, `ProjectServerApp.IserverDCOM`.

Задание имени сервера приводит к автоматической установке для выбранного сервера идентификатора GUID, который присваивается в качестве значения свойству `ServerGUID`. GUID (Globally Unique IDentifier — глобальный уникальный 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.

Чтобы протестировать соединение с сервером приложений, свойству `Connected` устанавливается 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 принимает значение `true` при выборе провайдера для клиентского набора данных `ClientDataSet`.

Рис. 18.6. Выбор сетевого компьютера

Клиентский набор данных `ClientDataSet` предназначен для работы с записями, поступающими с сервера приложений. Перечислим основные свойства этого компонента:

- `RemoteServer` типа `TCustomRemoteServer` (соединение, используемое для связи с сервером);
- `ProviderName` типа `AnsiString` (провайдер, обеспечивающий передачу данных);
- `Active` типа `bool` (признак, указывающий, открыт или закрыт набор данных);
- `PacketRecords` типа `int` (размер пакета данных);
- `FileName` типа `AnsiString` (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 удоб-

но 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 созданного нами сервера приложений расположен компонент провайдера `DataSetProvider1`, предоставляющий интерфейс, имя которого и нужно 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. 18.7 примере в форме приложения размещены компоненты `DBGrid` (сетка) и `DataSource`, свойство `DataSet` которого имеет значение `ClientDataSet1` (имя компонента `ClientDataSet` по умолчанию). После установки свойству `Active` клиентского набора данных значения `true` в сетке отображаются записи набора данных `Country`, отбор которых обеспечивает инструкция `SELECT` набора данных `Query1` сервера приложений.

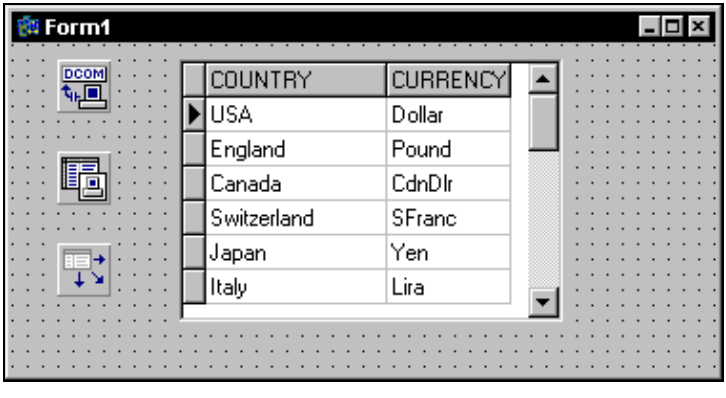


Рис. 18.7. Форма приложения "тонкого" клиента на этапе разработки

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 реализуется с помощью компонента `ClientDataSet`.

Для просмотра состояния текущей записи клиентского набора данных используется метод `Db::TUpdateStatus __fastcall UpdateStatus(void)`, возвращающий следующие значения:

- `usUnmodified` (нет изменений);
- `usModified` (изменена);
- `usInserted` (вставлена);
- `usDeleted` (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 `int`, 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. 18.8 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 `'Наличие/отсутствие изменения в записи'` и `'Число изменений в записях -'` соответственно.

Рис. 18.8. 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`:

```
void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
      TField *Field)
{
   switch  (ClientDataSet1->UpdateStatus())
      {
       case   usUnmodified: {Label1->Caption = "Запись не изменялась";
break;}
       case   usModified:   {Label1->Caption = "Запись изменена"; break;
}
       case   usInserted:   {Label1->Caption = "Запись вставлена"; break;
}
       case   usDeleted:    {Label1->Caption = "Запись удалена";  }
      }
       if (ClientDataSet1->ChangeCount > 0 )
              {ClientDataSet2->Data=ClientDataSet1->Delta;  }
       Label2->Caption="Число изменений в записях - " +
                    IntToStr(ClientDataSet1->ChangeCount);
}
```

В форме расположены следующие кнопки:

- **Обновить** (`Button1`);
- **Отменить** (`Button2`);
- **Сохранить** (`Button3`);
- **Загрузить** (`Button4`);
- **Выход** (`Button5`).

Назначение кнопок и обработчики событий их нажатия будут рассмотрены далее.

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 `int __fastcall ApplyUpdates(int MaxErrors)`. 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 зачастую вызваны конфликтами, связанными с редактированием этих же записей другими клиентами.

Для примера приведем следующую процедуру:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
```

```
  Label3->Caption = IntToStr(ClientDataSet1->ApplyUpdates(-1));
}
```

При нажатии кнопки `Button1` на сервер приложений пересылаются все изменения, сделанные в приложении клиента. Число ошибок, связанных с пересылкой записей, выводится в надписи `Label3`.

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:

```
typedef void __fastcall (__closure *TReconcileErrorEvent)
        (TCustomClientDataSet* DataSet, EReconcileError* E,
         Db::TUpdateKind UpdateKind, TReconcileAction &Action);
```

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`. Состав переключателей может изменяться в зависимости от вида ошибки.

Рассмотрим следующий пример:

```
// Подключение заголовочного файла диалога Reconcile Error Dialog
#include "recerror.h"
...
void __fastcall TForm1::ClientDataSet1ReconcileError(
      TCustomClientDataSet *DataSet, EReconcileError *E,
      TUpdateKind UpdateKind, TReconcileAction &Action)
{
       Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

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 события `OnReconcileError`.

Замечание

Чтобы можно было реализовать вызов функции `HandleReconcileError`, в файл главного модуля приложения (*см. пример*) следует включить предложение препроцессора `#include "recerror.h"`.

Кроме того, в среде C++ Builder для открытого приложения клиента нужно задать команду **Project/Options** и в диалоговом окне параметров проекта на вкладке **Directories/Conditionals** в поле **Include path** задать путь C:\Program

Files\Borland\CBuilder6\ObjRepos. Это связано с тем, что в каталоге ObjRepos находится заголовочный файл recerror.h.

При возникновении ситуации нарушения ограничения FOREIGN KEY, диалоговое окно устранения ошибки имеет вид, показанный на рис. 18.9.

С помощью функции `HandleReconcileError` можно также обрабатывать ошибки конфликта, связанного с обновлением записи. 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` (void) отменяет все изменения, не отправленные на сервер. Так, выполнение процедуры-обработчика

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
   ClientDataSet1->CancelUpdates();
}
```

приводит к тому, что при нажатии кнопки `Button2` все изменения, сделанные в приложении клиента, отменяются.

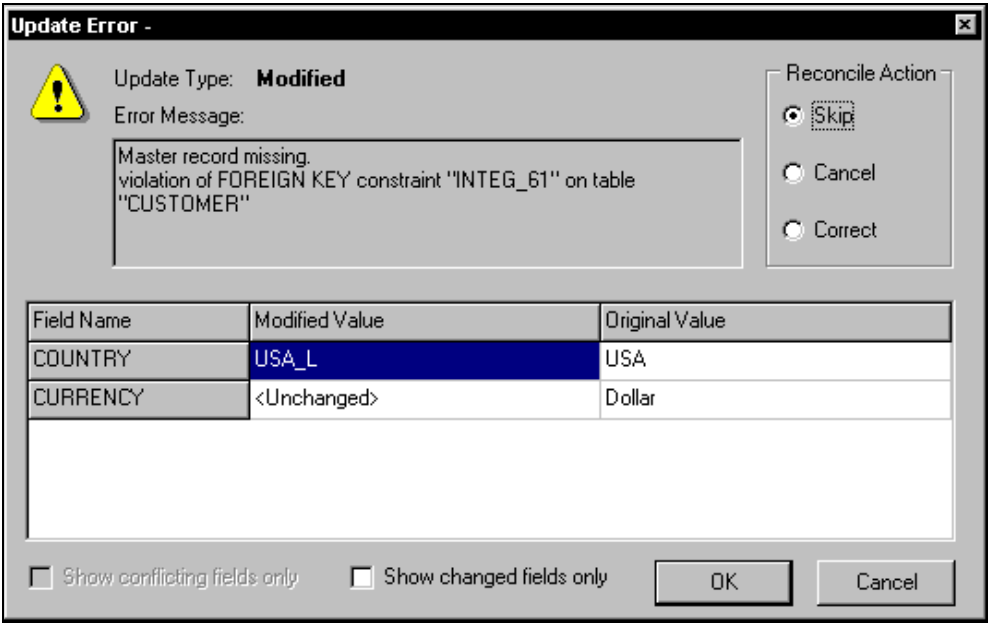


Рис. 18.9. Диалоговое окно устранения ошибок обновления данных

Метод `bool __fastcall UndoLastChange(bool FollowChange)` 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 AnsiString FileName = "")` загружает в клиентский набор данных данные из файла, имя которого указано параметром `FileName`. Если значением параметра является пустая строка, то данные читаются из файла, задаваемого свойством `FileName`.

Метод `SaveToFile(const AnsiString FileName = "", TDataPacketFormat Format = 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 свойству `PacketRecords` клиентского набора данных следует установить значение `-1`.

Рассмотрим в качестве примера обмен данными с диском:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
if (SaveDialog1->Execute())
    ClientDataSet1->SaveToFile(SaveDialog1->FileName);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  if (OpenDialog1->Execute())
       ClientDataSet1->LoadFromFile(OpenDialog1->FileName);
}
```

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 `AnsiString`. Если это свойство задано, то методы `LoadFromFile` и `SaveToFile` вызываются без параметров.

Предметный указатель

A

AddDatabase 420
Alias 447
ALTER TABLE 234, 358
AppendRecord 212
ApplyRange 197

B

Bands 305
BDE 8, 35
BDE Administrator 321

C

CancelRange 197
CancelUpdates 472
ChangeCount 468
CHECK 361
CheckClosed 430
CheckOpen 430
CheckValidStatement 430
Code 19
COLLATE 361
Column 156
Columns Editor 156
COM+ 456
Command 296
CommandText 295, 296, 297
CommandType 267, 295, 296
Commit 265, 420
Connected 289, 419
ConnectionName 263
ConnectionState 265
ConnectOptions 290
Constraints 462
CORBA 456
CREATE DOMAIN 368
CREATE GENERATOR 383
CREATE INDEX 235, 366
CREATE PROCEDURE 371
CREATE TABLE 231, 357
CREATE TRIGGER 379
CREATE VIEW 369

D

Data Controls 145
Database Desktop 59, 331
DataModule 82
DataSource1 79
DB 41
DBChart 171
DBCheckBox 147
DBComboBox 151
DBCtrlGrid 161, 162
DBGrid 152
DBGrid1 79
DBImage 167
DBListBox 151
DBLookupComboBox 152
DBNavigator 166
DBNavigator1 79
DBRadioGroup 148
DCOM 456
DCOMConnection 464
DefaultAction 421
Delete 214
DELETE 254
DROP INDEX 236
DROP TABLE 358

E

EDatabaseError 41
EDBClient 43
EDBEditError 41, 43
Edit 209
EventStatus 290
ExecDirect 268
ExecuteDirect 265
ExecuteOptions 297

F

Filter 193
FindKey 204
ForcedRefresh 423
FOREIGN KEY 365
Frame 304

G

gdb 355
GEN_ID 361, 383
GenerateParamNames 430
GROUP BY 246
GUID 464

I

IB 41
IBConsole 353, 435
IBTransaction 420
IN 244
IndexFieldNames 223
IndexName 223
INSERT 253
InsertRecord 213
Interactive SQL 449
InterBase Express 417
InTransaction 420

L

LIKE 244

M

MasterFields 223
MasterSource 223
MTS 456

N

Number 19

O

ODBC 329
OnConnectComplete 290
OnDisconnect 291
OnDrawColumnCell 157
OnWillConnect 290
Options 307

P

Page 305
Post 210
POST_EVENT 384
Preview 310
PRIMARY KEY 363
PrinterSetting 306
PrintIfEmpty 308

Q

qbe 336
QR 41
QRBand 305
QRDBText 317
QRExpr 317
QRlabel 317
QRShape 314
Query 182
QuickReport 303

R

Reconcile Error Dialog 470
RecordNumber 190
RecordStatus 293
REFERENCES 365
RemoveAllPasswords 74
RemoveDatabase 420
Rollback 420

S

SELECT 236
SetSchemaInfo 268

SHOW INDEX 367
ShowProgress 308
SnapToGrid 308
SOAP 456
SQL 227
SQL Builder 321
SQL Explorer 321
SQL Links 328
SQL Monitor 452
SQLConnection 260
SQLDataSet 266
SQLMonitor 281
SQLPASSTHRU 406
SQLQuery 274
SQLStoredProc 276
SQLTable 273
SQL-запрос 255
StartTransaction 420
StatementType 425
States 297
StoredProc 371

T

Table1 80
TADOCommand 296
TCustomADODataSet 292
TDBError 42
TIBCustomDataSet 422
TIBDataBase 417, 418
TIBDataSet 425
TIBQuery 424
TIBSQL 430
TIBTable 424
TraceList 281

U

UpdateRecordTypes 422
UpdateSQL 408

А

Администратор BDE 321
Архитектура:
  "клиент-сервер" 12, 342
  "файл-сервер" 11, 341
Атрибуты 29

Б

База данных 5
Банки данных 5
Бизнес-правила 28, 346

В

Визуальные компоненты для работы
  с данными 145
Внешний ключ 365
Вызов хранимой процедуры 378, 400
Вычисляемое:
  поле 121
  столбец 359

Г

Генератор 383
Главная:
  таблица 24, 365
  индекс 18, 105
  пароль 72
Глубина индекса 368
Группирование записей 246

Д

Диапазон ограничений 362
Динамическое поле 116
Добавление записей 214
Домен 361, 370
Допустимое значение 75
Доступ:
  к данным 87, 116
  к значению поля 127
  к полю 98
Драйвер 325, 401

З

Заголовок отчета 317
Задание индекса 64
Запись 16
Запрос SQL 109, 334

И

Избыточность данных 46
Изменение:
  данных 102, 141, 387, 412, 449
  поля связи 26
  структуры таблицы 78
Изоляция транзакций 405
Имя:
  индекса 65, 102
  поля 62, 127
  таблицы 100
Индекс 19, 366
Индексные поля 64
Инструкция:
  выбора строки 375
  цикла 375
Инструменты для работы с локальными БД 321
Использование переключателей 148
Источник данных 140
Итоговая полоса 319

К

Каскадное удаление 33, 381
Клиент 342
Ключ 18, 61, 62, 103, 363
Ключевые поля 18, 103
Кодировка символов 326, 354
Колонтитул 319
Компоненты:
  для создания приложений БД 36
  отчета 313
Конструирование запросов 321, 335
Конфигурационный файл 331
Копирование данных 331

Л

Логическая:
  таблица 87, 369
  поле 147
Локальная:
  архитектура 10
  БД 10

М

Маска 67, 138
Метаданные 348
Метод доступа к данным 21
Механизм:
  кэшированных изменений 407
  событий сервера 384
  транзакций 27
Многофайловая БД 355
Модели БД 7
Модификация:
  записей 249
  набора данных 206
Модифицированная сетка 161
Модуль данных 82
Монопольный доступ 102

Н

Набор данных 87, 100, 109
Навигатор 165
Настройка BDE 322

О

Объект поля 115
Ограничение:
  значения 360
  ссылочной целостности 365, 380
  БД 28
  на значения полей 66
Окно:
  обозревателя дерева объектов 35
  редактора кода 35
Описание
  полей таблицы 62
  столбца 359
Определение:
  данных 231
  индекса 106
Отбор:
  данных 236, 387
  записей 336, 376
Отношение подчиненности 24

Отображение данных 142
Отчет 303
  параметры страницы 305
  полосы 305
  предварительный просмотр 310

П

Параметры:
  BDE 322, 393, 396
  индекса 64, 104
Пароль 71
Первичный:
  индекс 18
  ключ 18, 363
Переход по закладкам 186
План выполнения запроса 389
Подчиненная таблица 24, 365
Поиск:
  записей 198
  по индексным полям 204
Поле 16, 98, 115
  выбора 120, 123
  связи 69, 123
Полоса отчета 311
Построение диаграмм 171
Права доступа 71, 385
Представление записей 152
Префиксы имен компонентов 41
Привилегия 385
Приложение-клиент 340
Приложения баз данных 6
Проектирование БД 45
Просмотр 369
  данных 449
Простой и комбинированный списки 151
Простой отчет 316
Псевдоним 323, 333, 393
Псевдофильтрация 200

Р

Разбалансировка индекса 368
Размер поля 62
Регистрация:
  БД 441
  сервера 438
Редактирование записей 208
Редактируемый набор данных 114
Редактор полей 117
Режим набора данных 95, 141
Реляционная БД 15
Реляционный способ доступа 342
Реструктуризация таблицы 358

С

Связывание таблиц 23, 321, 337, 366
Сервер 342
  приложений 457
Система управления базой данных 5
Системные установки 329
Системный администратор 355
Словарь данных 29
Сложный критерий 245
Соединение:
  с базой данных 350, 449
  таблиц 249
Создание:
  базы данных 449
  приложения 79
  таблицы 59, 357, 449
  удаленной БД 353
Сортировка 247
  набора данных 183
  строк 361
Составная инструкция 374
Состояние набора данных 90
Список подчиненных таблиц 78
Способ:
  взаимодействия с сервером 406
  доступа к данным 22
Средства CASE 54
Ссылочная целостность 69
Статическое поле 117
Столбец 343
  сетки 156
Структура:
  базы данных 347
  таблицы 357

Т

Таблица 15
  выбора 75
Текущая:
  запись 144

индекс 102
сеанс работы 395
Технология ADO 283
Тип:
полосы отчета 311
поля 62, 119, 128, 131
столбца 359
таблицы 29, 101
Транзакция 27
Триггер 372, 379

У

Удаление:
записей 214
таблицы 358
Управление:
БД 441
полями 237
Уровень доступа 182
Условие 361
отбора записей 241
Условная инструкция 374

Ф

Фильтрация 192
по диапазону 197
Формат:
поля 140
таблиц 29, 31, 323
Форматирование выводимой информации 139

Х

Хранимая процедура 370

Ш

Шаблон 67

Я

Язык хранимых процедур 372
Языковой драйвер 74, 326