

Министерство сельского хозяйства Российской Федерации
Адамовский сельскохозяйственный техникум - филиал
федерального государственного бюджетного образовательного учреждения
высшего профессионального образования
«Оренбургский государственный аграрный университет»

РАЗРАБОТКА БАЗ ДАННЫХ DELPHI

Учебное пособие

для студентов специальности 09.02.04 Информационные системы (по отраслям)

Адамовка
2014 г.

Одобрено и рекомендовано к печати – директор Адамовского сельскохозяйственного техникума - филиал ФГБОУ ВПО Оренбургский ГАУ В.А. Слободяник.

Рассмотрено и рекомендовано к печати на заседании методического совета филиала «__» сентября 2014 г. (протокол №1) председатель методического совета Л.В. Юрченкова.

Рецензент: Гайфуллина Т.Ф. – преподаватель специальных дисциплин

Киселева С.В.

Практикум по программированию на JavaScript: Учеб. Пособие для сред. Проф. Образования/ Светлана Владимировна Киселева. – Адамовская типография ГУП РИА «Оренбуржье»

Методические рекомендации предназначены в помощь студентам специальностей 09.02.04 Информационные системы (по отраслям) по выполнению выпускной квалификационной работы по теме «Разработка базы данных». Пособие содержит сведения по созданию приложений баз данных в среде визуального программирования Delphi. Рассматриваются навигационный и реляционный способы доступа к данным. Значительное внимание уделено конструкциям структурированного языка запросов SQL. Приводятся основные механизмы доступа к данным. Изложение материала сопровождается большим количеством примеров, которые могут быть полезны как при выполнении лабораторных работ, так и при решении реальных задач. Методика изложения материала позволяет использовать пособие для самостоятельной работы при наличии знаний по применению системы Delphi.

ПРЕДИСЛОВИЕ

В пособии рассматриваются некоторые вопросы применения системы программирования Delphi для разработки приложений баз данных. Выполнен обзор средств Delphi, используемых при создании программ для работы с базами данных: утилит, модулей данных, визуальных и невизуальных компонентов.

В Delphi реализованы разнообразные технологии доступа к данным, однако общие подходы и последовательность действий при разработке приложений баз данных почти одинаковы. Сначала достаточно подробно рассмотрена структура приложения, использующего Borland Database Engine, а затем приведены возможности и особенности других механизмов доступа к данным.

В пособие включены вопросы использования как навигационного, так и реляционного способа работы с данными. Навигационный способ работы с данными предполагает выполнение операций с отдельными записями, прост и логичен, его освоение позволяет закрепить навыки программирования, получить опыт создания полнофункционального приложения в системе Delphi. Достоинством этого способа является простота задания операций. Как правило, навигационный способ работы с данными используют в локальных базах данных. Однако в настоящее время навигационный подход уже не может составить конкуренцию реляционному способу работы с данными, поэтому значительное внимание уделено конструкциям структурированного языка запросов SQL и созданию в Delphi приложений, использующих SQL. Достаточно подробно рассматривается оператор SELECT и правила формирования сложных SQL-запросов.

Пособие создано для студентов первого курса, изучающих дисциплины «Управление данными», «Информационное обеспечение систем управления», и учитывает уровень их подготовки в области программирования и информационных технологий.

1. СРЕДСТВА DELPHI ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ

1.1. Borland Database Engine

Традиционным для системы Delphi способом работы с базами данных является использование процессора баз данных Borland Database Engine (BDE). Разработанный фирмой Borland унифицированный программный интерфейс BDE позволяет выполнять доступ к данным как с использованием традиционного record-ориентированного (навигационного) подхода, так и с использованием set-ориентированного (реляционного) подхода, принятого в SQL-серверах баз данных. Универсальный механизм доступа к данным, которым является BDE, применяется в средствах разработки фирмы Borland (Delphi, C++ Builder), а также в некоторых других продуктах.

BDE устанавливается вместе с Delphi, обеспечивает доступ к локальным базам данных, расположенным на том же компьютере, и к удалённым базам, расположенным на сервере. BDE предоставляет очень гибкий механизм управления базами данных, позволяющий приложениям, созданным в среде Delphi, получать информацию из баз данных наиболее популярных форматов.

BDE представляет собой набор динамических библиотек и драйверов, обеспечивающих доступ к данным. В составе BDE поставляются стандартные драйверы, обеспечивающие доступ к базам данных Paradox, dBase, FoxPro и текстовым файлам. Эти драйверы устанавливаются автоматически вместе с ядром процессора. Доступ к данным серверов SQL обеспечивает отдельная система драйверов Borland SQL Links. Эти драйверы нужны при разработке приложений для серверов Oracle, Sybase, Informix и InterBase. Драйверы SQL Links необходимо устанавливать дополнительно. Кроме того, в BDE есть возможность подключения любых драйверов ODBC. Подробный состав BDE, параметры драйверов и сведения по настройке можно найти в литературе [3].

В Delphi реализовано достаточно большое количество технологий доступа к данным (гл. 6), однако общие подходы и последовательность действий при разработке приложений баз данных почти одинаковы.

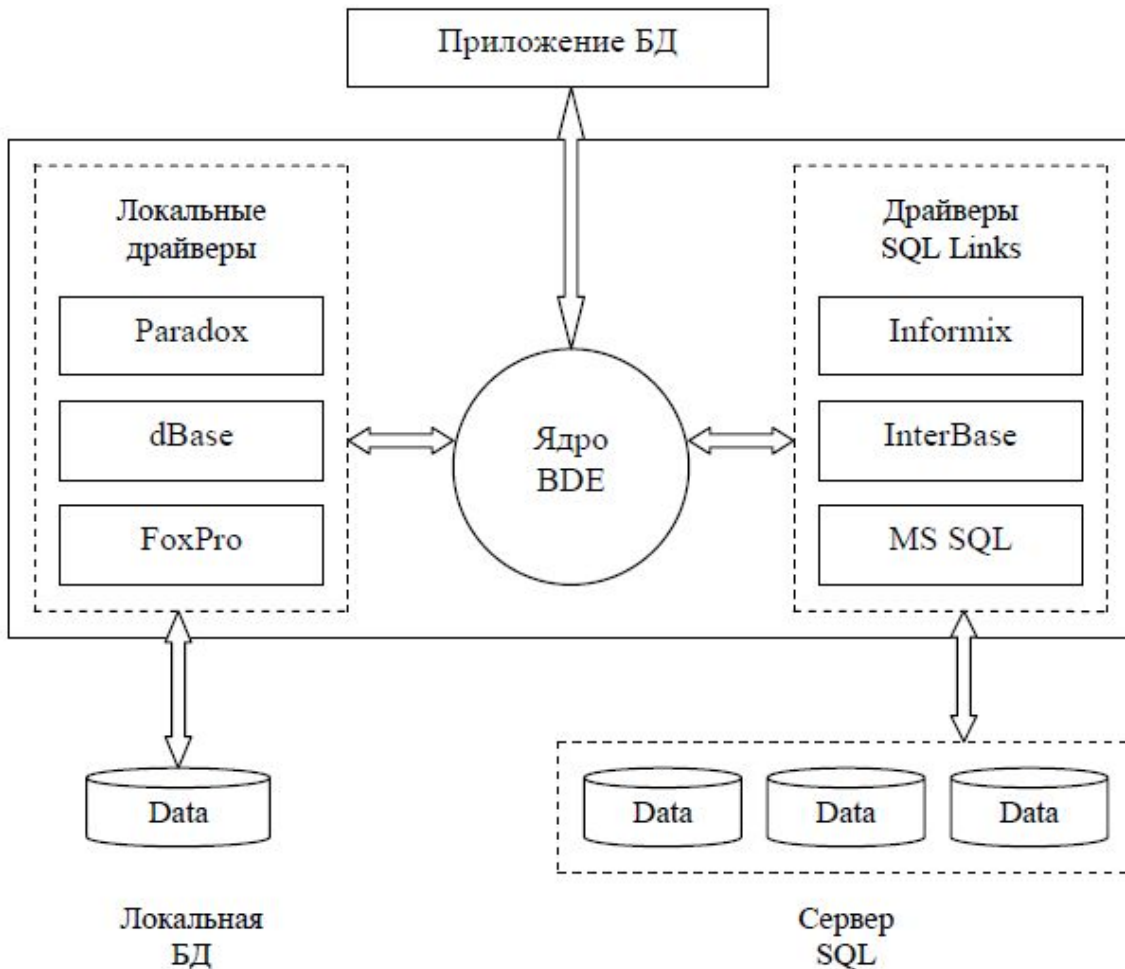


Рис. 1. Схема доступа к базе данных через BDE

BDE играет роль связующего звена (рис. 1) между базой данных и приложением, выполняет низкоуровневую работу. BDE не является частью программы, может располагаться на машине клиента или на сервере.

При разработке приложения программист может использовать низкоуровневый интерфейс BDE или применять компоненты, упрощающие разработку программ.

1.2. Утилиты для работы с базами данных в Delphi 1.2.1. BDE

Administrator

Утилита BDE Administrator (*bdeadmin.exe*) предназначена для конфигурирования BDE, позволяет устанавливать параметры псевдонимов баз данных, драйверов и параметры, общие для всех баз данных. Настройки BDE сохраняются в файле *idapi32.cfg*. Окно программы содержит две области (рис. 2). В левой области расположены страницы **Databases**

и **Configuration**. Правая область используется для вывода сведений об объекте, выбранном слева. На странице **Configuration** приведены сведения о драйверах баз данных и установках BDE. На странице **Databases** приведены псевдонимы имеющихся на компьютере баз данных. Здесь же можно создавать и редактировать псевдонимы.

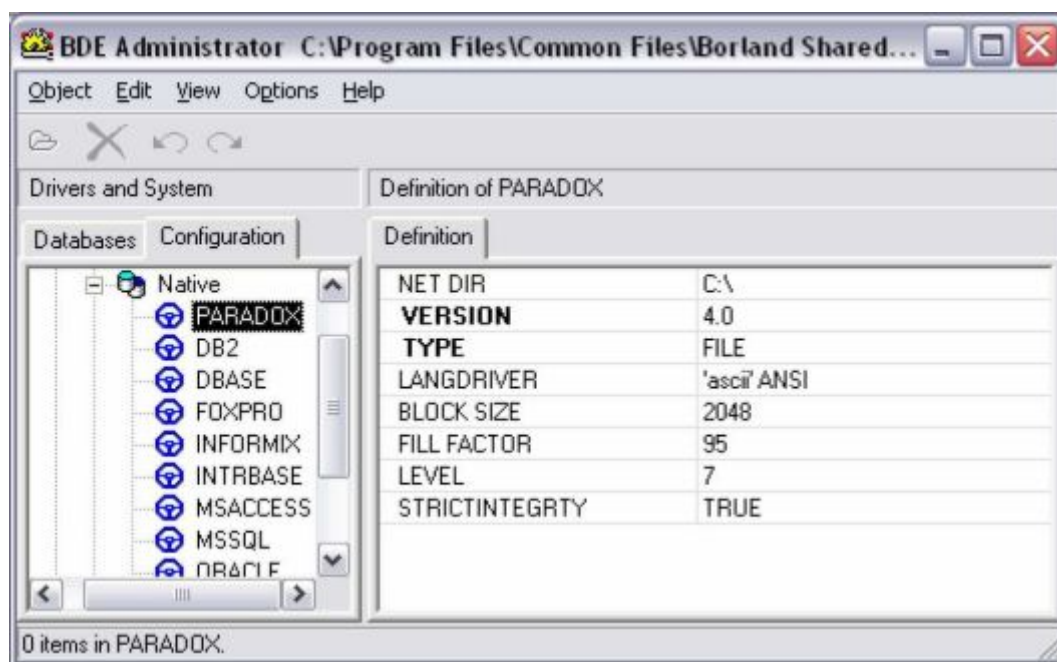


Рис. 2. Окно утилиты BDE Administrator

1.2.2. SQL Explorer

Универсальная утилита SQL Explorer (*dbexplor.exe*) позволяет:

- просматривать, создавать, корректировать псевдонимы;
- просматривать структуру и содержимое таблиц баз данных;
- редактировать таблицы;
- формировать запросы на языке SQL;
- создавать словари данных.

SQL Explorer (рис. 3) выполняет функции проводника по базам данных: позволяет узнать, из каких таблиц состоит БД, получить сведения о полях таблиц, индексах, ссылочной целостности и др. Является удобным средством для просмотра и редактирования таблиц. Кроме того, позволяет быстро создавать интерфейс приложения, перетаскивая мышью поля

на форму. Работа с псевдонимами в проводнике выполняется так же, как в утилите BDE Administrator. В SQL Explorer удобно тестировать SQL-запросы.

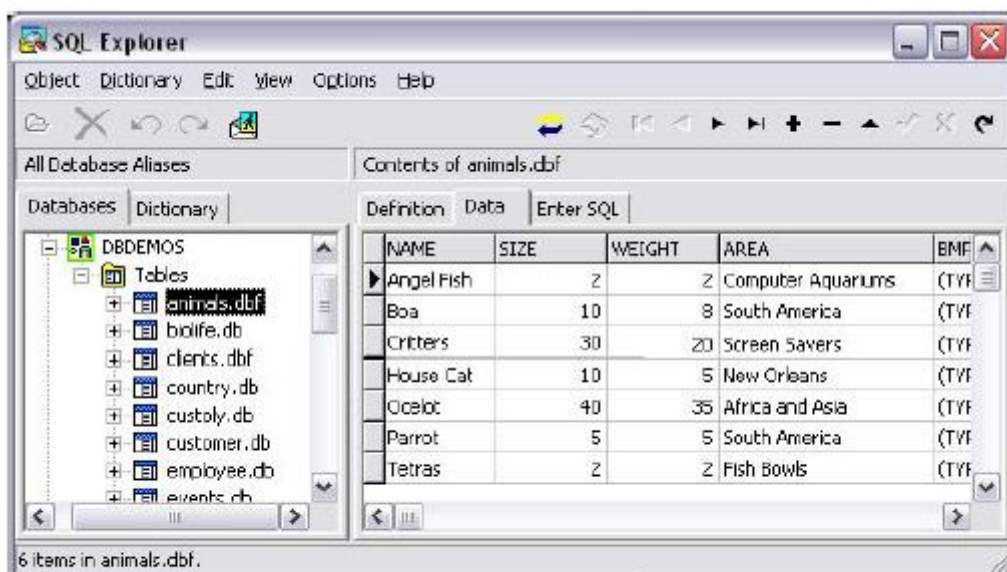


Рис. 3. Окно утилиты SQL Explorer

1.3. Псевдонимы

При работе с базами данных во многих случаях удобнее пользоваться *псевдонимами*, а не просто указывать путь доступа к таблицам базы данных. *Псевдоним (alias - алиас)* - это известное разработчику и BDE имя базы данных. В BDE с псевдонимом ассоциируются параметры, используемые для соединения с базой данных: формат БД, путь к её файлам, языковой драйвер, имя сервера, имя пользователя, режим открытия и т.п.

Псевдоним сохраняется в отдельном конфигурационном файле на диске и позволяет исключить из программы прямое указание пути доступа к базе данных. Такой подход даёт возможность располагать данные в любом месте, не перекомпилируя при этом программу.

Для создания псевдонима в утилитах BDE Administrator и SQL Explorer необходимо выполнить следующие действия:

- на левой панели выбрать страницу **Database**;
- через всплывающее меню или меню **Object** выбрать команду **New**;
- в окне **New Database Alias** (рис. 4) выбрать драйвер для работы с БД и нажать **ОК**.
При работе с БД Paradox выбрать Standard;

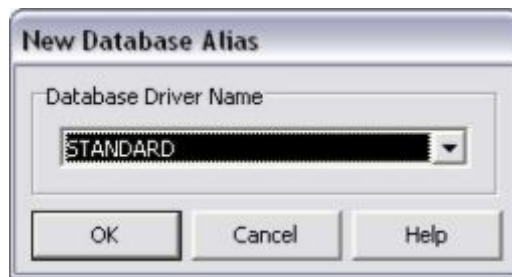


Рис. 4. Окно для выбора драйвера

на левой панели записать имя;

на странице *Definitions* (правая панель) в поле *Path* (рис. 5) указать путь к файлам БД; щёлкнуть на строке *Path* и с помощью кнопки обзора найти нужную папку; через всплывающее меню для левой панели или меню **Object** выбрать команду

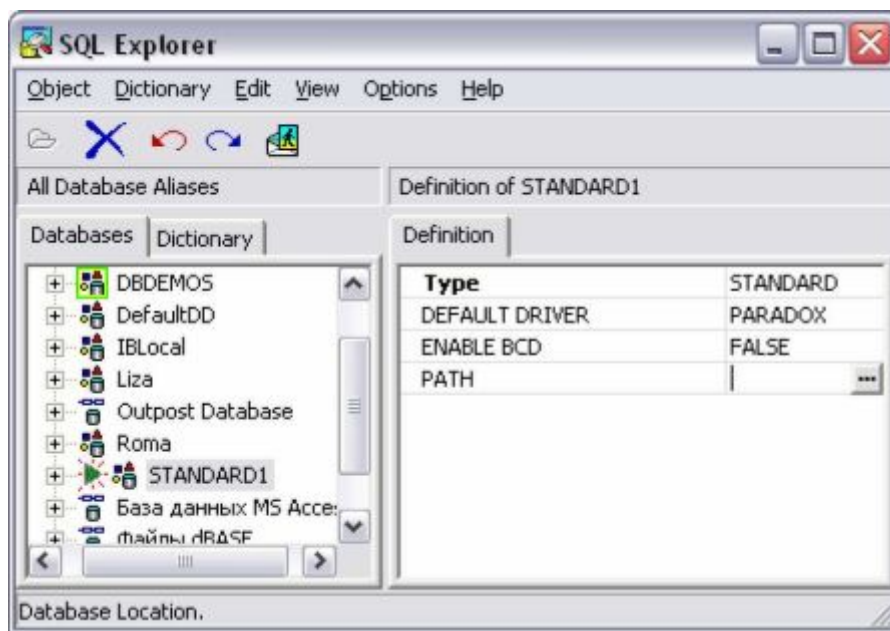


Рис. 5. Задание расположения базы

Apply.

Дополнительная информация, сообщаемая при создании псевдонима, зависит от типа выбранной базы данных. После создания нового псевдонима его имя вносится в общий список псевдонимов.

1.4. Обзор компонентов для работы с базами данных

Система Delphi содержит богатую библиотеку компонентов, значительно упрощающих разработку приложений для баз данных. Компоненты освобождают программиста от работы на нижнем уровне, позволяют быстро создавать надёжные приложения.

Набор компонентов и распределение их по страницам зависят от версии системы Delphi. В версиях до седьмой включительно при разработке любого приложения были доступны все имеющиеся компоненты. В современных версиях предусмотрена возможность создания приложений для платформ Win32 и Net. Кроме того, разные виды приложений используют разные наборы компонентов.

Все приводимые в пособии примеры предполагают использование обычных приложений Delphi, чтобы их можно было выполнить как в Delphi 7, так и в Delphi 2005 и Delphi 2006. В современных версиях системы такие приложения называют VCL Form Application - Delphi for Win 32. При таких условиях компоненты, используемые для работы с БД, находятся на страницах:

Data Access - не визуальные компоненты, предназначенные для организации доступа к данным;

Data Controls - визуальные компоненты для отображения данных;

dbExpres - компоненты для создания приложений, использующих технологию dbExpress;

BDE - компоненты для создания приложений, использующих BDE;

ADO (Delphi 7) или **dbGo** (Delphi 2005 и Delphi 2006) - компоненты для создания приложений по технологии ADO;

InterBase - компоненты для работы с сервером InterBase.

Таким образом, в Delphi предусмотрены специальные наборы компонентов, обеспечивающие доступ к данным при использовании разных технологий, и наборы компонентов, отображающие данные. Компоненты доступа к данным позволяют осуществлять соединения с БД, производить выборку, копирование данных и т. п.

Компоненты для отображения данных позволяют выводить данные в виде таблиц, полей, списков. Отображаемые данные могут быть текстового, графического или произвольного форматов.

Компоненты для работы с базами данных можно разделить на три группы: множества данных (data sets);

визуальные компоненты баз данных (data-aware controls) и источники данных (data sources).

Множества данных - это невидимые компоненты, которые взаимодействуют с **VDE** и обеспечивают доступ к данным в таблицах. Наиболее важные из них - компоненты **Table** и **Query**.

Визуальные компоненты баз данных - это управляющие элементы пользовательского интерфейса для просмотра и редактирования данных. Многие из них дублируют обычные управляющие компоненты: **DBEdit**, **DBCheckBox**, **DBRadioGroup**, **DBImage** и др.

Источники данных - это невидимые компоненты, исполняющие роль трубопроводов между множествами данных и визуальными компонентами баз данных. Используя введённые понятия, можно уточнить структуру приложения, осуществляющего доступ к данным через VDE

(

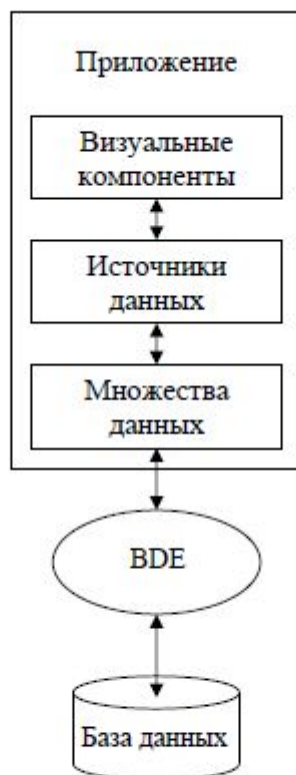


Рис. 6. Структура приложения БД, использующего VDE

1.5. Модули данных

Модуль данных - это контейнер для невидимых компонентов доступа к базе данных. Для создания модуля данных надо выполнить команду **File|New|Other** и в окне **New**

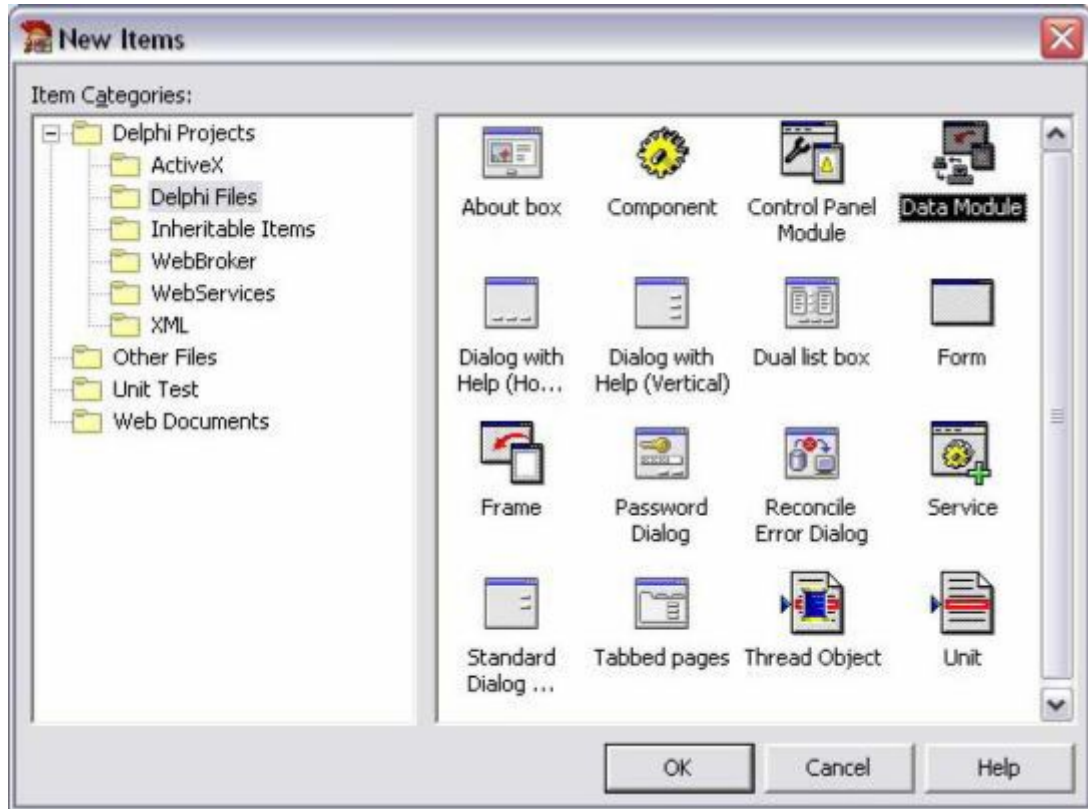


Рис. 7. Создание модуля

Items выбрать **Data Module** (рис. 7).

Модуль данных является объектом класса **TDataModule**, в него можно помещать только невидимые компоненты и задавать для компонентов доступа к данным обработчики событий. Для модуля данных определено всего несколько свойств (**Name**, **Tag**) и событий (**OnCreate**, **OnDestroy**), так как в отличие от формы его непосредственным предком является класс **TComponent**. Использование модуля данных позволяет отделить логику обработки данных от логики работы пользовательского интерфейса.

Для форм и модулей данных, создаваемых в приложении, **Delphi** использует сквозную нумерацию. Для подсоединения модуля данных используется команда **File|Use Unit**.

При разработке приложений целесообразно поместить множества данных и источники данных в модуль данных, а визуальные компоненты - на формы.

1.6. Невизуальные компоненты для работы с данными 1.6.1. Компонент Table

Компонент **Table** (табл. 1) обеспечивает доступ к таблицам базы данных, создавая *набор данных*, структура полей которого повторяет таблицу БД. Набором данных называют записи одной или нескольких таблиц, переданные в приложение в результате активизации компонента доступа к данным.

Таблица 1

Важнейшие свойства компонента **Table**

Свойство	Описание
Active	Если равно true, то таблица открыта, если false - то закрыта
CachedUpdates	Если равно true, то операции по модификации таблицы сразу не выполняются, а кэшируются (накапливаются). Для физического выполнения скопившихся операций вызывается метод ApplyUpdates
DataBaseName	Задаёт базу данных, которой принадлежит таблица. Содержит псевдоним или путь к файлам, составляющим базу данных
Exclusive	Если равно true, то всем остальным пользователям таблицы будет отказано в доступе
Filter	Булевское выражение, задающее фильтр для записей
Filtered	Включает (true) или выключает (false) фильтрацию записей
FilterOptions	Параметры, определяющие режимы фильтрации
IndexFieldNames	Имена индексированных полей
IndexFiles	Список индексных файлов, используемых при работе с таблицами в формате dBase
IndexName	Вторичный индекс таблицы. Свойства IndexName и IndexFieldNames являются взаимоисключающими
MasterFields	Содержит имена полей, по которым данная подчинённая таблица связывается с главной таблицей
MasterSource	Указывает источник данных (DataSource) главной таблицы, если данная таблица является подчинённой
ReadOnly	Запрещает/разрешает (true/false) модификацию данных в таблице
TableName	Имя таблицы БД
TableType	Физический тип таблицы. Это свойство игнорируется при работе с серверной базой данных

С помощью компонента **Table** можно организовать доступ к любой записи таблицы или их подмножеству. Компонент **Table** содержит все необходимые свойства, события и методы для создания, удаления, модификации, сортировки, фильтрации и поиска записей в таблице.

1.6.2. Компонент Query

Компоненты **Table** и **Query** являются наследниками класса **TDataSet** (гл. 6), поэтому у них очень много общих свойств и методов. **Query** обладает большими возможностями, позволяет формировать запросы к базе данных на языке SQL, создавать логические таблицы. Приёмы использования компонента **Query** излагаются в гл. 5.

1.6.3. Компонент DataSource

Компонент **DataSource** (табл. 2) обеспечивает взаимодействие набора данных с компонентами для отображения данных. С каждым компонентом доступа к данным должен быть связан как минимум один компонент **DataSource**. С одним компонентом **DataSource** может быть связано несколько визуальных компонентов.

Полный перечень свойств и методов компонентов можно посмотреть в справочной
Таблица 2

Важнейшие свойства компонента **DataSource**

Свойство	Описание
AutoEdit	Определяет, переходит ли связанная с источником данных таблица БД в режим редактирования записи, если пользователь начинает печатать символы в одном из управляющих элементов, связанных с источником данных. Если <code>true</code> , то переходит
DataSet	Указывает компонент (Table или Query), поставляющий данные из таблицы
Enabled	Определяет, обновляется ли содержимое управляющих элементов, связанных с источником данных, при изменении текущей записи в таблице. Позволяет включить (<code>true</code>) или отключить (<code>false</code>) все подсоединенные визуальные компоненты

системе. В Delphi 2006 это проще всего сделать с помощью поиска по ключевому слову (`index`).

1.6.4. Компоненты полей

Компоненты множества данных **Table**, **Query** хранят информацию о полях своей таблицы в виде массива компонентов полей. Абстрактный класс **TField** предоставляет доступ к полям таблицы, обладает мощными потомками, которые применяются автоматически или задаются в Редакторе полей **Fields Editor**. Потомки класса **TField** (табл. 3) отличаются от базового класса и друг от друга особенностями, связанными с обработкой данных различных типов.

Таблица 3

Потомки класса **TField**

Класс	Представляемый тип данных
TAutoIncField	Целые числа, увеличивающиеся на единицу с каждой следующей записью. Диапазон чисел как у класса TIntegerField (32 разряда)
TBCDField	Вещественные числа с фиксированным числом десятичных знаков после запятой. Точность - 18 знаков
TBlobField	Двоичные данные неограниченного объёма (BLOB - Binary Large Object - большой двоичный объект)
TBooleanField	Логический, принимающий значения true и false
TBytesField	Двоичные данные фиксированной длины
TCurrencyField	Денежные единицы. Диапазон и точность соответствуют классу TFloatField
TDateField	Значение даты
TDateTimeField	Значение даты и времени
TFloatField	Вещественные числа
TGraphicField	Графическое изображение произвольных размеров
TIntegerField	Целые числа (32 разряда)
TMemoField	Текст произвольной длины
TSmallintField	Целые числа в диапазоне от -32 768 до 32 767 (16 разрядов)
TStringField	Строка текста длиной до 255 символов
TTimeField	Значение времени
TVarBytesField	Двоичные данные переменной длины. Реальная длина хранится в первых двух байтах
TWordField	Целые числа в диапазоне от 0 до 65 535 (16 разрядов)

Так как класс **TField** является *абстрактным*, то в работающем приложении его экземпляры никогда не создаются. Вместо них создаются экземпляры производных классов, каждый из которых представляет определённый тип данных.

Компоненты полей являются невизуальными, служат для доступа к соответствующим полям записи. Компоненты полей (табл. 4) описывают тип и формат вывода данных в колонке, определяют возможность отображения данных из колонки в визуальных компонентах и др. Кроме описательной информации о колонках объекты типа TField хранят значения полей текущей записи таблицы.

Таблица 4

Некоторые свойства компонентов полей Свойство Описание

Alignment	Выравнивание поля по левому, правому краю или по центру. Присутствует только в числовых полях
Calculated	Вычисляемое поле. Если равно true, то значение поля вычисляется в обработчике события OnCalcFields
Currency	Если равно true, то значение вещественного поля отображается в денежном формате
DisplayFormat	Форматная строка, используемая при отображении значений числовых полей и полей даты и времени
DisplayLabel	Заголовок колонки, отображаемый компонентом DBGrid
DisplayWidth	Число символов, отводимое для колонки компонентом DBGrid
EditFormat	Форматная строка, используемая при редактировании значений числовых полей в управляющих элементах
EditMask	Шаблон редактирования, присутствует только в строковых полях и полях, содержащих дату и время
FieldName	Имя физической колонки в таблице базы данных
Index	Номер поля в таблице базы данных
MaxValue	Максимальное значение числового поля
Precision	Количество десятичных знаков, к которому должно быть приведено значение вещественного поля перед округлением (по умолчанию принимается 15)
ReadOnly	Запрет/разрешение на изменение поля
Size	Число байтов, зарезервированных для поля в таблице БД
Required	Определяет, может ли поле быть пустым. Если равно true, то значение поля всегда должно быть задано
Visible	Определяет, отображается ли колонка в компоненте DBGrid

При работе с таблицей БД с помощью компонентов множества данных (например, **Table** или **Query**) для каждой колонки автоматически генерируются

объекты TField. Эти компоненты создаются, когда таблица открывается, и удаляются при её закрытии. Если компоненты TField генерируются динамически, то они меняются при изменении структуры таблицы. Благодаря такому подходу можно открывать и использовать таблицы, структура которых заранее неизвестна. Это очень удобно. Однако в некоторых случаях целесообразно создать постоянные поля, которые позволяют:

- создавать вычисляемые поля, значения которых определяются при работе приложения по данным из других полей;
- ограничивать состав полей;
- изменять порядок полей в таблице;
- скрывать некоторые поля при работе приложения;
- задавать формат отображения или редактирования данных на этапе разработки приложения.

Если приложение точно знает таблицы своей БД и рассчитывает на определённую структуру колонок, то целесообразно создать для каждой используемой таблицы постоянные поля. Благодаря тому, что компоненты полей не будут генерироваться динамически, приложение станет использовать созданные на этапе проектирования и сохраняемые вместе с таблицами компоненты полей, что повышает устойчивость приложения к изменению структуры базы данных.

1.7. Редактор полей и его использование

Постоянные компоненты полей создаются с помощью Редактора полей Fields Editor, существующего в множествах данных **Table, Query**. Окно редактора полей открывается через контекстное меню этих компонентов. В редакторе полей *отображаются только постоянные поля*. Поэтому, если все поля динамические, окно редактора полей будет пустым. Если хотя бы одно поле является постоянным, то динамические поля создаваться не будут, т.е. остальные поля будут считаться отсутствующими. Ниже приведена последовательность действий при создании постоянных полей (рис. 8).

1. Щёлкнуть правой кнопкой мыши по компоненту **Table** и выбрать из контекстного меню команду **Fields Editor**. Откроется окно Редактора полей, которое будет пустым, так как

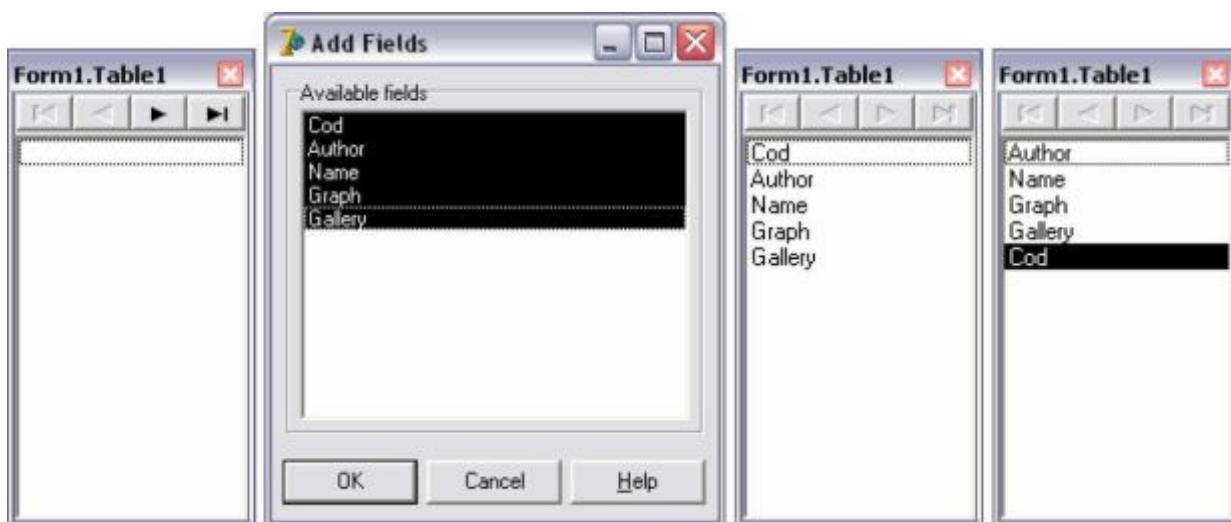


Рис. 8. Создание постоянных

по умолчанию все поля динамические.

2. Щёлкнуть правой кнопкой мыши по окну Редактора полей и выбрать из контекстного меню команду **Add Fields**. В окне диалога **Add Fields** появится перечень всех полей таблицы.

3. В списке всех доступных динамических полей следует выбрать нужные поля и нажать **ОК**. В окне Редактора полей появятся выбранные поля и одновременно будут внесены изменения в текст программного модуля: в объявлении класса появятся компоненты полей.

type

```
TForm1 = class(TForm)
  DataSource1: TDataSource;
  Table1: TTable;
  Table1Cod: TAutoIncField;
  Table1Author: TIntegerField;
  Table1Name: TStringField;
  Table1Graph: TGraphicField;
  Table1Gallery: TIntegerField; end;
```

В приведённом примере все поля таблицы объявлены как постоянные. Порядок следования полей определяется их расположением в Редакторе полей. При необходимости порядок следования можно изменить, перемещая поля

мышью или клавиатурными комбинациями. Для уточнения параметров постоянного поля его надо выбрать в Редакторе полей и задать значения свойств в Инспекторе объектов.

Помимо приведённых выше постоянных полей, основанных на физических полях таблицы, можно создать постоянные поля ещё двух типов: вычисляемые и поля просмотра (**Lookup**). Значение вычисляемого поля определяется во время работы приложения по заданной формуле с использованием данных из других полей.

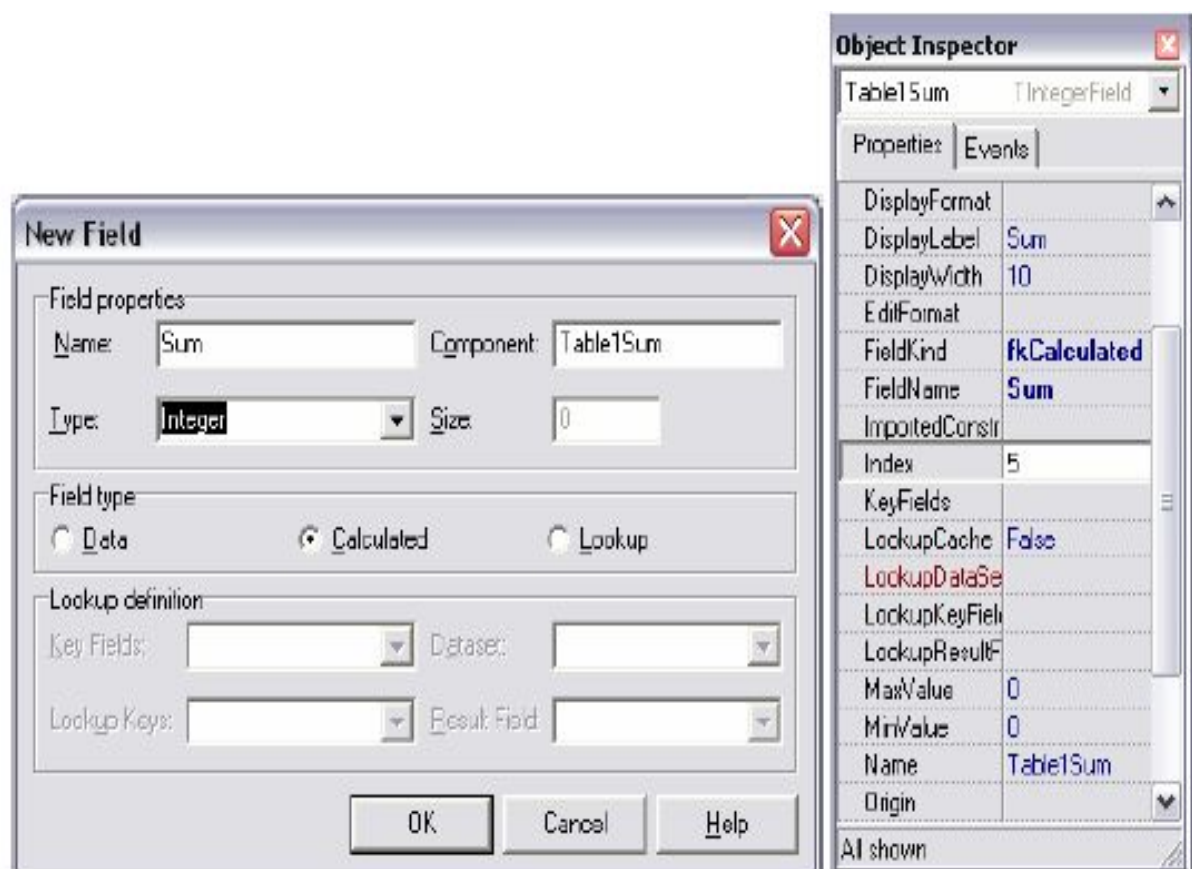


Рис. 9. Создание вычисляемого поля

Для создания вычисляемого поля (рис. 9) необходимо:

- открыть окно редактора полей и через контекстное меню командой **New Field** вызвать диалоговое окно для создания новых полей;
- в строке *Name* записать имя нового поля. Строку *Component* (название компонента) система формирует сама;
- в выпадающем списке *Type* выбрать тип создаваемого поля;
- нажать **ОК**. В Редакторе полей появится имя нового поля, у которого в свойстве FieldKind должно быть записано Calculated;

- задать способ вычисления значения в созданном поле. С этой целью для компонента **Table** определить обработчик события OnCalcFields, реализующий необходимый алгоритм вычислений. Поле просмотра (поле выбора, отыскиваемое поле) позволяет выбирать значения из предлагаемого списка. С полем просмотра связывается список, который наполняется значениями из другой таблицы. По сути, значение поля отыскивается в реляционно-связанной таблице.

Для создания поля просмотра в таблице, доступ к которой осуществляется через **Table1**, необходимо выполнить следующие действия:

1. Добавить компонент **Table2** и связать его с таблицей, в которой находятся представляющие интерес сведения (с таблицей просмотра);
2. Открыть Редактор полей для компонента **Table1**;
3. Через контекстное меню Редактора полей задать **New Field**;
4. В окне **New Field** создать новое поле. Для этого:
 - ввести имя нового поля, не совпадающее с имеющимися в таблице именами,
 - указать тип. Тип должен соответствовать типу поля в таблице просмотра,
 - если поле строкового типа, то задать его размер,
 - в группе зависимых переключателей *Field type* выбрать значение **Lookup**;
5. Связать новое поле с полем в таблице просмотра:

- открыть выпадающий список *Key Fields* и выбрать поле, по которому связаны таблицы **Table1** и **Table2**,
- в выпадающем списке *Dataset* выбрать значение **Table2** - таблицу, из которой будут браться данные,
- в списке *LookupKeys* выбрать поле таблицы просмотра, которое ассоциируется с полем таблицы **Table1**,
- в списке **Result Field** выбрать введённое имя нового поля.

1.8. Визуальные компоненты для работы с данными

Большинство визуальных компонентов для работы с данными похожи на соответствующие компоненты для обычных приложений. Основное отличие заключается в том, что компоненты для работы с базами данных содержат свойства (табл. 5), позволяющие указать источник данных (*DataSource*) и поле, из которого компонент получает данные (*DataField*).

Названия компонентов начинаются с букв **DB**. Большинство компонентов предназначено для отображения текущего значения одного поля: **DBEdit**, **DBCheckBox**, **DBRadioGroup**, **DBText** и др. Для вывода и редактирования данных поля Мемо используются компоненты **DBMemo** и **DBRichEdit**. Для просмотра изображений предназначен компонент **DBImage**. Отображать данные графически позволяет компонент **DBChart**.

Кроме того, имеются компоненты, предназначенные для использования только в приложениях баз данных. Это **DBNavigator** и компоненты **DBLook-upComboBox**, **DBLookupListBox** для синхронного просмотра данных из связанных таблиц.

1.8.1. Компонент **DBNavigator**

Таблица 5

Общие свойства визуальных компонентов

Свойство	Описание
<i>DataField</i>	Поле связанного с компонентом набора данных
<i>DataSource</i>	Связанный с компонентом источник данных
<i>Field</i>	Обеспечивает доступ к классу <i>TField</i> используемого поля
<i>ReadOnly</i>	Запрещает/разрешает (<i>true/false</i>) пользователю изменять данные

Компонент **DBNavigator** является удобным средством перемещения по записям таблицы. Представляет собой панель с кнопками, построенную по типу панелей управления электронными устройствами. **DBNavigator** обеспечивает:

- прокрутку записей поодиночке вперёд и назад;
- переход к первой или последней записи;
- вставку новой записи;
- удаление текущей записи;
- переход в режим редактирования текущей записи;
- внесение изменений в таблицу;
- отмену сделанных в текущей записи изменений;
- обновление отображаемых значений. Это необходимо в тех случаях, когда данные в таблице БД могут изменяться другими приложениями.

Не все из перечисленных возможностей навигатора бывают нужны. Можно оставить только те кнопки, которые реализуют требуемую функциональность. Для управления видимостью кнопок используется свойство `Visible-Buttons`: *для каждой кнопки отдельно* выбирается `true` или `false`.

Свойство `flat` управляет внешним видом кнопок: предусмотрены объёмное (`false`) и плоское (`true`) изображения.

Свойства `Hints` и `ShowHint` предназначены для вывода подсказок. По умолчанию подсказки к кнопкам выводятся на английском языке и содержат текст, приведённый в свойстве `Hints`. Если требуется изменить подсказку к отдельной кнопке или создать подсказки на русском языке, то надо открыть строковый редактор свойства `Hints` и записать текст подсказок.

1.8.2. Компонент **DBGrid**

Визуальный компонент **DBGrid** (табл. б) предназначен для организации табличного просмотра и редактирования данных. Внешний вид данных, отображаемый **DBGrid**, по умолчанию соответствует структуре набора данных. Компонент **DBGrid** часто называют *сеткой*.

Для перемещения по записям используются полосы прокрутки и клавиши управления курсором. Для изменения данных достаточно установить курсор в нужную ячейку и ввести другое значение. Новая пустая строка создаётся в позиции указателя нажатием на клавишу **Insert**. Чтобы изменения, сделанные при редактировании и добавлении записи были внесены в таблицу, необходимо нажать на клавишу **Enter** или перейти на другую строку. До того как данные были переданы в таблицу, можно клавишей **Esc** отменить изменения. Для удаления записи используется комбинация клавиш **Ctrl+Delete**.

Таблица 6

Некоторые свойства компонента **DBGrid**

Свойство	Описание
Align	Определяет способ выравнивания внутри владельца
Columns	Содержит список объектов, описывающих колонки в таблице
DataSource	Указывает источник (компонент DataSource), из которого извлекаются отображаемые данные
DefaultDrawing	Если равно true, то ячейки таблицы отображаются в обычном стиле. Если значение равно false, то в обработчике события OnDrawColumnCell можно определить свой способ рисования ячеек
FixedColor	Задаёт цвет фиксированных строк и колонок таблицы
Options	Определяет режимы работы компонента
TitleFont	Определяет шрифт, используемый при отображении названий колонок

Свойство **Options** является составным. Его значения определяют особенности поведения и внешнего представления компонента на экране (табл. 7). По умолчанию свойство **Options** содержит комбинацию значений [dgEditing, DgTitles, dgIndicator, DgColumnResize, DgColLines, DgRowLines, DgTabs, DgConfirmDelete, DgCancelOnExit].

По умолчанию для каждого поля исходной таблицы в **DBGrid** создаётся отдельный столбец. Такие столбцы называются динамическими. Характеристики динамического столбца определяются свойствами поля, для отображения которого этот столбец используется. Например, название столбец получает по названию поля, а ширина столбца определяется типом данных поля. Это не всегда удобно. При необходимости в **DBGrid** можно перейти к статическим столбцам, параметры которых задаются независимо от свойств поля. Для формирования статических столбцов используется Редактор колонок.

Компонент **DBGrid** обычно используют для просмотра данных. Для ввода и изменения данных используют компоненты, работающие с одним полем, так, чтобы на

Таблица 7

Значения свойства Options

Параметр	Значение
<code>dgEditing</code>	Разрешает редактировать данные
<code>dgAlwaysShowEditor</code>	Компонент всегда находится в режиме редактирования
<code>DgTitles</code>	Для колонок отображаются заголовки
<code>dgIndicator</code>	Таблица имеет узкую колонку, в которой отображается индикатор текущей записи
<code>DgColumnResize</code>	Колонки можно изменять по ширине и перемещать мышью
<code>DgColLines</code>	Колонки таблицы разделены линиями
<code>DgRowLines</code>	Строки таблицы разделены линиями
<code>DgTabs</code>	Клавиши Tab и Shift+Tab используются для перехода между колонками
<code>DgRowSelect</code>	В таблице выделяется вся строка, а не отдельная ячейка. Значения <code>dgEditing</code> , <code>dgAlwaysShowEditor</code> игнорируются
<code>DgAlwaysShowSelection</code>	Выделенный элемент виден даже тогда, когда компонент не обладает фокусом ввода
<code>DgConfirmDelete</code>	При попытке удалить запись клавишами Ctrl+Delete выдаётся сообщение с просьбой подтвердить операцию
<code>DgCancelOnExit</code>	Добавленная, но ещё не заполненная запись, не сохраняется в таблице при потере фокуса ввода
<code>DgMultiSelect</code>	Можно выделить несколько строк, расположенных подряд или вразброс

экране были видны поля только одной записи.

1.9. Редактор колонок

Редактор колонок позволяет в компоненте **DBGrid** задать отображаемые поля и уточнить требования к оформлению колонок. На экран Редактор колонок выводится двойным щелчком по **DBGrid** или через контекстное меню.

В окне Редактора колонки можно удалять, добавлять, менять местами. Свойства выбранной колонки становятся доступными для редактирования в Инспекторе объектов. Настройка параметров строки заголовка и рабочих ячеек колонки выполняется отдельно (рис. 10). Параметры заголовка колонки задаются сложным свойством Title из раздела Localizable (Delphi 2006), в котором можно указать: название (Caption), шрифт (Font) и др. Свойства рабочих ячеек сгруппированы в разделе Visual. Для восстановления параметров колонки используется команда Restore Defaults (рис. 11).

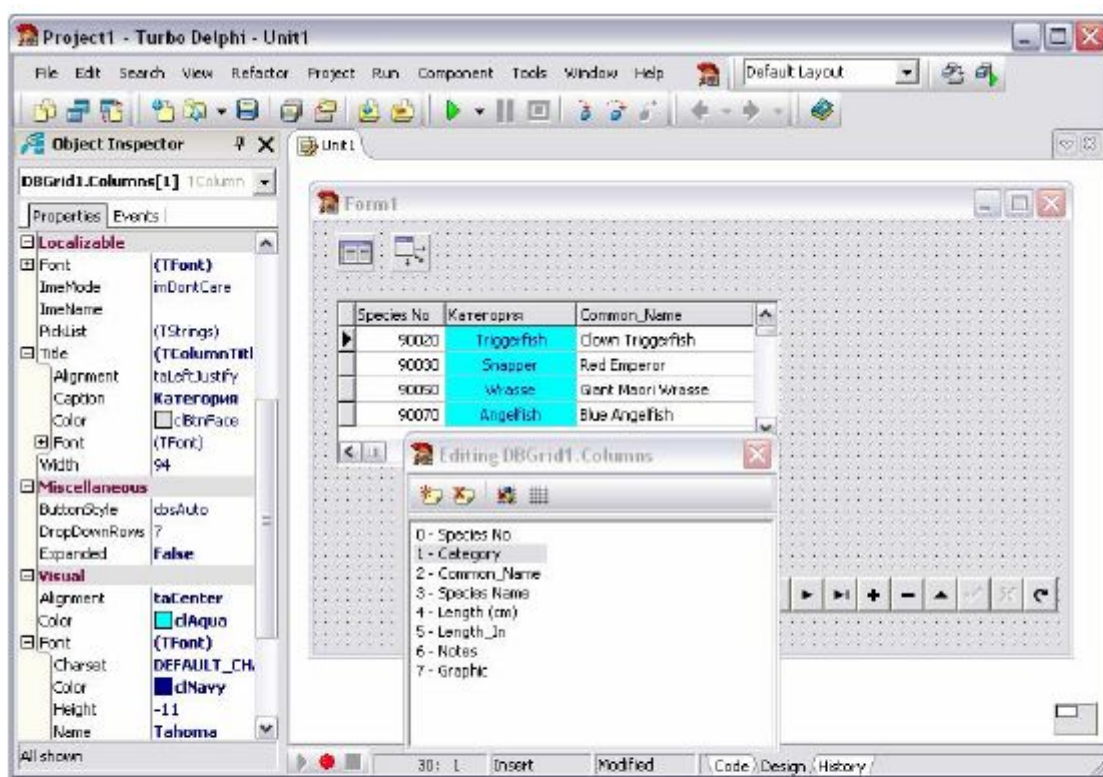


Рис. 10. Задание параметров колонок в Инспекторе объектов (Delphi 2006)

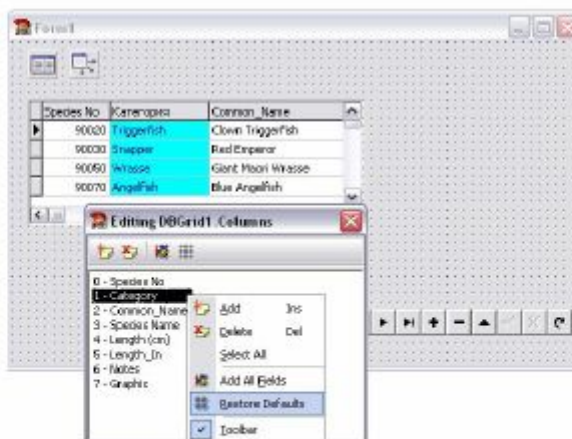


Рис. 11. Редактор колонок Columns Editor

Следует помнить, что упрощённое представление таблицы облегчает просмотр, но ограничивает редактирование, так как скрытые колонки считаются отсутствующими и в них нельзя занести данные.

1.10. Контрольные вопросы

1. Для чего предназначен BDE?
2. Для чего предназначены утилиты SQL Explorer, BDE Administrator?
3. Как загрузить утилиту SQL Explorer, BDE Administrator?
4. Что такое псевдоним БД?
5. Как создать псевдоним БД?
6. Где и как используется псевдоним?
7. Что такое Data Module?
8. Как подсоединить Data Module к приложению?
9. Перечислите визуальные компоненты для работы с базами данных.
10. Перечислите невидимые компоненты для работы с базами данных.
11. Перечислите основные свойства компонента **Table**.
12. Перечислите основные свойства компонента **DataSource**.
13. Компоненты полей. Что это такое?
14. Перечислите основные компоненты полей.
15. Для чего используется Редактор полей?
16. Как войти в Редактор полей?
17. В каких случаях окно Редактора полей пустое?
18. Как создать постоянные поля?
19. Как узнать, какие поля используются: динамические или постоянные?
20. Какие поля называют вычисляемыми?
21. Какие поля называют отыскиваемыми?
22. Как создать вычисляемое поле?
23. Как создать отыскиваемое поле?

24. С каким компонентом работает Редактор колонок?
25. Для чего используется Редактор колонок?
26. Как открыть Редактор колонок?
27. Почему в некоторых случаях вычисляемое поле автоматически не отображается в компоненте **DBGrid**?

2. НАВИГАЦИОННЫЙ СПОСОБ ДОСТУПА К ДАННЫМ

2.1. Операции с таблицей базы данных

Навигационный способ доступа к данным предполагает выполнение операций с отдельными записями. Достоинством этого способа является простота задания операций. Основной недостаток состоит в том, что приложение получает все записи набора, что приводит к значительной загрузке сети. Как правило, навигационный способ работы с данными используют в локальных базах данных.

В любом наборе данных существует текущий указатель, который задаёт запись, над которой будут выполняться операции изменения данных. Компоненты **Table** и **Query** содержат методы, позволяющие перемещаться по записям (изменять положение текущего указателя).

Используя навигационный способ доступа к данным, можно выполнять следующие операции:

- навигацию по набору данных;
- редактирование записей;
- добавление и удаление записей;
- сортировку записей;
- фильтрацию записей;
- поиск записей;
- создание, удаление, переименование таблиц.

Большинство действий можно выполнять как с помощью компонентов, так и программно.

2.2. Навигация по набору данных 2.2.1.

Простейшее приложение для работы с БД

В соответствии со схемой работы с базой данных с использованием BDE для создания приложения необходимо выполнить приведённые ниже действия.

1. Разместить на форме компоненты **Table**, **DataSource**, **DBGrid**.

2. Связать компонент **Table** с таблицей БД, которая находится на диске. Для этого в Инспекторе объектов:

- в свойстве `DatabaseName` выбрать в выпадающем списке псевдоним или указать путь к базе данных;
- в свойстве `TableName` указать название таблицы;
- свойство `Active` установить в значение `true`.

3. Связать компоненты **DataSource** и **Table**: в свойстве `DataSet` компонента **DataSource** установить значение `Table1`.

4. Подсоединить **DBGrid** к источнику данных: в свойстве `DataSource` компонента **DBGrid** задать значение `DataSource1`.

В данном примере все свойства задавались в Инспекторе объектов, но это можно сделать программно.

Для удобства работы можно добавить на форму навигатор **DBNavigator** и связать его с источником данных: в свойстве `DataSource` задать значение `DataSource1`.

2.2.2. Открытие и закрытие DataSet

В Инспекторе объектов доступ к множеству данных регулируется свойством `Active` компонента **Table**.

Если же требуется открыть таблицу программно, то можно воспользоваться одним из двух способов: задать свойство `Active` или вызвать метод `Open`. Результат будет одинаковым.

```
Table1.Active:=True; или
```

```
Table1.Open;
```

Для закрытия таблицы тоже существует два способа:

```
Table1.Close; или
```

```
Table1.Active:=False;
```

2.2.3. Перемещение по записям

Обширный набор методов и свойств класса `TDataSet` (табл. 8) обеспечивает всё, что нужно для доступа к любой конкретной записи таблицы.

Некоторые методы и свойства класса TDataSet

Свойство или метод	Описание
Procedure First	Перемещение к первой записи в таблице
Procedure Last	Перемещение к последней записи в таблице
Procedure Next	Перемещение на одну запись вперёд
Procedure Prior	Перемещение на одну запись назад
Property BOF: Boolean	Имеет значение true в начале таблицы
Property EOF: Boolean	Имеет значение true в конце таблицы
Function MoveBy(Distance: Integer) : Integer	Перемещение по таблице на заданное число записей вперёд или назад. Для перемещения назад следует задать отрицательное число, например, Table1.MoveBy(-1)

При использовании компонента **DBGrid** можно перемещаться по записям таблицы с помощью полос прокрутки. При использовании **DBNavigator** для задания перемещения предусмотрены кнопки. Однако иногда нужно перемещаться по таблице программным путем, без использования возможностей, встроенных в визуальные компоненты. Для этого используются приведённые в таблице методы и свойства. Чаще всего для задания перемещения используют кнопку и создают обработчик, в котором вызывается нужный метод. Например, перемещение к следующей записи обеспечивает код: **procedure** TForm1.Button1Click(Sender:TObject); **begin**

Table1.Next; **end;**

Свойства булевского типа BOF и EOF используются для проверки, находится ли указатель в начале таблицы или в конце. Этих свойств нет в Инспекторе объектов, они доступны только программно во время работы приложения. Кроме того, значения этих свойств можно только прочитать.

Свойство BOF возвращает true в трёх случаях:

- после открытия файла;
- после вызова TDataSet.First;
- после того, как вызов TDataSet.Prior не выполняется.

Пример организации прохода по таблице с использованием BOF и Prior: **while not Table1.Bof do begin**

```
. . . //действия с данными Table1.Prior;
```

end;

Цикл будет продолжаться до тех пор, пока вызов Table1.Prior не сможет больше выполнить перемещение на предыдущую запись в таблице. В этот момент BOF вернет true и программа выйдет из цикла.

Свойство EOF возвращает true в следующих случаях:

- после того, как был открыт пустой файл;
- после вызова TDataSet.Last;
- после того, как вызов TDataSet.Next не выполняется.

Код, приведенный ниже, позволяет пробежать по всем записям в таблице от начала до конца: Table1.First; **while not Table1.EOF do begin**

```
. . . //действия с данными Table1.Next;
```

end;

2.3. Доступ к полям

Таблица состоит из записей, а записи - из полей. При работе с полями программно надо уметь правильно к ним обращаться.

Существует несколько способов получения доступа к полям записи. Прежде всего это возможности класса TDataSet:

свойство Fields[Index:Integer]:TField, свойство FieldValues[const

FieldName:string]:Variant, метод FieldByName(const FieldName:string):TField, а

также свойство Value:Variant класса TField.

Свойство `Fields` позволяет обратиться к полю по индексу. Индекс задаёт номер поля, к которому будет получен доступ. Поля нумеруются с нуля.

Метод `FieldByName` подразумевает обращение к полю по имени. Обращение по имени очень удобно в некоторых случаях. Например, если нет уверенности в местонахождении поля или если структура записи могла измениться.

При корректном обращении к одному и тому же полю разными способами результат будет одинаковым. Наличие нескольких способов получения результата нужно исключительно для того, чтобы обеспечить программистов гибким и удобным набором инструментов для программного доступа к содержимому набора данных.

При работе с полями используются потомки класса `TField`, которые обеспечивают простой и гибкий способ доступа к данным, связанным с конкретным полем. В классе `TField` определены свойства (разд. 1.6.4), которые позволяют выбрать тип результата: `AsBoolean`, `AsCurrency`, `AsDateTime`, `AsFloat`, `AsInteger`, `AsString`, `AsVariant`.

При необходимости Delphi сможет выполнить преобразования, которые имеют смысл. Например, преобразовывать поле `Boolean` в `Integer` или `Float` или поле `Integer` в `String`. Но преобразование `String` в `Integer` выполнено не будет. Для доступа к полям `Date` или `DateTime` можно использовать `AsString` и `AsFloat`.

Свойство `Fields` позволяет получить доступ не только к содержимому полей, но также и к их именам. Для обращения к имени используется свойство `FieldName` класса `TField`.

При обращении к полю через свойство `FieldValues` следует помнить, что возвращаемое значение имеет тип `Variant`, а это не всегда удобно. Например, сравнение значений лучше выполнять в том формате, который имеет поле. Свойство `FieldValues` можно интерпретировать как массив полей текущей записи, индексы элементов которого задаются именами полей. Так как свойство `FieldValues` объявлено как основное (используется по умолчанию), то конструкции `DataSet.FieldValues['Name']` и `DataSet['Name']` эквивалентны.

Использование свойства Value класса TField возможно только для постоянных полей. После создания в Редакторе полей постоянного поля в описании появится элемент, являющийся одним из компонентов полей (наследником TField), например, Table2Population: TFloatField;

Постоянное поле (в данном случае Table2Population) имеет свойство Value, которое обеспечивает доступ к данным. Для свойства Value возможны два варианта использования. Разрешено применять конструкцию, которая возвращает значение типа Variant (например, Table2Population.Value), либо применить свойство, которое обеспечивает преобразование результата в нужный тип (например, Table2Population.AsString): **procedure**

```
TForm1.Button2Click(Sender:TObject); Var rr:extended; begin  
    // использование свойства Value класса TField  
    Edit9.Text:=Table2Population.AsString; rr:=Table2Population.Value;  
    Edit10.Text:=floattostr(rr); end;
```

Далее приводится пример кода, иллюстрирующего разные варианты обращения к полям:

```
procedure TForm1.Button1Click(Sender:TObject); Var  
s1,s2,s3,s4,s5:string;  
    r1,r2,r3:extended; begin  
    s1:=Table1.Fields[1].AsString;  
    s2:=Table1.FieldName('Capital').AsString;  
    s3:=Table1.FieldValues['Capital'];  
    r1:=Table1.FieldValues['Population'];  
    r3:=Table1['Population'];  
    s4:=Table1.Fields[1].FieldName;// Получение имени поля
```



```
s5:=Table1.FieldByName('Population').AsString;  
r2:=Table1.FieldByName('Population').AsFloat;  
Edit1.Text:=s1; Edit2.Text:=s2;  
Edit3.Text:=s3; Edit4.Text:=s4;  
Edit5.Text:=floattostr(r1); Edit6.Text:=s5;  
Edit7.Text:=floattostr(r2);  
Edit8.Text:=floattostr(r3); end;
```

При работе с записями полезны следующие свойства наборов данных: FieldCount - количество полей в записи; RecordCount - количество записей в наборе данных; RecNo - номер записи.

2.4. Модификация набора данных

Модификация набора данных подразумевает изменение, добавление и удаление данных. Действия с данными можно выполнять через визуальные компоненты или программно. При модификации данных программно используются методы класса TDataSet, процедуры:

- Append - добавление записи в конец набора данных;
- Insert - добавление записи в текущую позицию;
- Cancel - отмена действия;
- Delete - удаление записи;
- Edit - задание режима редактирования;
- Post - внесение изменений.

Принципиальная возможность модификации данных определяется значением свойства CanModify. Если TTable находится в состоянии Readonly, то CanModify имеет значение false, редактирование запрещено. В противном случае CanModify возвращает true и можно редактировать или добавлять записи в таблицу.

2.4.1. Редактирование

Для изменения данных в таблице программным путём необходимо выполнить действия:

- перейти в режим редактирования (метод Edit);
- задать новые значения полей;
- занести изменения в таблицу или отменить выполненные действия (Post или Cancel).

Действия выполняются над текущей записью. Например:

```
Table1.Edit;
```

```
Table1.FieldName('CustName').AsString:='Fred'; Table1.Post;
```

Первая строка переводит БД в режим редактирования. Следующая строка присваивает полю CustName значение Fred. Процедура Post инициирует запись данных на диск.

Вместо явного вызова метода Post можно задать переход к другой записи, так как перемещение на другую запись сопровождается автоматическим сохранением данных на диске. Table1.Edit;

```
Table1.FieldName('CustNo').AsInteger:=1234; Table1.Next;
```

Итак, действия выполняются над полями текущей записи и вызов в режиме редактирования методов First, Next, Prior, Last приводит к сохранению данных (выполнению Post). Следует иметь в виду, что автоматическое сохранение данных при смещении текущего указателя на другую запись характерно только для локальных баз данных. До тех пор, пока не был вызван напрямую или косвенно метод Post, можно отменить изменения (вернуться к состоянию, которое было при переходе в режим редактирования) процедурой Cancel.

2.4.2. Добавление записи

Добавление записи предполагает выполнение действий:

- задание режима вставки записи методом Append или Insert;
- формирование значений полей;
- занесение данных в таблицу или отмена добавления записи.

Если данные в таблице упорядочены по индексированному полю, то результат использования методов Append и Insert будет одинаков: добавленная запись займёт место в таблице в соответствии со значением индекса. Различие между методами Append и Insert сказывается только в таблицах, у которых нет индексов. Table1.Insert;

```
Table1.FieldName('Name').AsString:='Russia';
```

```
Table1.FieldName('Capital').AsString:='Moscow';
```

```
Table1.Post;
```

При переводе таблицы в режим вставки появляется новая запись с незаполненными полями. Затем следует занести в поля значения. Если после вызова Insert (Append) возникла необходимость отказаться от вставки новой записи, то следует до вызова Post применить метод Cancel.

Помимо задания значений отдельным полям можно использовать метод SetFields. В качестве параметра этот метод использует массив констант и позволяет задать значения сразу всем полям. Например, Table1.Append;

```
Table1.SetFields[9000,2118,Now,Now,47]; Table1.First;
```

Вместо трёх этапов добавления записи можно получить аналогичный результат применением процедур InsertRecord и AppendRecord, в которых передаются константные массивы со значениями полей новой записи (как в методе SetFields).

2.4.3. Удаление записей и таблиц

Удаление записей выполняется вызовом метода Delete. При этом текущая запись удаляется, и курсор перемещается к следующей записи.

Table1.Delete;

Для таблиц различают операции очистки и физического уничтожения. Действия по очистке и уничтожению выполняются только для закрытых наборов данных.

Очистка таблицы:

Table1.Close;

Table1.Exclusive:=true;

Table1.EmptyTable;

Table1.Exclusive:=false;

Table1.Open;

Физическое уничтожение таблицы:

Table1.Close; Table1.DeleteTable;

2.4.4. Защита таблицы от изменений

Защиту данных от изменений можно осуществлять на разных уровнях.

Способ 1. Использование свойства Readonly визуальных компонентов. Все компоненты для отображения данных имеют свойство Readonly, которое по умолчанию установлено в значение false, что позволяет изменять любые поля записей. Если сделать только читаемым какой-либо визуальный компонент, то это частично ограничит доступ к таблице. Например, если установить свойство Readonly компонента **DBGrid** в true, то пользователь не сможет изменять данные при табличном просмотре записей, однако этот запрет не распространяется на внесение изменений программным путём и на внесение изменений через другой визуальный компонент, подсоединённый к той же таблице.

Способ 2. Использование свойства Readonly компонента **Table**. По умолчанию это свойство имеет значение false, и любые изменения разрешены. Управление свойством Readonly компонента **Table** позволяет более на

дѣжно защитить таблицу от изменений. Однако применение этого метода не всегда целесообразно, так как при переключении режима таблицу надо сначала закрыть, а после повторного открытия текущей становится первая запись, и чтобы вернуться к первоначальной позиции, приходится использовать закладку. В итоге тратятся дополнительные ресурсы и время.

Способ 3. Использование свойства **AutoEdit** компонента **DataSource**. Этот способ позволяет одновременно управлять возможностью редактирования через все визуальные компоненты, подключенные к одному источнику данных. Если свойство **AutoEdit** компонента **DataSource** имеет значение **false**, то при вводе данных в визуальные компоненты изменений в таблице не происходит. Однако можно выполнить редактирование программно, поэтому следует сделать недоступными все компоненты, которые могут перевести таблицу в режим редактирования. В частности, присвоить **false** свойству **dgEditing** компонента **DBGrid**.

Для переключения режимов целесообразно разместить на форме кнопку или создать пункт меню и записать обработчик так, чтобы щелчок по кнопке (выбор команды меню) изменял режим на противоположный. При этом пользователю должно быть понятно, в каком режиме находятся таблицы. Это можно сделать с помощью надписей и путём изменения доступности компонентов. После запуска приложения обязательно должен быть включен режим просмотра.

Кнопки, использующиеся при модификации данных, в режиме просмотра должны быть недоступны. Свойство этих кнопок **Enabled** должно быть установлено в **false** в Инспекторе объектов или программно.

Перевод приложения в режим редактирования пользователь задаёт явно.

Пусть для этой цели предусмотрена кнопка **Изменить режим** (**CntrBtn**). **procedure**

```
TForm1.CntrBtnClick(Sender:TObject);
```

```
//Обработчик кнопки Изменить режим begin
```

```
CanBtn.Enabled:=not DataSource1.Autoedit; DelBtn.Enabled:=not
```

```
DataSource1.Autoedit;
```

```

InsBtn.Enabled:=not DataSource1.Autoedit; if not
DataSource1.Autoedit then begin
    DataSource1.Autoedit:=true; RegLabel.Caption:='Режим
редактирования'; end else begin
    if Table1.State <> dsBrowse then Table1.Post;
    DataSource1.Autoedit:=false; RegLabel.Caption:='Режим просмотра';
    end;
end;
procedure TForm1.InsBtnClick(Sender:TObject); begin
    Table1.Insert; end;
procedure TForm1.DelBtnClick(Sender:TObject); begin
Table1.Delete;
end;
procedure TForm1.CanBtnClick(Sender:TObject); begin
Table1.Cancel;
end;

```

2.5. Работа с записями

Работа с записями подразумевает выполнение операций сортировки, фильтрации и поиска.

Сортировкой называют упорядочение записей по некоторому признаку.

Под фильтрацией понимают отображение только тех записей, которые удовлетворяют заданному условию.

При выполнении поиска отыскивается ближайшая запись (одна), соответствующая сформулированному условию.

Действия могут выполняться в Инспекторе объектов или программно. С точки зрения реализации методы фильтрации и поиска можно разделить на две группы:

- выполняемые по любым полям;
- выполняемые только по индексированным полям.

2.6. Сортировка

Сортировка строк по какому-либо полю выполняется на уровне множества данных с помощью индексов. Индекс управляет логическим порядком записей в таблице БД. Индекс определяет поля, по которым выполняется сортировка записей, направление сортировки (по возрастанию или убыванию значений полей), чувствительность к прописным и строчным буквам.

Начальный порядок в таблице определяется первичным индексом. Если таблица имеет первичный ключ, то первичный индекс создаётся по тому же полю (полям). Чтобы изменить порядок записей, необходимо сделать активным другой индекс. Вторичных индексов может быть несколько, соответственно, могут быть заданы разные способы сортировки. Индексы обычно создают при разработке структуры таблицы и потом активизируют в нужный момент.

Существуют два взаимоисключающих способа задания индекса: через свойство `IndexName` или с помощью свойства `IndexFieldName`. В свойстве `IndexName` необходимо записать имя индекса. Это можно сделать для тех таблиц, у которых индексы имеют имена, в частности, для таблиц формата `Paradox`. В свойстве `IndexFieldName` следует перечислить через точку с запятой поля, входящие в индекс.

В таблицах формата `Paradox` можно использовать `IndexName` или `IndexFieldName`, но не одновременно. Например, таблица `Customer` базы данных

DBDEMOS имеет два индекса (рис. 12). Первичный индекс (Primary), как принято в таблицах Paradox, имени не имеет, построен по полю CustNo. Вторичный индекс ByCompany создан по полю Company. Какие индексы определены в таблице, указано в свойстве

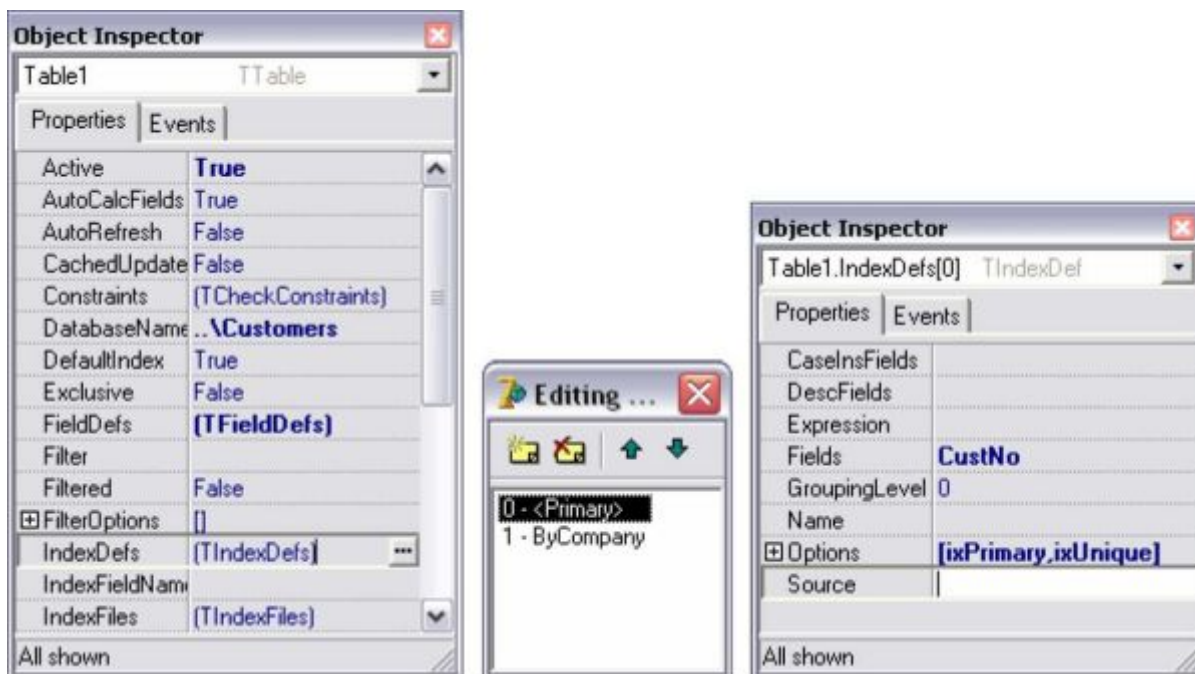


Рис. 12. Индексы и их свойства

IndexDefs компонента **Table**.

Пусть для выбора индекса используется компонент RadioGroup. Нужно значение можно задать с помощью свойства IndexName:

Case RadioGroup1.ItemIndex of 0: Table1.IndexName:= ""; //Выбран первичный индекс 1:

Table1.IndexName:='ByCompany'; //Выбран вторичный индекс

end;

Либо воспользоваться свойством IndexFieldNames:

Case RadioGroup2.ItemIndex of

0: Table1.IndexFieldNames:='CustNo';

1: Table1.IndexFieldNames:='Company'; **end;**

2.7. Фильтрация

Фильтрация позволяет исключить из просмотра (замаскировать) ненужные строки. *Фильтрация записей выполняется с помощью строкового свойства **Filter** и булевского свойства **Filtered***, которые имеются в множествах данных (компонентах **Table**, **Query** и др.). После того как в свойстве **Filter** записано условие, а в свойстве **Filtered** установлено значение **true**, все управляющие элементы, связанные с множеством данных, видят только те записи, которые удовлетворяют заданному критерию.

Синтаксис выражения в свойстве **Filter** аналогичен синтаксису логического выражения в языке Delphi, но в качестве идентификаторов можно использовать *только имена полей таблицы. Разрешены операции отношения и логические операции **not**, **and**, **or***. Нельзя использовать функции (в том числе стандартные математические) и сравнивать поля между собой.

Составное свойство **FilterOptions** позволяет уточнить алгоритм сравнения строковых значений, записанных в свойстве **Filter**. Если значение свойства **foCaseInsensitive** равно **true**, то при проверке строк не делается различий между прописными и строчными буквами. Если значение **foNoPartialCompare** равно **true**, то равными считаются строки, состоящие из одинаковых символов и совпадающие по длине.

В тех случаях, когда ограничения свойства **Filter** не позволяют задать требуемое условие отбора записей, можно выполнять фильтрацию с помощью события **OnFilterRecord**. Оно генерируется при смещении курсора, если значение свойства **Filtered** равно **true**. Обработчик события **OnFilterRecord** имеет параметр **Ассерт** булевского типа. По умолчанию **Ассерт** имеет значение **true** и условие включения записи определяется выражением в свойстве **Filter**. Чтобы исключить запись, переменной **Ассерт** следует присвоить значение **false**.

Фильтрация с использованием события **OnFilterRecord** позволяет использовать все возможности языка Delphi, но снижает производительность.

2.7.1. Свойство **Filter**

Свойство **Filter** имеет строковый тип и может быть задано в Инспекторе объектов или сформировано программно. Запись условия в Инспекторе объектов обычно не вызывает

затруднений. Например, для фильтрации по числовому полю LengthIn простое условие `Length_In>60`, записанное в свойстве Filter компонента **Table**, позволяет вывести записи, у которых значение поля LengthIn превышает 60. Условие `(Length_In>60)and(Length_In<100)` отбирает записи, у которых длина LengthIn больше 60 и меньше 100.

Для вывода записей, имеющих значение 'Cod' в поле Category строкового типа, достаточно записать условие: `Category='Cod'`. Если при этом в свойстве foCaseInsensitive задать true, то на результат не будет влиять регистр, то есть значения 'Cod' и 'cod' не различаются (рис. 13).

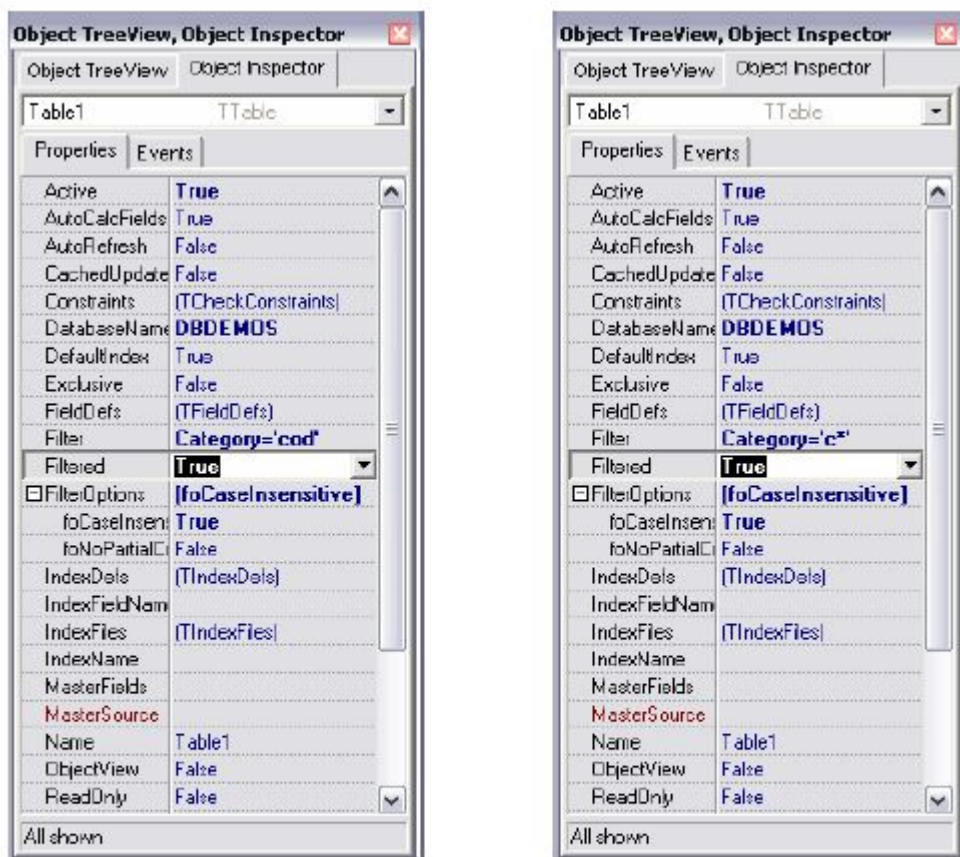


Рис. 13. Задание условия фильтрации в Инспекторе объектов

При организации фильтрации по строковым полям можно использовать маску. Условие Category='C*' в качестве результата выдаст все записи, у которых значение поля Category начинается с символа 'C'(рис. 14). При использовании маски свойство

Species No	Category	Common_Name	Species Name	Length
90080	Cod	Lunartail Rockcod	Variola louti	
90140	Cod	Lingcod	Ophiodon elongatus	

Рис. 14. Результаты

foNoPartialCompare должно быть false.

Программное задание свойства Filter обладает существенно большими возможностями, так как позволяет в условии использовать значения, вводимые пользователем во время работы приложения. Так как Filter - это свойство строкового типа, то надо сформировать строковую константу, задающую условие фильтрации. В этом случае все проблемы связаны с корректностью записи строкового выражения. Возможны два варианта:

- ' ' - простое строковое выражение, записанное в апострофах;
- ' '+' '+' ' . . . - сцепление нескольких строковых констант.

Примеры записи фильтра для числового поля вещественного типа:

```
Table1.Filter:='Length_In >60'; Table1.Filter:='Length_In
>'+Edit4.Text; Table1.Filter:='Length_In >23,625';
```

```
Table1.Filter:='Length_In >'+FloatToStr(23.625);
```

```
Table1.Filter:=
```

```
Length_In >'+Edit4.Text+'and Length_In<'+ Edit5.Text;
```

```
Table1.Filter:=
```

```
('Length_In>'+Edit4.Text+'and(Length_In<'+Edit5.Text+');
```

При фильтрации по полю строкового типа значение поля задаётся строковой константой, следовательно, должно быть записано в апострофах. По правилам языка Delphi для задания апострофа внутри строки его надо ввести дважды. Следовательно, если требуется вывести записи, у которых значение поля CommonName равно Lingcod, то в свойстве Filter надо записать:

```
Table1.Filter: = 'Common_Name="Lingcod";
```

Для фильтрации записей по первой букве поля Common_Name можно использовать маску. При этом строка, задающая условие фильтрации, формируется с помощью операции сцепления (+):

```
Table1.Filter: = 'Common_Name=' + "C*";
```

Если значение поля вводится в компонент **Edit**, то надо сформировать строковую константу по данным свойства Text.

```
Table1.Filter: = 'Common_Name=' + ""+Edit6.Text+ ' ' '; // В Edit6 C* Table1.Filter: =  
'Common_Name='+Edit6.Text;//В Edit6 'C*'
```

Варианты могут быть разные, в зависимости от того, в каком виде записывается интересующее пользователя значение поля. Лучше не требовать от пользователя знания каких-либо особых правил записи: Table1.Filter:='Common_Name='+''+Edit6.Text+'*'; //В Edit6 C

Для задания апострофа иногда удобно воспользоваться его кодом - #39. Надо только помнить, что апостроф, заданный в виде #39, всегда является отдельным элементом при сцеплении строк:

```
Table1.Filter: = 'Common_Name='+#39+Edit6.Text+'*'+#39;//В Edit6 C
```

Кроме того, можно воспользоваться функцией QuotedStr, которая возвращает строку в апострофах. Например:

```
Table1.Filter:='Common_Name<'+QuotedStr(Edit6.Text);
```

Если внутри строки есть апостроф, то он удваивается.

При необходимости выполнить фильтрацию по нескольким полям выбирают один из двух способов:

- записывают сложное условие с использованием логических связок;
- фильтруют по одному условию, затем по второму условию в уже отфильтрованном списке и т.д.

Если поле, по которому выполняется фильтрация, содержит дату, то при задании условия в Инспекторе объектов дату надо записать в апострофах. При выполнении фильтрации программно с датой надо работать как с полем строкового типа:

```
Table1.Filter:='Card_Exp <' + ""+(Edit4.Text)+"''"; Можно, например, использовать  
функцию QuotedStr: Table1.Filter:='Card_Exp<' + Quotedstr(Edit4.Text);
```

2.7.2. Выполнение вычислений

Часто требуется обработать данные по заданному полю: найти сумму, среднее, минимальное, максимальное значения и т.п. Причем при выполнении действий требуется учитывать данные не всех записей, а только тех, которые удовлетворяют некоторому условию. Организовать вычисления можно следующими способами:

- 1) в цикле перемещения по всем записям проверять условие и, если условие верно, выполнять вычисления;
- 2) отфильтровать данные по заданному условию, а затем выполнить требуемые действия для отфильтрованного набора записей.

В отфильтрованном наборе данных (Filtered равно True) методы First, Last, Next, Prior учитывают критерий фильтрации, т.е. задают перемещение в отобранных записях. Свойство RecordCount компонента **Table** содержит количество записей в отфильтрованном наборе. Ниже приведён пример работы с данными по методу 2:

- выполнена фильтрация по полю Category;
- определено количество записей в отфильтрованном наборе;
- найдено максимальное значение поля Length в отфильтрованном наборе.

```
procedure TForm1.Button8Click(Sender:TObject);  
var n, maxL:integer;  
begin  
Table1.Filtered:=False; if Edit8.Text<>"  
then
```

begin

```
Table.Filter:='Category='+'''+Edit8.Text+''';  
Table.Filtered:=True;  
n:=Table.RecordCount;  
Labell0.Caption:=  
    'Количество записей в заданной категории '+inttostr(n); Table.First;  
maxL:=Table.FieldName('Length').AsInteger; while not Table.Eof do  
begin  
    Table.Next;  
    if maxL<Table.FieldName('Length').AsInteger then  
        maxL:=Table.FieldName('Length').AsInteger; end;  
Labelll.Caption:=  
    'Максимальная длина в заданной категории '+inttostr(maxL); end  
else ShowMessage('Не задана категория'); end;
```

В примере проверяется, ввёл ли пользователь интересующее его название категории.

Если данные не введены, то вычисления не выполняются и выводится сообщение.

2.7.3. Событие OnfilterRecord

Если ограничения на запись выражений в свойстве Filter не позволяют задать нужное условие, например сравнить поля таблицы между собой, то можно воспользоваться событием OnfilterRecord (рис. 15).

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;  
                                     var Accept: Boolean);  
begin  
case RadioGroup1.ItemIndex of  
    0: Accept:=Table.FieldName('OnHand').AsInteger>  
        Table.FieldName('OnOrder').AsInteger;  
    1: Accept:=Table.Fields[3].AsInteger<  
        Table.Fields[4].AsInteger;  
    2: Accept:=Table.FieldValues['OnHand']=  
        Table.FieldValues['OnOrder'];  
end;  
end;  
procedure TForm1.Button1Click(Sender: TObject); begin  
    table.Filtered:=true; end;
```

В примере использованы разные способы обращения к полям. Во всех вариантах получен верный результат. Однако при использовании свойства FieldValues следует помнить, что это свойство имеет тип Variant. Лучше действия выполнять над данными, типы которых

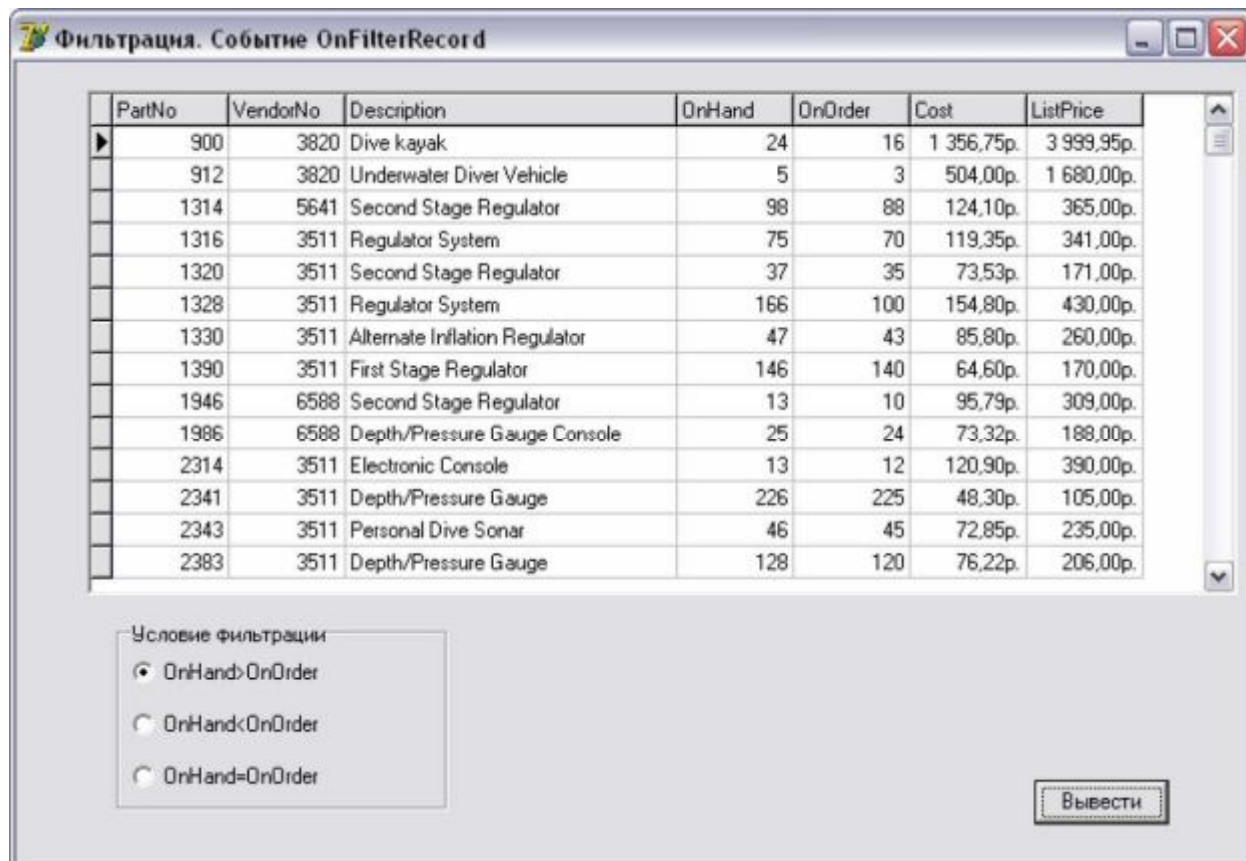


Рис. 15. Результат использования события OnfilterRecord

соответствуют типам полей.

При создании приложения для работы с базой данных разработчик предусматривает удобные средства для получения необходимых пользователю данных. В дополнение к чётко сформулированным и реализованным задачам по извлечению данных бывает целесообразно предусмотреть в приложении вариант задания условий фильтрации по любому набору полей, который можно реализовать с помощью события OnfilterRecord.

В этом случае пользователь отмечает флажками интересующие поля и вводит в них значения (рис. 16). В обработчике события OnfilterRecord формируется значение Accert, учитывающее установки пользователя. Для

удобства вычислений целесообразно объявить логические переменные для каждого поля, участвующего в фильтрации. Значение любой из введённых логических переменных будет равно true, если условие фильтрации выполняется или если поле не участвует в текущий

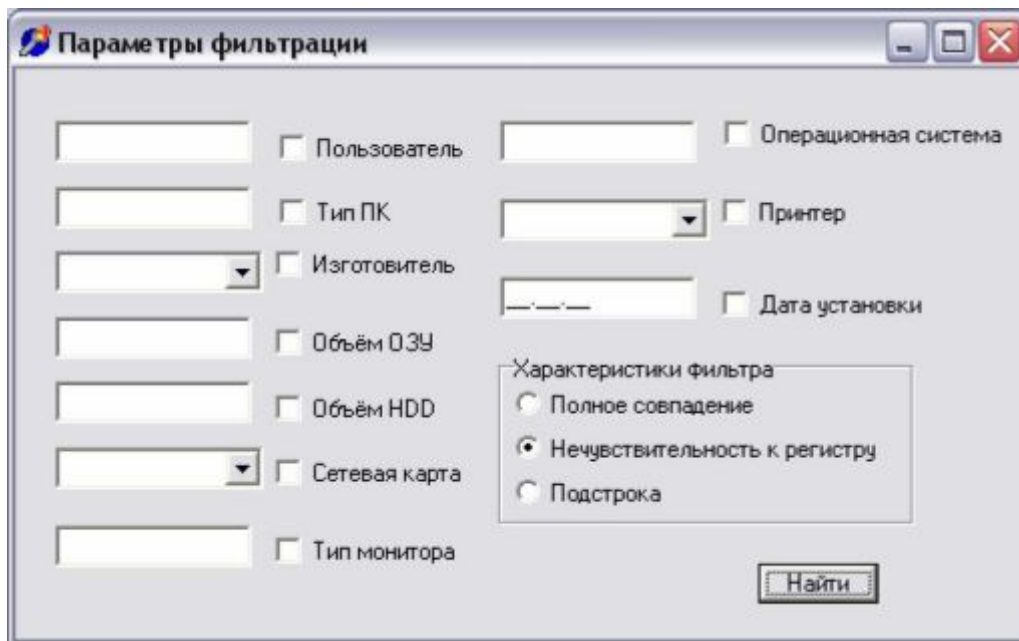


Рис. 16. Задание параметров

момент в фильтрации.

```

procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
                                     var Accept: Boolean); var
a1,a2,a3,a4,a5,a6,a7,a8,a9,a10: boolean; Begin
if Filtrform.CheckBox1.Checked
    then a1:=form1.Table1.Fields[1].asString=Edit1.Text else a1:=true;

if Filtrform.CheckBox4.Checked
    then a4:=form1.Table1.Fields[4].asInteger>=
                                     strToInt(Edit1.Text)
    else a4:=true;

Accept:=a1 and a2 and a3 and a4 and a5 and a6 and a7 and a8 and a9 and a10; end;

```

2.7.4. Фильтрация диапазона

Методы фильтрации диапазона работают только с индексированными полями. По умолчанию берётся текущий индекс. Если в свойствах

TTable.IndexName или TTable.IndexFieldName индекс не указан, то используется первичный ключ (главный индекс). Для фильтрации диапазона по другому индексированному полю следует явно задать нужный индекс. При выполнении фильтрации можно использовать методы класса TTable:

procedure SetRangeStart;

procedure SetRangeEnd;

procedure ApplyRange;

procedure CancelRange;

procedure EditRangeStart;

procedure EditRangeEnd;

procedure SetRange;

Последовательность действий при выполнении фильтрации диапазона:

- процедурой SetRangeStart перевести таблицу в режим диапазона;
- задать начало диапазона;
- вызвать процедуру SetRangeEnd;
- задать конец диапазона;
- вызвать метод ApplyRange для выполнения фильтрации. **procedure**

TForm1.Button3Click(Sender:TObject); **begin**

Table1.IndexName:= ""; //Выбран первичный индекс Table1.CancelRange;

//Отменена фильтрация диапазона

With Table1 **do begin**

SetRangeStart;

if Edit3.text<>"**then** Fields[0].AsString:=Edit3.text; SetRangeEnd;

if Edit4.text<>"**then** ields[0].AsString:=Edit4.text; ApplyRange; **end; end;**

Метод ApplyRange приводит запрос в действие: выводит записи, попадающие в указанный диапазон. Метод CancelRange отменяет фильтрацию. Если для того же индекса задать новое условие фильтрации, не отменив предыдущее, то записи будут отбираться из уже отфильтрованного набора данных, что часто приводит к ошибкам. Действия по заданию диапазона целесообразно начать с отмены предыдущей фильтрации.

Свойство `KeyExclusive` отвечает за включение в отфильтрованный набор записей, попадающих на границу диапазона. По умолчанию `KeyExclusive` имеет значение `false`, границы включаются в диапазон. Для исключения из диапазона граничного значения надо присвоить `KeyExclusive` значение `true`. Причём для каждой границы свойство `KeyExclusive` устанавливается отдельно, то есть можно одну границу включить в диапазон, а другую - нет. Методы `EditRangeStart`, `EditRangeEnd` работают аналогично `SetRangeStart` и `SetRangeEnd`.

Ниже приведён пример фильтрации диапазона в таблице `customer.db`. В этой таблице определено два индекса: главный по полю `CustNo` и вторичный `ByCompany` по полю `Company`. Так как поле `CustNo` целочисленное, то введённые пользователем значения преобразуются к целому типу.

```
procedure TForm1.RadioGroup1Click(Sender:TObject); begin
case RadioGroup1.ItemIndex of
0:Table1.IndexFieldNames:="";
1:Table1.IndexFieldNames:='Company'; end; end;

procedure TForm1.Button5Click(Sender:TObject); begin
Table1.CancelRange;
with Table1 do begin
    EditRangeStart;
    if RadioGroup1.ItemIndex=0 then FieldByName('CustNo').AsInteger:=strtoint(Edit3.Text)
    else FieldByName('Company').AsString:=Edit3.Text;
    EditRangeEnd;
    KeyExclusive:=false;
    if RadioGroup1.ItemIndex=0 then FieldByName('CustNo').AsInteger:=strtoint(Edit4.Text)
    else FieldByName('Company').AsString:=Edit4.Text;
    ApplyRange; end;
end;
```

При смене текущего индекса фильтрация по индексу, который был установлен ранее, перестаёт действовать, в результирующий набор включаются все

записи. В приведённом примере для смены индекса используется компонент **RadioGroup**.

Метод `SetRange` даёт тот же эффект, что и последовательное применение процедур `SetRangeStart`, `SetRangeEnd`, `ApplyRange`. В качестве параметров задаются массивы констант, определяющие начало и конец диапазона. Ниже приведёна реализация фильтрации диапазона с использованием метода

`SetRange`.

```
procedure TForm1.Button6Click(Sender:TObject);  
var RangeS, RangeE:longint;  
begin  
Table1.CancelRange; case  
RadioGroup1.ItemIndex of 0: begin  
    RangeS:=strtoint(Edit3.Text);  
    RangeE:=strtoint(Edit4.Text);  
    Table1.SetRange([RangeS],[RangeE]); end;  
1:Table1.SetRange([Edit3.Text],[Edit4.Text]);  
end;  
end;
```

2.8. Поиск

2.8.1. Поиск записи по любому полю

Для поиска записи по любому полю предназначены методы `Locate` и `Lookup`. Функция `Locate` ищет первую запись, удовлетворяющую условию поиска, и делает её текущей. Если запись найдена, функция `Locate` возвращает `true`, иначе `false`.

```
Locate(KeyF:String;KeyV:String;Optns:TLocateOptions):Boolean;
```

В описании функции приняты обозначения: `KeyF` - имя поля, по которому выполняется поиск; `KeyV` - искомое значение поля; `Optns` - параметры поиска.

Параметры поиска `[loCaseInsensitive,loPartialKey]` влияют на чувствительность к регистру и возможность поиска по частичному совпадению.

Часто целесообразно использовать конструкцию, позволяющую сообщить о том, что запись не найдена.

```
procedure TForm1.Button1Click(Sender:TObject); begin  
if not Table1.Locate('Species Name', Edit1.Text,  
[loCaseInsensitive,loPartialKey]) then
```

```
  begin
```

```
    MessageDlg('Запись не найдена!',mtInformation,[mbOK],0); exit;
```

```
  end; end;
```

Иногда бывает удобно ввести переменные для подготовки параметров функции Locate. Ниже записан вариант реализации примера с использованием переменных.

```
procedure TForm1.Button1Click(Sender: TObject); var  
KeyFields,KeyValues:String; Options:TLocateOptions;
```

```
begin
```

```
KeyFields:='Species Name';
```

```
KeyValues:=Edit1.Text;
```

```
Options:=[loCaseInsensitive, loPartialKey]; if not Table1.Locate(KeyFields, KeyValues, Options)
```

```
then begin
```

```
MessageDlg('Запись не найдена!',mtInformation, [mbOK], 0); exit; end; end;
```

Если поле, по которому осуществляется поиск, входит в индекс, то этот индекс используется для поиска. Если поле входит в несколько индексов, то неизвестно, какой из них будет использован. При поиске по полям, не входящим в индекс, применяются фильтры BDE.

Синтаксис функции Lookup аналогичен синтаксису метода Locate. Функция Lookup находит запись, но не делает её текущей. Указатель текущей записи не меняется. Функция Lookup выполняет поиск только на точное совпадение. Если запись найдена, то Lookup возвращает значения некоторых полей. Если запись не найдена, то возвращается Null .

2.8.2. Поиск записи по индексированному полю

Поиск может выполняться по первичному ключу (главному индексу) или по вторичному индексу. Перед выполнением поиска должен быть задан нужный индекс. Для выполнения поиска используются методы класса TTable:

SetKey - перевод таблицы в режим поиска;

GotoKey - выполнение поиска;

GotoNearest - выполнение поиска при частичном совпадении; FindKey - задание критерия и выполнение поиска;

FindNearest - задание критерия и выполнение поиска при частичном совпадении.

Последовательность действий при организации поиска:

- задать индекс через свойство IndexName или IndexFieldName;
- перейти в режим поиска методом SetKey;
- задать искомое значение поля;
- выполнить поиск методом GotoKey или GotoNearest. Аналогичный результат даёт использование методов FindKey и

FindNearest, но применение этих методов позволяет сократить запись. Процедуры FindKey и FindNearest принимают в качестве параметра искомые значения индексных полей.

В простейшем варианте обработчик события для кнопки **Button1**, задающей поиск по полю CustNo выглядит так:

```
procedure TForm1.Button1Click(Sender: TObject); begin  
Table1.SetKey;//Перевод Table1 в режим поиска.  
Table1.FieldName('CustNo').AsString:=Edit1.Text;//Критерий Table1.GotoKey;  
//Выполнение поиска end;
```

Если поиск выполняется не по первичному индексу, то необходимо в свойстве IndexName определить имя индекса, который будет использоваться. Например, если таблица *customer* имеет вторичный индекс по полю City, то надо в свойстве IndexName указать имя индекса.

```
Table1.IndexName:='CityIndex'; Table1.Active  
:=True;  
Table1.SetKey;  
Table1.FieldName('City').AsString:=Edit1.Text; Table1.GotoKey;
```

Поиск не будет выполняться, если не назначить правильно индекс. Следует обратить внимание на то, что IndexName - это свойство TTable, его нет в

других потомках TDataSet или TDBDataSet. При поиске некоторого значения в БД всегда существует вероятность того, что поиск окажется неудачным. Желательно предусмотреть обработку исключительной ситуации.

Иногда требуется найти не точно совпадающее значение, а близкое к нему, для этого следует вместо GotoKey пользоваться методом GotoNearest.

2.8.3. Инкрементальный локатор

Инкрементальный локатор - это механизм точного или приближённого поиска записей с последующим позиционированием на них курсора компонента **Table**. При реализации обычно применяют:

- компонент **Edit** для задания искомого значения;
- управляющий элемент (кнопку **Button**, пункт меню и т.п.), в обработчике которого реализуется поиск.

Инкрементальный локатор удобно использовать, если по мере ввода символа в **Edit** выполняется переход на запись, которая ближе всего к искомой.

Пусть данные в таблице отсортированы по полю Товар (диван, кресло, кровать, сервант, стенка, стол компьютерный, стол обеденный, стол раздвижной, стул, шкаф). Значение для поиска вводится в однострочный редактор **Edit1**. Для реализации поиска будем использовать команду:

```
Table1.FindNearest([Edit1.text]);
```

Если в **Edit1** ввели символ 'с', то результатом применения метода будет перемещение курсора на первую строку, начинающуюся на 'с' - сервант. Если затем в редакторе **Edit1** добавить символ 'т', то курсор сместится на первую запись, начинающуюся на 'ст' (стенка).

2.9. Положение курсора. Закладки

Закладка (Bookmarks) - это средство, позволяющее запомнить положение курсора в таблице. Часто бывает полезно отметить текущее местоположение в таблице, чтобы в дальнейшем быстро возвратиться к этому месту. При работе с закладками можно использовать один из двух подходов. В любом из вариантов закладка - это переменная, которая хранит информацию о положении курсора.

Способ 1. Использование методов класса TBookmark. Для выполнения

действий с переменной типа TBookmark можно использовать методы:

function GetBookmark:TBookmark - установить закладку в таблице; **procedure**

GotoBookmark(Bookmark:TBookmark) - перейти на закладку; **procedure**

FreeBookmark(Bookmark:TBookmark) - освободить память.

Предлагается реализовать такой алгоритм работы с закладкой:

- объявить переменную типа TBookmark
- установить закладку функцией GetBookmark;
- выполнить нужные перемещения по таблице;
- перейти на закладку с помощью процедуры GotoBookmark;
- освободить память процедурой FreeBookmark.

Вызов GetBookmark возвращает переменную типа TBookmark. Класс TBookmark содержит достаточно информации, чтобы найти местоположение текущего указателя. Поэтому для немедленного перехода на закладку можно просто передать значение переменной типа TBookmark функции GotoBookmark. Так как вызов GetBookmark выделяет память для TBookmark, то необходимо вызывать FreeBookmark до окончания программы и перед каждой попыткой повторного использования TBookmark (в GetBookmark) для освобождения памяти.

Способ 2. Использование свойства Bookmark таблицы TTable. Свойство Bookmark класса TDataSet имеет тип TBookmarkstr. Алгоритм решения задачи:

- запомнить значение свойства Bookmark в локальной переменной;
- выполнить нужные перемещения по таблице;
- восстановить значение свойства Bookmark из локальной переменной. Для запоминания положения курсора помимо закладок можно использовать свойство RecNo, содержащее номер текущей записи.

Способ 3. Использование свойства RecNo таблицы TTable. Свойство RecNo класса TBDEDataSet имеет тип Longint. Алгоритм решения задачи:

- запомнить значение свойства RecNo в локальной переменной;

- выполнить нужные перемещения по таблице;
- восстановить значение свойства RecNo из локальной переменной. *Пример 2.1.*

Разработать функцию пользователя для вычисления среднего

значения поля. При этом положение указателя в таблице не должно меняться и данные на экране не должны перерисовываться. Использовать закладку по способу 1.

```
function MyBookmark(Table:TDataSet):Extended;
    //Вычисление среднего значения поля Amt Paid var B:TBookmark;
begin
    Result:=0; try
    B:=Table.GetBookmark; // Установили закладку Table.DisableControls;
    Table.First; while not Table.Eof do begin
        Result:=Result+Table.FieldValues['Amt_Paid']; Table.Next; end;
    Result:=Result/Table.RecordCount; Table.GotoBookmark(B); // Перешли на
    закладку Table.FreeBookmark(B); finally Table.EnableControls;
    end; end;
```

В приведённом примере использованы методы DisableControls и EnableControls класса TDataSet, которые применяются для того, чтобы временно запретить, а затем разрешить вывод записей. Если этого не сделать, то при программном перемещении по записям каждый раз при вызове метода Next визуальные компоненты, с которыми связана таблица, будут перерисовываться. Перерисовка требует существенно больше времени, чем собственно сканирование, и сопровождается мельканием на экране.

При реализации функции использован блок try...finally, что позволяет в любом случае восстановить вывод данных на экран.

При вызове функции MyBookmark1 в параметре Table указывается таблица, которая будет использоваться. Однако поле указано конкретно (AmtPaid), что существенно ограничивает область применения функции.

Пример 2.2. Разработать функцию пользователя для вычисления суммы значений по заданному полю. При этом положение указателя в таблице не должно меняться и данные на экране не должны перерисовываться. Использовать закладку по способу 2.

```
function MyBookmark2(Table:TDataSet;f:string):Extended;
    //Сумма по заданному полю var B:TBookmarkstr;
//Использование свойства Bookmark begin
Result := 0;
Try
    B:=Table.Bookmark; //Запомнили положение курсора Table.DisableControls;
    //Отключили вывод на экран Table.First; while not Table.Eof do begin
        Result:=Result+Table.FieldValues[f]; Table.Next; end;
    Table.Bookmark:=B; //Восстановили положение курсора Table.EnableControls;
//Разрешили вывод на экран except
    ShowMessage('Ошибка при выполнении функции MyBookmark2'); end; end;
```

При реализации функции использован блок try...except, что позволяет обработать ошибки. Функция MyBookmark2 имеет два параметра. При вызове функции в качестве параметра Table задаётся таблица, которая будет использоваться. Параметр f предназначен для задания поля, что делает функцию MyBookmark2 более универсальной, чем MyBookmark1.

Пример 2.3. Разработать функцию для вычисления максимального значения поля, указанного пользователем. При этом положение указателя в таблице не должно меняться и данные на экране не должны перерисовываться. Для задания позиции курсора использовать свойство RecNo.

```
function Remember_rec(Table:TDataSet;f:string):Extended;
    //Максимальное значение заданного поля var B:longint;
//Использование свойства RecNo begin
Result:=0;
try
    with Table do
```

```

begin
  B:=RecNo; //Запомнили номер записи
  DisableControls;
  First;
  try
    Result:=FieldValues[f];
  except
    ShowMessage('Ошибка при задании поля в Remember_rec'); exit; end;
  While not Eof do begin
    Next;
    if (Result<FieldValues[f]) then Result:=FieldValues[f]; end;
  EnableControls;
  RecNo:=B; //Переместили курсор в начальное положение end; except
  ShowMessage('Ошибка при выполнении функции Remember_rec'); end; end;

```

При реализации функции Remember_rec использованы вложенные блоки try...except, что позволяет локализовать ошибки. Внутренний блок реагирует на ошибки при задании имени поля.

2.10. Создание диаграмм

Для построения диаграмм в приложениях, использующих базы данных, предназначен компонент **DBChart**, который позволяет строить диаграммы различных типов, в том числе и объёмные. Важнейшим свойством **DBdiart** является Series[Index:Longint]:TChartSeries, представляющее собой массив диаграмм, выводимых в области компонента. Каждая серия содержит последовательность значений, по которым выполняется построение графика (диаграммы). Один компонент **DBChart** позволяет построить несколько графиков в одних координатах.

Для отображения серии данных используются классы TAreaSeries, TBarSeries, TLineSeries, TFastLineSeries, THorizBarSeries, TPie-Series , TPointSeries и др., соответствующие разным типам диаграмм.

При построении диаграмм часть свойств задаётся в целом для компонента **DBdiart**, а некоторые свойства уточняются для отдельных серий. Задать значения свойств для получения качественной иллюстрации можно на этапе проектирования или программно во время работы.

Так как параметров очень много, то для их настройки на этапе проектирования обычно используют окно редактора, который вызывается через контекстное меню **DBChart**. По сравнению с работой в Инспекторе объектов это проще и нагляднее. В редакторе две страницы: **Chart** и **Series**, каждая из которых имеет несколько вкладок. На вкладках страницы **Series** задаются параметры, относящиеся к конкретной серии данных.

Для добавления графика в окне редактора используется кнопке **Add** на вкладке **Series** страницы **Chart**. Щелчок по кнопке **Add** выводит галерею доступных типов графиков. После выбора типа графика появляется значок, соответствующий этому типу, и надпись **Series1** (рис. 17). Становятся доступными кнопки **Delete**, **Clone**, позволяющие удалить и продублировать диаграмму, а также кнопка **Title**, открывающая диалоговое окно для задания имени серии.

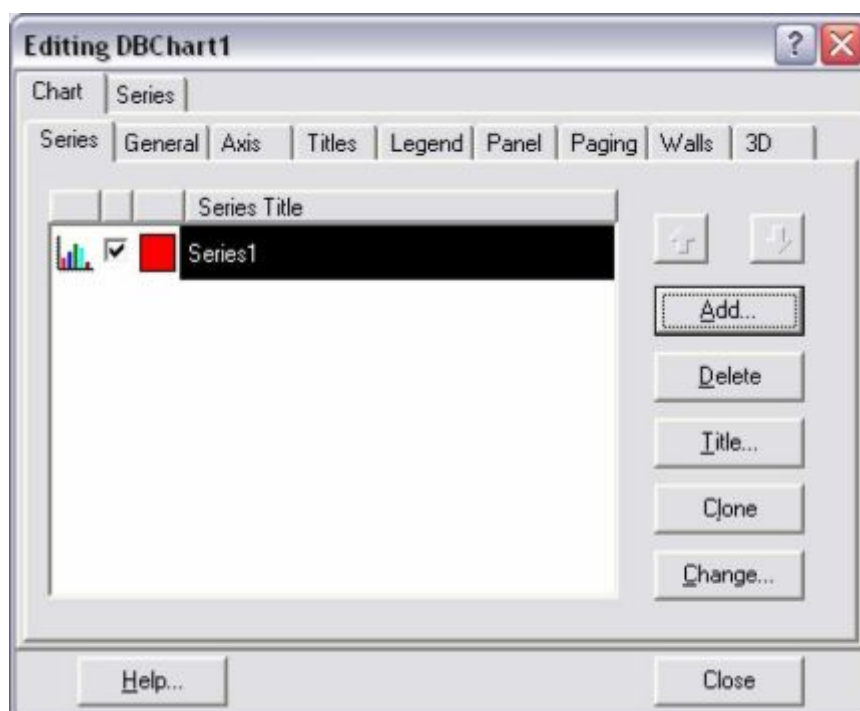


Рис. 17. Задание типа графика

В программном коде в разделе описания класса формы появляется тип, соответствующий выбранному графику, например: Series1:TBarSeries. В дальнейшем тип графика можно будет изменить. Для этого используется кнопка **Change**. Для задания данных необходимо перейти на страницу Series (рис. 18).

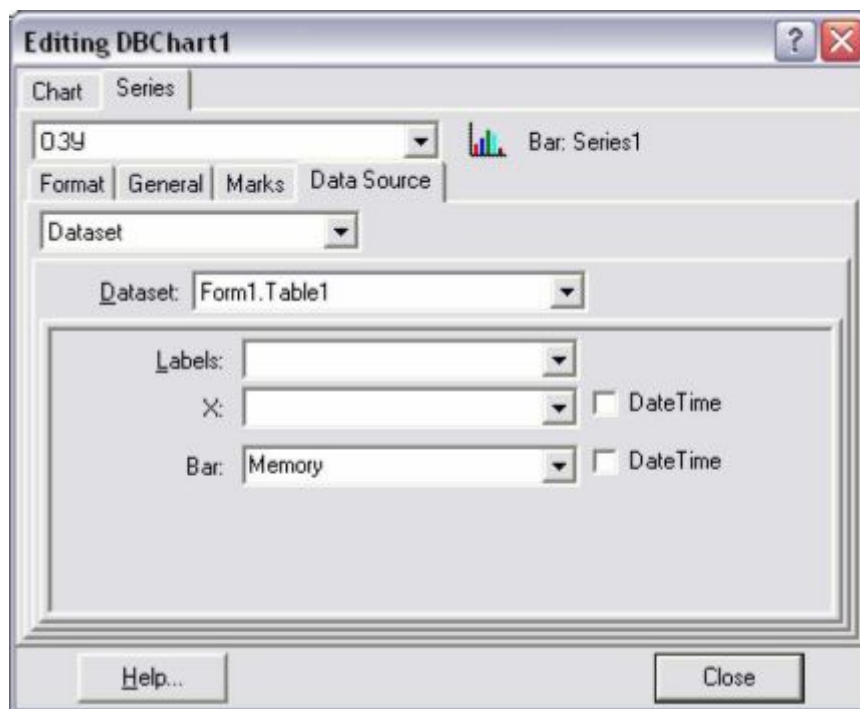


Рис. 18. Задание данных для построения

На вкладке **DataSource** в качестве источника данных надо выбрать Data-set и в поле **Dataset** указать компонент (**Table, Query**), из которого будут поступать данные, а затем выбрать поле. К этому моменту компонент **Table (Query)** должен быть уже установлен и связан с соответствующей таблицей.

В зависимости от типа диаграммы при построении могут использоваться данные из одного или двух полей. Поясняющие надписи задаются метками (Labels), значения которых берутся из символьных полей или полей типа Дата-время. Если в списке **Dataset** выбрано No Data, то источник данных на этапе проектирования не задаётся, он должен быть определён при выполнении приложения. В качестве источника данных можно выбрать функцию. Функция применяется к уже имеющимся сериям данных. Например, позволяет для двух графиков построить третий, который будет их разностью (Subtract) или суммой.

2.11. Контрольные вопросы

1. Какие используются методы для программного перемещения по записям?
2. Как программно открыть (закрыть) набор данных?
3. Для чего и как используются свойства EOF и bOF класса TDataSet?
4. Как программно выполнить удаление записи?
5. Для чего используются процедуры Edit, Post, Insert, Append, Delete?
6. Как программно выполнить редактирование записи?
7. Как добавить запись?
8. Как очистить таблицу БД?
9. Как удалить таблицу из БД?
10. Назовите способы обращения к полю записи.
11. Как изменить порядок записей в таблице?
12. Что такое фильтрация данных?
13. Для чего и как используется свойство Filter компонента **Table**?
14. Приведите правила записи условия в свойстве Filter компонента **Table**.
15. Для чего и как используется свойство Filtered компонента **Table**?
16. Когда целесообразно использовать событие OnfilterRecord?
17. Как ограничить диапазон просматриваемых записей?
18. Как найти нужную запись? Перечислите разные способы.
19. Какие используются методы для поиска записи по индексированному (неиндексированному) полю?
20. Что такое закладка?
21. Для чего и как используется свойство Bookmark класса TTable?
22. Как установить закладку?
23. Перечислите методы работы с закладками.
24. Как используются закладки?
25. В каких случаях целесообразно отключать визуальные компоненты от источника данных?

3. ОСНОВЫ SQL

3.1. Стандарты и реализации языка SQL

SQL - это язык, который обеспечивает доступ к информации и позволяет управлять реляционными базами данных. Язык SQL - Structured Query Language - структурированный язык запросов, обычно произносится как СИКВЭЛ или ЭСКЮЭЛЬ.

Основы языка были заложены Э. Коддом. Язык SQL оперирует данными, представленными в виде логически связанных таблиц. Особенность языка заключается в ориентации на конечный результат обработки данных, а не на процедуру обработки (как в процедурных языках). Для решения задачи надо в терминах языка сформулировать, **что** требуется получить, а возможности того, **как** это сделать, реализованы в языке.

Язык SQL используется на разных платформах, позволяет работать с реляционными базами данных различных форматов. К настоящему времени выпущено несколько стандартов языка: SQL 86, SQL 89, SQL 92, SQL 99. Стандарты разрабатываются и принимаются двумя организациями: **ANSI** (Американским национальным институтом стандартов) и **ISO** (Международной организацией по стандартизации).

Стандарт SQL 86 определяет минимальный стандартный синтаксис. Был выпущен **ANSI** и поддержан **ISO**. Стандарт SQL 89 ввёл набор дополнительных операторов. Стандарт SQL 92 определил три уровня соответствия: основной, средний и полный. В большинстве случаев производители, объявившие о поддержке этого стандарта, реализовывали только основной уровень.

Стандарт SQL 99 ввёл в язык объектные и некоторые процедурные расширения. В этом стандарте определено обязательное ядро и уровни расширений. Ядро включает в себя основной уровень стандарта SQL 92. Уровни расширения не являются обязательными для реализации.

На практике большинство коммерческих СУБД расширяют SQL по своему усмотрению, добавляя различные особенности, которые, по мнению производителей, будут весьма полезны. Как правило, вносимые расширения диктуются текущими потребностями и могут быть учтены в новых стандартах. Таким образом, несмотря на наличие стандартов, реализации языка SQL значительно отличаются друг от друга. Приступая к использованию SQL для работы с конкретной базой данных, надо ознакомиться с описанием языка по документации соответствующей СУБД.

3.2. Группы операторов SQL

Язык SQL определяет набор операторов (команд), типы данных, набор встроенных функций. Так как язык SQL предназначен для манипулирования данными в реляционных базах данных, определения структуры баз данных и для управления правами доступа к данным в многопользовательской среде, то в него в качестве составных частей входят:

язык манипулирования данными (Data Manipulation Language, DML);

язык определения данных (Data Definition Language, DDL);

язык управления данными (Data Control Language, DCL).

Это не отдельные языки, а различные команды одного языка. Деление проведено только лишь с точки зрения функционального назначения команд.

Язык манипулирования данными (DML, ЯМД) состоит из четырёх основных команд:

SELECT (выбрать) - извлечь данные из одной или нескольких таблиц; INSERT (вставить) - добавить строки в таблицу; UPDATE (обновить) - изменить значения полей в таблице; DELETE (удалить) - удалить строки из таблицы.

Некоторые авторы наиболее широко используемую команду SELECT выделяют в отдельную группу - язык запросов (DQL). Язык запросов составляет единственная команда SELECT со всеми своими многочисленными опциями и предложениями.

Язык определения данных (DDL, ЯОД) используется для создания (CREATE), изменения (ALTER), удаления (DROP) структуры базы данных и ее составных частей - таблиц, индексов, представлений (виртуальных таблиц), а также триггеров и хранимых процедур. Таким образом, DDL управляет объектами базы данных. У каждой СУБД свой набор объектов. Основными командами языка определения данных являются:

CREATE DATABASE - создать базу данных;

CREATE TABLE - создать таблицу;

CREATE VIEW - создать виртуальную таблицу;

CREATE INDEX - создать индекс;

CREATE TRIGGER - создать триггер;

CREATE PROCEDURE - создать хранимую процедуру;

ALTER DATABASE - модифицировать базу данных;

ALTER TABLE - модифицировать таблицу;

ALTER VIEW - модифицировать виртуальную таблицу;

ALTER INDEX - модифицировать индекс;

ALTER TRIGGER - модифицировать триггер;

ALTER PROCEDURE - модифицировать сохраненную процедуру;

DROP DATABASE - удалить базу данных;

DROP TABLE - удалить таблицу;

DROP VIEW - удалить виртуальную таблицу;

DROP INDEX - удалить индекс;

DROP TRIGGER - удалить триггер;

DROP PROCEDURE - удалить хранимую процедуру.

Язык управления данными (DCL) используется для управления правами доступа к данным и для управления выполнением процедур в многопользовательской среде. Более точно его можно было бы назвать *языком управления доступом*. Он состоит из двух основных команд:

GRANT - дать права;

REVOKE - забрать права.

С точки зрения прикладного интерфейса существуют две разновидности языка SQL: интерактивный и встроенный. Интерактивный SQL позволяет в интерактивном режиме вводить запросы, посылать их на выполнение и получать результаты в предназначенном для этого окне. Интерактивный SQL используется в специальных утилитах (например, SQL Explorer).

Встроенный SQL применяется в прикладных программах, написанных на других языках программирования, позволяет посылать запросы и обрабатывать полученные результаты, в том числе комбинируя навигационный (record-ориентированный) и реляционный (set-ориентированный) подходы.

Переходя от навигационного способа работы с данными к реляционному, следует учитывать, что в языке SQL СУБД Paradox запись (Record) обозначается Row, а поле (Field) - Column.

3.3. Выполнение запросов в SQL Explorer

Утилита SQL Explorer позволяет вводить и выполнять SQL-запросы. Если в окне утилиты в меню **Options** выбрать команду **Query**, то откроется окно для уточнения правил, используемых при записи запроса (рис. 19). Это окно позволяет задать синтаксис, принятый в конкретной реализации языка SQL. По умолчанию используется синтаксис, принятый в продуктах фирмы Borland.



Рис. 19. Окно для уточнения синтаксиса

Для формирования запроса следует в области *Databases* утилиты выбрать базу данных, открыть список таблиц и указать нужную таблицу (рис. 20).

Справа появится поле с тремя закладками:

- закладка *Definition* содержит описание таблицы;
- закладка *Data* - данные;
- закладка *Enter SQL* - инструменты для записи и выполнения запроса. Надо перейти на страницу *Enter SQL* и ввести текст запроса.

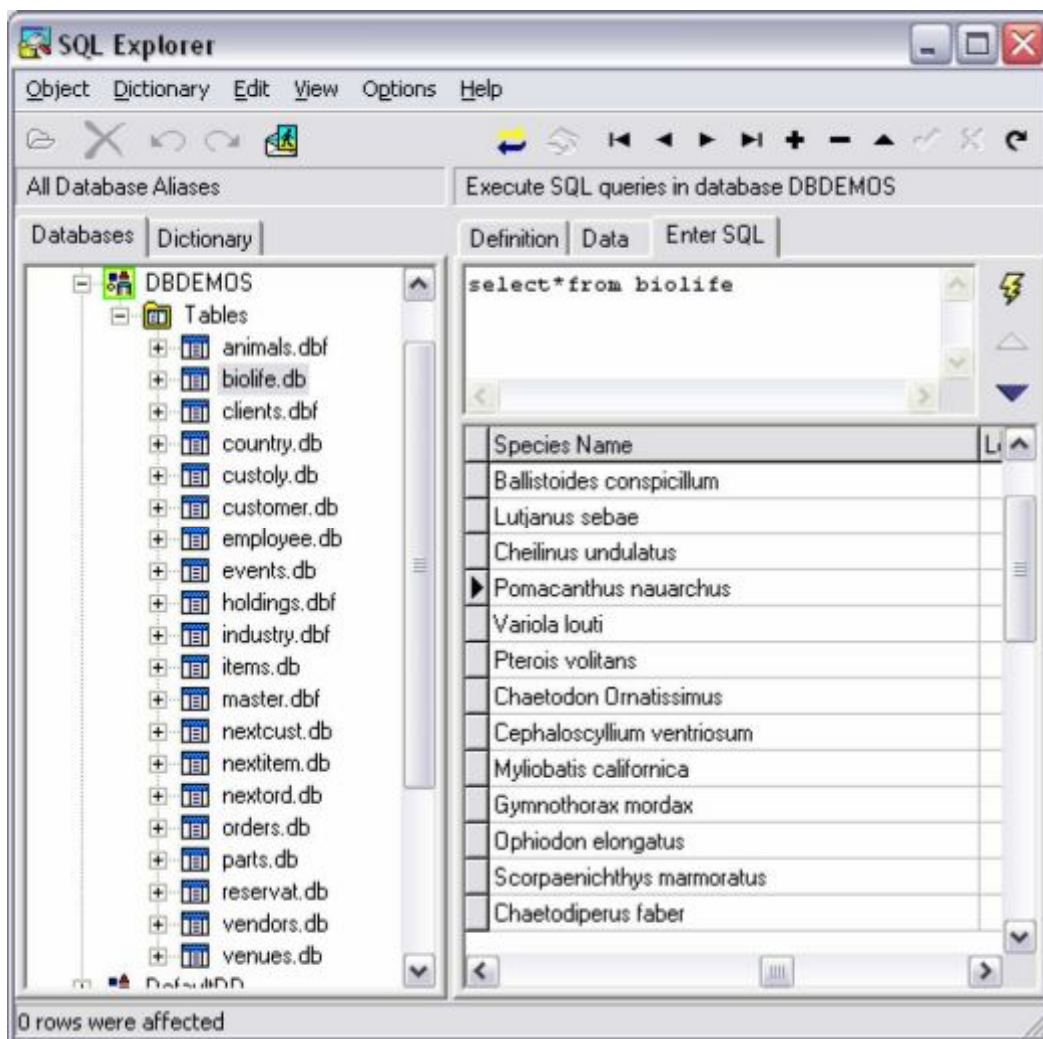


Рис. 20. Работа с запросами в SQL Explorer

Страница для работы с запросами содержит поле редактора для записи текста запроса и расположенные справа кнопки для задания действий с запросами: **Execute Query** - выполнить;

Previous Query, Next Query - кнопки для перемещения по запросам. Сверху находятся кнопки для навигации и работы с записями. Эти кнопки имеют такие же обозначения, как кнопки компонента **DBNavigator**.

После ввода запроса надо нажать кнопку **Execute Query** - появится таблица с результатами.

Кроме того, через контекстное меню редактора запросов можно загрузить текст запроса из файла.

Для ввода нового запроса следует нажать кнопку **Next Query** и записать запрос. Иногда удобно скопировать старый запрос в буфер, нажать кнопку **Next Query**, вставить текст из буфера и внести в него изменения.

Протестированные запросы можно через буфер передать в нужную программу или сохранить в файле. Для сохранения в файле следует открыть контекстное меню для поля редактора текста запроса и выбрать соответствующую команду.

3.4. Применение языка SQL

Реляционные базы данных имеют мощный теоретический фундамент, основанный на математической теории множеств. Основными операциями над отношениями являются: *выборка* (Restriction), *проекция* (Projection), *соединение* (Join), *объединение* (Union), *пересечение*, *разность*, *декартово произведение* (Cross Join).

Далее рассматриваются конструкции языка SQL и приводятся примеры, иллюстрирующие суть операций. При этом необходимо учитывать следующие моменты:

- в примерах используются таблицы поставляемой с системой Delphi демонстрационной базы данных DBDemos;
- в командах SQL зарезервированные слова записываются прописными буквами;
- для лучшего усвоения материала целесообразно сразу выполнять примеры и анализировать получаемые наборы данных;
- удобным средством для тестирования примеров является SQL Explorer.

Операции *выборки* и *проекции* являются унарными, поскольку они работают с одним отношением.

Операция *выборки* - это построение горизонтального подмножества, т.е. получение части строк, обладающих заданными свойствами. Операция *выборки* работает с одной таблицей и позволяет получить либо все, либо те строки таблицы, которые удовлетворяют заданному условию (предикату). Напомним, что под предикатом понимается некоторое специфицированное условие, значение которого имеет булевский тип.

Например, вывести все строки таблицы *country*:

```
SELECT * FROM country
```

или из таблицы *orders* извлечь сведения о заказах, оплаченных в кредит:

```
SELECT * FROM orders WHERE PaymentMethod="Credit"
```

 Операция *проекции* - это построение вертикального подмножества отношения, т. е. выделение подмножества столбцов таблицы. Операция *проекции* применяется к одной таблице и в качестве результата выдаёт таблицу, у которой оставлена только часть атрибутов и исключены строки-дубликаты.

Например, вывести названия фирм, город и страну из таблицы *vendors*:

```
SELECT
```

```
VendorName, City, Country FROM vendors
```

 На практике очень часто требуется получить

подмножество столбцов и строк таблицы - выполнить комбинацию *выборки* и *проекции*. Для этого достаточно перечислить столбцы таблицы и наложить ограничения на строки.

Например, получить фамилии работников, которых зовут Roger:

```
SELECT firstName, lastName FROM employee WHERE firstName="Roger"
```

Декартово произведение $R \times S$ двух отношений (двух таблиц) определяет новое отношение - результат сцепления каждой записи из отношения R с каждой записью из отношения S .

Пусть таблица R имеет поля a_1, a_2 и таблица S имеет поля b_1, b_2 . Оператор

```
SELECT R.a1, R.a2, S.b1, S.b2 FROM R, S
```

 сформирует результирующую таблицу, причём если одна из исходных таблиц имеет N записей и K полей, а другая - M записей и L полей, то их *декартово произведение* будет содержать $N \times M$ записей и $K+L$ полей. Исходные таблицы могут содержать поля с

одинаковыми именами, тогда имена полей полученного набора данных будут содержать названия таблиц для обеспечения уникальности имён.

Как правило, пользователя интересует только та часть записей декартова произведения, которая удовлетворяет некоторому условию, поэтому вместо декартова произведения используется одна из самых важных операций реляционной алгебры - операция *соединения*.

Если в предложении FROM указано более одной таблицы, то эти таблицы соединяются. По умолчанию результирующая таблица представляет собой перекрёстное соединение (Cross Join) или декартово произведение.

Операция объединения (UNION) позволяет объединить результаты отдельных запросов по нескольким таблицам в единую результирующую таблицу. Таким образом, предложение UNION объединяет вывод двух или более SQL-запросов в единый набор строк и столбцов. При этом результаты запросов должны быть совместимы, т. е. иметь одинаковое количество полей с совпадающими типами данных (быть совместимыми по *объединению*).

Пример 3.1. Из таблиц employee и country получить список работников и заказчиков, проживающих во Франции:

```
SELECT first_name,last_name,job_country FROM employee WHERE job_country =  
"France"
```

UNION

```
SELECT contact_first,contact_last,country FROM customer WHERE country = "France"
```

Операция *пересечения* отношений R и S определяет отношение (таблицу), которое содержит записи, присутствующие как R, так и в S. Отношения R и S должны быть совместимы по *объединению*. Таким образом, *пересечением* двух таблиц - R и S является таблица, содержащая все строки, присутствующие в обеих исходных таблицах одновременно.

Разность двух отношений R и S состоит из записей, которые имеются в отношении R, но отсутствуют в отношении S. Причем отношения R и S должны быть совместимы по *объединению*. Таким образом, *разностью* двух таблиц R и

S является таблица, содержащая все строки, которые присутствуют в таблице R, но отсутствуют в таблице S.

Наиболее важной командой языка манипулирования данными является команда SELECT.

3.5. Формирование запросов средствами языка SQL

Оператор SQL состоит из *зарезервированных слов* и из слов, определяемых пользователем. *Зарезервированные слова* являются постоянной частью языка SQL, имеют фиксированное значение, их нельзя разбивать на части. Слова, определяемые пользователем, представляют собой имена различных объектов базы данных и записываются в соответствии с синтаксическими правилами. Слова в операторе располагаются строго в определённой последовательности.

Имена формируются из символов алфавита, заданного стандартом языка. Разрешено использовать строчные и прописные буквы латинского алфавита (A-Z, a-z), цифры (0-9) и символ подчёркивания (_). Имя может иметь длину до 128 символов, должно начинаться с буквы и не может содержать пробелы.

Большинство элементов языка нечувствительны к регистру. Язык SQL имеет свободный формат, то есть там, где допустим один пробел между элементами, разрешено ставить любое количество пробелов и пустых строк. Это позволяет SQL-операторы и их фрагменты записывать с использованием отступов и выравнивания, что облегчает чтение и понимание запросов.

Точка с запятой является стандартным разделителем, но в некоторых реализациях (в частности, в компоненте **Query**) разделитель в конце команды необязателен.

Набор объектов, используемых в базе данных, зависит от СУБД. К основным объектам относятся *таблицы, представления, хранимые процедуры, триггеры, индексы, ключи*, создаваемые пользователем *функции, ограничения целостности* и др.

Представлениями (просмотрами) называют виртуальные таблицы, содержимое которых определяется запросом. Подобно реальным таблицам пред

ставления содержат именованные столбцы и строки с данными. Для конечных пользователей представление выглядит как таблица, но в действительности оно только представляет данные, расположенные в одной или нескольких таблицах. Информация, которую видит пользователь через представление, не сохраняется в базе данных как самостоятельный объект. Таблицы - на диске, представления - в оперативной памяти.

Хранимые процедуры представляют собой группу команд SQL, объединённых в один модуль. Такая группа команд компилируется и выполняется как единое целое.

Триггеры называется специальный класс хранимых процедур, автоматически запускаемых при добавлении, изменении или удалении данных из таблицы. Триггеры выполняются до или (и) после события изменения записи в таблице.

3.6. Оператор SELECT 3.6.1. Общая

форма команды SELECT

Общая форма оператора SELECT приводится в стандартах. В более простых случаях достаточно воспользоваться только некоторыми возможностями оператора SELECT:

SELECT [DISTINCT] список_выбираемых_полей

FROM список таблиц или представлений

[WHERE условие_отбора_строк]

[GROUP BY спецификация группировки

[HAVING условие_отбора_групп]]

[UNION другое_выражение_Select]

[ORDER BY спецификация сортировки];

Квадратные скобки означают необязательность использования дополнительных конструкций команды.

Простейшие конструкции языка SQL позволяют: - назначать поля, которые должны быть выбраны;

- назначать к выборке все поля;
- управлять вертикальным и горизонтальным порядком выбираемых данных;
- подставлять собственные заголовки полей в результирующей таблице;
- производить вычисления в списке выбираемых элементов;
- использовать литералы в списке выбираемых элементов;
- ограничивать число возвращаемых строк;
- формировать сложные условия поиска;
- устранять одинаковые строки из результата.

3.6.2. Поля и предложение FROM

После слова **SELECT** приводится список выражений, определяющий значения, формируемые запросом. В самом простом случае список выражений является *списком полей таблицы*.

В предложении **FROM** перечисляются все объекты (один или несколько), из которых производится выборка данных. Каждая таблица или представление, которые упоминаются в запросе, должны быть перечислены в предложении **FROM**. В простейшем случае после слова **FROM** записывается имя таблицы, из которой извлекаются данные. Если требуется извлечение значений всех полей, то вместо списка полей можно указать символ *****. Например, чтобы получить сведения из всех полей таблицы **country**, надо записать: **SELECT * FROM country**

Для получения данных из определённых полей используется команда, в которой после слова **SELECT** перечислены только нужные поля: **SELECT Last_Name,First_Name,City,Country,Phone FROM custoly**

Если требуется вывести имя, фамилию, телефон, а затем другие поля, то надо просто перечислить имена полей в требуемом порядке: **SELECT First_Name,Last_Name,Phone,City,Country FROM custoly**

Для уточнения объекта, которому принадлежит поле, перед именем поля указывается имя объекта. Задание составного имени
имя таблицы.имя поля

является обязательным при использовании нескольких таблиц или представлений, а также при использовании имён полей с пробелами. При задании полей, имена которых содержат пробел, надо использовать кавычки или апострофы.

Например, при выводе данных из таблицы `biolife` для полей `Species Name` и `Length (cm)` используются составные имена:

```
SELECT Category,Common_Name,biolife."Species Name", biolife."Length (cm)"
FROM biolife
```

```
SELECT Category,Common_Name,biolife.'Species Name', biolife.'Length (cm)'  
FROM biolife
```

```
SELECT biolife.Category,biolife.Common_Name,  
        biolife.'Species Name',biolife.'Length (cm)'  
FROM biolife
```

3.6.3. Литералы

Для придания большей наглядности получаемому результату можно использовать литералы. Литералы - это строковые константы, которые применяются наряду с наименованиями столбцов и позволяют сопровождать выводимые данные пояснениями. Строковая константа записывается в кавычках или апострофах.

```
SELECT LastName,"получает",Salary," в год" FROM employee или
```

```
SELECT LastName,'получает'Salary,' в год' FROM employee
```

К сожалению, могут возникнуть проблемы с кириллицей.

3.6.4. Конкатенация

Имеется возможность сцеплять данные из двух или более столбцов, имеющих строковый тип, друг с другом, а также соединять их с литералами. Для этого используется операция конкатенации, которая задаётся двумя вертикальными чёрточками (`||`).

```
SELECT FirstName||' '||LastName,HireDate FROM employee
```

Этот запрос выводит список сотрудников с указанием даты поступления на работу.

3.6.5. Использование квалификатора AS

Для придания наглядности получаемым результатам наряду с литералами в списке выбираемых элементов можно использовать квалификатор AS. Данный квалификатор заменяет в результирующей таблице существующее название поля на заданное. Таким способом можно дать название создаваемому в запросе полю (например, вычисляемому) или заменить реальное имя на другое, более простое либо более понятное пользователю.

```
SELECT VenueNo,Event_Name AS Name,  
       Event_Description AS Description FROM events
```

```
SELECT VenueNo,Event_Name AS Name,  
       Event_Date AS events.'Date',Event_Time AS events.'Time'
```

```
FROM events
```

В последнем запросе вместо имён полей EventName, EventDate, EventTime для столбцов используются названия Name, Date, Time. Поскольку идентификаторы Date и Time в таблицах формата Paradox используются для задания типов данных, то при формировании названий столбцов пришлось использовать составные имена.

3.7. Предложение WHERE 3.7.1.

Ограничения на число выводимых строк

Число возвращаемых в результате запроса строк может быть ограничено путем использования предложения WHERE, содержащего условия отбора. Так как в языке SQL применяется трёхзначная логика, то условие отбора для отдельных строк может принимать значения true, false или unknown. Значение unknown получается при сравнении значения null с любым другим значением, включая null. Запрос возвращает в качестве результата только те строки, для которых предикат имеет значение true. При формировании условия используются следующие операции: сравнения (=, <>, >, <, >=, <=), BETWEEN, IN, LIKE, IS NULL, EXIST, ANY, ALL, SOME.

3.7.2. Операции сравнения

Для выполнения сравнения *элементы должны иметь сравнимые типы*. Если в базе данных определены домены, то сравниваемые элементы должны относиться к одному домену. Элементом сравнения могут выступать: значение поля, литерал, арифметическое выражение, значение, возвращаемое итоговой или другой встроенной функцией, значение, возвращаемое подзапросом.

Пример 3.2. Получить список сотрудников с именем Brown (Lee):

```
SELECT LastName, FirstName, Salary FROM employee WHERE  
LastName='Brown'
```

```
SELECT LastName, FirstName, Salary FROM employee WHERE  
LastName='Lee '
```

При сравнении литералов конечные пробелы игнорируются. Обе команды выдают верный результат.

Пример 3.3. Получить список сотрудников с зарплатой меньше 27 000:

```
SELECT LastName, FirstName, Salary FROM employee WHERE  
Salary<27000
```

Пример 3.4. Получить список фирм-заказчиков с указанием города и страны за исключением заказчиков из Канады:

```
SELECT Company,City,Country FROM customer WHERE  
Country <>'Canada'
```

Пример 3.5. Получить список заказчиков из US с указанием названия фирмы, города и штата:

```
SELECT Company,City,State FROM customer WHERE Country  
='US'
```

Поле, участвующее в формировании условия, необязательно включать в перечень выводимых полей.

3.7.3. Операция BETWEEN

Предикат BETWEEN задает диапазон значений, для которого выражение принимает значение true. Разрешено также использовать конструкцию NOT BETWEEN.

Пример 3.6. Получить список сотрудников, у которых зарплата лежит в диапазоне от 25 000 до 30 000:

```
SELECT LastName,FirstName,Salary FROM employee WHERE Salary  
BETWEEN 25000 and 30000
```

Тот же запрос с использованием операторов сравнения будет выглядеть следующим образом:

```
SELECT LastName,FirstName,Salary FROM employee WHERE  
Salary>=25000 and Salary<=30000
```

Таким образом, при использовании BETWEEN значения, попадающие на границу диапазона, включаются в результирующий набор. Значения, определяющие нижнюю и верхнюю границы диапазона, могут не являться реальными величинами базы данных. И это очень удобно, так как не всегда известны имеющиеся в базе данных значения. Предикат BETWEEN позволяет сравнивать не только числа, но и строки и даты.

Пример 3.7. Получить список сотрудников, фамилии которых начинаются с Nelson и заканчиваются Osborn:

```
SELECT LastName,FirstName,Salary FROM employee WHERE LastName  
BETWEEN "Nelson" AND "Osborne"
```

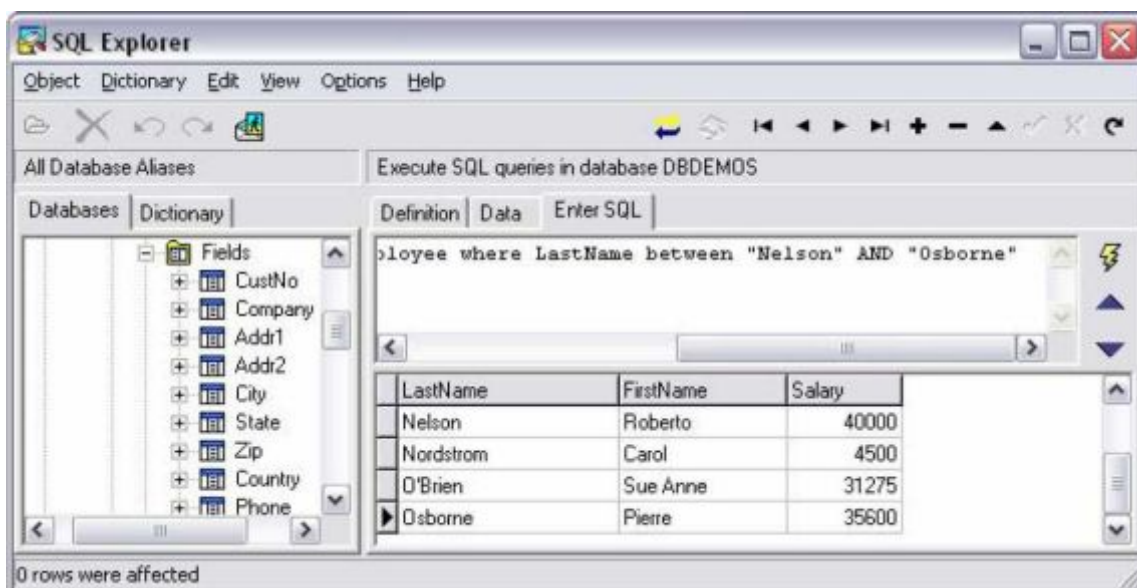


Рис. 21. Результат реализации операции BETWEEN в примере 3.7

Пример 3.8. Вывести список сотрудников, фамилии которых находятся между Nel и Osb:

```
SELECT LastName,FirstName,Salary FROM employee  
WHERE LastName BETWEEN "Nel" AND "Osb"
```

В таблице базы данных employee значений Nel и Osb нет.

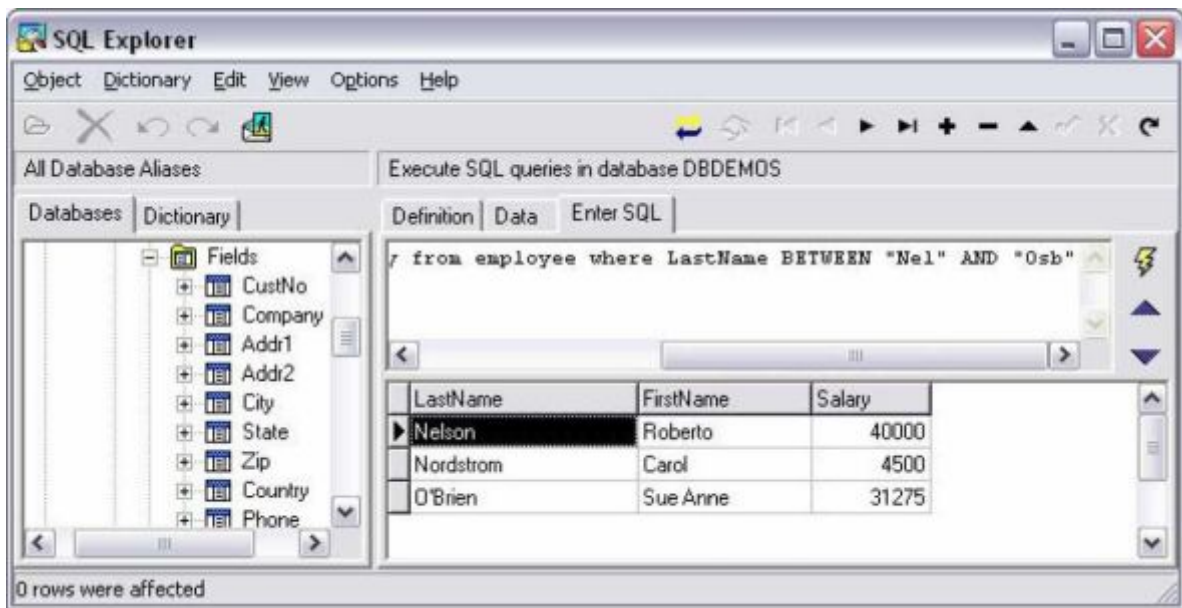


Рис. 22. Результат реализации операции BETWEEN в примере 3.8

Однако сотрудники с фамилиями, начинающимися с символов, для которых выполняется условие «больше или равно Nel» и «не более Osb», попадут в выборку. Фамилии, начинающиеся с символов O, Os, Osb, попадают в заданный диапазон, а Osborne - нет.

Предикат BETWEEN с отрицанием NOT (NOT BETWEEN) позволяет получить записи, у которых значение заданного поля лежит вне введённого диапазона (меньше нижней границы и больше верхней границы).

Пример 3.9. Найти рыб с длиной меньше 10 и больше 80 дюймов (маленьких и очень больших):

```
SELECT Category,Common_Name,Length_In FROM biolife WHERE Length
In NOT BETWEEN 10 AND 80
```

3.7.4. Операция IN

Предикат IN проверяет, совпадает ли заданное значение (например, значение столбца или функция от него) с одним из перечисленных в списке. Элементы списка записываются через запятую в круглых скобках. Если проверяемое значение равно какому-либо элементу в списке, то предикат принимает значение true. Разрешено использовать конструкцию NOT IN.

Пример 3.10. Вывести список компаний из городов Santa Maria, San Jose, Downey:

```
SELECT Company,City FROM customer  
WHERE City in ('Santa Maria','San Jose','Downey')
```

Пример 3.11. Вывести заказы и даты их оплаты, у которых средством оплаты была не Visa и не AmEx:

```
SELECT OrderNo,SaleDate,PaymentMethod FROM orders WHERE  
PaymentMethod not in ('Visa','AmEx')
```

3.7.5. Предикат LIKE

Предикат LIKE используется только с символьными данными. Он проверяет, соответствует ли данное символьное значение указанной подстроке с указанной маской. Предусмотрена также конструкция NOT LIKE.

В подстроке можно применять любые разрешённые символы (с учетом верхнего и нижнего регистров), а также специальные символы:

% - замещает любое количество символов (в том числе и 0),

_ - замещает только один символ.

Пример 3.12. Получить список сотрудников, фамилии которых начинаются с буквы F:

```
SELECT FirstName,LastName FROM employee WHERE  
LastName LIKE"F%"
```

Пример 3.13. Получить список сотрудников, у которых имя заканчивается на «er»:

```
SELECT FirstName,LastName FROM employee WHERE  
FirstName LIKE"%er"
```

3.7.6. Предикат IS NULL

В SQL-запросах NULL означает, что значение столбца *неизвестно*. Условия поиска, в которых значение столбца сравнивается с NULL, всегда принимают значение unknown и, соответственно, приводят к ошибке. Таким образом, в запросах нет смысла применять выражения вида

```
WHERE имя_поля=Ж^
```

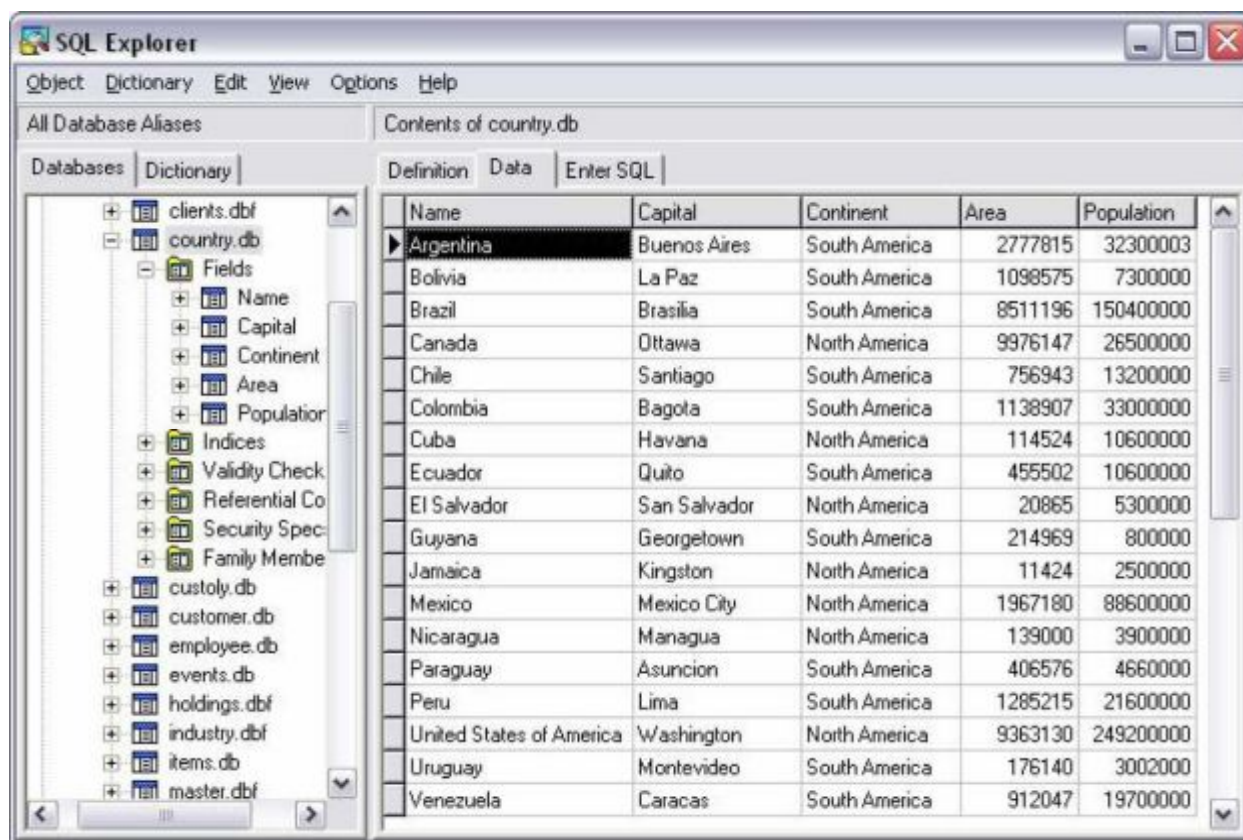
Если требуется определить, имеет ли поле значение, используется предикат IS NULL. Условие, содержащее IS NULL, принимает значение true только

тогда, когда значение поля или выражения имеет значение NULL (пусто, не определено). Разрешено также использовать конструкцию IS NOT NULL, которая означает *не пусто*, имеет *какое-либо значение*. Предикат IS NULL возвращает только значения true или false.

3.7.7. Логические операции

Логические операции AND, OR, NOT позволяют сформировать сложные условия отбора записей. Если в одном выражении используется несколько логических операций, то они выполняются с учётом приоритета: сначала выполняется отрицание NOT, затем AND (логическое И), только после этого OR (логическое ИЛИ). Для изменения порядка выполнения операций разрешается использовать скобки.

Пример 3.14. По таблице country (рис. 23) было выполнено два запроса.



Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777815	32300003
Bolivia	La Paz	South America	1098575	7300000
Brazil	Brasilia	South America	8511196	150400000
Canada	Ottawa	North America	9976147	26500000
Chile	Santiago	South America	756943	13200000
Colombia	Bagota	South America	1138907	33000000
Cuba	Havana	North America	114524	10600000
Ecuador	Quito	South America	455502	10600000
El Salvador	San Salvador	North America	20865	5300000
Guyana	Georgetown	South America	214969	800000
Jamaica	Kingston	North America	11424	2500000
Mexico	Mexico City	North America	1967180	88600000
Nicaragua	Managua	North America	139000	3900000
Paraguay	Asuncion	South America	406576	4660000
Peru	Lima	South America	1285215	21600000
United States of America	Washington	North America	9363130	249200000
Uruguay	Montevideo	South America	176140	3002000
Venezuela	Caracas	South America	912047	19700000

Рис. 23. Исходная таблица

Проанализировать результаты.

3.7.8. Работа с датами

В разных СУБД отличаются встроенные функции для работы с датами и используются разные форматы для представления даты, даты и времени. Причём отличаются как внутреннее, так и внешнее представления. Внешне дата может быть представлена строками различных форматов, например:

"October 27,2008"

"27-OCT-2008"

"10-27-08"

"10/27/08"

"27.10.08"

Пример 3.15. Вывести список сотрудников, принятых на работу после заданной даты.

Использовать разные варианты форматов даты:

```
SELECT FirstName,LastName,HireDate FROM employee WHERE  
HireDate>'1.01.94'
```

```
SELECT FirstName,LastName,HireDate FROM employee WHERE  
HireDate>'1/01/90'
```

```
SELECT FirstName,LastName,HireDate FROM employee WHERE  
HireDate>'27-Oct-1992'
```

Таблицы формата Paradox поддерживают только часть из перечисленных выше форматов. В запросах приведены варианты разрешённых форматов даты.

Пример 3.16. Выполнить сравнение результатов двух вариантов запроса,

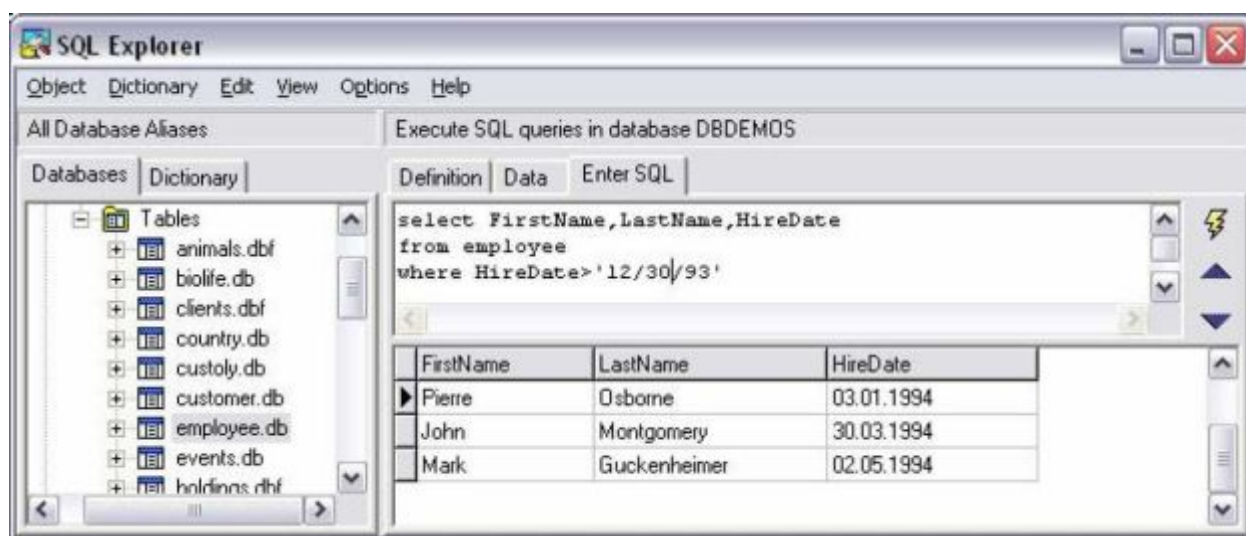


Рис. 26. Дата задана строкой '12/30/93 '

отличающихся только форматом записи даты.

При работе с датами часто возникают проблемы в связи с особенностями форматов записи. Следующие два примера показывают, что для получения корректного результата надо знать, в какой последовательности задаются год, месяц и день.

```
SELECT FirstName,LastName,HireDate FROM employee
        WHERE HireDate>'12/30/93'
```

Дата 12/30/93 означает: 12 - месяц, 30 -день, 93 - год.

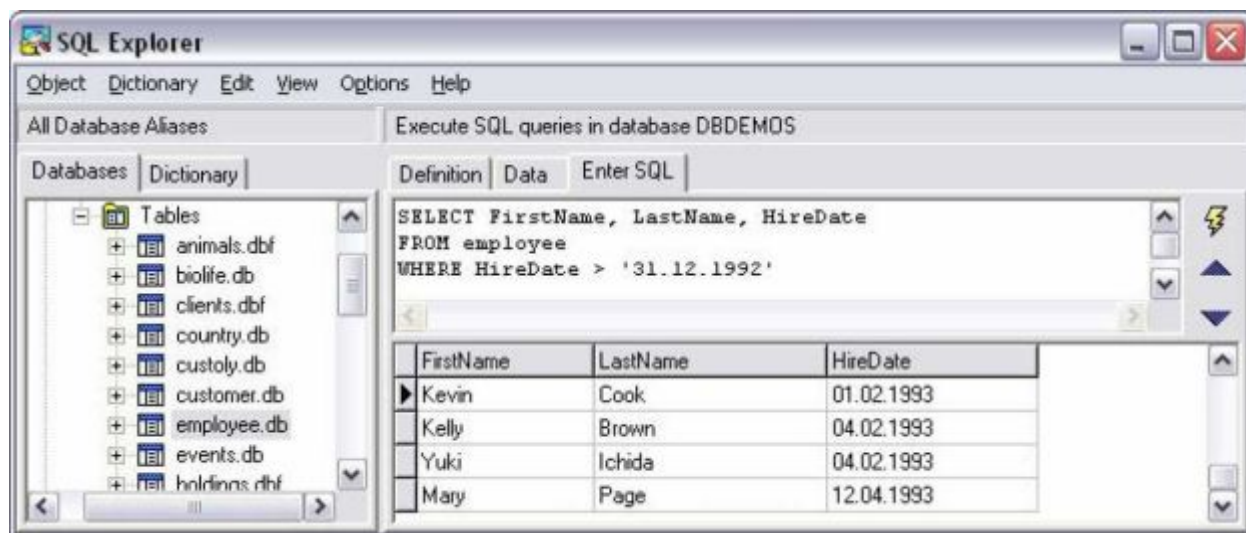


Рис. 27. Дата задана строкой '31.12.1992 '

Привычная форма даты 31.12.1992 означает: 31-день, 12-месяц, 1992-год. Значения дат можно сравнивать друг с другом, вычитать одну из другой. **Пример 3.17.** Получить список служащих, проработавших на предприятии к 1/01/2000 более 8 лет:

```
SELECT FirstName,LastName,HireDate FROM employee WHERE
'1/01/2000'-HireDate > 8*365+2
```

Пример 3.18. Получить список сотрудников, поступивших на работу до 1.01.89 и после 31.12.93, то есть раньше 1989 г. и позже 1993 г.:

```
SELECT FirstName,LastName,HireDate FROM employee
WHERE HireDate NOT BETWEEN "1-JAN-1989" AND "31-DEC-19 93"
```

Кроме абсолютных дат некоторые реализации языка SQL (например, Inter Base) позволяют оперировать относительными значениями: yesterday (вчера), today (сегодня), now (сейчас, включая время), tommorrow (завтра).

Дата может неявно конвертироваться в строку (из строки), если строка, представляющая дату, имеет один из разрешённых форматов и выражение не содержит неоднозначностей в толковании типов столбцов.

3.7.9. Изменение порядка выводимых строк (ORDER BY)

Предложение ORDER BY записывается в конце SQL-запроса и применяется для упорядочивания выводимых строк. Порядок строк определяется значениями полей, имена которых записаны после слов ORDER BY. По умолчанию упорядочивание выполняется по возрастанию. При необходимости способ упорядочивания можно задать явно опциями ASC (по возрастанию) и DESC (по убыванию).

Разрешается задавать порядок выводимых строк по нескольким полям, причём сначала упорядочивание выполняется по первому указанному полю, затем в пределах упорядоченной последовательности строк - по второму и т.д. Способ упорядочивания ASC или DESC задаётся для каждого поля отдельно.

Пример 3.19. Получить список сотрудников в порядке убывания зарплат:

```
SELECT LastName,FirstName,Salary FROM employee ORDER BY  
Salary DESC
```

Пример 3.20. Получить список сотрудников с расположением фамилий по алфавиту:

```
SELECT LastName,FirstName,Salary FROM employee ORDER BY  
LastName
```

Пример 3.21. Вывести перечень государств, упорядоченный по алфавиту названий континентов, а в пределах континента - по убыванию населения:

```
SELECT Continent,Name,Area,Population FROM country ORDER BY Continent  
ASC, Population DESC
```

3.7.10. Устранение дублирования

В таблицах реляционных баз данных по определению нет одинаковых строк. Однако при выполнении запросов часто выводится только часть полей и в результирующем наборе появляются строки с одинаковыми значениями. Для устранения дублирования служит модификатор DISTINCT. Этот модификатор указывается один раз в списке выбираемых элементов и действует на весь список.

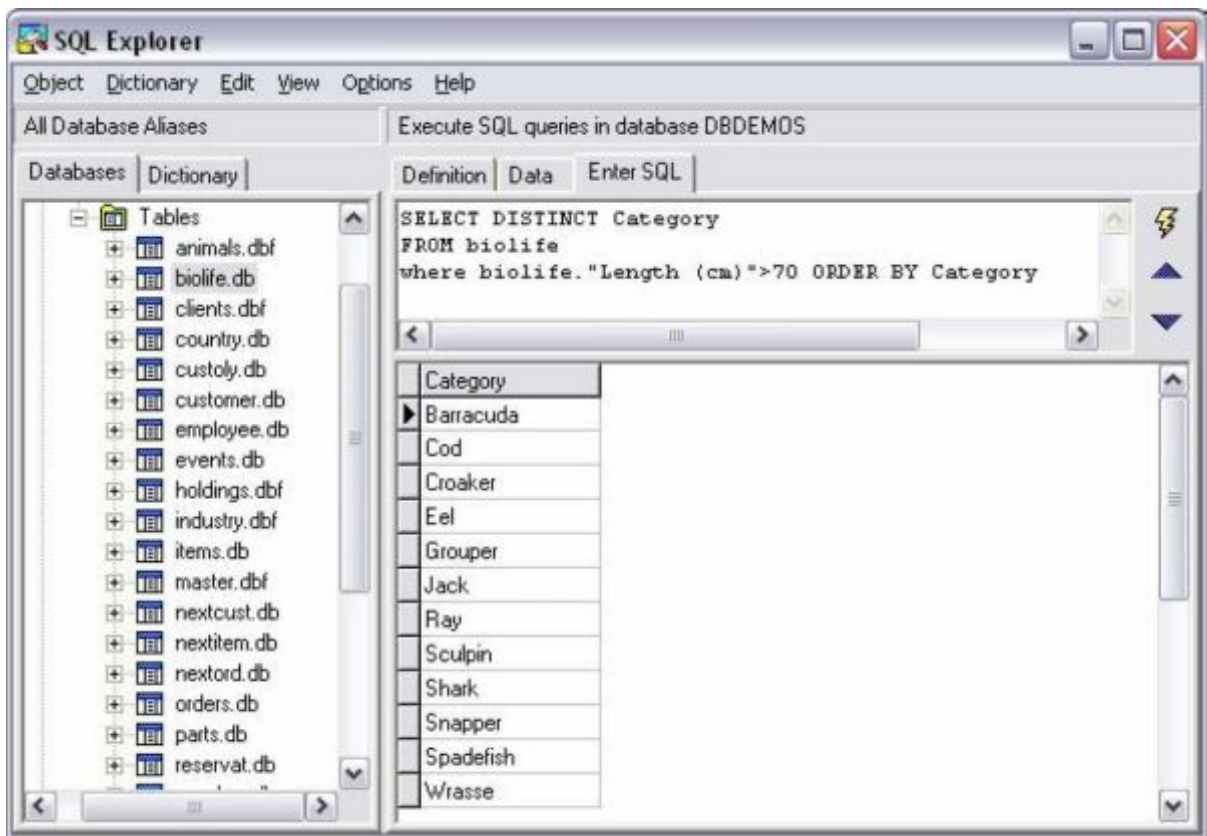


Рис. 28. Результат запроса без дублирования

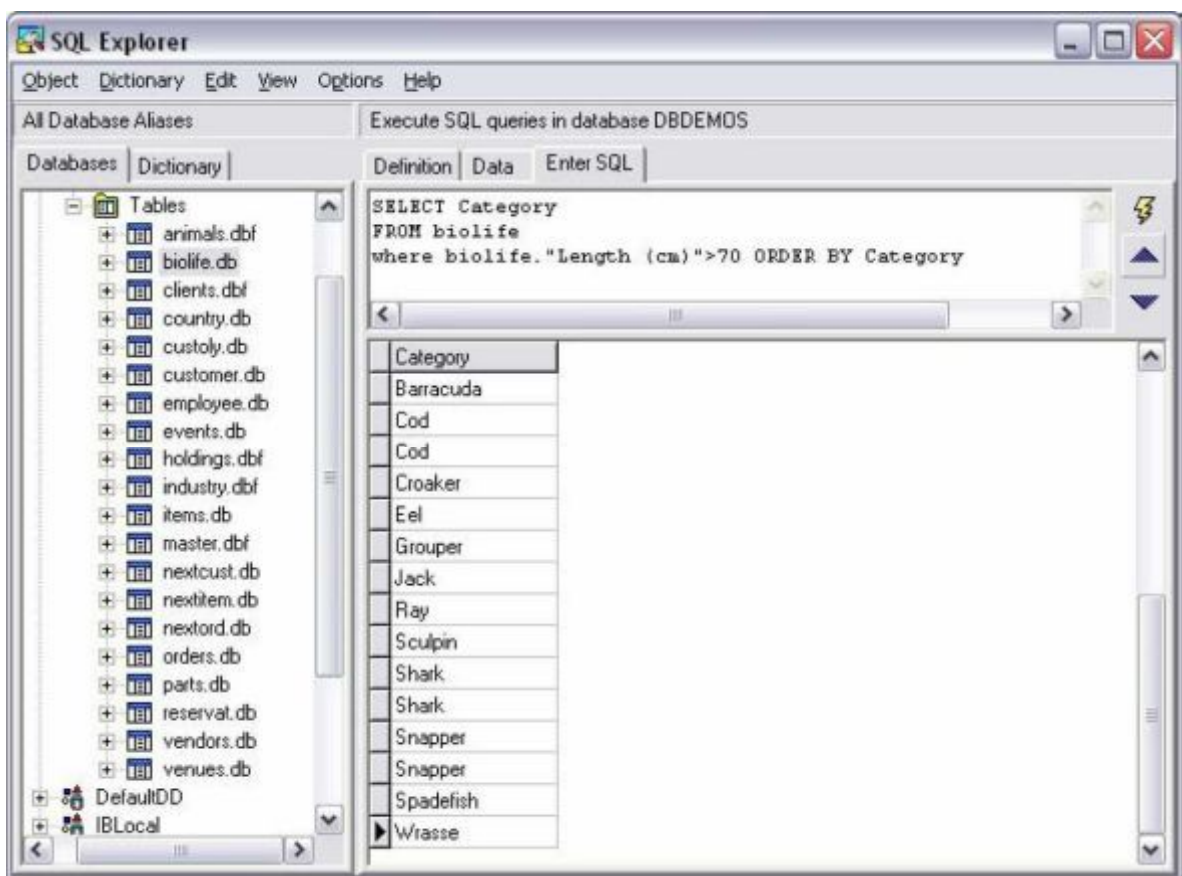


Рис. 29. Результат запроса с дублированием

Пример 3.22. По таблице *parts* получить список изделий:

```
SELECT DISTINCT Description FROM parts
```

Пример 3.23. По таблице *biolife* вывести перечень категорий, к которым относятся рыбы длиной более 70 см. Перечень упорядочить по алфавиту:

```
SELECT DISTINCT Category FROM biolife  
WHERE biolife."Length (cm)">70 ORDER BY Category
```

3.8. Вычисления в запросах 3.8.1.

Вычисляемые поля

При выводе данных из таблиц можно использовать значения полей для выполнения вычислений. Для создания вычисляемого поля надо в команде `SELECT` в списке выводимых значений записать выражение, которое будет вычисляться при выводе результатов запроса. Например, для получения из таблицы *employee* списка сотрудников с указанием зарплаты и зарплаты, увеличенной на 17%, достаточно выполнить запрос:

```
SELECT LastName,FirstName,Salary,Salary*1.17 FROM employee
```

При выполнении вычислений разрешено использовать арифметические операции сложения, вычитания, умножения, деления и встроенные функции. При этом выражение будет вычисляться с учётом общепринятого приоритета операций, для изменения порядка действий в выражениях применяются скобки.

При выводе результатов столбцу, содержащему вычисленные значения, автоматически даётся имя, построенное по введённому выражению. Язык SQL позволяет явным образом задать имя столбца с помощью фразы `AS`.

Пример 3.24. Вычислить общую стоимость изделий, имеющихся в наличии и заказанных. Задать имя вычисляемому полю: `SELECT`

```
Description,OnHand*ListPrice+OnOrder*ListPrice FROM parts  
SELECT Description,(OnHand+OnOrder)*ListPrice FROM parts
```

Оба запроса решают поставленную задачу. Во втором запросе для задания нужной последовательности вычислений в выражении использованы скобки.

```
SELECT Description,(OnHand+OnOrder)*ListPrice AS TotalCost FROM parts
```

В последнем варианте запроса вычисляемому полю присвоено имя.

3.8.2. Итоговые функции

Итоговые (агрегатные) функции предназначены для получения обобщающих сведений о результирующем наборе записей. К итоговым функциям относятся:

SUM - вычисление суммы значений по заданному полю; MAX -

определение максимального значения поля; MIN - определение

минимального значения поля;

AVG - вычисление арифметического среднего указанного поля (сумма значений, делённая на их количество);

COUNT - определение количества записей в выходном наборе.

Итоговые функции оперируют со значениями в указанном поле таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к нечисловым полям, а функции SUM и AVG могут применяться только к числовым полям.

При вычислении результатов любых функций сначала исключаются все пустые значения, а затем требуемая операция применяется к оставшимся конкретным значениям столбца. Особенно важно это учитывать при вычислении среднего значения. Вариант COUNT(*) применения функции COUNT является исключением из правил. В этом случае определяется количество *всех* строк в результирующей таблице независимо от того, что в них находится.

Пример 3.25. Определить количество записей, максимальную, минимальную и среднюю длину рыбы:

```
SELECT count(*) AS Number,max(Length_In) AS Max_Length,  
min(Length_In) AS min_Length,avg(Length_In) AS avg_Length FROM biolife
```

Итоговые функции используются в списке предложения SELECT и в составе предложения HAVING. Если список в предложении SELECT содержит итоговые функции, а в тексте запроса отсутствует фраза GROUP BY, обеспечивающая объединение данных в группы, то поля могут быть только аргументами итоговых функций.

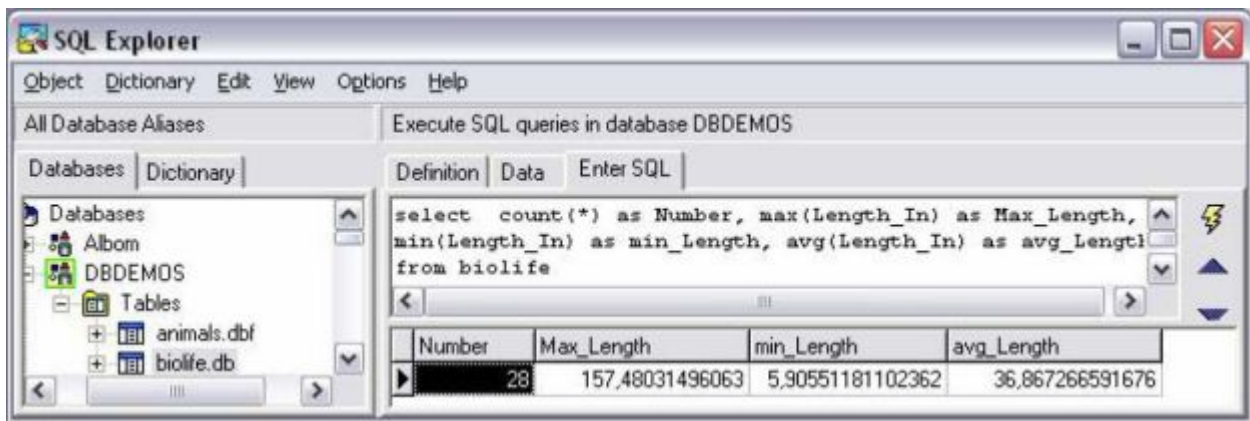


Рис. 30. Применение итоговых

3.8.3. Выполнение группировки

Предложение GROUP BY предназначено для подведения промежуточных итогов в запросах. Для каждой отдельной группы создаётся единственная итоговая строка. При наличии в команде SELECT фразы GROUP BY каждый элемент списка в предложении SELECT должен иметь единственное значение для всей группы. Предложение SELECT может включать только следующие типы элементов: имена полей, итоговые функции, константы и выражения, включающие комбинации перечисленных выше элементов.

Все имена полей, приведённые в списке предложения SELECT, должны присутствовать и в фразе GROUP BY, за исключением случаев, когда имя столбца используется в *итоговой функции*. При этом в предложении GROUP BY могут быть поля, отсутствующие в списке предложения SELECT.

Пример 3.26. Вывести сведения о категории, количестве записей, максимальной, минимальной и средней длины рыбы в *каждой категории*:

```
SELECT Category, count(*)AS Number,
max(Length_In) AS Max_Length,min(Length_In) AS min_Length,
avg(Length_In) AS avg_Length FROM
biolife GROUP BY Category
```

Если совместно с GROUP BY используется предложение WHERE, то оно обрабатывается первым, а группировка применяется только к тем строкам, которые удовлетворяют условию поиска.

Пример 3.27. Вывести сведения о количестве и средней стоимости заказов для заказчиков с номерами более 6800:

```
SELECT CustNo, count(*)AS Number, avg(ItemsTotal) FROM
orders WHERE CustNo>6800 GROUP BY CustNo
```

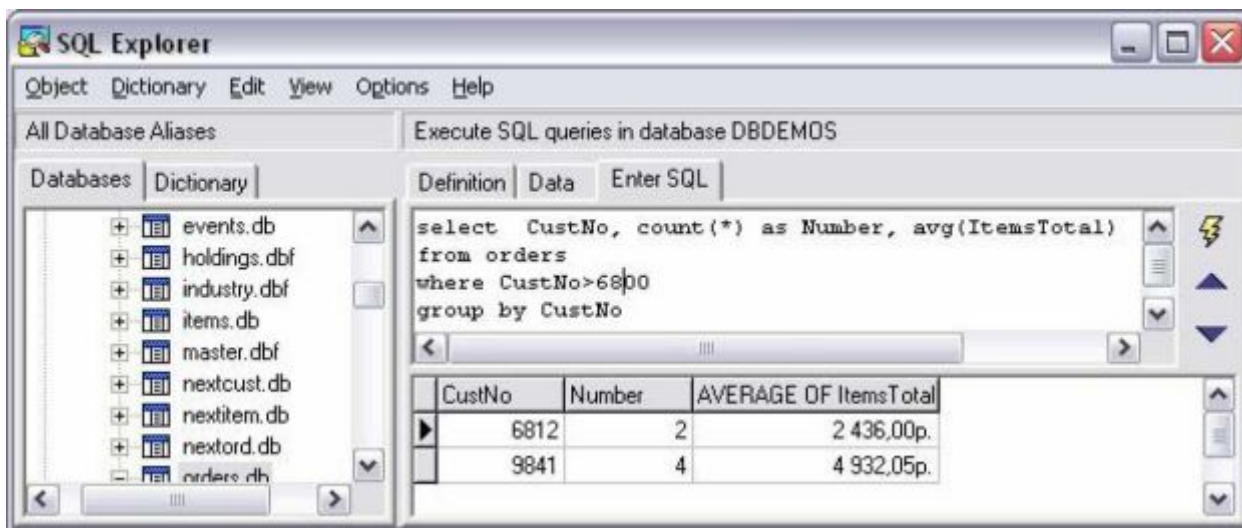


Рис. 31. Применение группировки

При проведении группировки все отсутствующие значения рассматриваются как равные. Если две строки таблицы в одном и том же группируемом столбце содержат NULL и идентичные значения во всех других непустых группируемых столбцах, они помещаются в одну и ту же группу.

3.8.4. Предложение HAVING

При помощи предложения HAVING отображаются все предварительно сгруппированные посредством GROUP BY блоки данных, удовлетворяющие заданным в HAVING условиям. Это дополнительная возможность выполнить фильтрацию в выходном наборе.

Условия в HAVING отличаются от условий в WHERE. Предложение HAVING позволяет исключить из результирующего набора группы с итоговыми значениями, которые не удовлетворяют поставленным условиям. **В предложении WHERE нельзя использовать итоговые функции.**

Пример 3.28. Вывести сведения о категориях (количестве записей, максимальной, минимальной и средней длине), в которых количество представителей не меньше двух:

```
SELECT Category, count(Category) AS Number,  
max(Length_In) AS Max_Length, min(Length_In) AS min_Length,  
avg(Length_In) AS avg_Length  
FROM biolife  
GROUP BY Category  
HAVING count(Category)>=2
```

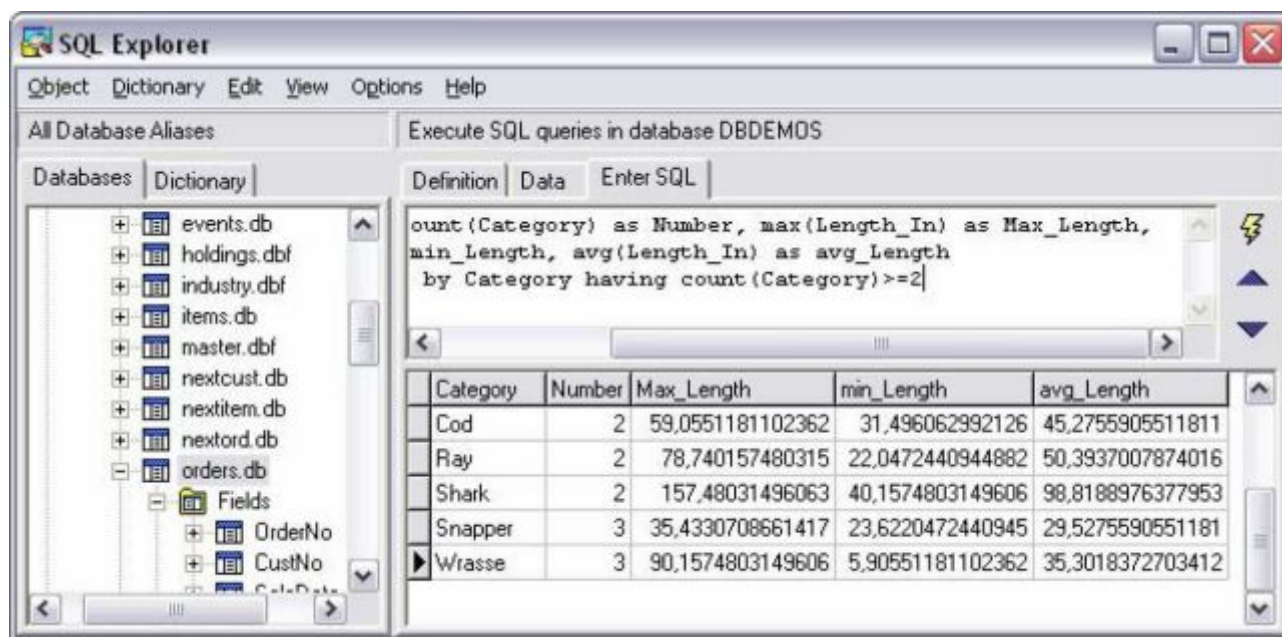


Рис. 32. Результат применения предложения HAVING

3.9. Контрольные вопросы и задания

1. В каких базах данных используется язык SQL?
2. Назовите типы команд языка SQL.
3. Для чего предназначен язык определения данных ЯОД ?
4. Для чего предназначен язык манипулирования данными ЯМД?
5. Какая из имеющихся в Delphi утилит позволяет выполнять команды SQL в интерактивном режиме?
6. Как называется операция, позволяющая получить подмножество столбцов таблицы?
7. В чём суть операции выборки?
8. Каков будет результат применения декартова произведения к двум отношениям?

9. Чем представление принципиально отличается от таблицы?
10. Перечислите служебные слова, используемые в команде SELECT.
11. Где указывается перечень выводимых полей?
12. Что записывается после слова FROM в команде SELECT?
13. В каких случаях при задании поля приходится использовать составные имена?
14. Какие операции можно использовать при формировании условия в предложении WHERE?
15. Поясните использование операций сравнения, BETWEEN, IN, LIKE.
16. Как изменить порядок выводимых строк в результирующем наборе?
17. Чем определяется порядок действий при вычислении значения условия?
18. Можно ли упорядочить выводимые записи по нескольким полям?
19. Какая логическая операция - or или and имеет более высокий приоритет?
20. Почему при работе с полями даты могут возникнуть проблемы?
21. Как устранить дублирование в результирующем наборе?
22. Какие операции можно использовать при выполнении вычислений в запросах?
23. Что в запросах вычисляют функции AVG и COUNT?
24. Для чего используется группировка в запросах?
25. В каких случаях для задания условия отбора записей необходимо использовать предложение HAVING?
26. Сформировать запросы к таблице *employee* из DBDemos:
 - подсчитать количество сотрудников, у которых заработная плата больше 35 000;
 - найти сотрудников, фамилия которых начинается на букву 'B';
 - найти сотрудников, поступивших на работу в 1993 г.;
 - вывести сведения о сотрудниках, у которых телефон начинается цифрами 22;
 - определить среднюю зарплату сотрудников, работающих с 1989 г.
27. Сформировать запросы к таблице *custoly* из DBDemos:
 - определить, кто не имеет e-mail;
 - найти людей, фамилия которых начинается на 'Sm';
 - найти людей, фамилия которых начинается на 'S', а телефон - на 8;
 - подсчитать количество покупателей из каждой страны;
 - отсортировать записи по фамилиям.

4. ФОРМИРОВАНИЕ СЛОЖНЫХ SQL-ЗАПРОСОВ

4.1. Соединение таблиц

Соединение - это вывод связанной информации из нескольких таблиц или запросов в виде одного логического набора данных. В операции соединения проявляется одна из наиболее важных особенностей запросов SQL - способность определять связи между многочисленными таблицами и выводить информацию из них с учётом этих связей. Именно эта операция придаёт гибкость и лёгкость языку SQL.

Соединяемые таблицы перечисляются через запятую в предложении FROM оператора SELECT. Правила соединения таблиц задаются в предложении WHERE запроса SELECT с помощью специального *условия соединения*.

Существует несколько видов операции соединения:

CROSS JOIN - перекрестное соединение;

INNER JOIN - внутреннее соединение, используется по умолчанию;

LEFT JOIN [OUTER] - левое внешнее соединение;

RIGHT JOIN [OUTER]- правое внешнее соединение;

FULL JOIN [OUTER]- полное внешнее соединение.

Внешние соединения поддерживаются стандартом ANSI-92 и содержат зарезервированное слово JOIN. Внутренние соединения (или просто соединения) могут записываться как без использования этого слова (стандарт ANSI-89), так и с использованием слова JOIN (стандарт ANSI-92).

При использовании стандарта ANSI 92 условия соединения записываются в предложении FROM по формату:

```
FROM имя_таблицы_1 {INNER|LEFT|RIGHT} JOIN имя  
таблицы_2 ON условие соединения
```

При формировании запроса придерживаются следующих правил: - слева и справа от зарезервированного слова JOIN указывают соединяемые таблицы;

- после слова ON записывают условия соединения;
- условия поиска, основанные на *правой* таблице, помещают в предложение ON;
- условия поиска, основанные на *левой* таблице, помещают в предложение WHERE.

Характерные черты операции соединения:

- в условиях соединения могут участвовать поля, относящиеся к одному и тому же типу данных, но они не обязательно должны иметь одинаковые имена. Часто в условиях соединения связывание выполняется по первичному ключу одной таблицы и внешнему ключу другой таблицы;
- соединяемые поля могут (но не обязаны!) присутствовать в списке выбираемых полей;
- разрешено использовать множественные условия соединения;
- условие соединения может комбинироваться с другими предикатами.

При задании соединения часто используют псевдонимы. Псевдонимы назначают в предложении FROM после имени таблицы. Эти псевдонимы не имеют ничего общего с псевдонимом базы данных, создаваемым для упрощения доступа к таблицам локальных баз данных.

Если названия полей таблиц совпадают, то псевдонимы позволяют корректно сформулировать запрос. Назначение псевдонима обязательно при выполнении соединения таблицы самой с собой. Применение псевдонимов позволяет упростить и сократить запись запроса.

Псевдонимы представляют собой любой допустимый идентификатор, написание которого подчиняется тем же правилам, что и написание имён таблиц.

4.2. Внутреннее соединение

Внутреннее соединение возвращает только те строки, для которых условие соединения принимает значение true. Выполнение соединений рассмотрим на таблицах *customer.db*, *employee.db*, *orders.db*, *items.db*, *parts.db* и *ven-dors.db*, имеющихся в демонстрационной базе данных **DBDemos** (рис. 33).

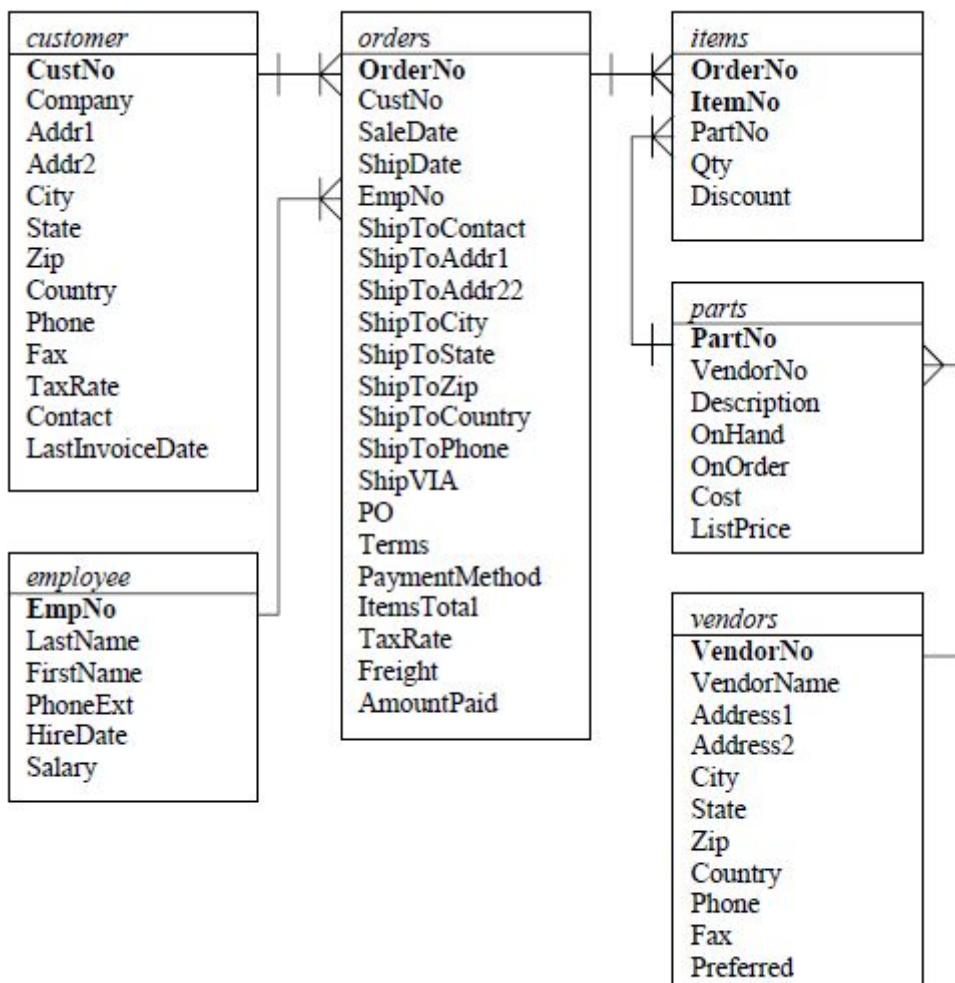


Рис. 33. Демонстрационная база данных «Торговая фирма»

Заказчики оформляют заказ на приобретение оборудования. Сведения о заказчиках приведены в таблице *customer*. Сведения о заказах заносятся в таблицу *orders*. Данные о сотрудниках, оформляющих заказы, хранятся в таблице *employee*.

Часто при получении информации из БД приходится извлекать данные из нескольких таблиц. Задача решается путём соединения двух, трёх и более таблиц так, чтобы результирующий набор данных содержал информацию из всех соединяемых таблиц.

Пример 4.1. Вывести сведения о сотрудниках (имя, фамилия) и оформленных ими в декабре 1994 г. заказах (дата продажи, стоимость, оплачено).

Фамилия и имя сотрудника приведены в таблице *employee*, а дата продажи, стоимость заказа и сведения об оплате - в *orders*. Таблицы *employee* и *orders* связаны отношением «один-ко-многим» по полю *EmpNo*.

Каждому сотруднику, сведения о котором приведены в таблице *employee*, может соответствовать несколько записей в таблице *orders* в зависимости от того, сколько заказов оформил этот сотрудник.

Анализ искомых полей показывает, что для решения задачи следует *соединить* данные из таблиц *orders* и *employee* так, чтобы результирующий набор данных содержал информацию из обеих таблиц. Новый набор данных должен содержать пять полей: *FirstName*, *LastName* из таблицы *employee* и *SaleDate*, *ItemsTotal*, *AmountPaid* из таблицы *orders*.

По условию задачи надо вывести сведения не о всех заказах, а только о тех, которые были заключены в декабре 1994 г. Следовательно, надо сформулировать условие отбора. Соединение таблиц *orders* и *employee* с дополнительным условием отбора строк может быть выполнено единственным SQL-запросом, который выглядит так:

```
SELECT FirstName,LastName,SaleDate,ItemsTotal,AmountPaid FROM employee,orders
WHERE SaleDate>"30.11.94" AND SaleDate<="31.12.94"
AND employee.EmpNo=orders.EmpNo
```

Этот запрос состоит из нескольких различных частей. После слова *SELECT* идёт список полей, которые надо включить в набор данных. Так как имена полей разные, то можно не уточнять, какое поле к какой таблице относится. Выражение *FROM* объявляет, что используются две таблицы, одна называется *orders*, а другая *employee*.

В выражении *WHERE* определены два условия:

- условие соединения *employee.EmpNo=orders.EmpNo*. Указаны поля связи для двух таблиц. Некоторые серверы могут вернуть *DataSet*, даже если не включить условие соединения в запрос, но почти всегда результирующий набор записей будет не тем, что требовалось получить;
- состоящее из двух частей условие отбора строк

SaleDate>"30.11.94" AND SaleDate<="31.12.94"



FirstName	LastName	SaleDate	ItemsTotal	AmountPaid
Phi	Forest	16.12.1994	17 781,00p.	17 781,00p.
Michael	Yanowski	09.12.1994	64 115,75p.	64 115,75p.
Kevin	Cock	10.12.1994	2 577,05p.	2 577,05p.
Ashok	Ramanathan	11.12.1994	1 999,00p.	1 999,00p.
Leslie	Phong	14.12.1994	158 922,65p.	158 922,65p.
Jennifer M.	Burbark	15.12.1994	6 935,00p.	6 935,00p.
Ann	Bennet	16.12.1994	1 400,00p.	1 400,00p.
Mark	Guckenheimer	20.12.1994	304,00p.	304,00p.
Chris	Papadopoulos	22.12.1994	16 939,50p.	16 939,50p.
Jennifer M.	Burbark	23.12.1994	11 568,00p.	11 568,00p.
Takashi	Yamamoto	25.12.1994	4 317,75p.	4 317,75p.
Mary	Page	30.12.1994	7 134,00p.	7 134,00p.

Рис. 34. Результат выполнения примера 4.1

Пример 4.2. Получить сведения о компаниях-заказчиках (название, город, страна) и сделанных ими заказах (дата продажи, стоимость, оплачено) после 01.01.1993 на сумму более 15 000.

```
SELECT Company, City, Country, SaleDate, ItemsTotal, AmountPaid FROM customer, orders
WHERE customer.CustNo=orders.CustNo and
      SaleDate>"01.01.1993" AND ItemsTotal>15000
```

При необходимости вывести сведения из трёх таблиц создаётся соединение трёх таблиц: записываются условия соединения между таблицами попарно. Синтаксис зависит от версии языка SQL.

Пример 4.3. Вывести сведения о заказе, заказчике и сотруднике, оформившем заказ, для заказов со стоимостью более 100 000.

```
SELECT orders.OrderNo, orders.ItemsTotal, customer.Company,
       employee.FirstName, employee.LastName
```

```

FROM employee, customer, orders WHERE
customer.CustNo=orders.CustNo
      AND employee.EmpNo=orders.EmpNo
      AND orders.ItemsTotal>=100000

```

OrderNo	ItemsTotal	Company	FirstName	LastName
1071	103 041,00p.	The Depth Charge	Bill	Parker
1096	123 740,00p.	Kirk Enterprises	Janet	Baldwin
1161	102 453,60p.	Blue Sports	Randy	Williams
1263	158 922,65p.	American SCUBA Supply	Leslie	Phong

Рис. 35. Результат выполнения примера

В приведённых выше примерах правила соединения таблиц записывались без слова JOIN и без использования псевдонимов.

Пример 4.4. Вывести сведения о заказах (код заказа, сумма, компания, сотрудник) со стоимостью менее 10 000, оформленных сотрудником по фамилии Cook.

```

SELECT o.OrderNo,o.ItemsTotal, c.Company,
      e.FirstName,e.LastName FROM employee e,
customer c, orders o WHERE c.CustNo=o.CustNo
      AND e.EmpNo= o.EmpNo
      AND o.ItemsTotal<10000
      AND e.LastName="Cook"

```

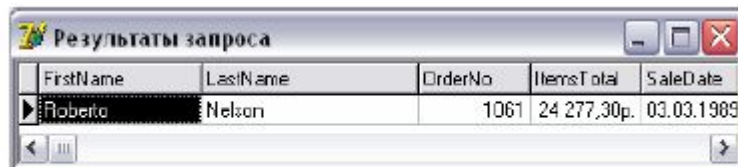
OrderNo	ItemsTotal	Company	FirstName	LastName
1030	559,60p.	Marina SCUBA Center	Kevin	Cook
1049	1 809,85p.	Neptune's Trident Supply	Kevin	Cook
1200	1 827,00p.	VIP Divers Club	Kevin	Cook
1260	2 577,85p.	Divers of Blue-green	Kevin	Cook

Рис. 36. Результат выполнения примера

Для краткости записи в запросе используется особенность SQL, которая позволяет ссылаться на таблицы с помощью псевдонимов. Для таблицы *orders* назначен псевдоним **o**, для таблицы *employee* - **e**, а для *customer* - **c**.

Пример 4.5. Получить сведения о заказах (номер, стоимость, дата продажи) со стоимостью больше 20 000, оформленных сотрудником по фамилии Nelson (имя, фамилия). Сформировать запрос с использованием слова JOIN и псевдонимов.

```
SELECT e.FirstName, e.LastName,  
       o.OrderNo, o.ItemsTotal, o.SaleDate FROM employee e  
JOIN orders o ON o.ItemsTotal >= 20000 AND e.EmpNo = o.EmpNo  
WHERE e.LastName = "Nelson"
```



FirstName	LastName	OrderNo	ItemsTotal	SaleDate
Roberto	Nelson	1061	24 277,30p.	03.03.1989

Рис. 37. Результат выполнения примера 4.5

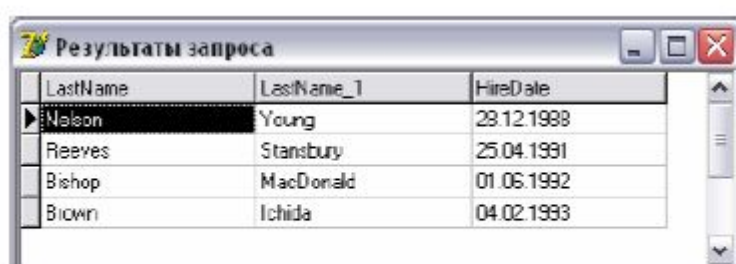
После слова ON записано условие соединения и условие отбора строк, основанное на «правой» таблице - orders. Условие, использующее данные из «левой» таблицы (employee), записано после слова WHERE.

4.3. Самосоединения

В некоторых задачах необходимо получить информацию, выбранную особым образом только из одной таблицы. Для этого используются самосоединения - соединение таблицы с собой с помощью псевдонимов. Самосоединения полезны в случаях, когда нужно получить пары аналогичных элементов из одной и той же таблицы.

Пример 4.6. Получить пары фамилий сотрудников, которые приняты на работу в один и тот же день

```
SELECT one.LastName, two.LastName, one.HireDate FROM employee one,  
employee two WHERE one.HireDate = two.HireDate AND one.EmpNo <  
two.EmpNo
```



LastName	LastName_1	HireDate
Nelson	Young	29.12.1998
Reeves	Stansbury	25.04.1991
Bishop	MacDonald	01.05.1992
Brown	Ichida	04.02.1993

Рис. 38. Результат выполнения примера 4.6

4.4. Внешнее соединение

Внутреннее соединение возвращает только те строки, для которых условие соединения принимает значение true. Иногда требуется включить в результирующий набор большее количество строк. Внешнее соединение возвращает *все* строки из одной таблицы и только те строки из другой таблицы, для которых условие соединения принимает значение *истина*. Строки второй таблицы, не удовлетворяющие условию соединения, получают значение null в результирующем наборе. Существует два вида внешнего соединения: LEFT JOIN и RIGHT JOIN.

4.4.1. Левое соединение

В левом соединении (LEFT JOIN) запрос возвращает все строки из левой таблицы (т.е. таблицы, стоящей *слева* от словосочетания LEFT JOIN) и только те строки из правой таблицы, которые удовлетворяют условию соединения. Если же в правой таблице не найдётся строк, удовлетворяющих заданному условию, в качестве значений второй таблицы подставляется null.

Пример 4.7. Вывести сведения о заказе билетов на все объявленные соревнования.

В таблице *events* находится перечень соревнований, а в таблице *venues* - список сооружений, используемых для проведения соревнований. Данные о сделанных заказах на билеты хранятся в таблице *reservat*. Сведения о заказчиках приведены в таблице *custoly*. Перечень полей в таблицах и связи между таблицами показаны на рис. 39.

Для получения результатов надо соединить данные из двух таблиц: *events* и *reservat*. Причём вполне возможна ситуация, при которой не на все соревнования были заказаны билеты. Так как в запросе надо обеспечить вывод всех запланированных соревнований, то следует применить внешнее соединение.

```
SELECT eventNo, event_Name, CustNo, NumTickets
FROM events LEFT JOIN reservat
ON events.eventNo=reservat.eventNo
```

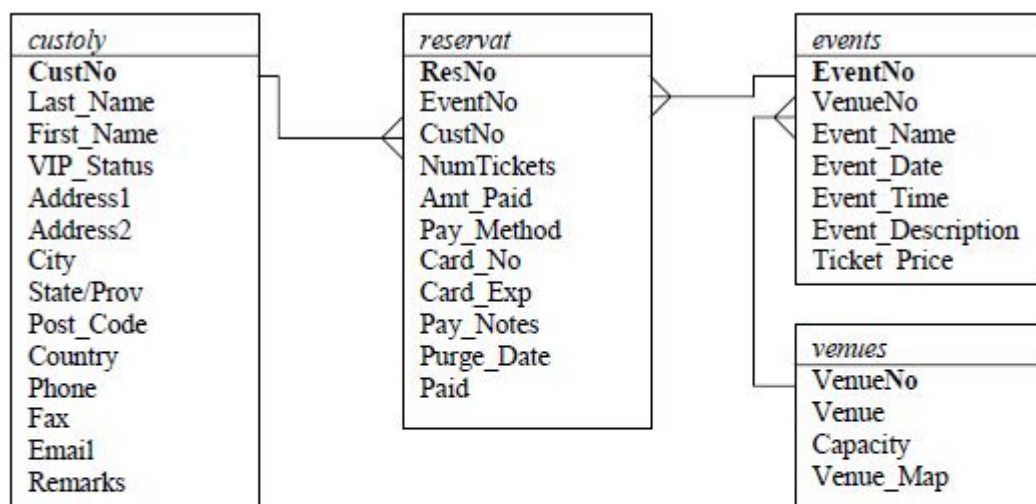


Рис. 39. Демонстрационная база данных «Заказ билетов»

Результаты запроса

eventNo	eventName	CustNo	NumTickets
13	Women's Soccer Finals	25	7
14	Surfing: Freestyle Finals	9	2
14	Surfing: Freestyle Finals	6	3
14	Surfing: Freestyle Finals	5	6
14	Surfing: Freestyle Finals	16	4
14	Surfing: Freestyle Finals	9	6
14	Surfing: Freestyle Finals	18	1
14	Surfing: Freestyle Finals	3	2
15	Swimming: 100 m Finals	9	1
16	Track & Field: Women's Finals	12	8
16	Track & Field: Women's Finals	7	4
16	Track & Field: Women's Finals	22	3
16	Track & Field: Women's Finals	10	3
16	Track & Field: Women's Finals	19	9
16	Track & Field: Women's Finals	4	9
16	Track & Field: Women's Finals	23	1
17	Archery Finals		
18	Men's Hockey Finals		
19	Men's Downhill Giant Slalom		
20	Men's Ski Jump Finals		
21	Women's Downhill Ski Finals		
22	Men's Relay		
23	Baseball Quarter-finals		

Рис. 40. Заказ билетов (LEFT JOIN)

В запросе использовано левое соединение, что обеспечивает вывод всех записей из таблицы *events* (рис. 40). Если использовать обычное соединение, то в результирующем наборе будут отсутствовать сведения о соревнованиях, на которые ещё не заказаны билеты

eventNo	event_Name	CustNo	NumTickets
13	Women's Soccer Finals	20	6
14	Surfing: Freestyle Finals	6	3
14	Surfing: Freestyle Finals	5	6
14	Surfing: Freestyle Finals	9	6
14	Surfing: Freestyle Finals	16	4
14	Surfing: Freestyle Finals	18	1
14	Surfing: Freestyle Finals	3	2
14	Surfing: Freestyle Finals	9	2
15	Swimming: 100 m Finals	9	1
16	Track & Field: Women's Finals	19	9
16	Track & Field: Women's Finals	12	8
16	Track & Field: Women's Finals	10	3
16	Track & Field: Women's Finals	23	1
16	Track & Field: Women's Finals	22	3
16	Track & Field: Women's Finals	4	9
16	Track & Field: Women's Finals	7	4

Рис. 41. Заказ билетов

(рис. 41).

```
SELECT eventNo, event_Name, CustNo, NumTickets
FROM events JOIN reservat
ON events.eventNo=reservat.eventNo
ORDER BY events.eventNo
```

4.4.2. Правое соединение

При правом соединении в результирующий набор будут включены все строки из правой таблицы и те строки из левой таблицы, для которых выполняется условие соединения.

Пример 4.8. Вывести сведения о стадионах и проводимых соревнованиях.

```
SELECT Venue, Event_Name FROM Events
RIGHT JOIN Venues ON
Events.VenueNo=Venues.VenueNo
```

При реализации запроса было использовано правое соединение. В соответствии с этим из таблицы *venues* были получены сведения о всех стадионах. Из таблицы *events* получены данные о проводимых соревнованиях. Оказалось, что на двух стадионах соревнования не запланированы (рис. 42).

eventNo	event_Name	CustNo	NumTickets
13	Women's Soccer Finals	20	6
14	Surfing: Freestyle Finals	6	3
14	Surfing: Freestyle Finals	5	6
14	Surfing: Freestyle Finals	9	6
14	Surfing: Freestyle Finals	16	4
14	Surfing: Freestyle Finals	18	1
14	Surfing: Freestyle Finals	3	2
14	Surfing: Freestyle Finals	9	2
15	Swimming: 100 m Finals	9	1
16	Track & Field: Women's Finals	19	9
16	Track & Field: Women's Finals	12	8
16	Track & Field: Women's Finals	10	3
16	Track & Field: Women's Finals	23	1
16	Track & Field: Women's Finals	22	3
16	Track & Field: Women's Finals	4	9
16	Track & Field: Women's Finals	7	4

Рис. 41. Заказ билетов (JOIN)

Полное внешнее соединение

При полном внешнем соединении в результирующий набор будут выбраны все строки как из правой, так и из левой таблицы. При выполнении условия соединения строка содержит значения как из левой, так и из правой таблицы. Если условие соединения не выполняется, то вместо отсутствующих значений левой или правой таблицы заносится значение null.

Пример 4.9. Вывести сведения о всех стадионах, приведённых в таблице *venues*, и всех проводимых соревнованиях (таблица *events*). Для каждого соревнования указать, на каком стадионе оно проводится.

```

SELECT Venue, event_Name
FROM events FULL JOIN Venues
ON events.VenueNo=venues.VenueNo

```

Если для тех же таблиц выполнить левое соединение и сравнить результаты левого, правого и полного соединений, то можно убедиться в том, что полное соединение действительно включает все строки каждой из таблиц.

```

SELECT Venue, event_Name
FROM events LEFT JOIN venues
ON events.VenueNo=venues.VenueNo

```

Venue	event_Name
	Men's Hockey Finals
	Men's Downhill Giant Slalom
	Men's Ski Jump Finals
	Women's Downhill Ski Finals
	Men's Relay
	Baseball Quarter-finals
	Karate Finals
	Women's Hockey Finals
	Men's Gymnastics Finals
Memorial Stadium	Women's Basketball Finals
Main Auditorium	Women's High Dive Finals
Main Auditorium	Women's Soccer Finals
Main Auditorium	500 meter Men's Speed Skating
Coliseum	Archery Finals
Five Track	Men's Track and Field
Five Track	Track & Field: Women's Finals
Five Track	Polo Finals
Five Track	Lacrosse Semi-finals
Five Track	100 Meter Individual Medley
Guillermo Field	
Memorial Stadium	Women's Cycling 20mi
Special City Route	Kayak Level IV Finals
Ayung River	Surfing: Freestyle Finals
County Line Beach	Swimming: 100 m Finals
Parker Aquatic Ctr	

Рис. 43. Сведения о всех стадионах и всех соревнованиях (FULL JOIN)

4.5. Объединение запросов


Операция объединения (UNION) позволяет слить результаты отдельных запросов по нескольким таблицам в единую результирующую таблицу. Предложение UNION объединяет вывод двух или более SQL-запросов в единый набор строк и столбцов. При этом результаты запросов должны быть совместимы:

- иметь одинаковое количество полей;
- типы значений попарно объединяемых полей должны быть одинаковыми или совместимыми.

При выполнении соединения в результирующем наборе автоматически исключается дублирование строк.

Пример 4.10. Получить список заказчиков и поставщиков из штата Калифорния с указанием их номеров телефонов.

```
SELECT Company,State,Phone FROM customer
WHERE State="CA"
UNION
SELECT VendorName,State,Phone FROM vendors WHERE
State="CA"
```



Company	State	Phone
American SCUBA Supply	CA	213-654-0092
Blue Glass Happiness	CA	213-555-1984
Catamaran Dive Club	CA	213-223-0941
Dive Time	CA	213-555-3708
Diver's Grotto	CA	213-432-0093
Fish Research Labs	CA	209-555-3292
Glen Specialties, Inc.	CA	714-555-5100
Marine Camera & Dive	CA	619-555-0604
San Pablo Dive Center	CA	823-044-2910
Scuba Professionals	CA	213-555-7850
Techniques	CA	415-555-1410
Undersea Systems, Inc.	CA	800-555-3483
Underwater Sports Co.	CA	408-867-0594

0

Язык SQL предусматривает возможность объединения запросов, полученных в результате соединения.

4.6. Использование вложенных запросов

4.6.1. Подзапросы

Часто при формировании условия в предложении WHERE требуется использовать значение, которое должно быть вычислено в момент выполнения оператора SELECT. Например, надо иметь максимальное, минимальное, среднее и т.п. значения. В таких случаях используют законченные операторы SELECT, внедрённые в тело другого оператора SELECT. Внешний и внутренний операторы SELECT строятся по одинаковым правилам. Внешний оператор использует результат внутреннего оператора.

Внутренние операторы располагают в предложениях WHERE или HAVING после операций сравнения (=, <, >, <=, >=, <>). Такие операторы называют подзапросами. Текст подзапроса помещают в круглые скобки. Ограничения при формировании подзапросов:

- ORDER BY не используется;
- список в SELECT состоит из имён отдельных столбцов или составленных из них выражений;
- по умолчанию имена столбцов относятся к таблице, указанной после FROM. Разрешено использовать имена столбцов внешнего запроса, явно указывая таблицу;
- если подзапрос является одним из двух операндов в операции сравнения, то записывается справа: Number=(SELECT ...) .

Существует два типа подзапросов - скалярный и табличный. Скалярный подзапрос возвращает одно значение. Табличный подзапрос возвращает множество значений.

4.6.2. Подзапросы, возвращающие одно значение

Пример 4.11. Найти название рыбы максимальной длины.

```
SELECT Common_Name FROM biolife WHERE  
biolife."Length (cm)"=  
    (SELECT max(biolife."Length (cm)" FROM biolife)
```


Пример 4.12. Определить дату продажи заказа с максимальной стоимостью. Вывести дату продажи и стоимость.

```
SELECT SaleDate,ItemsTotal FROM orders
WHERE ItemsTotal=(SELECT max(ItemsTotal) FROM orders)
```

Пример 4.13. Определить даты оформления заказов, стоимость которых превысила среднее значение, и указать для этих заказов превышение над средним уровнем.

```
SELECT OrderNo, SaleDate, ItemsTotal,
       ItemsTotal-(SELECT avg(ItemsTotal) FROM orders) FROM orders
WHERE ItemsTotal>(SELECT avg(ItemsTotal) FROM orders)
```

Пример 4.14. Вывести имя, фамилию, табельный номер сотрудника, оформившего заказ максимальной стоимости, и стоимость этого заказа.

Так как требуемые сведения находятся в разных таблицах, то для получения результата надо сформировать запрос, использующий соединение данных из двух таблиц. Кроме того, в запрос следует добавить подзапрос, определяющий максимальную стоимость заказа.

```
SELECT FirstName,LastName,EmpNo,ItemsTotal FROM employee e
join orders o ON E.EmpNo=o.EmpNo AND
ItemsTotal=(SELECT max(ItemsTotal) FROM orders)
```

4.6.3.Подзапросы, возвращающие множество значений

Подзапросы могут возвращать не одно, а множество значений. В таких случаях данные помещаются во временную таблицу, которая может использоваться только в том месте, где она появилась в *подзапросе*.

Если требуется сравнить полученные значения в предложениях WHERE или HAVING, то используют операции IN, NOT IN, ALL, SOME, ANY, EXISTS, NOT EXISTS, которые работают с множеством значений.

{WHERE|HAVING} выражение [NOT] IN (*подзапрос*);

{WHERE|HAVING} выражение

оператор_сравнения {ALL|SOME|ANY}(*подзапрос*); {WHERE|HAVING}

[NOT] EXISTS (*подзапрос*);

Операции IN и NOT IN были рассмотрены ранее. Напомним, при использовании этих операций проверяется, входит ли значение в полученное множество или сравниваемое значение не является элементом набора значений, возвращаемых подзапросом.

Ключевые слова ANY и ALL могут использоваться с подзапросами, возвращающими один столбец чисел. Ключевое слово SOME является синонимом слова ANY.

Если подзапросу предшествует ключевое слово ALL, условие сравнения считается выполненным, если оно выполняется для всех значений в результирующем столбце подзапроса.

Если записи подзапроса предшествует ключевое слово ANY, то условие сравнения считается выполненным, когда оно выполняется хотя бы для одного из значений в результирующем столбце подзапроса.

Если в результате выполнения подзапроса получено пустое значение, то для ключевого слова ALL условие сравнения будет считаться выполненным, а для ключевого слова ANY - невыполненным.

Операции EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами и позволяют узнать, является ли возвращаемая подзапросом таблица пустой. Результатом выполнения этих операций является логическое значение true или false. Результат операции EXISTS равен true в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. Если результирующая таблица пуста, операция EXISTS возвращает false. Для операции NOT EXISTS используются правила обработки, обратные по отношению к EXISTS.

4.7. Контрольные вопросы и задания

1. Какая операция называется соединением?
2. Приведите правила соединения таблиц.
3. В чём суть внутреннего (левого, правого, полного) соединения?

4. Для чего в запросах используются псевдонимы?
5. В каких случаях используется операция объединения?
6. Для чего используются подзапросы?
7. Приведите правила формирования подзапросов.
8. Сформировать запрос к таблице *order* из DBDemos, позволяющий получить сведения о заказе, у которого самая большая разница между суммой заказа и оплаченной суммой.
9. Сформировать запрос для получения сведений о покупателе, сделавшем наибольшее количество заказов (таблицы *orders* и *customer*).

5. РЕЛЯЦИОННЫЙ СПОСОБ ДОСТУПА К ДАННЫМ

5.1. Компонент Query

Класс TQuery, так же, как и TTable, является наследником класса TDataSet и предназначен для работы с наборами данных. Однако в отличие от TTable, который работает с одной таблицей, TQuery позволяет создавать наборы данных по нескольким таблицам.

Методика работы с компонентом **Query** похожа на методику работы с **Table**, но есть и различия. Компонент **Query** более мощный и универсальный по сравнению с **Table**, обеспечивает доступ к таблицам на языке **SQL** (табл. 9). Представляет данные в виде таблицы, колонки которой являются потомками класса TField. Однако таблица компонента - логическая, формируется в результате **SQL**-запроса.

Если запрос требует получения из базы данных сведений (используется оператор **SELECT**), то данные помещаются в локальную таблицу в виде временного файла в каталоге запуска программы и **Query** становится владельцем этой таблицы. Эти данные не предназначены для изменения. Для внесения изменений в таблицы используются специальные запросы: **INSERT**, **UPDATE**, **DELETE**.

Целесообразность использования того или иного компонента определяется следующими факторами:

- при работе с локальными и файл-серверными базами данных скорость при использовании **Query** ниже;
- возможности компонента **Query** существенно шире, так как он позволяет соединять данные из нескольких таблиц;
- при работе с серверными базами данных **Table** теряет все преимущества, так как работает с копией таблиц. В клиент-серверных приложениях **Table** не используют;
- **Query** предназначен для работы с распределёнными СУБД, но может использоваться и в локальных.

Некоторые свойства компонента **Query**

Свойство	Описание
Active	Если true, то запрос открыт
CachedUpdates	Если true, то изменения запоминаются для переноса в таблицу методом ApplyUpdates
DatabaseName	Псевдоним или путь к файлам таблиц базы данных
DataSource	Источник данных для параметрического запроса
Local	Для локальной БД - true
ParamCheck	Если true, то SQL-запрос может быть параметрическим
Params[Index]	Массив параметров динамического SQL-запроса
Prepared	True, если запрос подготовлен методом Prepare
RequestLive	Если true, то в результате запроса происходит попытка создать изменяемую таблицу
SQL	Текст SQL-запроса. Имеет тип TStrings
UniDirectional	Оптимизирует доступ к таблице. Если true, то можно перемещаться по таблице более быстро, но только вперед

Процедура ExecSQL выполняет запросы без открытия набора данных. Функция

ParamByName(Name) обеспечивает доступ к параметру по имени Name.

Процедура Prepare готовит запрос к выполнению.

Процедура UnPrepare отменяет последствия вызова Prepare.

Процедура GetDetailLinkField (MasterFields, DetailFields) заполняет список MasterFields полями главной таблицы и DetailFields - полями подчинённой таблицы.

5.2. Реализация запросов 5.2.1.

Классификация запросов

В зависимости от времени создания запроса они делятся на *статические* и *динамические*. *Статические* (неизменяемые) запросы создаются во время разработки приложения. В таких запросах возможно применение компонентов

полей. Статический запрос **SELECT** не только создаётся, но и выполняется на этапе дизайна. *Динамические* запросы формируются во время выполнения программы. Для подготовки запроса используется метод `Add` или запрос загружается из файла.

Различают обычные и *параметрические* запросы. *Параметрическим* является запрос, в **SQL**-операторе которого могут изменяться отдельные составляющие. Изменяемые части оператора оформляются как параметры. Начальные значения параметров могут задаваться в Инспекторе объектов. Изменяемые параметры вводятся разными способами во время выполнения программы. При этом сам запрос может быть сформулирован программно или в Инспекторе объектов.

Наибольшими возможностями обладают динамические параметрические запросы.

5.2.2. Выполнение запроса

Компонент **Query** может возвращать набор данных (выборка данных из одной или более таблиц с помощью оператора **SELECT**) и выполнять действия над таблицами (**INSERT**, **UPDATE**, **DELETE** и т.д.).

В соответствии с этим существует два различных способа выполнения запроса **SQL**. В случае использования **SELECT** следует открывать **Query** методом `Open` или заданием свойству `Active` значения `true`.

Если выражение **SQL** не подразумевает возвращение курсора (набор данных не возвращается), то надо вызвать метод `ExecSQL`.

Закрытие набора данных, созданного командой **SELECT**, осуществляется методом `Close` или заданием свойству `Active` значения `false`. Причём метод `Close` можно вызывать без опасений. Даже если запрос уже закрыт, исключительная ситуация генерироваться не будет. Для компонента **Query**, не возвращающего набор данных, `Close` последствий не имеет.

5.2.3. Простейшее приложение

Основные приёмы применения компонента **Query** легко понять на примере.

Пример 5.1. Создать приложение для вывода данных из таблицы *Заказов (Orders)* с помощью статического запроса. Последовательность действий

Поместить на форму компоненты **Query1**, **DataSource1** и **DBGrid1**.

Задать значения свойств компонента **Query1**:

- в свойстве `DatabaseName` указать псевдоним (alias) базы данных (DBDemos).
Вместо псевдонима можно задать путь к папке, в которой находятся таблицы БД;
- в свойстве `SQL` ввести запрос. Для этого в Инспекторе объектов для свойства `SQL` открыть **StringListEditor** и записать текст запроса, например: `Select *from orders`
- свойство `Active` установить в `true`.

Для компонента **DataSource1** в свойстве `DataSet` указать **Query1**.

Для компонента **DBGrid1** в свойстве `DataSource` задать **DataSource1**.

В результате все поля таблицы *orders* будут выведены в компонент **DBGrid1**. Если требуется вывести только часть полей, то надо изменить текст запроса:

`Select OrderNo, CastNo, SaleDate, ShipDate from Orders` Так как любое изменение свойства `SQL` закрывает набор данных, то надо в свойстве `Active` задать `true`.

5.2.4. Программное использование Query

Доступ к свойству **SQL** возможен не только через Инспектор объектов во время создания проекта, но и программно, во время выполнения приложения (*run time*). При программном использовании **Query**, рекомендуется приведённая ниже последовательность действий.

Закреть текущий запрос.

Очистить список строк в свойстве `SQL`.

Добавить новые строки в запрос.

Инициировать выполнение запроса.

Так как свойство SQL имеет тип TStrings, то его можно формировать добавлением строк методом Add либо использованием метода LoadFromFile для загрузки текста из файла.

Пример 5.2. Сформировать программно запрос для вывода из таблицы *Country* всех сведений об Аргентине.

```
Query1.Close;  
Query1.SQL.Clear;  
Query1.SQL.Add('SELECT * FROM Country');  
Query1.SQL.Add('WHERE Name="Argentina"); Query1.Open;
```

Если версия языка **SQL** позволяет использовать шаблоны поиска без учета регистра (*case insensitive*), то, немного изменив запрос

```
Query1.SQL.Add('WHERE Name LIKE "C%");
```

удастся получить набор данных, содержащий все записи, в которых поле Name начинается с буквы 'C' .

Пример 5.3. Загрузить из файла запрос для вывода из таблицы *biolife* сведений (Category,Length_in) о рыбах категорий, названия которых начинаются на S , упорядоченных по длине.

Для использования запроса, сохранённого в текстовом файле с расширением *sql* или *txt*, надо текст запроса занести в свойство SQL. Например, можно добавить в проект компоненты **Button1**, **OpenDialog1** и записать обработчик щелчка по кнопке **Button1** (события OnClick):

```
procedure TForm1.Button1Click(Sender: TObject); begin  
  if OpenDialog1.Execute then with Query1  
    do  
    begin  
      Close;  
      SQL.LoadFromFile(OpenDialog1.FileName); Open;  
    end;  
end;
```

Предварительно в файле должен быть записан запрос:

```
SELECT category,Length_in FROM biolife WHERE category LIKE  
'S%' ORDER BY Length_in
```


5.3. Параметрические запросы

Параметрические запросы позволяют подставить значение переменной (параметра) вместо отдельных слов в выражениях WHERE или INSERT. *Параметр* - это имя с двоеточием перед ним. Имена параметров можно задавать любые, записанные в соответствии с правилами формирования имён. Имена параметров могут не совпадать с именами полей. Значение параметра может быть изменено в нужный момент.

SQL parser (программа, которая разбирает текст запроса) понимает, что имеет дело с параметром, а не константой, так как имени параметра предшествует двоеточие (например :NameStr). Двоеточие сообщает о необходимости заменить переменную NameStr некоторой величиной, которая будет задана позже. Есть два пути присвоить значение переменной в параметрическом запросе **SQL**: использовать свойство Params объекта **Query** или использовать свойство DataSource для получения информации из другого **DataSet**. Сам запрос можно сформировать в Инспекторе объектов или программно.

5.3.1. Задание параметров через свойство Params программно

Программно доступ к параметру можно выполнить несколькими способами:

- по индексу. Например, Params[0] используется для обращения к первому параметру в SQL-запросе;
- по имени методом ParamByName. Например, для обращения к параметру LastNameMy используется ParamByName(LastNameMy);
- используя свойство ParamValues класса TParams;
- используя свойство Value класса TParams.

При задании конкретного значения параметра для указания типа используется одно из свойств класса TParams (AsString, AsInteger и т.д.) или свойство Value типа Variant.

Например, для запроса

```
SELECT * FROM country WHERE Name=:Name
```

параметр Name можно задать разными способами:

```
Params[0].asString:='Argentina';  
ParamByName('Name').asString:='Argentina';  
Params[0].Value:='Argentina';  
Params[0].asString:=Edit1.text;  
Params.ParamValues['Name']:=Edit1.text;
```

В запросе

```
SELECT * FROM orders WHERE CustNo=:CustNo
```

используется целочисленный параметр CustNo. Для его задания можно использовать, например, один из приведённых ниже способов:

```
Params[0].AsInteger:=1356; ParamByName('CustNo').AsInteger:=15  
60; Params[0].Value:=1680; Params.ParamValues['CustNo']:=138 4;  
Params[0].AsInteger:=StrToInt(Edit2.text);
```

В тех случаях, когда известны возможные значения параметров, удобно использовать компонент ComboBox. Это позволит не записывать данные, а выбирать из списка. Если же нужного значения нет в списке, то его можно будет ввести, так как компонент ComboBox позволяет это сделать.

Если подставлять значение параметра в запрос через свойство Params, то обычно нужно сделать четыре шага:

- закрыть Query;
- подготовить запрос к выполнению, вызвав метод Prepare;
- присвоить необходимые значения свойству Params;
- открыть Query.

К моменту задания параметра SQL-запрос обязательно должен быть уже сформулирован. Прежде чем использовать переменную Params, можно вызвать Prepare. Этот вызов инициирует действия по обработке SQL-запроса и подготовке свойства Params к принятию соответствующего количества переменных. Метод Prepare проверяет синтаксис, компилирует и запоминает запрос в буфере BDE. При последующих вызовах запроса время на синтаксический анализ не тратится. Можно присвоить значения параметров без предварительного вызова Prepare, но это будет работать несколько медленнее. Если не указать яв

но Prepare, то каждый раз перед выполнением запроса будет использоваться метод Prepare, а после выполнения - UnPrepare. Метод UnPrepare освобождает ресурсы. Таким образом, второй шаг нужен при первом выполнении запроса, в дальнейшем его можно опустить.

Если в запросе содержится более одного параметра, то задать их можно, изменяя индекс у свойства Params либо используя доступ по имени параметра.

Итак, параметрические SQL-запросы используют переменные, которые начинаются с двоеточия и определяют места, куда будут переданы значения параметров. Для задания параметров в приложение включают компоненты, позволяющие вводить или выбирать нужные значения.

5.3.2. Использование запроса с шаблоном поиска

Пример 5.4. Из таблицы *employee* вывести сведения о сотрудниках, у которых фамилия начинается на выбранную букву. В приложении предусмотреть удобные средства для задания буквы.

Поместим на форму компоненты **Query**, **DataSource**, **DBGrid** и **TabControl**. Соединим компоненты и установим в свойстве DatabaseName компонента **Query1** псевдоним DBDemos. В Инспекторе объектов в свойстве SQL компонента **Query1** запишем текст запроса:

```
SELECT * FROM employee WHERE LastName LIKE:LastNameStr
```

В обработчике события для формы OnCreate напишем код, заполняющий закладки для компонента **TabControl1** и подготавливающий запрос:

```
procedure TForm2.FormCreate(Sender: TObject); var i:byte;  
begin  
Query1.Prepare; for i:=0 to 25  
do  
    TabControl1.Tabs.Add(chr(ord('A')+i)); end;
```

Записанный в свойстве SQL запрос выбирает записи из таблицы *employee*, в которых поле LastName похоже (LIKE) на значение параметра :LastNameStr. Параметр будет передаваться в момент переключения закладок: **procedure** TForm2.TabControl1Change(Sender: TObject);

```

begin
with Query1 do begin
    Close;
    Params[0].AsString:=
    TabControl1.Tabs.Strings[TabControl1.TabIndex]+'%'; Open;
end; end;

```

В результате выполнения запроса при переходе на нужную закладку в компоненте **DBGrid** будут отображаться сведения о сотрудниках, у которых фамилия начинается на выбранную букву.

5.3.3. Передача параметров через свойство **DataSource**

Рассмотрим создание связи «один-ко-многим» между двумя таблицами с помощью компонента **Query**. По сравнению с использованием **Table** этот способ более гибок, так как не требует индексации по полям связи.

Компонент **Query** имеет свойство **DataSource**, которое предназначено для создания связи с другим **DataSet**. Не имеет значения, каким способом получен другой набор данных: компонентами **Table**, **Query** или другим потомком **TDataSet**. Всё, что нужно для установления соединения, - это удостовериться, что у нужного **DataSet** есть связанный с ним **DataSource**.

Пример 5.5. Связать таблицы *customer* и *orders* так, чтобы каждый раз, когда пользователь выбирает имя заказчика, он видел список заказов, сделанных этим заказчиком.

Расположим на форме компоненты **Table1**, **DataSource1**, **DBGrid1** и свяжем их между собой и с таблицей *customer*.

Расположим на форме второй набор компонентов - **Query1**, **DataSource2**, **DBGrid2** и свяжем объекты между собой.

Создадим связь между таблицами *orders* (Заказы) и *customers* (Заказчики) так, что при просмотре конкретной записи о заказчике будут видны только заказы, связанные с ним. В свойстве SQL компонента **Query1** наберём текст запроса: `SELECT * FROM orders WHERE CustNo=:CustNo`

В этом запросе параметру CustNo должно быть присвоено значение из некоторого источника. Вместо того чтобы использовать свойство Params и вручную присваивать значения, можно воспользоваться данными из другой таблицы. Для этого в свойстве DataSource компонента **Query1** надо указать таблицу, из которой берутся данные. При получении данных из DataSource считается, что *в запросе после двоеточия стоит имя поля из DataSource*. При изменении текущей записи в главном **DataSet** запрос будет автоматически пересчитываться.

В свойстве DatabaseName для **Query1** укажем DBDemos. В свойстве DataSource для **Query1** укажем **DataSource1**. Присвоим свойству Active значение true и запустим программу. Таким образом, использование этого метода позволяет автоматически подставлять значения параметров. Важно корректно задать имя параметра, так как оно не может быть произвольным, а должно совпадать с именем поля, из которого берутся данные.

5.4. Изменение данных

Для изменения данных используются специальные запросы: INSERT, UPDATE , DELETE . Пример запроса для удаления записи: DELETE FROM country WHERE Name='Argentina';

Этот запрос удаляет любую запись из таблицы *country*, которая имеет значение Argentina в поле Name. Понятно, что в большинстве случаев целесообразно использовать параметрический запрос, чтобы менять название страны, сведения о которой требуется удалить: DELETE FROM country WHERE Name=:CountryName

Параметр CountryName может быть задан во время выполнения программы:
Query2.Prepare;
Query2.Params[0]='Argentina'; Query2.ExecSQL;

Сначала вызывается метод Prepare, чтобы разобрать **SQL**-запрос и подготовить свойство Params. Следующим шагом присваивается значение свойст-

в `Params`, а затем выполняется подготовленный **SQL**-запрос. Для выполнения запроса используется метод `ExecSQL`.

5.5. Рекомендации

1. Часть кода, реализующего открытие набора данных, целесообразно помещать в защищённый блок `Try...Except`.

2. При использовании параметрических запросов явно задавать методы `Prepare` и `UnPrepare`.

3. При работе с датами назначать параметру тип `ftDate` или `ftTime` и использовать явное преобразование `As`. Например:

```
Query1.ParamByName('DateParam').AsDate:=
```

4. При сравнении данных учитывать, что свойство `Value` имеет тип `Variant` и после преобразования может не получиться точного совпадения.

5. Корректно задавать параметру значение `Null`. При этом отсутствие значения - это не 0 и не пустая строка. На этапе дизайна для получения значения `Null` надо очистить у параметра свойство `Value` в Инспекторе объектов. Во время выполнения программы надо очистить параметр методом `Clear`, например: `Query2.ParamByName('LastName').Clear`

5.6. Контрольное задание

Создать в `Delphi` приложение, позволяющее получить сведения из таблицы `country` `DBDemos`:

- вывести сведения о заданном государстве;
- подсчитать количество государств на континентах;
- подсчитать количество государств на указанном континенте;
- вывести сведения о государствах, столицы которых начинаются на заданную букву;
- для всех государств определить плотность населения.

6. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ

6.1. Обзор средств доступа к данным

Существует несколько способов доступа к данным из средств разработки и клиентских приложений.

Системы управления базами данных содержат в своем составе библиотеки, предоставляющие специальный *прикладной программный интерфейс (Application Programming Interface, API)* для доступа к данным этой СУБД. Обычно такой интерфейс представляет собой набор функций, вызываемых из клиентского приложения. В случае настольных СУБД эти функции обеспечивают чтение/запись файлов базы данных, а в случае серверных СУБД инициируют передачу запросов серверу баз данных и получение от сервера результатов. Библиотеки, содержащие API для доступа к данным серверной СУБД, обычно входят в состав её клиентского программного обеспечения, устанавливаемого на компьютерах, где функционируют клиентские приложения.

В последнее время Windows-версии клиентского программного обеспечения наиболее популярных серверных СУБД, в частности Microsoft SQL Server, Oracle, Informix, содержат также СОМ-серверы, предоставляющие объекты для доступа к данным.

Использование клиентского API (или клиентских СОМ-объектов) является естественным и эффективным способом манипулирования данными в приложении. Однако в этом случае созданное приложение сможет использовать данные только конкретной СУБД, так как клиентские API и объектные модели не подчиняются каким-либо стандартам и различны для разных СУБД.

Другой способ работы с данными в приложении базируется на применении *универсальных механизмов* доступа к данным.

Универсальный механизм доступа к данным обычно реализован в виде библиотек и дополнительных модулей, называемых *драйверами* или *провайдерами*. Библиотеки содержат стандартный набор функций или классов, подчи

няющийся определённой спецификации. Дополнительные модули реализуют непосредственное обращение к функциям клиентского API конкретных СУБД.

Приложения, использующие универсальные механизмы доступа к данным, легко модифицировать, если необходима смена СУБД. Помимо очевидных достоинств универсальные механизмы имеют ряд недостатков:

- невозможность использования функциональности, специфичной для конкретной СУБД,
- снижение производительности приложений,
- усложнение процедуры поставки приложения.

Последний недостаток связан с тем, что в состав приложения нужно включать библиотеки, ответственные за реализацию универсальных механизмов, драйверы, а также обеспечивать настройки, необходимые для их правильного функционирования.

К универсальным механизмам доступа к данным относятся:

ODBC - Open Database Connectivity;

OLE DB - Object Linking and Embedding Database;

ADO - ActiveX Data Objects;

BDE - Borland Database Engine.

OLE DB и ADO - часть универсального механизма доступа к данным фирмы Microsoft (*Microsoft Universal Data Access*), позволяющая осуществить доступ как к реляционным, так и к нереляционным источникам данных, таким как файловая система, данные электронной почты, многомерные хранилища данных и др.

Итак, приложение, использующее базы данных, может применять следующие механизмы доступа к данным:

- непосредственный вызов функций клиентского API или обращение к COM-объектам (Component Object Model) клиентских библиотек;
- вызов функций ODBC API (или применение классов, инкапсулирующих подобные вызовы);
- непосредственное обращение к интерфейсам OLE DB;

- применение ADO (или применение классов, инкапсулирующих обращение к объектам ADO);
- применение ADO+OLE DB+ODBC;
- применение BDE+SQL Links (или применение классов, инкапсулирующих обращение к функциям BDE);
- применение BDE+ODBC Link+ODBC.

6.2. Особенности использования BDE

Физически BDE представляет собой набор библиотек доступа к данным, реализующих BDE API - набор вызываемых из приложения функций для работы с данными. Эти функции, в свою очередь, могут обращаться к функциям клиентского API или ODBC API, а также непосредственно выполнять считывание и запись файлов некоторых СУБД (dBase, Paradox).

Для доступа к базе данных с помощью BDE на компьютере, содержащем клиентское приложение, должны быть установлены библиотеки BDE общего назначения, а также *драйвер* для данной СУБД. В случае серверных СУБД такие драйверы носят название *SQL Links*. Эти драйверы содержат стандартный набор функций, созданных на основе функций клиентских API соответствующих СУБД. Среди BDE-драйверов имеется драйвер, созданный с использованием ODBC API, - так называемый *ODBC Link*, который применяется вместе с ODBC-драйвером для выбранной СУБД.

Для доступа к данным Paradox, dBase и текстовым файлам есть BDE-драйверы прямого доступа, манипулирующие файлами этих СУБД.

Для серверных СУБД Oracle, Sybase, IBM DB2, Informix, InterBase имеются драйверы SQL Links. Помимо этого доступ к ним может быть осуществлен с помощью ODBC Link и ODBC-драйверов. Для этих СУБД нередко имеется по несколько ODBC-драйверов разных производителей.

Доступ к данным других СУБД осуществим только с помощью ODBC Link и соответствующего ODBC-драйвера.

Применение BDE абсолютно неоправданно, когда для хранения данных используются Microsoft SQL Server и Microsoft Access. Доступ к данным этих СУБД целесообразно осуществлять с помощью ADO и OLE DB.

Применение BDE в ряде случаев связано с определенными ограничениями, причиной которых часто является отсутствие необходимых BDE-драйверов, которые, в отличие от ODBC-драйверов и OLE DB-провайдеров, не производит никто кроме фирмы Borland.

6.3. ODBC

Open Database Connectivity - широко распространённый программный интерфейс фирмы Microsoft. Для доступа к данным конкретной СУБД с помощью ODBC кроме собственно клиентской части этой СУБД нужен *ODBC Administrator* (приложение, позволяющее определить, какие источники данных доступны для данного компьютера с помощью ODBC, и описать новые источники данных), и *ODBC-драйвер* для доступа к этой СУБД.

ODBC-драйвер представляет собой динамически загружаемую библиотеку, которую клиентское приложение может загрузить в свое адресное пространство и использовать для доступа к источнику данных.

Для каждой используемой СУБД нужен собственный ODBC-драйвер, так как ODBC-драйверы используют функции клиентских API, разные для различных СУБД. С помощью ODBC можно манипулировать данными любой СУБД (и даже данными, не имеющими прямого отношения к базам данных, например данными в файлах электронных таблиц или в текстовых файлах), если для них имеется ODBC-драйвер.

Для работы с данными можно использовать как непосредственные вызовы ODBC API, так и другие универсальные механизмы доступа к данным, например OLE DB, ADO, BDE, реализующие стандартные функции или классы на основе вызовов ODBC API в драйверах или провайдерах, специально предназначенных для работы с любыми ODBC-источниками. Так как ODBC является

на данный момент наиболее часто используемым универсальным механизмом доступа к данным (ODBC-драйвер можно найти практически к чему угодно, и нередко не один), хорошие перспективы применения подобных компонентов очевидны.

Производительность приложений, использующих компоненты для доступа к ODBC-источникам, обычно выше производительности приложений, которые используют BDE и ODBC Link, за счет отказа от использования дополнительных библиотек BDE. По этой же причине упрощена поставка таких приложений, поскольку не требуется включать BDE в дистрибутив и обеспечивать его настройку на компьютере пользователя. Однако следует позаботиться об установке ODBC, наличии соответствующего ODBC-драйвера и описании ODBC-источника данных.

6.4. OLE DB

6.4.1. Основные понятия

Технология доступа к данным от Microsoft под названием OLE DB представляет собой набор COM-интерфейсов (Component Object Model).

OLE DB - это метод доступа к любым данным через стандартные COM-интерфейсы вне зависимости от типа данных и места их расположения. В качестве данных могут выступать базы данных, текстовые документы, таблицы Excel и любые другие источники данных. В отличие от доступа, предоставляемого посредством драйверов ODBC, OLE DB позволяет реализовать доступ как к SQL-серверам с применением языка SQL, так и к любым другим произвольным источникам данных.

Средства, предоставляющие доступ к источнику данных с использованием технологии OLE DB, называются *OLE DB-провайдерами*.

Для доступа к источнику данных с помощью OLE DB требуется, чтобы на компьютере, где используется клиентское приложение, был установлен OLE DB-провайдер для данной СУБД. OLE DB-провайдер представляет собой динамическую библиотеку, загружаемую в адресное пространство клиентского

приложения и используемую для доступа к источнику данных. Для каждого типа СУБД нужен собственный OLE DB-провайдер, так как эти провайдеры базируются на функциях клиентских API, разных для различных СУБД.

Если для доступа к конкретному источнику данных существует только ODBC-драйвер, то для применения технологии OLE DB можно использовать OLE DB-провайдер, предназначенный для доступа к ODBC-источнику данных. Этот провайдер использует не API клиентской части какой-либо СУБД, а интерфейс ODBC API, поэтому он применяется вместе с ODBC-драйвером для выбранной СУБД.

Так как архитектура OLE DB основана на COM, то механизм создания результирующих наборов состоит из последовательностей шагов:

- создание объекта;
- запрос указателя на интерфейс созданного объекта;
- вызов метода интерфейса.

Аналогично комплексу действий, который производится после создания результирующего набора при применении технологии ODBC - выполнения связывания, в технологии OLE DB используется механизм аксессоров.

6.4.2. Объектная модель OLE DB

Спецификация OLE DB описывает набор интерфейсов, реализуемых объектами OLE DB. Каждый объектный тип определен как набор интерфейсов. Спецификация OLE DB определяет набор интерфейсов базового уровня, которые должны реализовываться любыми OLE DB-провайдерами.

В базовую модель OLE DB входят объекты **DataSource**, **Session**, **Rowset**.

Объект **DataSource** (источник данных) предназначен для соединения с источником данных и создания одного или нескольких сеансов. Этот объект управляет соединением, использует информацию о полномочиях и аутентификации пользователя.

Объект **Session** (сеанс) управляет взаимодействием с источником данных -выполняет запросы и создает результирующие наборы, позволяет возвращать метаданные. В сеансе может создаваться одна или несколько команд.

Объект **Rowset** (результатирующий набор) представляет собой данные, извлекаемые в результате выполнения команды или создаваемые в сеансе.

Спецификация OLE DB определяет объект **Command** (команда), предназначенный для выполнения текстовой команды. В качестве такой команды может выступать и SQL-оператор. При этом выполнение команды может создавать результирующий набор.

Некоторые OLE DB-провайдеры поддерживают работу со схемой (Schema), которая предоставляет метаданные по базе данных. Метаданные становятся доступны как обычные результирующие наборы. В заголовочном файле *oledb.h* содержатся уникальные идентификаторы всех доступных типов результирующих наборов схемы данных (например, для получения информации по таблицам базы данных следует указать уникальный идентификатор DBSCHEMA_TABLES). Столбец результирующего набора с именем TABLE_NAME содержит имя таблицы, столбец TABLETYPE указывает тип таблицы.

Для обеспечения расширенных возможностей управления транзакциями объектная модель OLE DB включает объект **Transaction**.

OLE DB-провайдеры, как и все COM-компоненты, регистрируются в реестре Windows. Для поиска информации о зарегистрированных источниках данных используются специальные объекты, называемые нумераторами.

Для каждого объектного типа спецификация OLE DB определяет набор интерфейсов, который должен обязательно быть реализован для данного объекта. Такие интерфейсы отмечаются как [mandatory]. Интерфейсы, которые могут отсутствовать, отмечаются как [optional].

6.4.3. Создание результирующего набора

При реализации доступа к БД посредством OLE DB-провайдера сначала следует создать объект данных и установить соединение с базой данных. Далее необходимо создать объект *сеанс*. И только потом можно создавать результирующий набор одним из имеющихся способов.

Для объекта *сеанс* вызывается метод IOpenRowset, выполняющий непосредственное создание результирующего набора (интерфейс IOpenRowset должен поддерживаться любым провайдером).

Для объекта *сеанс* вызывается метод `IDBCreateCommand`, создающий объект *Command*. Далее для объекта *команда* вызывается метод `iCommand`. Затем вызывается метод, формирующий результирующий набор данных.

Чтобы результирующий набор, хранимый на сервере, можно было использовать, необходимо выполнить связывание и извлечение данных. Для этого следует определить структуры, описывающие столбцы, создать *аксессор* и вызвать один из методов получения строк результирующего набора. Затем вызвать метод для записи данных в структуру, определенную аксессором.

После получения и обработки строк их следует освободить, а после просмотра всего результирующего набора освободить аксессор и сам результирующий набор.

6.5. ADO

Компанией Microsoft был предложен механизм доступа к данным ActiveX Data Objects (ADO), построенный на использовании интерфейсов OLE DB. ADO - это набор библиотек, содержащих COM-объекты, реализующие прикладной программный интерфейс для доступа к данным и используемые в клиентских приложениях. Технология ADO использует библиотеки OLE DB, предоставляющие низкоуровневый интерфейс для доступа

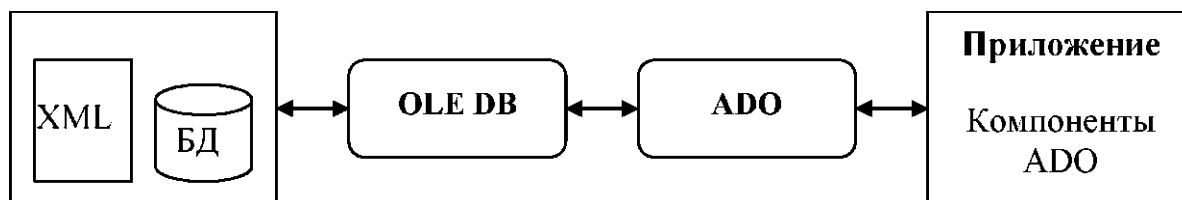


Рис. 45. Схема доступа к данным, использующая ADO

к данным.

ADO становится всё более популярным способом доступа к данным, так как включен в ядро операционных систем семейства Windows, и входит в состав таких популярных продуктов, как MS Office и MS Internet Explorer.

Согласно терминологии ADO любой источник данных (база данных, электронная таблица, файл) называется хранилищем данных, с которым приложение взаимодействует при посредстве провайдера.

Приложение обращается к данным не напрямую, а через объект OLE DB, который «умеет» работать с разнообразными данными. Технология ADO включает в себя набор объектов и механизмов, обеспечивающих взаимодействие объектов с данными и приложениями. Очень важную роль играют провайдеры ADO, координирующие работу приложений с хранилищами данных различных типов.

Набор объектов и соответствующий провайдер могут быть созданы для любого хранилища данных без внесения изменений в структуру ADO.

Для каждого используемого типа хранилища данных должен существовать провайдер ADO. Провайдер знает, какие данные и где расположены, умеет обращаться к данным с запросами и интерпретировать возвращаемую служебную информацию и результаты запросов для передачи приложениям. При установке соединения через соответствующие компоненты становится доступен список установленных в операционной системе провайдеров.

В технологии ADO используются объекты-перечислители, источники данных, сессии, транзакции, наборы рядов, команды.

Объекты-перечислители выполняют поиск объектов ADO, осуществляющих доступ к источнику данных.

Для соединения с хранилищем данных используются два типа объектов: источники данных и сессии.

Объект-набор рядов обеспечивает работу с данными.

Объект-команда объединяет текстовую команду и механизмы обработки команд. Команды позволяют использовать для работы с данными язык SQL.

6.6. Механизмы доступа к данным, поддерживаемые Delphi

Имеющиеся в Delphi классы и компоненты позволяют быстро и эффективно разрабатывать приложения баз данных. Для удобства использования компоненты разбиты на группы:

1. Компоненты для доступа к данным, реализующие:

- доступ через процессор баз данных BDE, используя ODBC-драйверы или внутренние драйверы BDE;

- доступ через ADO-объекты, в основе которого лежит применение технологии OLE DB;
 - доступ к локальному или удалённому SQL-серверу InterBase;
 - доступ посредством драйверов dbExpress;
 - доступ к БД при многозвенной архитектуре (компоненты страницы DataSnap);
2. Компоненты для связи источников данных с визуальными компонентами, предоставляющими интерфейс пользователя;
 3. Визуальные компоненты, реализующие интерфейс пользователя;
 4. Компоненты для визуального проектирования отчётов.

Разные механизмы работы с данными имеют схожие схемы. Ранее подробно была рассмотрена схема работы через BDE с использованием внутренних драйверов (с таблицами форматов Paradox, dBase, FoxPro). Работа через BDE с использованием ODBC-драйверов организуется аналогично.

В модуль данных (или в форму) добавляется компонент набора данных (объект класса TDataSet) и устанавливается связь с источником данных, определяемая свойством DataSourceName. Связь может быть указана по имени БД, каталогу или псевдониму (ограничения зависят от типа источника данных).

В модуль данных (или в форму) добавляется компонент источника данных (TDataSource), являющийся связующим звеном между набором данных и элементами управления, отображающими данные. Свойство DataSource компонента типа TDataSource указывает набор данных, формируемый компонентами таких классов, как TTable или TQuery.

В форму добавляются элементы управления для работы с данными, такие как TDBGrid, TDBEdit, TDBCheckBox и т.п. Они связываются с источником данных через свойство DataSource.

При реализации схем доступа к данным через ADO, dbExpress применяются другие компоненты, но подход остаётся тем же.

Предком всех классов наборов данных является класс TDataSet. В зависимости от механизма доступа, используемого приложением, базовыми классами набора данных могут быть:

TTable, TQuery, TStoredProc - для однозвенных или двухзвенных приложений, использующих BDE;

TClientDataSet - для реализации клиентского набора данных и для многозвенной архитектуры, использующей распределенный доступ;

TADODataset - для приложений, использующих ADO-объекты;

TSQLDataSet - для доступа к базе данных посредством dbExpress. Этот класс реализует направленный набор данных. Для такого набора данных не создается кэш памяти на клиенте, и среди методов доступа возможны только методы Next и First. Редактирование записей в направленном наборе данных возможно только явным выполнением SQL-оператора UPDATE или при установке соединения с клиентским набором данных через провайдер;

TSQLTable и TSQLQuery - для доступа к базе данных посредством dbExpress.

На рис. 46 приведена иерархия классов наборов данных библиотеки VCL системы Delphi. При работе с компонентами наборов данных можно обойтись без явного использования компонентов, реализующих соединение с базой данных. Однако некоторые возможности, такие как управление транзакциями или кэшированные обновления, невозможны без компонентов типа TDatabase или TADOConnection.

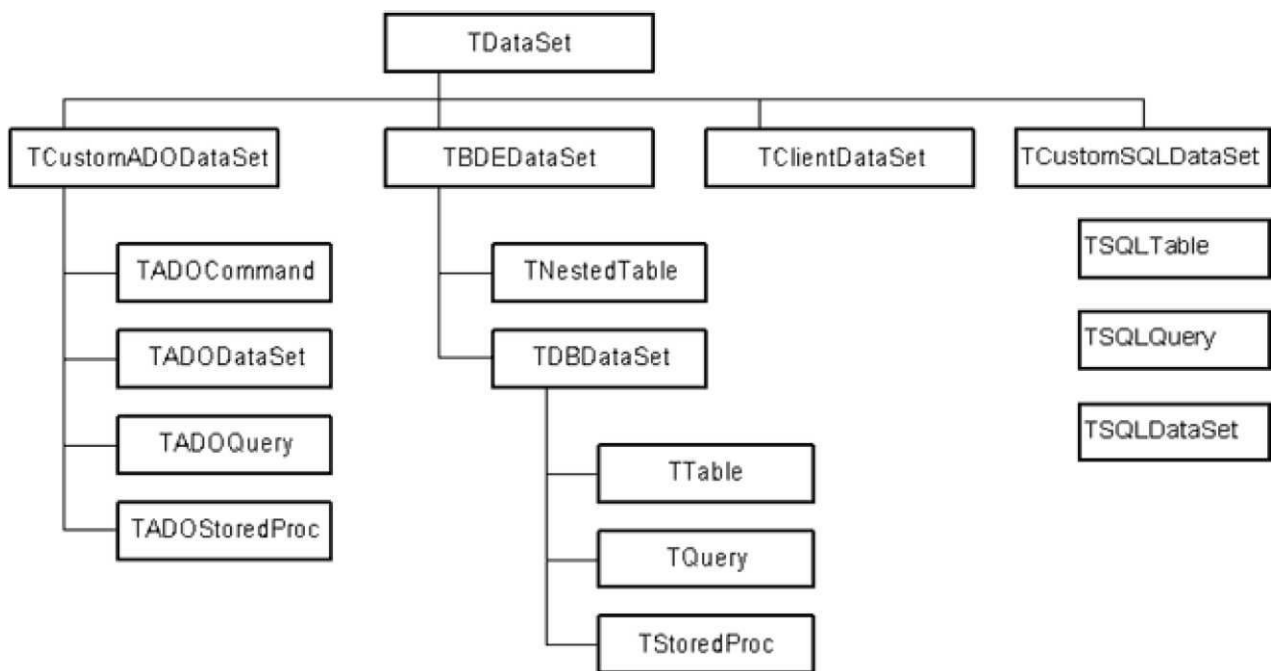


Рис. 46. Иерархия классов, реализующих доступ к

Компонент база данных TDatabase применяется для соединения с источником данных через драйверы BDE или внешние ODBC-драйверы. Компонент ADOConnection используется для создания объекта-соединения при доступе через OLE DB, который реализуется посредством ADO-объектов VCL-библиотеки.

По умолчанию при переходе от одной записи набора данных к другой происходит запись всех сделанных изменений в базу данных. Для того чтобы можно было отменять сделанные изменения или выполнять обновление нескольких записей, применяют кэшированные обновления. Они позволяют значительно снизить сетевой трафик за счет того, что все сделанные изменения хранятся во внутреннем кэше и при переходе от одной записи к другой информация в базу данных не передаётся. Чтобы включить режим кэшированного обновления, следует установить значение свойства CachedUpdates равным True для компонента набора данных. Для присвоения кэшированного обновления вызывается метод ApplyUpdates, а для отмены - CancelUpdates.

6.7. Технология ADO

Для применения технологии ADO в Delphi 7 предназначены семь компонентов, расположенных на закладке ADO палитры компонентов.

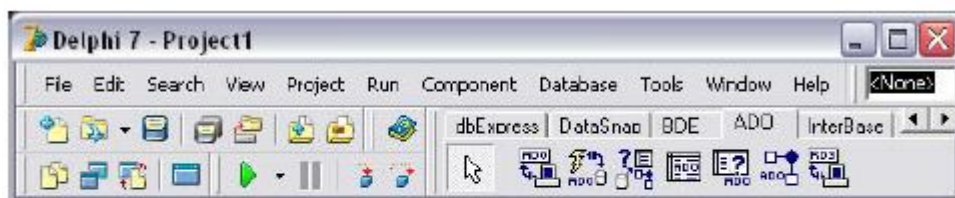


Рис. 47. Компоненты страницы ADO

Таблица 10

Назначение компонентов страницы ADO

Название	Описание
ADOConnection	Функционально аналогичен компоненту Database закладки BDE. Позволяет указывать местоположение базы данных и работать с транзакциями
ADOCommand	Предназначен для выполнения SQL-команды без возврата результирующего набора данных

Название	Описание
ADODataset	Предназначен для получения набора данных из одной или нескольких таблиц БД. Позволяет работать с возвращённым набором данных визуальным компонентом
ADOTable	Аналог компонента Table , расположенного на закладке BDE. Используется для доступа к таблице с помощью механизма ADO
ADOQuery	Аналог Query . Позволяет формировать запросы к БД, которые возвращают данные из базы (например, командой SELECT) или не формируют результирующего набора данных (например, INSERT)
ADOStoredProc	Предназначен для вызова процедуры, хранимой на сервере базы данных. Является потомком TDataSet, в отличие от BDE и InterBase позволяет возвращать набор данных, поэтому может выступать источником данных в компонентах типа DataSource
RDSCConnection	Управляет механизмом, который позволяет клиенту получать доступ к объектам, расположенным в другом адресном пространстве или на другом компьютере

Класс TADOConnection обеспечивает соединение с данными, доступ к которым реализуется через ADO-объекты. Компоненты **ADOConnection** используют для доступа к данным OLE DB-провайдеры.

Компоненты **ADOCommand** и **ADODataset** связываются с источником данных посредством объекта **ADOConnection**, указывая ссылку на него как значение свойства Connection.

Для идентификации соединения необходимо определить значение свойства ConnectionString (строка соединения) компонента **ADOConnection**, которое может основываться на указании datalink-файла или строки соединения. Если в качестве значения свойства ConnectionString указано имя datalink-файла, то настройку соединения можно выполнять автономно от приложения (например, указывая имя базы данных Microsoft SQL Server на текущем ПК).

Реализацию доступа к данным через ADO проще всего рассмотреть на примерах.

Пример 6.1. Разработать в Delphi приложение для просмотра объектов базы данных MS Access, используя механизм ADO.

Последовательность действий

1. Откроем Delphi, создадим приложение и разместим на форме три компонента:

ADOTable с закладки **ADO**; **DataSource** с

закладки **Data Access**; **BDGrid** с закладки

Data Controls.

2. Свяжем компоненты между собой:

- установим значение свойства DataSource компонента **DBGrid1** в **DataSource1**;
- в свойстве DataSet компонента **DataSource1** укажем **ADOTable1**.

3. Зададим параметры соединения компонента **ADOTable1**:

- в свойстве **ConnectionString** нажмём кнопку с многоточием, в окне редактора параметров соединения установим переключатель в положение **Use Connection String** и нажмём кнопку **Build**;



Рис. 48. Окно задания свойства ConnectionString

- в появившемся окне на вкладке **Поставщик данных** выберем свойство Microsoft Jet 4.0 OLE DB Provider;
- перейдём на вкладку **Подключение**, укажем путь к БД, введём имя пользователя и при необходимости зададим пароль;
- нажмём кнопку **Проверить подключение**;
- после завершения проверки подключения перейдём на вкладку **Дополнительно** и поставим галочки напротив свойств ReadWrite, Share Deny None;

- выберем вкладку **Все** и проверим сделанные установки. При необходимости изменить значение выберем нужную строчку, нажмём кнопку **Изменить значение**, в появившемся окне выберем значение и нажмём **ОК**;
- нажмём два раза **ОК**;
- в свойстве TableName компонента **ADOTable1** укажем нужную таблицу;
- в свойстве Active установим значение true.

Если всё сделано правильно, то после задания таблицы компонент **DBGrid1** заполнится данными (рис. 51).

При создании проекта в Delphi 2006 всё выполняется аналогично. Надо только учесть некоторые различия в интерфейсе:

- 1) компоненты для доступа к данным по технологии ADO находятся в разделе dbGo;
- 2) свойства в Инспекторе объектов объединены в группы по смыслу (раньше располагались по алфавиту).

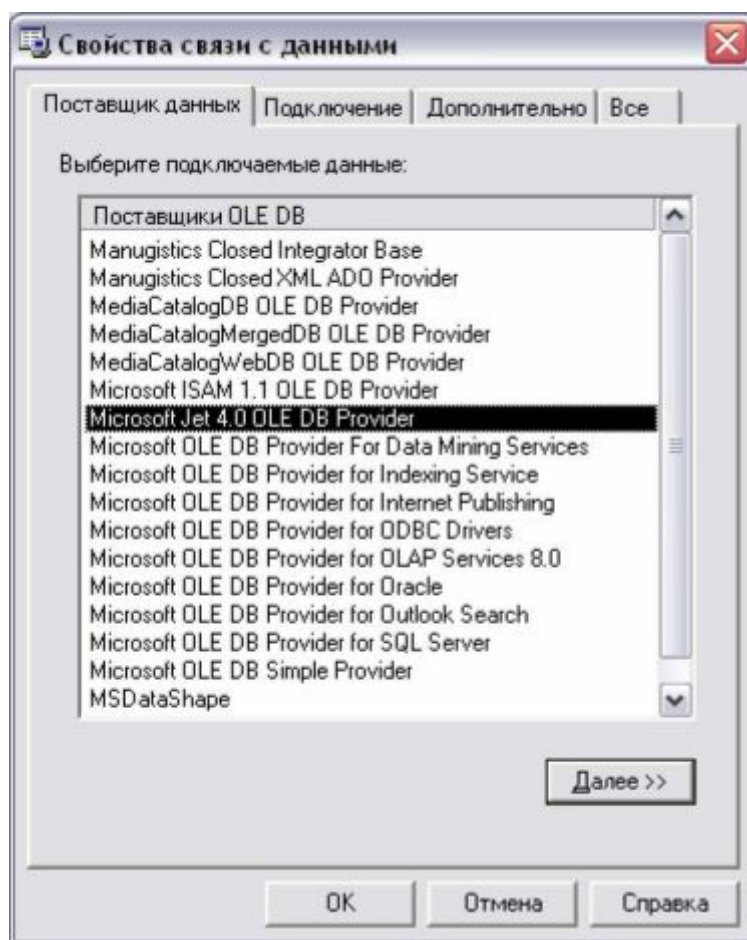


Рис. 49. Выбор OLE DB-провайдера

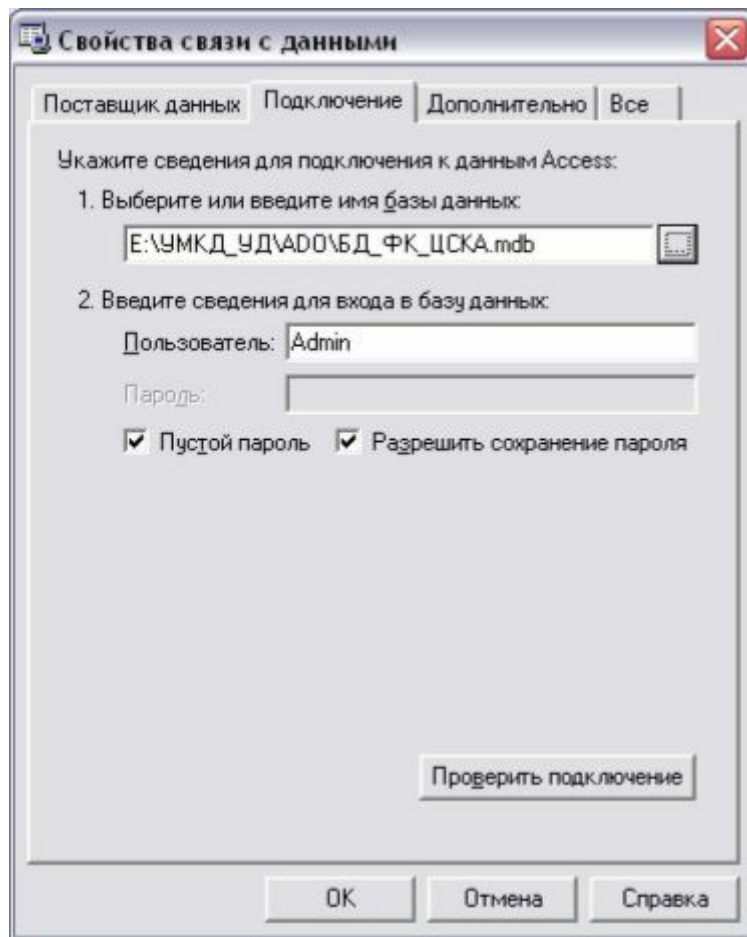


Рис. 50. Задание источника

Технология ADO. Использование файла mdb

Номер в команде	Игрок(ФИО)	Гражданство	Позиция	Дата рождения	Код игры
2	Шемберас Д.	Литва	Защитник	02.08.1978	0
4	Игнашевич С.	Россия	Защитник	14.07.1979	0
6	Березуцкий А.	Россия	Защитник	20.06.1982	0
7	Карвалью Д.	Бразилия	Полузащитник	01.03.1983	0
8	Гусев Р.	Россия	Полузащитник	17.07.1977	0
9	Олич И.	Хорватия	Нападающий	14.09.1979	0
10	Жо	Бразилия	Нападающий	20.03.1987	0
11	Вагнер С.	Бразилия	Нападающий	11.06.1984	0
18	Жирков Ю.	Россия	Полузащитник	20.08.1983	0
20	Дуду	Бразилия	Полузащитник	15.04.1983	0
24	Березуцкий В.	Россия	Защитник	20.06.1982	0
25	Рахимич Э.	Россия	Полузащитник	04.04.1976	0
35	Акинфеев И.	Россия	Вратарь	08.04.1986	0

Рис. 51. Приложение для просмотра таблицы базы данных

6.8. Технология dbExpress

ODBC Express представляет собой набор компонентов и классов для Delphi и C++ Builder, применяемых для доступа к ODBC-источникам данных и инкапсулирующих вызовы ODBC API. Для работы приложений, использующих эти компоненты и классы, требуются библиотеки ODBC (доступные на Web-сервере Microsoft) и ODBC-драйвер для выбранной СУБД. Сами компоненты и классы ODBC Express располагаются внутри исполняемого файла приложения.

6.9. Контрольные вопросы

1. Как расшифровывается аббревиатура API?
2. Чем по сути является API?
3. Назовите недостатки универсальных механизмов доступа к данным.
4. При использовании ODBC для каждой СУБД нужен собственный ODBC-драйвер или есть универсальный?
5. Как организовать доступ к данным по технологии OLE DB, если для нужной СУБД нет OLE DB-драйвера, но есть ODBC-драйвер?
6. Приведите схему доступа к данным с применением ADO.
7. Почему активно используется доступ к данным с использованием ADO?
8. Какие объекты используются в технологии ADO?
9. Какие компоненты Delphi используются для организации доступа к данным по технологии ADO?
10. Как задаются параметры соединения при разработке в Delphi приложения, использующего технологию ADO?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Delphi 7*. Наиболее полное руководство / А.Д. Хомоненко и др. - СПб.: БХВ - Санкт-Петербург, 2014. - 1216 с.
2. *Архангельский А.Л.* Delphi 7. - М.: Бином, 2004. - 1120 с.
3. *Дарахвелидзе П.Г., Марков Е.П.* Delphi 2005 для Win 32. - СПб.: БХВ - Санкт-Петербург, 2010. - 1136 с.
4. *Карпова Т.С.* Базы данных: модели, разработка, реализация. - СПб.: Питер, 2012. - 304 с.
5. *Калверт Ч.* Delphi 4. Энциклопедия пользователя. - Киев.: ДияСофт, 2010. - 800 с.
6. *Фаронов В.В.* Delphi 2005. Язык, среда, разработка приложений. - СПб.: Питер, 2011. - 580 с.
7. *Фаронов В.В., Шумаков П.В.* DELPHI 4. Руководство разработчика баз данных. - М.: Нолидж, 2012. - 560 с.
8. *Шейкер Т.Д.* Разработка приложений в системе Delphi: Учеб. пособие. - Владивосток: Изд-во ДВГТУ, 2010. - 172 с.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ	3
1. СРЕДСТВА DELPHI ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ.....	4
1.1. Borland Database Engine	4
1.2. Утилиты для работы с базами данных в Delphi.....	5
1.3. Псевдонимы	7
1.4. Обзор компонентов для работы с базами данных	9
1.5. Модули данных.....	11
1.6. Невизуальные компоненты для работы с данными.....	12
1.7. Редактор полей и его использование.....	16
1.8. Визуальные компоненты для работы с данными	20
1.9. Редактор колонок.....	23
1.10. Контрольные вопросы	25
2. НАВИГАЦИОННЫЙ СПОСОБ ДОСТУПА К ДАННЫМ.....	26
2.1. Операции с таблицей базы данных	26
2.2. Навигация по набору данных	26
2.3. Доступ к полям.....	29
2.4. Модификация набора данных.....	32
2.5. Работа с записями	37
2.6. Сортировка.....	38
2.7. Фильтрация	40
2.8. Поиск.....	50
2.9. Положение курсора. Закладки.....	53
2.10. Создание диаграмм.....	57
2.11. Контрольные вопросы	60
3. ОСНОВЫ SQL.....	61
3.1. Стандарты и реализации языка SQL	61
3.2. Группы операторов SQL.....	62
3.3. Выполнение запросов в SQL Explorer.....	64
3.4. Применение языка SQL	66
3.5. Формирование запросов средствами языка SQL.....	69
3.6. Оператор SELECT.....	70
3.7. Предложение WHERE	73
3.8. Вычисления в запросах.....	84
3.9. Контрольные вопросы и задания.....	88
4. ФОРМИРОВАНИЕ СЛОЖНЫХ SQL-ЗАПРОСОВ.....	90
4.1. Соединение таблиц	90
4.2. Внутреннее соединение	91
4.3. Самосоединения.....	96

4.4. Внешнее соединение.....	97
4.5. Объединение запросов.....	101
4.6. Использование вложенных запросов	103
4.7. Контрольные вопросы и задания.....	105
5. РЕЛЯЦИОННЫЙ СПОСОБ ДОСТУПА К ДАННЫМ.....	107
5.1. Компонент Query	107
5.2. Реализация запросов	108
5.3. Параметрические запросы	112
5.4. Изменение данных	116
5.5. Рекомендации.....	117
5.6. Контрольное задание	117
6. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ	118
6.1. Обзор средств доступа к данным	118
6.2. Особенности использования BDE.....	120
6.3. ODBC.....	121
6.4. OLE DB.....	122
6.5. ADO	125
6.6. Механизмы доступа к данным, поддерживаемые Delphi	126
6.7. Технология ADO.....	129
6.8. Технология dbExpress	134
6.9. Контрольные вопросы	134
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	135

