БАЗЫ ДАННЫХ

Оглавление

1. Базы данных и управление ими, архитектура системы баз данных

Любая организация нуждается в своевременном доступе к информации. Ценность информации в современном мире очень высока. Роль распорядителей информации в современном мире чаще всего выполняют базы данных. Базы данных обеспечивают надежное хранение информации, структурированном виде и своевременный доступ к ней. Практически любая современная организация нуждается в базе данных, удовлетворяющей те или иные потребности по хранению, управлению и администрированию данных.

База данных может быть определена как совокупность предназначенных для машинной обработки и хранения данных, которые могут использоваться одним или несколькими пользователями. Её можно также рассматривать как подобие электронной картотеки, т.е. контейнер для некоторого набора файлов данных (или таблиц), занесенных в компьютер.

Пользователям этой системы предоставляется возможность выполнять (или передавать системе запросы на выполнение) множество различных операций над такими файлами, например:

- добавлять новые пустые файлы в базу данных;
- вставлять новые данные в существующие файлы;
- получать данные из существующих файлов;
- удалять данные из существующих файлов;
- изменять данные в существующих файлах;
- удалять существующие файлы из базы данных.

Базы данных позволяют хранить, структурировать информацию и извлекать оптимальным для пользователя образом. Использование клиент-серверных технологий позволяют сберечь значительные средства, а главное и время для получения необходимой информации, а также упрощают доступ и ведение, поскольку они основываются на комплексной обработке данных и централизации их хранения. Кроме того, ЭВМ позволяет хранить любые форматы данных, текст, чертежи, данные в рукописной форме, фотографии, записи голоса и т.д.

Информация в БД должна быть:

- непротиворечивой
- неизбыточной
- целостной

Для использования столь огромных объемов хранимой информации, помимо развития системных устройств, средств передачи данных, памяти, необходимы средства обеспечения диалога человек – ЭВМ, которые позволяют пользователю вводить запросы, читать файлы, модифицировать хранимые данные, добавлять новые данные или принимать решения на

основании хранимых данных. Для обеспечения этих функций созданы специализированные средства – системы управления базами данных (СУБД). Современные СУБД – многопользовательские системы управления базой данных, которые специализируется на управлении массивом информации одним или множеством одновременно работающих пользователей.

Современные СУБД обеспечивают:

– набор средств для поддержки таблиц и отношений между связанными таблицами;

– развитый пользовательский интерфейс, который позволяет вводить и модифицировать информацию, выполнять поиск и представлять информацию в графическом или текстовом режиме;

– средства программирования высокого уровня, с помощью которых можно создавать собственные приложения.

Все запросы пользователей на получение доступа к базе данных обрабатываются СУБД. Все имеющиеся средства добавления файлов (или таблиц), выборки и обновления данных в этих файлах или таблицах также предоставляет СУБД. Основная задача СУБД – дать пользователю базы данных возможность работать с ней, *не вникая во все подробности работы на уровне аппаратного обеспечения.* (Пользователь СУБД более отстранен от этих подробностей, чем прикладной программист, применяющий языковую среду программирования.) Иными словами, СУБД позволяет конечному пользователю рассматривать базу данных как объект более высокого уровня по сравнению с аппаратным обеспечением, а также предоставляет в его распоряжение набор операций, выражаемых в терминах языка высокого уровня.

В состав языковых средств современных СУБД входят:

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

Прикладные программы, или приложения, служат для обработки данных, содержащихся в БД. Пользователь осуществляет управление БД и работу с её данными именно с помощью приложений, которые называются приложениями БД. Иногда термин «база данных» трактуют в более широком

смысле и обозначают им не только саму БД, но и приложения, обрабатывающие её данные.

В зависимости от взаимного расположения приложения и БД можно выделить:

локальные БД;

удалённые БД

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

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

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

Архитектура файл-сервер обычно применяется в сетях с небольшим количеством пользователей, для её реализации подходят локальные СУБД, например, MS Access, Paradox или dBase. Достоинством этой архитектуры является простота реализации, а также то, что приложение фактически разрабатывается в расчёте на одного пользователя.

Недостатками такого варианта архитектуры можно считать следующие:

В связи с тем, что на каждом компьютере имеется своя копия БД, изменения, сделанные в ней одним пользователем, в течение некоторого времени не известны другим пользователям. Поэтому требуется постоянное обновление БД. Кроме того, возникает необходимость синхронизации работы отдельных пользователей, связанных с блокировкой в таблицах записей, которые в данный момент редактирует другой пользователь.

Удалённая БД размещается на компьютере-сервере сети, а приложение, работающее с этой БД, находится на компьютере пользователя. В этом случае мы имеем дело с архитектурой *клиент-сервер*, когда информационная система делится на неоднородные части – сервер и клиент БД. В связи с тем, что компьютер-сервер отделён от клиента, его также называют *удалённым сервером*.

Клиент – это приложение пользователя. Для получения данных клиент формирует и отсылает запрос удалённому серверу, на котором размещена БД. Запрос формулируется на языке SQL, который является стандартным

средством доступа к серверу при использовании реляционных моделей данных. После получения запроса удалённый сервер направляет его программе SQL Server (серверу БД) – специальной программе, управляющей удалённой БД и обеспечивающей выполнение запроса и выдачу результата клиенту. Вся обработка запроса выполняется на удалённом сервере.

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

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

2. Реляционная модель данных

Любая модель данных должна содержать три компонента:

1) ***структуру данных*** – описывает точку зрения пользователя на представление данных;

2) ***набор допустимых операций***, выполняемых на структуре данных. Модель данных предполагает, как минимум, наличие языка определения данных (DDL – Data Definition Language), описывающего структуру их хранения, и языка манипулирования данными (DML – Data Manipulation Language), включающего операции извлечения и модификации данных;

3) ***ограничения целостности*** – механизм поддержания соответствия данных предметной области на основе формально описанных правил.

В процессе исторического развития в СУБД использовались следующие модели данных: *иерархическая, сетевая, реляционная.* В последнее время все большее значение приобретает объектно ориентированный подход к представлению данных.

Здесь мы рассмотрим *реляционную модель*.

В основе реляционной модели данных лежат три основные концепции: *реляционные отношения, ограничения целостности данных и алгебра реляционных операторов*

Переход к реляционным базам данных обычно датируется публикацией статьи сотрудника IBM Эдгара Кодда «Реляционная модель данных для больших совместно используемых банков данных», опубликованной в 1970 году. В этой и последующих статьях Э. Кодда и других авторов (среди которых следует отметить автора популярнейшего учебника по базам данных К. Дейта) были развиты основные положения реляционной модели данных.

Интересно отметить, что само понятие «модель данных», означающее переход на более абстрактный уровень оперирования данными, было сформулировано именно Коддом и в базах данных первого поколения не использовалось. Кодд предложил использовать для обработки данных аппарат теории множеств.

Особенностью реляционной модели является совершенно новый взгляд на построение базы данных и действия с данными. Основной структурной единицей реляционной модели является *отношение* (relation), которое с теоретической точки зрения является легко описываемым и хорошо изученным математическим объектом, а с практической – может, с некоторыми оговорками, трактоваться как таблица простейшей структуры, не имеющая повторяющихся строк. Это резко отличает ее от иерархической и сетевой моделей данных, в которых узлы графа, описывающего структуру базы данных, могли содержать сколь угодно сложные информационные объекты. Упрощение структуры данных, тем не менее, не привело к потере функциональных возможностей обработки данных. Кодду удалось показать, что отношение (таблица) является в достаточной степени насыщенным информационным объектом, чтобы обеспечить произвольную обработку данных, если они изначально записаны в табличной форме. Простота структурных единиц реляционной модели позволила привлечь формальные математические методы для описания обработки данных. С этой целью Э. Коддом были разработаны языки *реляционной алгебры* и *реляционного исчисления*, обладающие необходимой полнотой, требуемой для такой обработки. Названные языки высокоуровневые, операндами и результатами в которых являются отношения, а детализация алгоритмов их реализации возложена на систему управления базой данных.

Положив теорию отношений в основу реляционной модели, Э. Кодд обосновал реляционную замкнутость отношений и ряда некоторых специальных операций, которые применяются сразу ко всему множеству строк отношения, а не к отдельной строке. Указанная реляционная замкнутость означает, что результатом выполнения операций над отношениями является также отношение, над которым в свою очередь можно осуществить некоторую операцию. Из этого следует, что в данной модели можно оперировать реляционными выражениями, а не только отдельными операндами в виде простых имен таблиц.

Одним из основных преимуществ реляционной модели является ее однородность. Все данные рассматриваются как хранимые в таблицах и только в таблицах. Каждая строка такой таблицы имеет один и тот же формат.

К числу достоинств реляционного подхода можно отнести:

– наличие небольшого набора абстракций, которые позволяют сравнительно просто моделировать большую часть распространенных предметных областей и допускают точные формальные определения, оставаясь интуитивно понятными;

– наличие простого и в то же время мощного математического аппарата, опирающегося главным образом на теорию множеств и математическую логику и обеспечивающего теоретический базис реляционного подхода к организации БД;

– возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

2.1. Реляционное отношение

Рассмотрим следующий пример. Имеется список организаций-поставщиков, поставлявших некоторые товары в определенном количестве по определенной цене за 1 ед. Каждый поставщик описывается следующими характеристиками (свойствами, атрибутами): *код_п* (код поставщика), *имя_п* (имя поставщика), *гор* (город), в котором находится офис данного поставщика. Для описания товара используются такие характеристики, как *код_т* (код товара), *назв_т* (название товара).

Всю информацию о поставках и свойствах объектов данной предметной области можно представить в виде таблицы, например, такого вида:

Таблица «Поставщик-товар»

код_п	*имя_п*	*гор*	*код_т*	*назв_т*	*цена_1*	*кол_во*
п01	Скан	Ярославль	т14	монитор	7 500	10
п01	Скан	Ярославль	т09	принтер	3 400	4
п02	Альфа	Москва	т14	монитор	7 200	6
п03	Тензор	Ярославль	т03	HD диск	4 300	12
п11	ИП Иванов С. Н.	Тутаев	т23	стол	2 000	3

Имена столбцов таблицы – сокращенные названия характеристик (*атрибутов*) объектов рассматриваемой предметной области (поставщиков и товаров), а также свойств поставок (цена, количество поставляемых изделий). Каждая строка таблицы описывает одну конкретную поставку. Эту таблицу можно трактовать нестрого как задание *реляционного отношения*.

Реляционная база данных – это конечный (ограниченный) набор *отношений*. Отношения используются для представления сущностей, а также для представления связей между сущностями. *Отношение* – это двумерная *таблица*, имеющая уникальное имя и состоящая из *строк* и *столбцов*, где *строки соответствуют записям, а столбцы – атрибутам*. Каждая строка в таблице представляет некоторый объект реального мира или соотношения между объектами.

Атрибут – это поименованный столбец отношения. Свойства сущности, его характеристики определяются значениями атрибутов. Порядок следования атрибутов не влияет на само отношение.

Для строгого определения потребуются следующие понятия.

Типы данных, используемые в реляционной модели

Реляционная модель требует, чтобы типы используемых данных были *простыми*. *Простые, или атомарные, типы данных* не обладают внутренней структурой. Требование, чтобы тип данных был простым, нужно понимать так, что *в реляционных операциях не должна учитываться внутренняя структура данных*. Конечно, должны быть описаны действия, которые можно производить с данными как с единым целым, например данные числового типа можно складывать, для строк возможна операция конкатенации и т. д.

Прежде чем говорить о целостности сущностей, опишем использование null-значений в реляционных базах данных.

Null-значения данных

Основное назначение баз данных состоит в том, чтобы хранить и предоставлять информацию о реальном мире. Для представления этой информации в базе данных используются привычные для программистов типы данных – строковые, численные, логические и т. п. Однако в реальном мире часто встречается ситуация, когда данные неизвестны или неполны. Например, место жительства или дата рождения человека могут быть неизвестны (база данных разыскиваемых преступников). Если вместо неизвестного адреса уместно было бы вводить пустую строку, то что вводить вместо неизвестной даты?

Для того чтобы обойти проблему неполных или неизвестных данных, в базах данных могут использоваться типы данных, пополненные так называемым ***null-значением***. null-значение – это, собственно, не значение, а некий маркер, показывающий, что значение неизвестно.

Таким образом, в ситуации, когда возможно появление неизвестных или неполных данных, разработчик имеет на выбор два варианта.

Первый вариант состоит в том, чтобы ограничиться использованием обычных типов данных и не использовать null-значения, а вместо неизвестных данных вводить либо нулевые значения, либо значения специального вида – например, договориться, что строка «АДРЕС НЕИЗВЕСТЕН» и есть те данные, которые нужно вводить вместо неизвестного адреса. В любом случае на пользователя (или на разработчика) ложится ответственность за правильную трактовку таких данных. В частности, может потребоваться написание специального программного кода, который в нужных случаях «вылавливал» бы такие данные. Проблемы, возникающие при этом, очевидны – не все данные становятся равноправны, требуется дополнительный программный код, отслеживающий эту неравноправность, в результате чего усложняется разработка и сопровождение приложений.

Второй вариант состоит в использовании null-значений вместо неизвестных данных. При таком подходе есть менее очевидные и более глубокие проблемы. Наиболее бросающейся в глаза проблемой является необходимость использования трехзначной логики при оперировании с

данными, которые могут содержать null-значения. В этом случае при неаккуратном формулировании запросов даже самые естественные запросы могут давать неправильные ответы.

Практически все реализации современных реляционных СУБД позволяют использовать null-значения, несмотря на их недостаточную теоретическую обоснованность.

Домены

В реляционной модели данных с понятием тип данных тесно связано понятие домена, которое можно считать уточнением типа данных.

Определение 2.1. Домен – это подмножество значений некоторого типа данных имеющих определенный смысл.

Домен характеризуется следующими свойствами:

- домен имеет *уникальное имя* (в пределах базы данных);
- домен определен на некотором *простом* типе данных или на другом домене;
- домен может иметь некоторое *логическое условие*, позволяющее описать подмножество данных, допустимых для данного домена;
- домен несет определенную *смысловую нагрузку*.

Например, домен *D*, имеющий смысл «возраст сотрудника» можно описать как следующее подмножество множества *N* натуральных чисел:

$$D = \{n \in N : n \geq 18 \ \ and \ \ n \leq 60\}.$$

Если тип данных можно считать множеством всех возможных значений данного типа, то домен напоминает подмножество в этом множестве.

Отличие домена от понятия подмножества состоит именно в том, что *домен отражает семантику*, определенную предметной областью. Может быть несколько доменов, совпадающих как подмножества, но несущих различный смысл. Например, домены «Вес детали» и «Имеющееся количество» можно одинаково описать как множество неотрицательных целых чисел, но смысл этих доменов будет различным, и это будут *различные* домены.

Основное значение доменов состоит в том, что *домены ограничивают сравнения*. Некорректно с логической точки зрения сравнивать значения из различных доменов, даже если они имеют одинаковый тип. В этом проявляется смысловое ограничение доменов. Синтаксически правильный запрос «выдать список всех деталей, у которых вес детали больше имеющегося количества», не соответствует смыслу понятий «количество» и «вес».

Замечание. Не все домены обладают логическим условием, ограничивающим возможные значения домена. В таком случае множество возможных значений домена совпадает с множеством возможных значений типа данных.

Замечание. Не всегда очевидно, как задать логическое условие, ограничивающее возможные значения домена. Например, по-видимому, невозможно задать условие на строковый тип данных, задающий домен «Фамилия сотрудника». Ясно, что строки, являющиеся фамилиями, не должны начинаться с цифр, служебных символов, мягкого знака и т. д. Но вот является ли допустимой фамилия, состоящая из бессмысленного набора букв типа «Нпрнекккыыыов»? Очевидно, нет! Трудности такого рода возникают потому, что смысл реальных явлений далеко не всегда можно формально описать. Просто человек интуитивно понимает, что такое фамилия, но никто не может дать такое формальное определение, которое отличало бы фамилии от строк, фамилиями не являющимися. Выход из этой ситуации простой – положиться на разум сотрудника, вводящего фамилии в компьютер.

Определение 2.2. Атрибут отношения есть пара вида (*имя_атрибута*: *имя_домена*).

Имена атрибутов должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов.

Реляционное отношение (далее просто отношение), определенное на множестве доменов с именами D_1, D_2,…, D_n – это именованный объект, содержащий две части: ***заголовок*** и ***тело.***

Заголовок $\{A_1{:}D_1, \ldots A_n{:}D_n\}$ – это множество имен атрибутов или, точнее, пар вида (*имя_атрибута* : *имя_домена*). Тело – это множество ***кортежей*** $\{t_1, t_2, \ldots, t_m\}$, где *i*-й кортеж имеет вид $t_i = \{A_1{:}v_{i1}, \ldots A_n{:}v_{in}\}$, где v_{ij} – значение *j*-го атрибута A_j в *i*-м кортеже, $v_{ij} \in D_j$.

$$\left\{\begin{array}{l} \{A_1 : v_{11},\ A_2 : v_{12}, \ldots, A_n : v_{1n}\} \\ \{A_2 : v_{21},\ A_2 : v_{22}, \ldots, A_n : v_{2n}\} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \{A_m : v_{m1},\ A_2 : v_{m2}, \ldots, A_n : v_{mn}\} \end{array}\right\}$$

В дальнейшем значение атрибута A_i в кортеже t будем обозначать с помощью точечной нотации $t.A_i$.

Множество кортежей называется *телом отношения.* Тело отношения отражает состояние сущности, поэтому во времени оно постоянно меняется. Тело отношения характеризуется *кардинальным числом,* которое равно количеству содержащихся в нем кортежей.

Одной из главных характеристик отношения является его степень.

Степень отношения определяется количеством атрибутов, которое в нем присутствует. Эта характеристика отношения имеет еще названия: ранг и арность. Отношение с одним атрибутом называется унарным, с двумя атрибутами – бинарным, с тремя – тернарным, с n атрибутами n-арным.

Определение степени отношения осуществляется по заголовку отношения.

Свойства отношений:

- в теле отношения нет одинаковых кортежей;
- кортежи не упорядочены сверху вниз;
- атрибуты не упорядочены слева направо.

Подчеркнем, что все значения атрибутов – атомарные, т. е. относятся к простым типам данных в указанном выше смысле. В этом случае говорят, что отношение находится в *первой нормальной форме* (1НФ) или *нормализовано*.

Отношение удобно представлять в виде таблицы, в которой имена столбцов – это атрибуты, а строки представляют кортежи, как показано ниже.

Представление реляционного отношения

A_1	A_2	.....	A_n
v_{11}	v_{12}	.....	v_{1n}
v_{21}	v_{22}	.....	v_{2n}
.....	.....	.....	.....
v_{m1}	v_{m2}	.....	v_{mn}

В дальнейшем термины «отношение» и «таблица» будут использоваться как синонимы, хотя, строго говоря, отношение и таблица – это не совсем одно и то же. В отличие от отношения, в таблице столбцы (атрибуты) и строки (кортежи) уже идут в определенном порядке.

Определение 2.3. Реляционная база данных (РБД) – это БД, воспринимаемая пользователем как набор нормализованных отношений разной арности.

Выражение «воспринимаемая пользователем» является решающим: идея реляционной модели применяется к концептуальному (и внешнему) уровню системы, а не к внутреннему уровню.

В дальнейшем в некоторых примерах, чтобы не конкретизировать тело отношения (заполнение таблицы конкретными строками), будем использовать т. н. *схемы отношений. Схема отношения* R – это запись вида $R\ \{A_1, A_2, ..., A_n\}$., где R – имя отношения, $A_1, A_2, ..., A_n$ – имена атрибутов. Примером является следующая схема:

ПОСТАВЩИК-ТОВАР {код_пост, имя_пост, город, код_тов, назв_т, цена_1, кол_во}.

В схеме $R\ \{A_1, A_2, ..., A_n\}$ символ R играет роль *переменной отношения*. Аналогично переменным определенного типа в языках программирования переменная R может принимать некоторые значения, т. е. определенные заполнения тела отношения. Использовать понятие «переменная отношения» удобно при обсуждении разного рода теоретических вопросов, когда конкретное тело отношения не играет роли.

Почему сегодня так широко используется реляционная модель данных? Дело в том, что отношения можно рассматривать как математические объекты, а это дает возможность работы с ними (и, следовательно, с РБД) строго формализованными математическими процедурами.

2.2. Целостность базы данных: потенциальные и внешние ключи

В любой момент РБД содержит некоторую определенную конфигурацию данных, и эта конфигурация должна отражать реальную действительность (целостность данных). Следовательно, определение РБД нуждается в расширении, включающем *правила целостности данных,* назначение которых в том, чтобы информировать СУБД о разного рода ограничениях реального мира.

Есть правила целостности, которые относятся к конкретной БД, например:

– количество поставляемых деталей > 0;

– код поставщика имеет формат Пdd , где *d* – цифра;

– атрибут *город* принимает значения из определенного списка.

Это специфические для конкретных отношений правила целостности, их выполнение – задача разработчика БД. Однако есть два общих особых правила целостности, которые должны выполняться для *любой* РБД. Эти правила связаны с понятием *потенциальных* и *внешних ключей*.

2.2.1. Потенциальные ключи

Пусть $R\{A_1, ..., A_n\}$ – схема некоторого отношения. Пусть $K \subseteq \{A_1, ..., A_n\}$ – некоторое подмножество атрибутов.

Определение 2.4. Подмножество *K* называется потенциальным ключом, если оно удовлетворяет следующим свойствам:

1. Уникальность ключа – в любой момент в R нет двух различных кортежей с одинаковым значением K;

2. Неизбыточность ключа – $\forall K' \subset K$ не выполняется свойство уникальности.

Пример. В отношении *ПОСТАВЩИК_ТОВАР{код_пост, имя_пост, город, код_тов, название, цена_1, кол_во}* потенциальный ключ *K* = {*код_пост*, *код_товара*}.

Замечание. Возможны отношения, в которых единственным потенциальным ключом будет комбинация *всех* атрибутов, например, это справедливо для отношения со схемой *ПОСТАВЩИК-ТОВАР_1(код_пост, код_тов).*

Для чего нужны потенциальные ключи? Они обеспечивают основной механизм адресации на уровне кортежей. Иными словами, единственный гарантируемый системой способ точно указать на какой-либо кортеж отношения – это указать значение некоторого потенциального ключа.

Если в реляционном отношении есть несколько потенциальных ключей, то один из них указывается в качестве *первичного* ключа, остальные считаются *альтернативными* ключами.

Первое правило целостности данных (целостность сущностей) – *в любом отношении реляционной базы данных должен быть указан первичный ключ.*

В принципе при создании таблицы СУБД должна запрашивать у разработчика первичный ключ и при работе контролировать состояние таблицы после каждой ее модификации, не допуская наличия разных кортежей с одинаковым значением первичного ключа.

Далее в схеме отношения атрибуты, входящие в первичный ключ, будут выделяться подчеркиванием, например

СТУДЕНТ-ОЦЕНКА{*номер_студ_билета, фамилия, группа, дисциплина, оценка*}.

Второе правило, которое называется *правилом целостности по ссылкам*, является более сложным. Для этого потребуется ввести понятие внешних ключей.

2.2.2. Внешние ключи

Понятие внешних ключей поясним на примере реляционной базы данных со следующей схемой:

ТОВАР(код_товара, название, фирма_производитель) – справочник деталей;

ПОСТАВЩИК(код_поставщика, имя_поставщика, город) – справочник поставщиков;

ПОСТАВЩИК-ТОВАР(код_пост, код_тов, кол_во) – данные о поставках.

Рассмотрим отношение *ПОСТАВЩИК-ТОВАР*. Конкретное значение атрибута *код_пост* допустимо для отношения *ПОСТАВЩИК-ТОВАР* лишь в том случае, если такое же значение существует в качестве первичного ключа *K={код_поставщика}* в отношении *ПОСТАВЩИК*. Другими словами, не имеет смысла включать в *ПОСТАВЩИК-ТОВАР* поставку для поставщика *П07*, если в справочнике поставщиков *ПОСТАВЩИК отсутствует* поставщик с кодом *П07*. Аналогичная ситуация для деталей.

***Определение 2.5*.** Пусть $R1$ – отношение. Тогда внешний ключ $L1$ в $R1$ – это подмножество атрибутов отношения $R1$, такое что

- существует отношение $R2$ с первичным ключом $L2$;
- каждое значение $L1$ в текущем значении $R1$ совпадает со значением $L2$ некоторого кортежа в текущем значении $R2$ (обратное не требуется!).

Иначе говоря, внешний ключ – это атрибут (или подмножество атрибутов), чьи значения совпадают с имеющимися значениями первичного ключа другого отношения. В приведенном выше примере атрибут *код_пост* в отношении *ПОСТАВЩИК-ТОВАР* является внешним ключом,

связывающим это отношение с отношением *ПОСТАВЩИК,* т. к. в *ПОСТАВЩИК* атрибут *код_поставщика,* имеющий тот же смысл, что и *код_пост,* является первичным ключом.

Подобное взаимоотношение между таблицами называется *связью (relationship)*. Связь между двумя таблицами устанавливается путем присвоения значений внешнего ключа одной таблицы значениям первичного ключа другой.

Как правило, внешний ключ отношения *R* является частью потенциального ключа этого же отношения. Однако это не обязательно, например, в БД со схемой

ОТДЕЛ(номер, название, фонд_зарплаты),
СЛУЖАЩИЙ(таб_номер, ФИО, номер_отдела, зарплата, должность)

в отношении *СЛУЖАЩИЙ* атрибут *номер_отдела* является внешним ключом (относительно *ОТДЕЛ*), но он не входит ни в один потенциальный ключ этого отношения.

Второе правило целостности данных (*целостность по ссылкам*) – *БД не содержит несогласованных значений внешних ключей.*

Для отображения связей между отношениями (ссылок по внешним ключам) удобно использовать диаграммы следующего вида:

Рис. 2.1 Связь между таблицами по внешним ключам

При наличии связи $R1 \xrightarrow{L} R2$ по внешнему ключу *L* отношение *R*1 называется *главным* (master-table), а отношение *R*2 – *подчиненным* (detail-table) в данной связи. Так, на рисунке выше отношения *ПОСТАВЩИК* и *ТОВАР* – главные по отношению к подчиненному отношению *ПОСТАВЩИК-ТОВАР*.

2.2.3. Как обеспечивается ссылочная целостность

Второе правило целостности выражено исключительно в терминах *состояний* БД. Любое состояние БД, не удовлетворяющее этому правилу, некорректно. Одна из возможностей избежать некорректности – это запретить любые операции, приводящие к этому состоянию. Однако в некоторых случаях предпочтительнее допустить такую операцию, но при необходимости выполнить некоторые *компенсирующие* операции. Например, если пользователь удаляет из отношения *ПОСТАВЩИК* поставщика с

кодом ’П01’ система сама может удалить все поставки этого поставщика из отношения *ПОСТАВЩИК-ТОВАР* (т. н. *каскадное удаление*). Следовательно, у разработчика БД должна быть возможность определить, какие операции должны быть запрещены, а какие разрешены, нужны ли для разрешенных операций компенсирующие, и если да, то какие именно (язык SQL позволяет это делать).

После назначения внешнего ключа СУБД имеет возможность автоматически отслеживать вопросы «ненарушения» связей между отношениями, а именно:

- если пользователь попытается вставить в подчиненную таблицу запись, для внешнего ключа которой не существует соответствия в главной таблице (например, там нет еще записи с таким первичным ключом), СУБД сгенерирует ошибку;
- если пользователь попытается удалить из главной таблицы запись, на первичный ключ которой имеется хотя бы одна ссылка из подчиненной таблицы, СУБД также сгенерирует ошибку;
- если пользователь попытается изменить первичный ключ записи главной таблицы, на которую имеется хотя бы одна ссылка из подчиненной таблицы, СУБД также сгенерирует ошибку.

Замечание. Существуют два подхода к удалению и изменению записей в главной таблице:

- запретить удаление всех записей, а также изменение первичных ключей главной таблицы, на которые имеются ссылки подчиненной таблицы;
- распространить всякие изменения в первичном ключе главной таблицы на подчиненную таблицу, а именно:
  – если в главной таблице удалена запись, то в подчиненной таблице должны быть удалены все записи, ссылающиеся на удаляемую;
  – если в главной таблице изменен первичный ключ записи, то в подчиненной таблице должны быть изменены (автоматически) все внешние ключи записей, ссылающихся на изменяемую.

2.3. Средства манипулирования реляционными данными

Третий компонент реляционной модели (помимо понятия отношений и целостности данных) включает в себя набор операторов, которые дают возможность разработчику реляционной БД генерировать новые отношения из старых (базовых) для решения прикладных задач. Существуют два подхода к построению таких операторов – *реляционная алгебра* и *реляционное исчисление.*

В реализациях конкретных реляционных СУБД сейчас не используется в чистом виде ни реляционная алгебра, ни реляционное исчисление. Фактическим стандартом доступа к реляционным данным стал язык SQL (Structured Query Language). Язык SQL представляет собой смесь операторов

реляционной алгебры и выражений реляционного исчисления, использующий синтаксис, близкий к фразам английского языка и расширенный дополнительными возможностями, отсутствующими в реляционной алгебре и реляционном исчислении. Вообще, язык доступа к данным называется ***реляционно полным***, если он по выразительной силе не уступает реляционной алгебре (или, что то же самое, реляционному исчислению), т. е. любой оператор реляционной алгебры может быть выражен средствами этого языка. Именно таким и является язык SQL.

Первая редакция этого языка была опубликована в 1986 году и уточнена в 1989 году. Она обладала заметной неполнотой и в 1992 году была заменена новой редакцией SQL-92 (или SQL2). Вторая редакция появилась в период бурного роста числа автоматизированных систем, использующих базы данных, и явилась важным ориентиром для разработчиков СУБД. Совместимость с этим стандартом и сейчас выступает важной характеристикой той или иной системы управления базами данных.

Безусловно, расширение возможностей компьютерной техники и рост потребностей конечных пользователей порождают новые средства реализации этих потребностей со стороны систем управления базами данных. Это приводит к тому, что используемые в реальных системах языки работы с данными являются расширениями подмножеств языка SQL (кстати, то же самое происходит и с языками программирования). Критический анализ этих сокращений и расширений приводит к созданию новых редакций стандарта языка SQL. Так, за SQL-92 последовали SQL:1999 (SQL3), SQL:2003 и SQL:2008. Однако стоит заметить, что хотя реальные СУБД в ходе своего развития и приближаются к текущему стандарту, но не соответствуют ему, и для эффективного их использования все равно следует изучать документацию по конкретной СУБД. Тем не менее, знание стандарта позволяет оценить перспективы развития СУБД и избежать решений, которые не будут впоследствии соответствовать стандарту и которые придется серьезно дорабатывать.

2.3.1. Реляционная алгебра Кодда

Реляционная алгебра представляет собой набор операторов, использующих отношения в качестве аргументов и возвращающих отношения в качестве результата. Таким образом, реляционный оператор выглядит как функция с отношениями в качестве аргументов:

$$R = f(R_1, R_2, \ldots, R_n).$$

Реляционная алгебра является замкнутой, т. к. в качестве аргументов в реляционные операторы можно подставлять другие реляционные операторы, подходящие по типу:

$$R = f\left(f_1\left(R_{11}, \ldots, R_{1k_1}\right), f_2\left(R_{21}, \ldots, R_{2k_2}\right), \ldots\right).$$

Таким образом, в реляционных выражениях можно использовать вложенные выражения сколь угодно сложной структуры.

Каждое отношение обязано иметь уникальное имя в пределах базы данных. Имя отношения, полученного в результате выполнения реляционной операции, определяется в левой части равенства. Однако можно не требовать наличия имен от отношений, полученных в результате реляционных выражений, если эти отношения подставляются в качестве аргументов в другие реляционные выражения. Такие отношения будем называть *неименованными отношениями*. Неименованные отношения реально не существуют в базе данных, а только вычисляются в момент вычисления значения реляционного оператора.

Существует много подходов к определению наборов операторов и способов их интерпретации, но в принципе все они являются более или менее равносильными. Здесь будет рассмотрен классический подход, предложенный Кристофером Коддом. В этом варианте множество операторов состоит из восьми операций, составляющих две группы по четыре оператора в каждой:

- Традиционные операторы над множествами: *объединение, пересечение, вычитание* и *декартово произведение* (все они модифицированы с учетом того, что их операндами являются отношения, а не произвольные множества).
- Специальные реляционные операторы: *выборка, проекция, соединение и деление*.

Кроме того, в состав алгебры включается операция *присваивания*, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция *переименования атрибутов*, дающая возможность корректно сформировать схему результирующего отношения.

Операция присваивания имеет вид R = <выражение реляционной алгебры>, где R – переменная отношения. Операция переименования атрибута – это выражение вида R RENAME A_1, A_2 ,… AS B_1,B_2 ,…, где R – переменная отношения, A_1, A_2 ,… – имена атрибутов отношения R, B_1,B_2,… – новые имена. Это выражение приводит к получению отношения с тем же заголовком и телом, что и отношение, которое является текущим значением переменной R, за исключением того, что в нем атрибуты A_i называются теперь B_i.

Два отношения называются *совместимыми по типу*, если они имеют одинаковые заголовки.

Теоретико-множественные операторы

Объединение. Объединением двух совместимых по типу реляционных отношений A и B (A UNION B) называется отношение C с тем же заголовком, что у A и B, и с телом, состоящим из множества всех кортежей, принадлежащим A или B или обоим вместе:

$$t \in A \ UNION \ B \Leftrightarrow t \in A \ OR \ t \in B.$$

Оператор объединения поясняется на рис. 2.2 (а).

Пересечение. Пересечением двух совместимых по типу отношений *R* и *S* (синтаксис *R* INTERSECT *S*) называется отношение с тем же заголовком, что у *R* и *S*, и с телом, состоящим из множества всех кортежей, принадлежащим *R* и *S*:

$$t \in A \ INTERSECT \ B \Leftrightarrow t \in A \ AND \ t \in B.$$

Оператор пересечения поясняется на рис. 2.2 (б).

Вычитание. Вычитанием двух совместимых по типу отношений A и B (A MINUS B) называется отношение с тем же заголовком, что у A и B, и с телом, состоящим из множества всех кортежей, принадлежащим A и не принадлежащим B:

$$t \in A \ MINUS \ B \Leftrightarrow t \in A \ AND \ t \notin B.$$

Оператор вычитания поясняется на рис. 2.2 (в).

Декартово произведение. Пусть имеются два отношения *R* $\{A_1, A_2, \ldots, A_m\}$ и *S* $\{B_1, B_2, \ldots, B_n\}$. Тогда результатом операции произведения *R* TIMES *S* является отношение *T* $\{A_1, \ldots, A_m, B_1, \ldots, B_n\}$.

Оператор декартового произведения поясняется на рис. 2.2 (г).

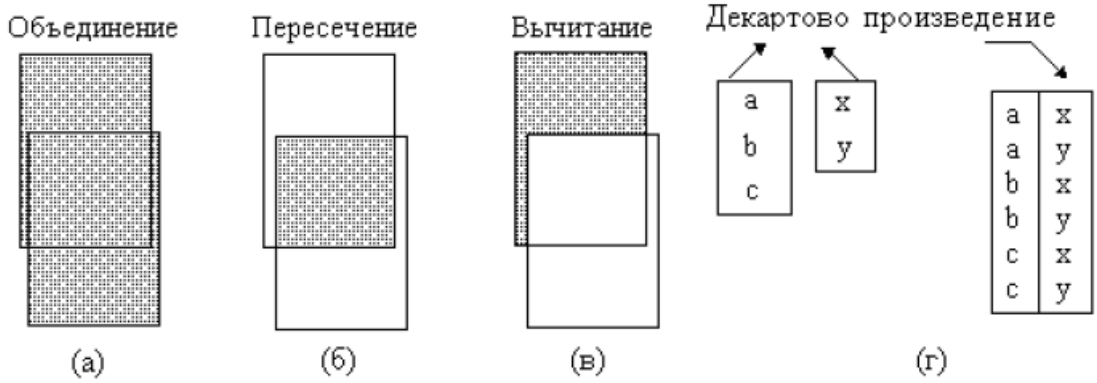


Рис. 2.2. Иллюстрация теоретико-множественных операций

Здесь может возникнуть проблема – как получить корректно сформированный заголовок отношения-результата? Поскольку заголовок результирующего отношения является сцеплением заголовков отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Эти соображения приводят к введению понятия совместимости по взятию декартова произведения. Два отношения совместимы по взятию декартова произведения в том и только том случае, если пересечение множеств имен атрибутов, взятых из их схем отношений, пусто. Любые два отношения всегда могут стать совместимыми по взятию декартова

произведения, если применить операцию переименования одноименных атрибутов.

Следует заметить, что операция взятия декартова произведения не является слишком осмысленной на практике. Во-первых, мощность тела ее результата очень велика даже при допустимых мощностях операндов, а, во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как будет показано далее, основной смысл включения операции расширенного декартова произведения в состав реляционной алгебры Кодда состоит в том, что на ее основе определяется действительно полезная операция соединения.

Специальные операторы

Выборка (θ-выборка). Пусть $\theta \in \{=, >, <, \geq, \leq, \neq\}$ – символ операции сравнения, R – отношение. Тогда θ-выборка (R WHERE $X\,\theta\,Y$) – это отношение с тем же заголовком, что и R, содержащее кортежи t из R, для которых условие $X\,\theta\,Y$ истинно. Здесь X – атрибут, а Y – атрибут, определенный на том же домене, что и X, или литерал (т. е. символьная строка или число).

Пример. Рассмотрим отношение *ПОСТАВЩИК-ТОВАР* (табл. 2.1). Тогда выражение

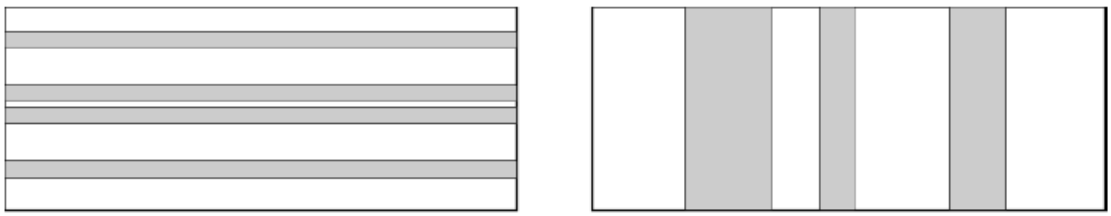
ПОСТАВЩИК-ТОВАР WHERE *назв_т* = ‘Монитор’ определяет отношение с телом вида

код_п	*имя_п*	*гор*	*код_т*	*назв_т*	*цена_1*	*кол_во*
П01	Скан	Ярославль	Т14	монитор	7 500	10
П11	Альфа	Москва	Т14	монитор	7 200	3

Обычно определение выборки расширяется до формы, в которой условие в выражении WHERE будет содержать произвольное число логических сочетаний простых условий, например,

ПОСТАВЩИК-ТОВАР WHERE (*назв_т* = ‘монитор’)
AND (*гор* = ‘Москва’ OR *гор* = ‘Воронеж’).

На интуитивном уровне оператор выборки лучше всего представлять как взятие некоторой «горизонтальной» вырезки из отношения-операнда (выборки некоторых строк из таблицы), как показано на рис. 2.3 (а).



(а) Выборка (б) Проекция

Рис. 2.3. Операции выборки и проекции

Проекция. Проекция отношения R на атрибуты $A,B,\ldots,C$ (синтаксис $R[A,B,\ldots,C]$) – это отношение с заголовком $A,B,\ldots,C$ и телом, состоящим из кортежей $\{A{:}a,\ B{:}b,\ldots,C{:}c\}$, таких, для которых в отношении R значение

атрибута *A* равно *a*, атрибута *B* равно *b*,..., атрибута *C* равно *c*. Таким образом, проекция – это подмножество кортежей, получаемое исключением тех атрибутов, которые не указаны в списке атрибутов, и последующим исключением дублирующих подкортежей. Тем самым при выполнении операции проекции выделяется «вертикальная» вырезка отношения-операнда (рис. 2.3 (б)).

Пример. Рассмотрим снова отношение *ПОСТАВЩИК-ТОВАР* (см. табл. 2.1). Выражение

ПОСТАВЩИК-ТОВАР[назв_ т, кол_во]

формирует отношение со следующими кортежами:

Назв_т	*кол_во*
Монитор	10
Принтер	16
Монитор	3
Процессор	3

Соединение. Оператор соединения (называемый также *соединением по условию, или θ-соединением*) требует наличия двух операндов – соединяемых отношений и третьего операнда – простого условия. Пусть соединяются отношения *R* и *S*. Тогда, по определению, результатом операции соединения (*R* JOIN *S*) WHERE usl совместимых по взятию декартова произведения отношений *R* и *S* является отношение, получаемое путем выполнения операции выборки по условию usl декартова произведения отношений *R* и *S*:

(*R* JOIN *S*) WHERE usl ≡ (*R* TIMES *S*) WHERE usl.

Хотя операция соединения в приведенной интерпретации не является примитивной (поскольку определяется с использованием операций декартова произведения и проекции), в силу особой практической важности она включается в базовый набор операций реляционной алгебры Кодда. Заметим также, что в практических реализациях соединение обычно не выполняется именно как выборка декартова произведения. Имеются более эффективные алгоритмы, гарантирующие получение такого же результата.

Важным частным случаем операции соединения по условию является *естественное соединение.* Операция естественного соединения применяется к паре отношений *R*{*A*,*B*} и *S*{*B*,*C*}, обладающих (возможно, составным) общим атрибутом *B* (т. е. атрибутом с одним и тем же именем и определенным на одном и том же домене). Пусть *ABC* обозначает объединение заголовков отношений A и B. Тогда естественное соединение отношений *R* и *S* (*R NATURAL JOIN S*) – это спроецированный на *ABC* результат соединения *R* и *S* по условию *R.B* = *S.B* [1]. Хотя операция

[1] Здесь *R.B* и *S.B* представляют собой так называемые **квалифицированные (уточненные)** имена атрибутов (часто такой способ именования называют точечной нотацией). Мы будем использовать подобную нотацию в тех случаях, когда требуется явно показать, схеме какого отношения принадлежит данный атрибут.

естественного соединения выражается через операции переименования, соединения по условию и проекции, для нее обычно используется сокращенная форма, называемая NATURAL JOIN.

Пример. Пусть отношения *ПОСТАВЩИК* и *ПОСТАВЩИК-ТОВАР* имеют следующее заполнение:

ПОСТАВЩИК

код_п	*имя_п*	*гор*
П01	Скан	Ярославль
П02	Альфа	Москва
П03	Тензор	Ярославль

ПОСТАВЩИК-ТОВАР

код_п	*код_т*	*цена_1*	*кол_во*
П01	Т06	23	10
П01	Т04	234	5
П03	Т06	120	6

Выражение *ПОСТАВЩИК* NATURAL JOIN *ПОСТАВЩИК-ТОВАР* создает новое отношение со следующими кортежами:

код_п	*имя_п*	*гор*	*код_т*	*цена1*	*кол_во*
П01	Скан	Ярославль	Т06	123	10
П01	Скан	Ярославль	Т04	234	5
П03	Тензор	Ярославль	Т06	120	6

Замечание. В синтаксисе естественного соединения не указываются, по каким атрибутам производится соединение. Естественное соединение производится *по всем* одинаковым атрибутам.

Замечание. Естественное соединение эквивалентно следующей последовательности реляционных операций:

– переименовать одинаковые атрибуты в отношениях,
– выполнить декартово произведение отношений,
– выполнить выборку по совпадающим значениям атрибутов, имевших одинаковые имена,
– выполнить проекцию, удалив повторяющиеся атрибуты,
– переименовать атрибуты, вернув им первоначальные имена.

Операция деления отношений. Эта операция наименее очевидна из всех операций реляционной алгебры Кодда и поэтому нуждается в более подробном объяснении. Пусть заданы два отношения – $R\{A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m\}$ (делимое) и $S\{B_1, B_2, \ldots, B_m\}$ (делитель). Будем считать, что атрибут B_i отношения R и атрибут B_i отношения S ($i = 1, \ldots, m$) не только обладают одним и тем же именем, но и определены на одном и том же домене.

По определению, результатом деления R на S (R DIVIDE BY S) является отношение $T\{A_1, A_2, \ldots, A_n\}$ (частное), которое состоит из кортежей $t = \{a_1, a_2, \ldots, a_n\}$ таких, что для *всех* кортежей $\{b_1, b_2, \ldots, b_m\} \in S$ в отношении R найдется кортеж $\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\}$. Другими словами, тело декартова произведения T TIMES S содержится в теле отношения R . Поясним это определение на простейшем примере.

Пример. Рассмотрим отношения следующего вида:

R

A	*B*
a1	b1
a1	b2
a2	b1

S

B
b1
b2

Тогда команда *T* = *R* DIVIDE BY *S* сформирует отношение *T* , тело которого содержит один кортеж

A
a1

Ситуацию, в которой удобно использовать операцию реляционного деления, проиллюстрируем на более содержательном примере. Типичные запросы, реализуемые с помощью операции деления, обычно в своей формулировке имеют слово «все» – «Какие поставщики поставляют *все* детали?», «Какие сотрудники работают во *всех* проектах?» и т. п. Пусть в БД некоторой организации поддерживаются два отношения: *ПРОЕКТЫ*{*проект, объем_финанс*}, представляющее справочник всех проектов, выполняемых в организации, и *ПРОЕКТЫ-СОТРУДНИКИ{проект, сотр_имя, отд_номер* }, где для каждого проекта указаны все задействованные в этом проекте сотрудники и отделы, в которых они работают. Требуется составить список всех тех сотрудников, которые работают ***во всех*** проектах. Требуемый список может быть получен в виде отношения *СОТРУД-ВСЕ-ПРОЕКТЫ*{*имя_сотруд, отдел*} следующим образом:

СОТРУД-ВСЕ-ПРОЕКТЫ = ПРОЕКТ-СОТРУДНИК
DIVIDE BY *ПРОЕКТЫ[проект]*.

Этот результат поясняется ниже на конкретных значениях переменных отношений *ПРОЕКТЫ* и *ПРОЕКТ-СОТРУДНИК*.

Отметим, что операция реляционного деления не является примитивной – она выражается через операции декартова произведения, взятия разности и проекции.

Основная цель создания реляционной алгебры – возможность формирования запросов к базе данных на формальном языке, операторами которого являются введенные операторы алгебры Кодда.

ПРОЕКТ-СОТРУДНИК-ОТДЕЛ

проект	***сотр_имя***	***отд_номер***
Пр1	Меньшиков О.Е.	1
Пр1	Домогаров А.Ю.	2
Пр1	Безруков С.В.	2
Пр2	Меньшиков О.Е.	1
Пр2	Домогаров А.Ю.	2
Пр2	Безруков С.В.	2
Пр3	Домогаров А.Ю.	2
Пр3	Боярская Е.М.	2
Пр3	Безруков С.В.	2

ПРОЕКТЫ

проект	***объем_финансирования***
Пр1	***100000***
Пр2	***234000***
Пр3	150000

СОТРУД-ВСЕ-ПРОЕКТЫ

сотр_имя	***отд_номер***
Домогаров А.Ю.	2
Безруков С.В.	2

3. Нормализация базы данных

Каждый программист обычно по-своему проектирует базу данных для программы, над которой работает. У одних это получается лучше, у других – хуже. Качество спроектированной БД в немалой степени зависит от опыта и интуиции программиста, однако существуют некоторые правила, помогающие улучшить проектируемую БД. Такие правила носят *рекомендательный* характер, и называются **нормализацией** базы данных.

Процесс нормализации данных заключается в устранении избыточности данных в таблицах.

Существует несколько *нормальных форм*, но для практических целей интерес представляют только первые три *нормальные формы*.

Первая нормальная форма (1НФ) требует, чтобы каждое поле таблицы БД было неделимым (атомарным) и не содержало повторяющихся групп.

Неделимость означает, что в таблице не должно быть полей, которые можно разбить на более мелкие поля. Например, если в одном поле мы объединим фамилию студента и группу, в которой он учится, требование неделимости соблюдаться не будет. Первая нормальная форма требует, чтобы мы разбили эти данные по двум полям.

Под понятием *повторяющиеся группы* подразумевают поля, содержащие одинаковые по смыслу значения. Взгляните на рисунок:

№	Студент 1	Студент 2	Студент 3
1	Иванов И.И.	Петров П.П.	Сидоров С.С.

Повторяющиеся группы

Верно, такую таблицу можно сделать, однако она нарушает правило *первой нормальной формы*. Поля «Студент 1», «Студент 2» и «Студент 3» содержат одинаковые по смыслу объекты, их требуется поместить в одно поле «Студент». Ведь в группе не бывает по три студента, правда? Их гораздо больше. И представляете, как будет выглядеть таблица, содержащая данные на тридцать студентов? Это тридцать одинаковых полей!

В этом случае таблицу следует преобразовать к виду, представленному на рисунке ниже.

№	Студент

Преобразованная структура таблицы

В приведенной выше таблице поля описывают студентов в формате «Фамилия И.О.». Однако если оператор будет вводить эти описания в формате «Фамилия Имя Отчество», то нарушается также правило неделимости. В этом случае каждое такое поле следует разбить на три отдельных поля, так как поиск может вестись не только по фамилии, но и по имени или по отчеству.

№	Фамилия	Имя	Отчество

Вторая нормальная форма (2НФ) требует, чтобы таблица удовлетворяла всем требованиям первой нормальной формы, и чтобы любое неключевое поле однозначно идентифицировалось полным набором ключевых полей. Рассмотрим пример: некоторые студенты посещают спортивные платные секции, и вуз взял на себя оплату этих секций. Взгляните на рисунок:

№ студента	Секция	Плата
100	Плавание	100
110	Скейтборд	150
112	Теннис	175
254	Плавание	100
260	Теннис	175

Нарушение второй нормальной формы

В чем здесь нарушение? Ключом этой таблицы служат поля «№ студента» – «Секция». Однако данная таблица также содержит отношение «Секция» – «Плата». Если мы удалим запись студента № 110, то потеряем данные о стоимости секции по скейтборду. А после этого мы не сможем ввести информацию об этой секции, пока в нее не запишется хотя бы один студент. Говорят, что такое отношение подвержено как аномалии удаления, так и аномалии вставки. В соответствие с требованиями *второй нормальной формы*, каждое неключевое поле должно однозначно зависеть от ключа. Поле «Плата» в приведенном примере содержит сведения о стоимости данной секции, и ни коим образом не зависит от ключа – номера студента. Таким образом, чтобы удовлетворить требованию *второй нормальной формы*,

данную таблицу следует разбить на две таблицы, каждая из которых зависит от своего ключа:

№ студента	Секция
100	Плавание
110	Скейтборд
112	Теннис
254	Плавание
260	Теннис

Ключ: № студента

Секция	Плата
Плавание	100
Скейтборд	150
Теннис	175

Ключ: Секция

Правильная вторая нормальная форма

Мы получили две таблицы, в каждой из которых не ключевые данные однозначно зависят от своего ключа.

Третья нормальная форма (3НФ) требует, чтобы в таблице не имелось транзитивных зависимостей между неключевыми полями, то есть, чтобы значение любого поля, не входящего в первичный ключ, не зависело от другого поля, также не входящего в первичный ключ. Допустим, в нашей студенческой базе данных есть таблица с расходами на спортивные секции:

Секция	Плата	Кол-во студентов	Общая стоимость
Плавание	100	2	200
Скейтборд	150	1	150
Теннис	175	2	350

Нарушение третьей нормальной формы

Как нетрудно заметить, ключевым полем здесь является поле «Секция». Поля «Плата» и «Кол-во студентов» зависят от ключевого поля и не зависят друг от друга. Однако поле «Общая стоимость» зависит от полей «Плата» и «Кол-во студентов», которые не являются ключевыми, следовательно, нарушается правило *третьей нормальной формы*.

Поле «Общая стоимость» в данном примере можно спокойно убрать из таблицы, ведь если потребуется вывести такие данные, нетрудно будет перемножить значения полей «Плата» и «Кол-во студентов», и создать для вывода вычисляемое поле.

Таким образом, нормализация данных подразумевает, что вы вначале проектируете свою базу данных: планируете, какие таблицы у вас будут, какие в них будут поля, какого типа и размера. Затем вы приводите каждую таблицу к первой нормальной форме. После этого приводите полученные таблицы ко второй, затем к третьей нормальной форме, после чего можете утверждать, что ваша база данных нормализована.

Однако такой подход имеет и недостатки: если вам требуется разработать программный комплекс для крупного предприятия, база данных будет довольно большой. При нормализации данных, вы можете получить сотни взаимосвязанных между собой таблиц. С увеличением числа нормализованных таблиц уменьшается восприятие программистом базы

данных в целом, то есть вы можете потерять общее представление вашей базы данных, запутаетесь в связях. Кроме того, поиск в чересчур нормализованных данных может быть замедлен. Отсюда вывод: при работе с данными большого объема ищите компромисс между требованиями нормализации и собственным общим восприятием базы данных.

4. Проектирование баз данных

4.1. Реляционные отношения между таблицами

В частном случае БД может состоять из одной таблицы. Но чаще всего реляционная БД состоит из нескольких взаимосвязанных таблиц. Организация связи между таблицами называется *связыванием.* Связи между таблицами можно устанавливать как на этапе разработки БД, так и при разработке приложения. Для связывания таблиц используются соответствующие *поля связи.* Поле связи – особое поле таблицы, которое однозначно идентифицирует запись. В подчиненной таблице для связи с главной также используется особый набор полей, называемый *внешним ключом.* Поле связи и первичный ключ должны быть индексированы, так как это ускоряет доступ к связанным записям. В общем случае поле связи и внешний ключ представляют собой индексы.

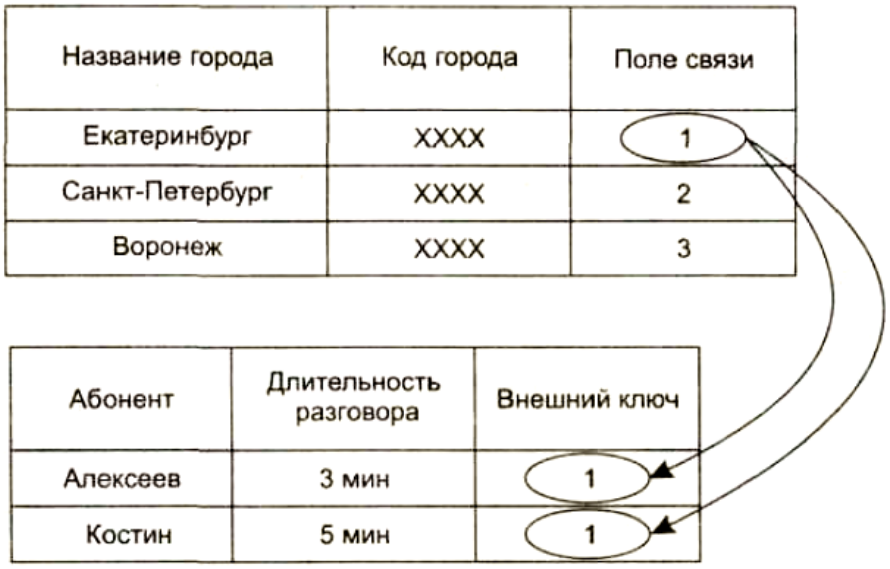


Рис. 4.1. Пример связи между таблицами

В качестве примера на рис. 4.1 приведены данные о длительности разговоров абонентов телефонной сети. «Поле связи» главной таблицы содержит некое значение. С этим значением связано поле подчиненной таблицы «внешний ключ», которое указывает на двух абонентов с фамилиями Алексеев и Костин. Использование поля связи позволяет получить данные о разговорах тех абонентов, которые звонили в Екатеринбург. В качестве поля связи можно было использовать поле «Код города», поскольку оно является уникальным.

Связь между таблицами определяет отношение подчиненности, при котором одна таблица является главной, другая подчиненной. Главная таблица обычно называется Master, подчиненная – Detail.

Различают следующие разновидности связи:

- отношение «один-к-одному»;
- отношение «один-ко-многим»;
- отношение «многие-к-одному»;
- отношение «многие-ко-многим».

Отношение *«один-к-одному»* имеет место, когда одной записи родительской таблицы соответствует одна запись в подчиненной. При этом в подчиненной таблице может содержаться, а может и не содержаться запись, соответствующая записи в главной таблице. Данное отношение обычно используется при разбиении таблицы с большим числом полей на несколько таблиц, чтобы таблица не «распухала» от второстепенной информации. В этом случае в первой таблице остаются поля с наиболее важной и часто используемой информацией, а остальные поля переносятся в другие таблицы. Пример данного отношения изображен на рис. 4.2.

Таблица «Студенты»

№	ФИО	Группа
1	Иванов И. А.	ОПИ-99–2
2	Кузнецов А. Г.	АГП-00–1
3	Перминов И. В.	ИТР-01–2

Таблица «Сдача зачетов»

№	Дата	Оценка
1	24.12.2014	5
2	18.05.2014	5
3	15.06.2013	4

Рис. 4.2. Связь «один-к-одному»

Отношение *«один-ко-многим»* подразумевает, что одной записи главной таблицы может соответствовать несколько записей в одной или нескольких подчиненных таблицах. Этот вид отношения встречается наиболее часто.

Таблица «Студенты»

№	ФИО	Группа
1	Иванов И. А.	ОПИ-99–2
2	Кузнецов А. Г.	АГП-00–1
3	Перминов И. В.	ИТР-01–2

Таблица «Сдача зачетов»

№	Дата	Предмет	Оценка
1	24.12.2004	Химия	5
2	18.05.2004	Физика	5
3	15.06.2003	Математика	4
2	18.05.2003	Математика	4
3	15.06.2003	Математика	4

Рис. 4.3. Связь «один-ко-многим»

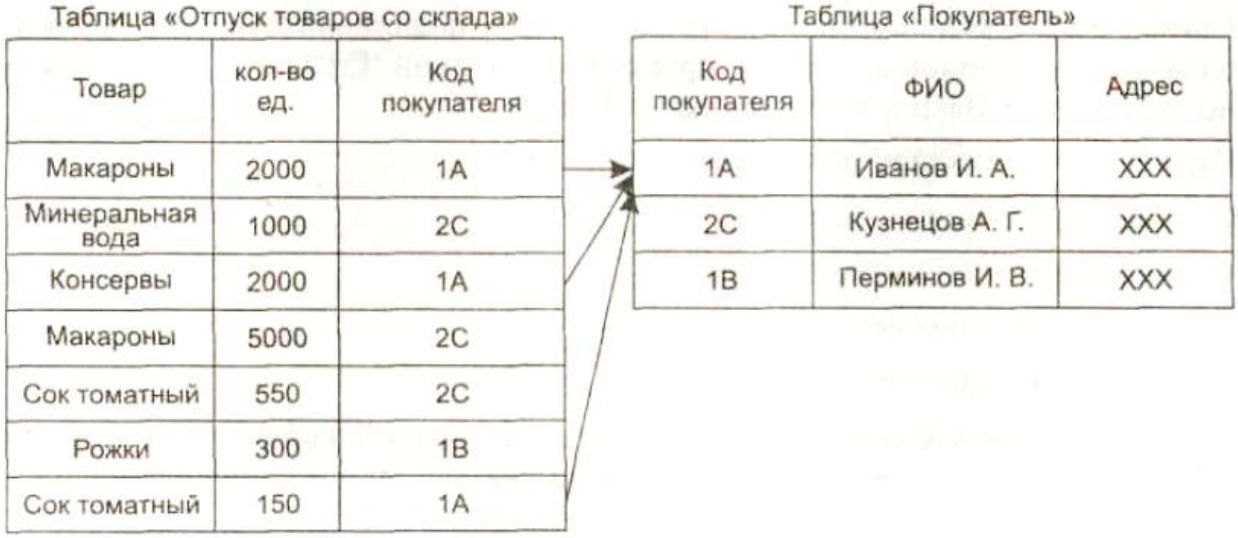


Рис. 4.4. Связь «многие-ко-многим»

Различают две разновидности связи «один-ко-многим». В первом случае для каждой записи главной таблицы должны существовать записи в подчиненной. Во втором – наличие записей в подчиненной таблице необязательно. В примере, приведенном на рис. 4.3, одному студенту в главной таблице соответствует несколько различных предметов. Связь осуществляется по полю «Номер».

Отношение *«многие-ко-многим»* имеет место, когда одной записи главной таблицы соответствует несколько записей подчиненной, а одной записи из подчиненной таблицы может соответствовать несколько записей из главной.

Как видно из рис. 4.4, один вид товара могут купить несколько покупателей, в то же время один покупатель может купить несколько товаров.

Несмотря на то, что многие СУБД не поддерживают данный вид отношения, его можно реализовать неявным способом, если возникнет такая необходимость.

4.2. Ссылочная целостность

На рис. 4.5 представлена таблица, находящаяся в отношении «один-ко-многим». Связь производиться по полю «Номер».

Рис. 4.5. Связанные таблицы

Потеря связей между записями может произойти в нескольких случаях:

- Если будет изменено значение поля связи в родительской таблице без изменения значений в полях дочерней таблицы.
- Если будет изменено значение поля (полей) связи в дочерней таблице без изменения соответствующего значения в родительской таблице.

Рассмотрим простой случай. Неожиданно у одного из клиентов картинной галереи сменился личный номер. Таким образом, в таблице «Покупатель» у Иванова оказался номер 11.

Так как у Иванова А. Г. номер 11, а заказанные им товары находятся под номером 1, то налицо нарушение целостности и достоверности данных. В новом, измененном варианте, Иванов ничего не заказал, а в таблице «Проданные картины» появились «бесхозные данные».

Рассмотрим второй вариант, изображенный на рис. 4.7. Данные в поле «Номер» были изменены на другие. Таким образом, в полях, имеющих прежнее значение «1», данные стали иметь значение «6». В этой ситуации тоже теряются нужные данные.

Таблица «Покупатель»

ФИО	Номер
Иванов А. Г.	11
Илюшин Ж. Е.	2
Коровьев Ф. Н.	3

Таблица «Проданные картины»

Название	Художник	Цена	Дата	Номер
Облака	Юров В. А.	1200	11.10.2004	1
Лес	Комаров А. Я.	2000	09.06.2004	1
Мороз	Ильин С. В.	5600	07.04.2004	3
Шторм	Юров В. А.	12000	11.12.2004	2
Сияние	Егоров М. Е.	0,98	18.07.2004	1

Рис. 4.6. Нарушение целостности БД

Таблица «Покупатель»

ФИО	Номер
Иванов А. Г.	1
Илюшин Ж. Е.	2
Коровьев Ф. Н.	3

Таблица «Проданные картины»

Название	Художник	Цена	Дата	Номер
Облака	Юров В. А.	1200	11.10.2004	6
Лес	Комаров А. Я.	2000	09.06.2004	6
Мороз	Ильин С. В.	5600	07.04.2004	3
Шторм	Юров В. А.	12000	11.12.2004	2
Сияние	Егоров М. Е.	0,98	18.07.2004	6

Рис. 4.7. Нарушение целостности БД из дочерней таблицы

В обоих примерах имело место нарушение целостности БД, то есть информация, хранящаяся в БД, стала недостоверной из-за искажения связей.

Нарушение ссылочной целостности может возникнуть в нескольких случаях:

- удаление записи из главной таблицы, без удаления связанных записей в дочерней таблице;
- изменение значения поля связи главной таблицы, без изменения ключа дочерней;
- изменение ключа дочерней таблицы без изменения поля связи главной.

Для предотвращения потери ссылочной целостности, используется *механизм каскадных изменений.* Суть его довольно проста:

- При изменении значения поля связи в главной таблице должен быть синхронно изменен ключ, то есть его значение.
- При удалении записи в родительской таблице обязательно следует удалить все связанные записи.

Ограничения на изменение полей связи и их каскадное удаление могут быть наложены на таблицы при их создании. Эти ограничения обычно хранятся в системных таблицах наряду с индексами, триггерами и хранимыми процедурами. В некоторых случаях забота о сохранении ссылочной целостности ложиться на плечи разработчика.

4.3. Понятие транзакции

Транзакция – это последовательность действий с базой данных, в которой либо все действия выполняются успешно, либо не выполняется ни одно из них. Для того чтобы наглядно продемонстрировать суть транзакции, стоит рассмотреть простой пример. На склад пришла новая партия какого-либо товара. Необходимо принять его и занести информацию о нем в базу данных. Возникает некая цепочка действий:

- Увеличить количество единиц товара на складе.
- Ввести дату поступления новой партии.
- Ввести номер площадки, где будет храниться новая партия товара.

Предположим, что на последнем шаге произошла какая-то ошибка. Товар был зарегистрирован, его количество было увеличено, но место его расположения на складе потеряно. Такая ситуация недопустима. Транзакция должна выполняться полностью. Только тогда изменения сохранятся в базе данных. В противном случае, если один из операторов транзакции по какой-либо причине не был выполнен, измененные данные в базе данных не сохраняются, а транзакция отменяется.

Транзакция может быть неявной и явной. *Неявная* транзакция стартует автоматически, а по завершении также автоматически подтверждается или отменяется. Явной транзакцией управляет программист, используя для этого средства языка SQL.

5. Технологии доступа к данным

Технологией доступа к данным называется система интерфейсов, обеспечивающая взаимодействие между приложением и базой данных. Во многих системах управления базами данных имеются библиотеки, содержащие интерфейсы прикладного программирования *(Application Programming Interface – API),* представляющие собой функции, при помощи которых можно выполнять с данными те или иные действия.

Для того чтобы наиболее полно использовать возможности того или иного сервера баз данных, необходимо работать с ним напрямую, через API. Однако это означает полную зависимость приложения от того или иного сервера и сложность перехода на другую платформу, так как будет необходимо переписывать большое количество кода.

Этот вопрос призваны решить различные технологии доступа к данным. Они являются прослойкой между API конкретного сервера и приложением пользователя, предоставляя программисту простой унифицированный механизм работы с данными. На сегодняшний день существует множество технологий доступа к данным, таких как BDE, OLE, ODBC, ADO, и до сих пор разрабатываются новые, более надежные, удобные в работе и более быстродействующие технологии.

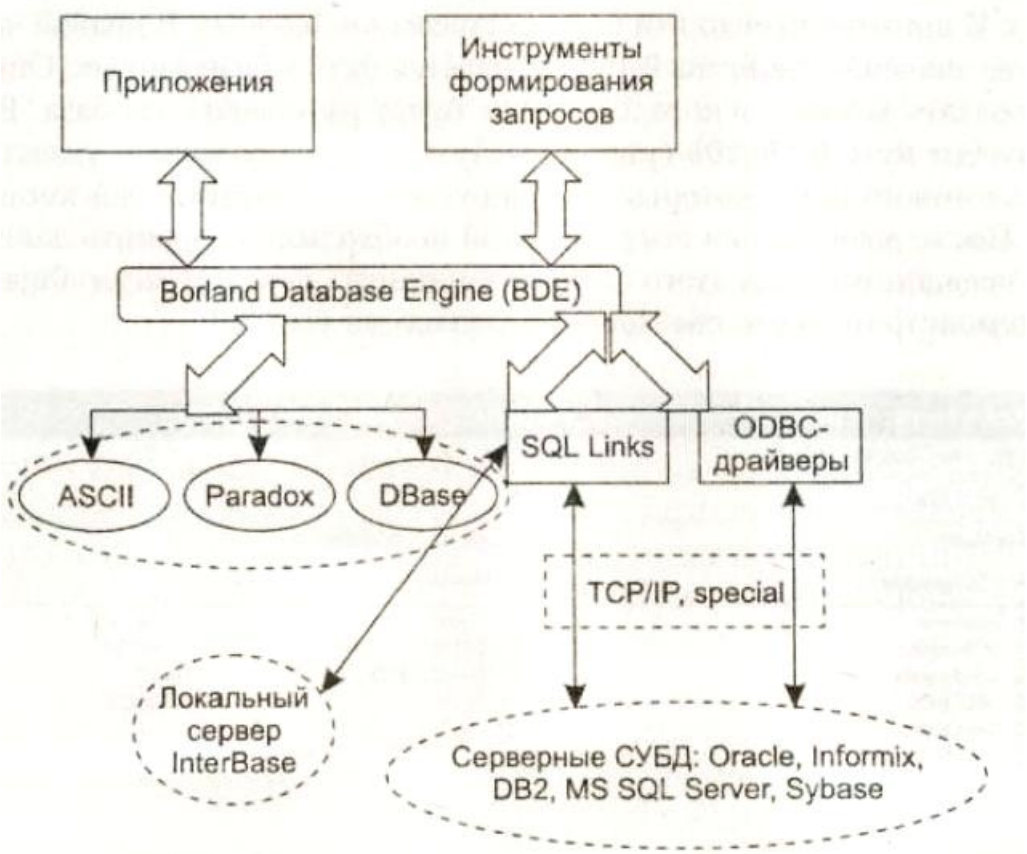
5.1. BDE

Фирма Borland разработала собственную технологию доступа к данным SQL Links, имеющую возможность взаимодействовать с ODBC через специальные «интерфейсы-мосты». Технология BDE является набором динамических библиотек, которые предоставляют интерфейсы, позволяющие передавать запросы на получение или модификацию данных из приложения в нужную базу данных и получать результат обработки. В процессе работы библиотеки используют вспомогательные файлы языковой поддержки и информацию о настройках среды.

Для разработчика BDE предоставляет множество преимуществ:

- непосредственный доступ к локальным базам данных (dBase, Paradox, текстовые файлы);
- доступ к SQL-серверам (Oracle, Sybase, MS SQL Server, InterBase, Informix, DB2) с помощью набора драйверов Borland SQL Links;
- доступ к любым источникам данных, имеющим драйвер ODBC (Open DataBase Connectivity), например к файлам электронных таблиц Excel и серверам баз данных, не имеющим драйверов SQL Links;
- создание приложений «клиент – сервер», использующих разнородные данные;
- использование SQL (Structured Query Language – язык запросов к серверным СУБД), в том числе и при работе с локальными данными;
- изоляцию приложения от средств языковой поддержки.

На рисунке ниже представлена схема, на которой показана связь приложений и BDE:



5.2. Архитектура приложений баз данных

Приложение баз данных предназначено для взаимодействия с некоторым источником данных – базой данных. Взаимодействие подразумевает получение данных, их представление в определенном формате для просмотра пользователем, редактирование в соответствии с реализованными в программе бизнес-алгоритмами и возврат обработанных данных обратно в базу данных.

В качестве источника данных могут выступать как собственно базы данных, так и обычные файлы – текстовые, электронные таблицы и т. д. Но здесь мы будем рассматривать приложения, работающие с базами данных.

Как известно, базы данных обслуживаются специальными программами – системами управления базами данных (СУБД), которые делятся на *локальные,* преимущественно однопользовательские, предназначенные для настольных приложений, и *серверные* – сетевые (часто удаленные), многопользовательские, функционирующие на выделенных компьютерах – серверах. Главный критерий такой классификации – объем базы данных и средняя нагрузка на СУБД.

Тем не менее, несмотря на разнообразие реализаций, общая архитектура приложения баз данных остается неизменной.

Само приложение включает механизм получения и отправки данных, механизм внутреннего представления данных в том или ином виде, пользовательский интерфейс для отображения и редактирования данных, бизнес-логику для обработки данных.

Механизм получения и отправки данных обеспечивает соединение с источником данных (часто опосредованно). Он должен "знать", куда ему обращаться и какой протокол обмена использовать для обеспечения двунаправленного потока данных.

Механизм внутреннего представления данных является ядром приложения баз данных. Он обеспечивает хранение полученных данных в приложении и предоставляет их по запросу других частей приложения.

Пользовательский интерфейс обеспечивает просмотр и редактирование данных, а также управление данными и приложением в целом.

Бизнес-логика приложения представляет собой набор реализованных в программе алгоритмов обработки данных.

Между приложением и собственно базой данных находится специальное программное обеспечение (ПО), связывающее программу и источник данных и управляющее процессом обмена данными. Это ПО может быть реализовано самыми разнообразными способами, в зависимости от объема базы данных, решаемых системой задач, числа пользователей, способами соединения приложения и базы данных. Промежуточное ПО может быть реализовано как окружение приложения, без которого оно вообще не будет работать, как набор драйверов и динамических библиотек, к которым обращается приложение, может быть интегрировано в само приложение. Наконец, это может быть отдельный удаленный сервер, обслуживающий тысячи приложений.

Источник данных представляет собой хранилище данных (саму базу данных) и СУБД, управляющую данными, обеспечивающую целостность и непротиворечивость данных.

В этом курсе мы подробно остановимся на способах разработки приложений баз данных в Delphi. При разнообразии способов реализации и обилии технических деталей общая архитектура приложений баз данных в Delphi следует описанной выше общей схеме.

В Delphi 7 реализовано достаточно большое число разнообразных технологий доступа к данным. Но последовательность операций при конструировании приложений баз данных остается почти одинаковой. И в работе используются по сути одни и те же компоненты, доработанные для применения с той или иной технологией доступа к данным.

Здесь рассматриваются общие подходы к разработке приложений баз данных в Delphi, базовые классы и механизмы, которые не изменятся, выберите ли вы для вашего приложения Borland Database Engine (BDE), Microsoft ActiveX Data Objects (ADO) или dbExpress.

5.3. Как работает приложение баз данных

В Репозитории Delphi отсутствует отдельный шаблон для приложения баз данных. Поэтому, как и любое другое приложение Delphi, приложение баз данных начинается с обычной формы. Безусловно, это оправданный подход, т. к. приложение баз данных имеет *пользовательский интерфейс.* И этот интерфейс создается с использованием стандартных и специализированных визуальных компонентов на обычных формах.

Визуальные компоненты отображения данных расположены на странице **Data Controls** Палитры компонентов. В большинстве они представляют собой модификации стандартных элементов управления, приспособленных для работы с набором данных*).*

Приложение может содержать произвольное число форм и использовать любой интерфейс (MDI или SDI). Обычно одна форма отвечает за выполнение группы однородных операций, объединенных общим назначением.

В основе любого приложения баз данных лежат *наборы данных,* которые представляют собой группы записей (их удобно представить в виде таблиц в памяти), переданных из базы данных в приложение для просмотра и редактирования. Каждый набор данных инкапсулирован в специальном компоненте доступа к данным. В VCL Delphi реализован набор базовых классов, поддерживающих функциональность наборов данных, и практически идентичные по составу наборы дочерних компонентов для технологий доступа к данным. Их общий предок – класс TDataSet.

Для обеспечения связи набора данных с визуальными компонентами отображения данных используется специальный компонент TDataSource. Его роль заключается в управлении потоками данных между набором данных и связанными с ним компонентами отображения данных. Этот компонент обеспечивает передачу данных в визуальные компоненты и возврат результатов редактирования в набор данных, отвечает за изменение состояния визуальных компонентов при изменении состояния набора данных, передаетсигналы управления от пользователя (визуальных компонентов) в наборданных. Компонент TDataSource расположен на странице **Data** Access Палитры компонентов.

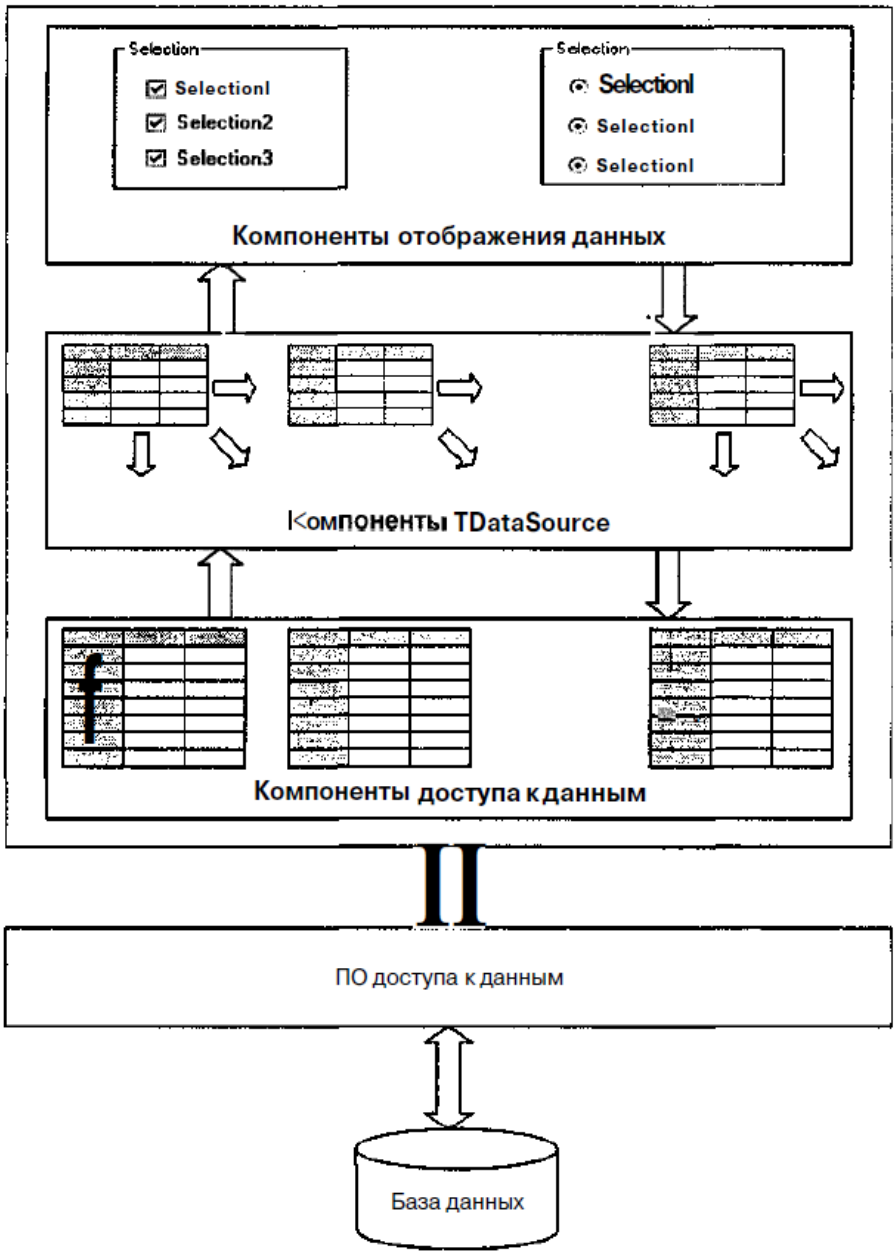
Таким образом, базовый механизм доступа к данным создается триадой компонентов:

• компоненты, инкапсулирующие набор данных (потомки класса TDataSet);

• компоненты TDataSource;

• визуальные компоненты отображения данных.

Рассмотрим схему взаимодействия этих компонентов в приложении баз данных.



В приложении с источником данных или промежуточным программным обеспечением взаимодействует компонент доступа к данным, который инкапсулирует набор данных и обращается к функциям соответствующей технологии доступа к данным для выполнения различных операций. Компонент доступа к данным представляет собой "образ" таблицы базы данных в приложении. Общее число таких компонентов в приложении не ограничено.

С каждым компонентом доступа к данным может быть связан как минимум один компонент TDataSource. В его обязанности входит соединение набора данных с визуальными компонентами отображения данных. Компонент TDataSource обеспечивает передачу в эти компоненты текущих значений полей из набора данных и возврат в него сделанных изменений.

Еще одна функция компонента TDataSource заключается в синхронизации поведения компонентов отображения данных с состоянием набора данных. Например, если набор данных не активен, то компонент

TDataSource обеспечивает удаление данных из компонентов отображения данных и их перевод в неактивное состояние. Или, если набор данных работает в режиме "только для чтения", то компонент TDataSource обязан передать в компоненты отображения данных запрещение на изменение данных.

С одним компонентом TDataSource могут быть связаны несколько визуальных компонентов отображения данных. Эти компоненты представляют собой модифицированные элементы управления, которые предназначены для показа информации из наборов данных.

При открытии набора данных компонент обеспечивает передачу в набор данных записей из требуемой таблицы БД. Курсор набора данных устанавливается на первую запись. Компонент TDataSource организует передачу в компоненты отображения данных значений необходимых полей из текущей записи. При перемещении по записям набора данных текущие значения полей в компонентах отображения данных автоматически обновляются.

Пользователь при помощи компонентов отображения данных может просматривать и редактировать данные. Измененные значения сразу же передаются из элемента управления в набор данных при помощи компонента TDataSource. Затем изменения могут быть переданы в базу данных или отменены.

Теперь, имея общее представление о работе приложения баз данных, перейдем к поэтапному рассмотрению процесса создания такого приложения

5.4. Подключение набора данных

Компонент доступа к данным является основой приложения баз данных. На основе выбранной таблицы БД он создает набор данных и позволяет эффективно управлять им. В процессе работы такой компонент тесно взаимодействует с функциями соответствующей технологии доступа к данным.

Обычно доступ к функциональности технологии доступа к данным осуществляется через совокупность интерфейсов. Все компоненты доступа к данным являются невизуальными.

Для создания нового проекта достаточно выбрать команду New Application из меню File.

Затем на форму нового проекта необходимо перенести компонент, инкапсулирующий набор данных, и выполнить следующие действия. Последовательность действий рассмотрим для компонента, инкапсулирующего функции таблицы.

1. *Подключишь компонент к базе данных.* Для этого, в зависимости от конкретной технологии, используется или специальный компонент, устанавливающий соединение, или прямое обращение к драйверу, интерфейсу или динамической библиотеке.

2. Подключить к компоненту таблицу БД. Для этого используется свойство TableName, доступное в Инспекторе объектов. После выполнения действий первого этапа в списке этого свойства должны появиться имена всех доступных в подключенной базе данных таблиц. После выбора имени таблицы в свойстве TableName компонент оказывается связанным с ней.

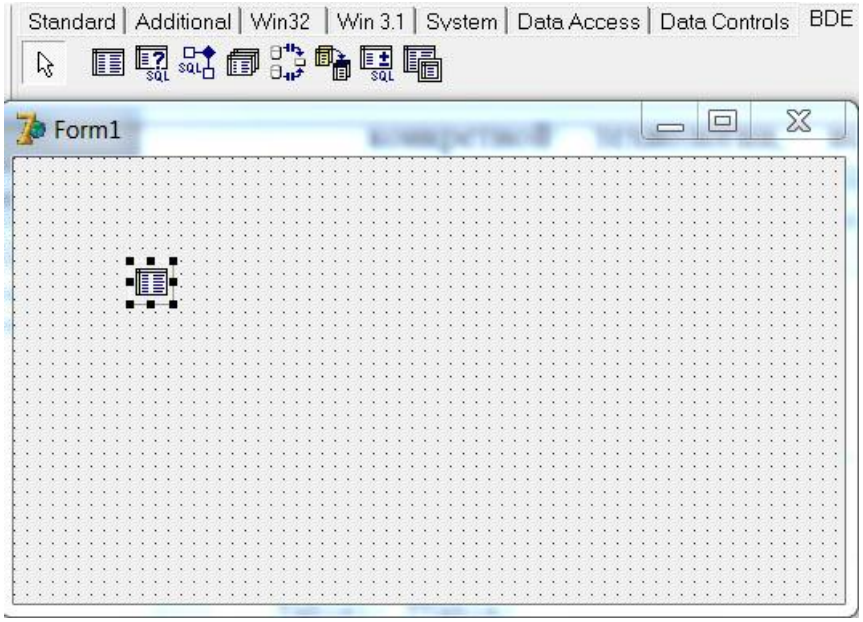
3. *Переименовать компонент.* Это не обязательное действие. Тем не менее, в любых случаях желательно присваивать компонентам доступа к данным осмысленные имена, соответствующие названиям подключенных таблиц.

Обычно название компонента копирует название таблицы (например, Orders или OrdTable или TblOrders).

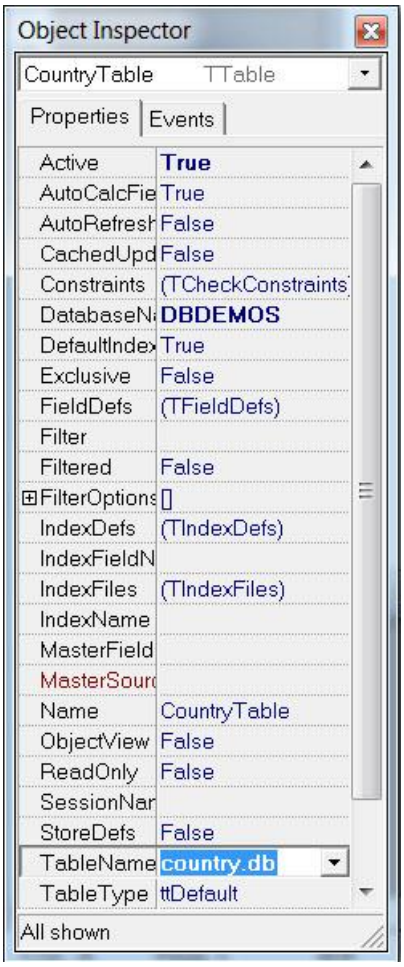
4. *Активизировать связь между компонентом и таблицей БД.* Для этого используется свойство Active. Если в Инспекторе объектов присвоить этому свойству значение True, то связь активизируется. Эту операцию можно выполнить и в исходном коде приложения. Также существует метод Open, который открывает набор данных, и метод close, закрывающий его.

В качестве примера попробуем создать простейшее приложение баз данных, работающее с таблицей COUNTRY.DB из стандартной демонстрационной базы данных DBDEMOS через драйвер процессора Borland Database Engine.

На форму нового проекта необходимо перенести компонент TTable со страницы **BDE** Палитры компонентов.



Свойство DatabaseName должно ссылаться на псевдоним DBDEMOS, который создается автоматически при установке Delphi, его можно выбрать из списка свойства DatabaseName. Для свойства TableName необходимо задать имя таблицы "COUNTRY.DB". Его также можно выбрать из списка. Двойной щелчок на свойстве Active в Инспекторе объектов присваивает ему значение True. После этого связь компонента с таблицей активизируется. Свойство Name имеет значение "CountryTable".



Открытие и закрытие набора данных можно предусмотреть как реакцию на действия пользователя или возникновение события. Чаще всего набор данных должен открываться при первом показе формы и закрываться при ее закрытии.

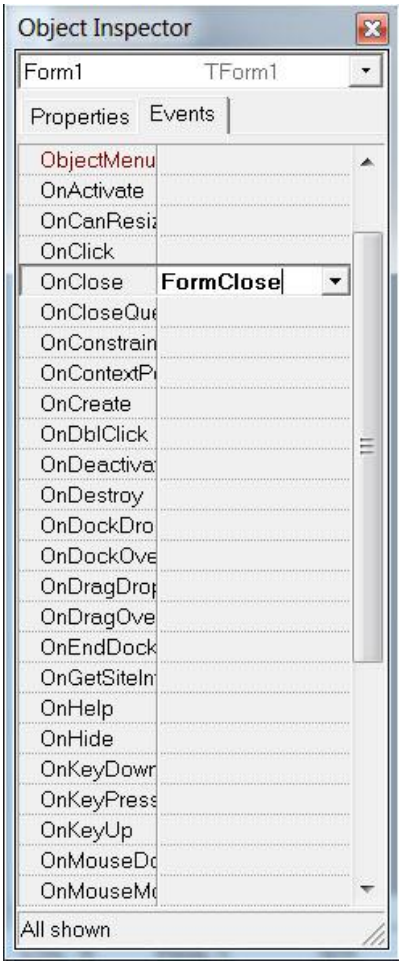
```
implementation

{$R *.dfm}

procedure TForm1.FormShow(Sender: TObject);
begin
  try
    CountryTable.Open;
  except
    ShowMessage('Table open error') ;
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  CountryTable.Close;
end;

end.
```

При открытии формы выполняется метод обработчик FormShow. В нем набор данных открывается при помощи метода Open. Обратите внимание на

использование конструкции try .. except, которая обеспечивает корректное завершение при возникновении исключительных ситуаций.

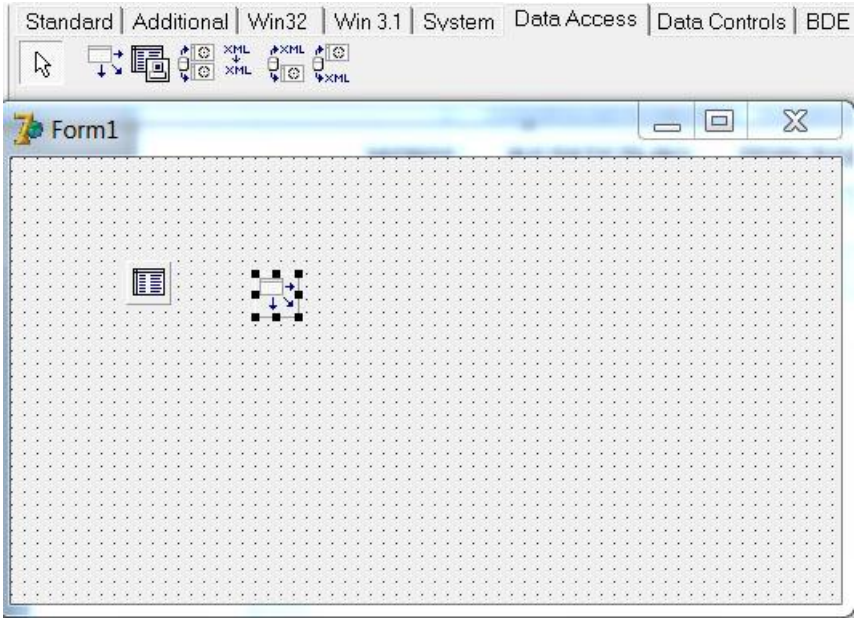
Так как ошибки в работе приложений баз данных могут привести к серьезным последствиям (потеря или искажение данных), то защитный код должен присутствовать во всех критических местах.

В методе-обработчике FormClose, который вызывается при закрытии формы, набор данных закрывается методом Close.



5.5. Настройка компонента *TDataSource*

На втором этапе разработки приложения баз данных необходимо перенести на форму и настроить компонент TDataSource (вкладка Data Access). Он обеспечивает взаимодействие набора данных с компонентами отображения данных. Чаще всего одному набору данных соответствует один компонент TDataSource, хотя их может быть несколько.



Для настройки свойств компонента необходимо выполнить следующие действия.

1. *Связать набор данных и компонент TDataSource.* Для этого используется свойство DataSet компонента TDataSource, доступное через Инспектор объектов. Это указатель на экземпляр компонента доступа к данным. В списке этого свойства в Инспекторе объектов перечислены все доступные компоненты наборов данных.

2. Переименовать компонент. Это не обязательное действие. Тем не менее желательно присваивать компонентам осмысленные имена, соответствующие названиям связанных наборов данных. Обычно название компонента комбинирует имя набора данных (например ordsource или dsOrders).

В приложении DemoDBApp компонент CountrySource связан с компонентом CountryTable. Поэтому свойство DataSet имеет значение CountryTable.

Примечание

Компонент TDataSource можно подключить не только к набору данных из той же формы, но и любой другой, модуль которой указан в секции **uses.**

Компонент TDataSource имеет ряд полезных свойств и методов.

Итак, связывание с компонентом набора данных выполняет свойство

```
property DataSet: TDataSet;
```

а определить текущее состояние набора данных можно, используя свойство

```
type TDataSetState = (dslnactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);
property State: TDataSetState;
```

При помощи свойства

```
property Enabled: Boolean;
```

можно включить или отключить все связанные визуальные компоненты. При значении False ни один связанный компонент отображения данных не будет работать.

Свойство

```
property AutoEdit: Boolean;
```

при значении True всегда будет переводить набор данных в режим редактирования при получении фокуса одним из связанных визуальных компонентов.

Аналогично, метод

```
procedure Edit;
```

переводит связанный набор данных в режим редактирования.

Метод

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

возвращает значение True, если компонент, указанный в параметре DataSet, действительно связан с данным компонентом TDatasource.

Метод-обработчик

```
type TDataChangeEvent = procedure(Sender: TObject; Field: TField) of object;
property OnDataChange: TDataChangeEvent;
```

вызывается при редактировании данных в одном из связанных визуальных компонентов.

Метод-обработчик

```
property OnUpdateData: TNotifyEvent;
```

вызывается перед сохранением изменений в базе данных.

Метод-обработчик

```
property OnStateChange: TNotifyEvent;
```

вызывается при изменении состояния связанного набора данных

5.6. Отображение данных

На третьем этапе создания приложения баз данных необходимо разработать пользовательский интерфейс на основе компонентов отображения данных.

Эти компоненты предназначены специально для решения задач просмотра и редактирования данных. Внешне большинство этих компонентов ничем не отличаются от стандартных элементов управления. Более того, многие из компонентов отображения данных являются наследниками стандартных компонентов – элементов управления.

Компоненты отображения данных должны быть связаны с компонентом TDataSource и через него с компонентом набора данных. Для этого используется их свойство DataSource. Оно присутствует во всех компонентах отображения данных.

Большинство компонентов предназначены для представления данных из одного единственного поля. В таких компонентах имеется еще одно свойство DataField, которое определяет поле связанного набора данных, отображаемое в компоненте.

Особое значение для приложений баз данных играет компонент TDBGrid, который представляет данные в виде таблицы. В столбцах таблицы размещаются поля набора данных, а в строках – записи. Для этого компонента не имеет смысла определять конкретное поле, но можно задать настраиваемый набор колонок, а для каждой из них определить поле набора данных.

Таким образом, для каждого визуального компонента отображения данных необходимо выполнить следующие операции:

1. *Связать компонент отображения данных и компонент TDataSource.* Для этого используется свойство DataSource, которое должно указывать на экземпляр требуемого компонента TDataSource. Один компонент отображения данных можно связать только с одним компонентом TDataSource. Необходимый компонент можно выбрать в списке свойств в Инспекторе объектов.

2. Задать поле данных. Для этого используется свойство DataField типа TFields. В нем необходимо указать имя поля связанного набора данных. После задания свойства DataSource поле можно выбрать из списка. Этот этап применяется только для компонентов, отображающих единственное поле.

Отдельное место среди компонентов отображения данных занимает компонент TDBNavigator. Он предназначен для перемещения по записям набора данных.

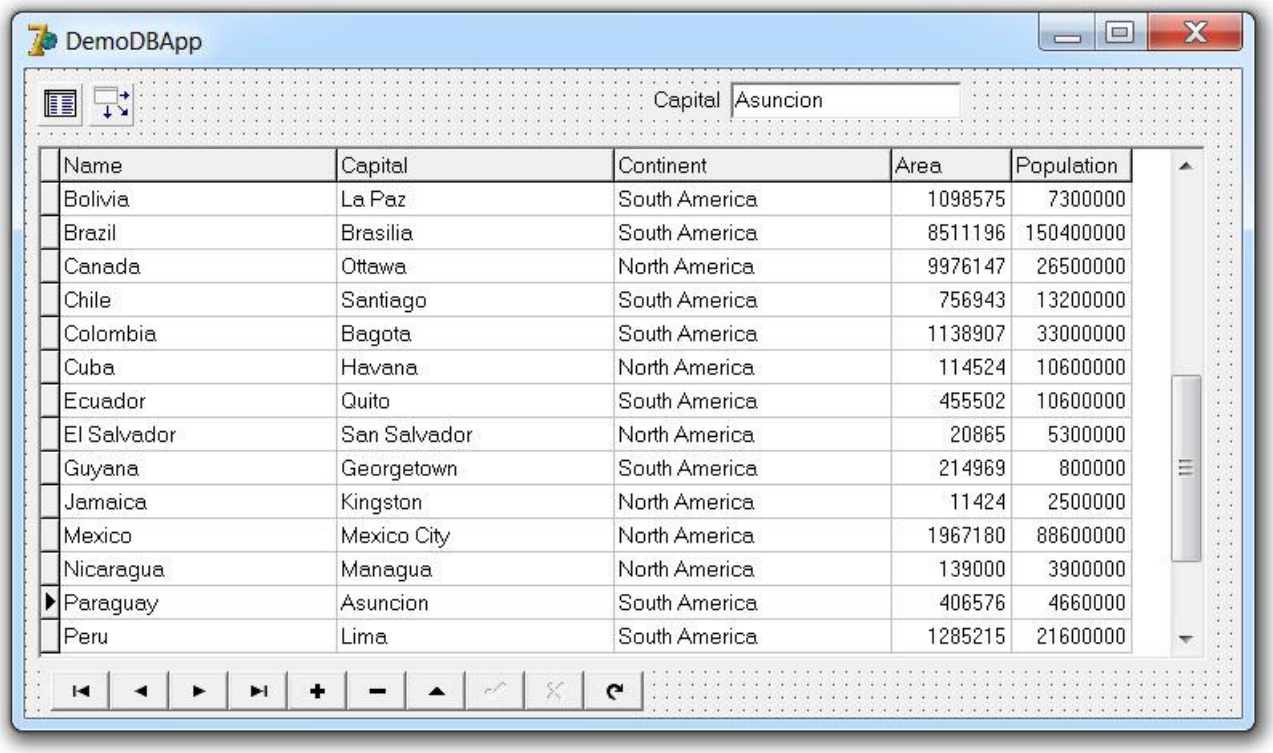
В приложении DemoDBApp использованы компоненты TDBGrid, TDBNavigator и TDBEdit.

Все три компонента отображения данных связаны с компонентом CountrySource типа TDataSource при помощи свойства DataSource.

Компонент TDBEdit отображает данные из поля Capital (столица государства) и позволяет редактировать их.

Компонент TDBGrid показывает набор данных целиком, данные в ячейках можно редактировать.

Компонент TDBNavigator позволяет перемещаться по записям набора данных CountryTable. При этом результат заметен во всех подключенных к набору данных компонентах отображения данных.



Резюме

Приложения баз данных могут получать доступ к источникам данных при помощи разнообразных технологий доступа, многие из которых используются и в приложениях Delphi. Тем не менее любое приложение баз данных в Delphi имеет стандартное ядро, структура которого определена архитектурой приложения баз данных.

Набор базовых компонентов и способов разработки является единой основой, на которой базируются технологии доступа к данным. Это позволило унифицировать процесс разработки приложений баз данных.

В основе процесса разработки лежит триада компонентов:

- невизуальные компоненты набора данных;
- невизуальные компоненты TDataSource;
- визуальные компоненты отображения данных.

6. Набор данных

Любое приложение баз данных должно уметь выполнять как минимум две операции. *Во-первых*, иметь информацию о местонахождении базы данных, подключаться к ней и считывать имеющуюся в таблицах БД информацию. Эта функция в значительной степени зависит от реализации конкретной технологии доступа к данным.

Во-вторых, обеспечивать представление и редактирование полученных данных. Множество записей одной или нескольких таблиц, переданные в приложение в результате активизации компонента доступа к данным, будем называть *набором данных*.

Каким образом получаются наборы данных? Когда мы открываем таблицу, специальный механизм делает выборку записей в соответствии с заданными параметрами, и возвращает нам эти записи в виде таблицы. Можно сказать, что наборы данных – это прослойка между нашим приложением и реальными таблицами, хранящимися в базе данных.

Понятно, что в объектно-ориентированной среде для представления какой-либо группы записей приложение должно использовать возможности некоторого класса. Этот класс должен инкапсулировать набор данных и обладать методами для управления записями и полями.

Таким образом, сам набор данных и класс набора данных является той осью, вокруг которой вращается любая деятельность приложения баз данных.

Пользователь просматривает на экране данные – это результат использования набора данных.

Пользователь решил изменить какое-то число – он изменит содержимое ячейки набора данных.

При закрытии приложение сохраняет все изменения – это набор данных передается в базу данных для сохранения.

При этом, используя одни базовые функции для обслуживания набора данных, компоненты должны обеспечивать доступ к данным в рамках различных технологий. Поэтому не удивительно, что разработчики VCL уделили особое внимание созданию максимально эффективной иерархии классов, обеспечивающих использование наборов данных (рис. 6.1).

Класс TDataSet является базовым классом иерархии, он инкапсулирует абстрактный набор данных и реализует максимально общие методы работ с ним. В него можно передать записи из таблицы базы данных или строки из обычного текстового файла – набор данных будет функционировать одинаково хорошо.

На основе базового класса реализованы специальные компоненты VCL для различных технологий доступа к данным, которые позволяют разработчику конструировать приложения баз данных, используя одни и те же приемы и настраивая одинаковые свойства.

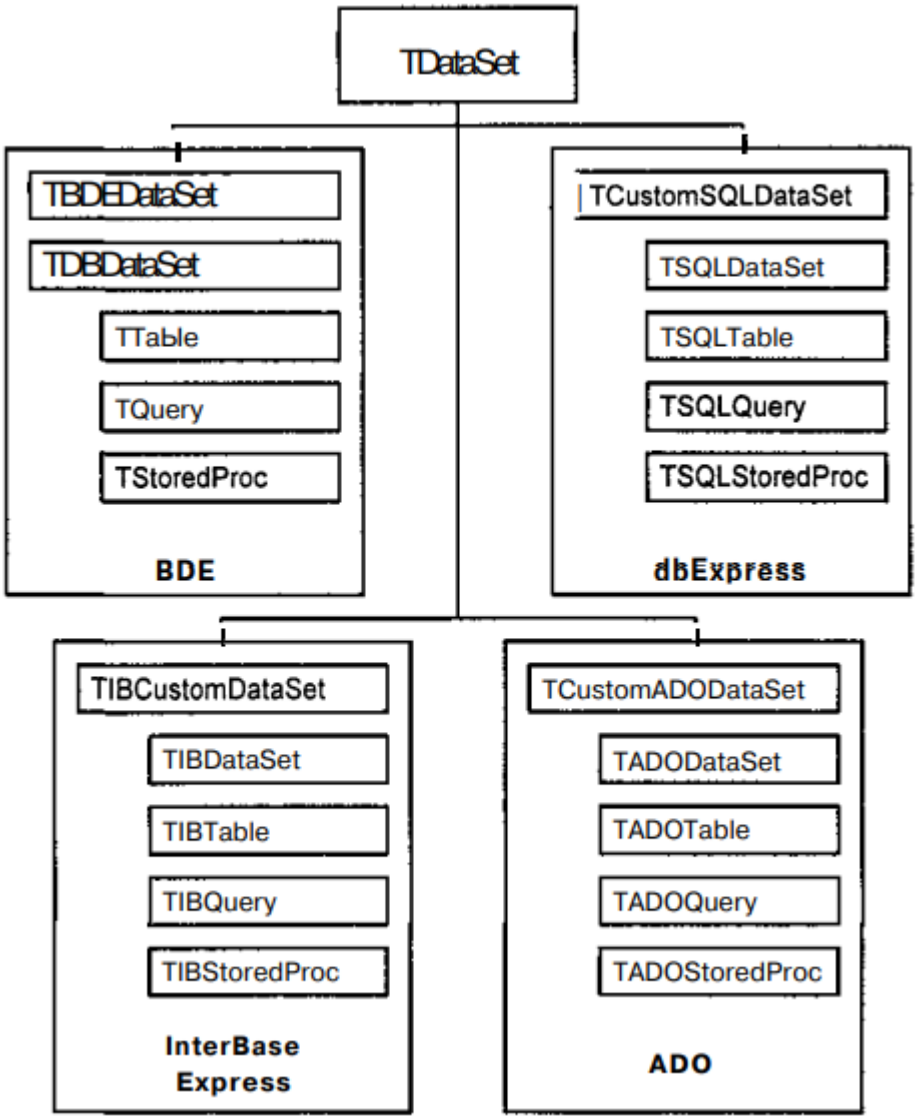


Рис. 6.1. Иерархия классов, обеспечивающих функционирование набора данных

В этой главе рассматриваются следующие вопросы:

- набор данных, инкапсулированный в классе TDataSet;
- что такое состояния набора данных;
- индексы, поля, параметры;
- прототипы компонентов для работы с таблицами, запросами и хранимыми процедурами;
- основные механизмы набора данных, реализованные в классе TDataSet.

6.1. Абстрактный набор данных

В основе иерархии классов, обеспечивающих функционирование наборов данных в приложениях баз данных Delphi, лежит класс TDataSet. Хотя он почти не содержит методов, реально обеспечивающих работоспособность основных механизмов набора данных, тем не менее, его значение трудно переоценить.

Этот класс задает структурную основу функционирования набора данных. Другими словами, это скелет набора данных, к методам которого необходимо лишь добавить требуемые вызовы соответствующих функций реальных технологий.

При решении наиболее распространенных задач программирования в процессе создания приложений баз данных класс TDataSet не нужен. Тем не менее, знание основных принципов работы набора данных всегда полезно. Кроме этого, класс TDataSet может использоваться разработчиками в качестве основы для создания собственных компонентов. Поэтому рассмотрим основные механизмы, реализованные в наборе данных.

Набор данных открывается и закрывается свойством

```
property Active: Boolean;
```

которому соответственно необходимо присвоить значение True или False. Аналогичные действия выполняют методы

```
procedure Open;
procedure Close;
```

После открытия набора данных можно перемещаться по его записям.

На одну запись вперед и назад перемещают курсор соответственно методы

```
procedure Next;
procedure Prior;
```

На первую и последнюю запись можно попасть, используя соответственно методы

```
procedure First;
procedure Last;
```

Признаком того, что достигнута последняя запись набора, является свойство

```
property Eof: Boolean;
```

которое в этом случае имеет значение True.

Аналогичную функцию для первой записи выполняет свойство

```
property Bof: Boolean;
```

Перемещение вперед и назад на заданное число записей выполняет метод

```
function MoveBy(Distance: Integer): Integer;
```

Параметр `Distance` определяет число записей. Если параметр отрицательный – перемещение осуществляется к началу набора данных, иначе – к концу.

Для ускоренного перемещения по набору данных можно отключить все связанные компоненты отображения данных. Это делается методом

```
procedure DisableControls;
```

Обратная операция выполняется методом

```
procedure EnableControls;
```

Общее число записей набора данных возвращает свойство

```
property RecordCount: Integer;
```

Однако использовать его нужно аккуратно, т. к. каждое обращение к этому свойству приводит к обновлению набора данных, что может вызвать проблемы для больших таблиц или сложных запросов. Если вам нужно определить, не является ли набор данных пустым (часто используемая операция), можно использовать метод

```
function IsEmpty: Boolean;
```

который возвращает значение True, если набор данных пуст, или уже упоминавшиеся свойства

```
if MyTable.Bof and MyTable.Eof then
   ShowMessage('DataSet is empty');
```

Номер текущей записи позволяет узнать свойство

```
property RecNo: Integer;
```

Размер записи в байтах возвращает свойство

```
property RecordSize: Word;
```

Каждая запись набора данных представляет собой совокупность значений полей таблицы. В зависимости от типа компонента и его настройки, число полей в наборе данных может изменяться. И совсем не обязательно набор данных должен содержать все поля таблицы базы данных.

Совокупность полей набора данных инкапсулирует свойство

```
property Fields: TFields;
```

а все необходимые параметры полей содержатся в свойстве

```
property FieldDefs: TFieldDefs;
```

Общее число полей набора данных возвращает свойство

```
property FieldCount: Integer;
```

а общее число полей типа BLOB содержится в свойстве

```
property BlobFieldCount: Integer;
```

Доступ к значениям полей текущей записи предоставляет свойство

```
property FieldValues[const FieldName: string]: Variant; default;
```

где в параметре `FieldName` задается имя поля.

В процессе программирования разработчик очень часто обращается к полям набора данных. Если структура полей набора данных жестко задана и не изменяется, это можно сделать так:

```
for i := 0 to MyTable.FieldCount –1 do
   MyTable.Fields[i].DiplayFormat := '#.###';
```

Иначе, если порядок следования полей и их состав меняется, можно использовать метод

```
function FieldByName(const FieldName: string): TField;
```

и делается это следующим образом:

```
MyTable.FieldByName('VENDORNO').AsInteger := 1234;
```

Имя поля, передаваемое в параметре `FieldName`, не чувствительно к регистру символов.

Метод

```
procedure GetFieldNames(List: TStrings);
```

вернет в параметр `List` полный список имен полей набора данных.

Более подробная информация о полях и способах работы с ними будет рассмотрена позже.

Класс TDataSet содержит ряд свойств и методов, которые обеспечивают редактирование набора данных.

Но сначала бывает полезно поинтересоваться, можно ли редактировать набор данных вообще. Это можно сделать при помощи свойства

```
property CanModify: Boolean;
```

которое принимает значение True для редактируемых наборов.

Перед началом редактирования набор данных нужно перевести в режим редактирования, использовав метод

```
procedure Edit;
```

Для сохранения сделанных изменений применяется метод

```
procedure Post; virtual;
```

Разработчик может вызывать его самостоятельно, или же метод `Post` вызывается самим набором данных при переходе на другую запись.

При необходимости все сделанные после последнего вызова метода `Post` изменения можно отменить методом

```
procedure Cancel; virtual;
```

Новая пустая запись добавляется в конец набора данных методом

```
procedure Append;
```

Новая пустая запись добавляется на место текущей методом

```
procedure Insert;
```

а текущая запись и все нижеследующие смещаются на одну позицию вниз.

> ***Замечание***. При использовании методов `Append` и `Insert` набор данных переходит в режим редактирования самостоятельно.

Дополнительно, у вас есть возможность добавить или вставить новую запись уже с заполненными полями. Для этого применяются методы

```
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
```

А делается это примерно так:

```
MyTab.AppendRecord([2345,'New Customer','+7(812)4569012',0,'']);
```

После вызова этих методов и их завершения набор данных автоматически возвращается в состояние просмотра.

Для существующей записи аналогичным образом можно заполнить все поля, использовав метод

```
procedure SetFields(const Values: array of const);
```

Текущая запись удаляется методом

```
procedure Delete;
```

При этом набор данных не выдает никаких предупреждений, а просто делает это.

Очистить содержимое всех полей текущей записи может метод

```
procedure ClearFields;
```

Обратите внимание, что поля становятся пустыми (NULL), а не сбрасываются в нулевое значение.

О том, редактировалась ли текущая запись, сообщает свойство

```
property Modified: Boolean;
```

если оно имеет значение True.

Набор данных можно обновить, не закрывая и не открывая его снова. Для этого применяется метод

```
procedure Refresh;
```

Однако он сработает только для таблиц и тех запросов, которые нельзя редактировать.

В каждый момент времени набор данных находится в определенном состоянии *(подробнее о состояниях будет позже).* Свойство

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit,
dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue,
dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc,
dsOpening);
property State: TDataSetState;
```

даёт информацию о текущем состоянии набора.

Методы-обработчики класса TDataSet предоставляют разработчику широчайшие возможности по отслеживанию событий, происходящих с набором данных.

По паре методов-обработчиков (до и после события) предусмотрено для следующих событий в наборе данных:

• открытие и закрытие набора данных;

• переход в режим редактирования;

• переход в режим вставки новой записи;

• сохранение сделанных изменений;

• отмена сделанных изменений;

• перемещение по записям набора данных;

• обновление набора данных.

Обратите внимание, что помимо методов-обработчиков режима вставки существует дополнительный метод

```
property OnNewRecord: TDataSetNotifyEvent;
```

который вызывается непосредственно при вставке или добавлении записи.

Дополнительно к этому могут использоваться методы-обработчики возникающих ошибок. Они предусмотрены для ошибок удаления, редактирования и сохранения изменений.

Метод-обработчик

```
property OnCalcFields: TDataSetNotifyEvent;
```

очень важен для задания значений вычисляемых полей. Он вызывается для каждой записи, которая отображается в визуальных компонентах, связанных с набором данных каждый раз, когда необходимо перерисовать значения полей в визуальных компонентах.

Если в методе-обработчике `OnCalcFields` производятся слишком сложные вычисления, частота его вызовов может быть уменьшена за счет свойства

```
property AutoCalcFields: Boolean;
```

По умолчанию оно равно значению True и расчет вычисляемых полей производится при каждой перерисовке. При значении False метод-обработчик `OnCalcFields` вызывается только при открытии, переходе в состояние редактирования и обновлении набора данных.

Все перечисленные выше обработчики имеют одинаковый тип

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

И метод-обработчик

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
            var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

вызывается для каждой записи набора данных при свойстве

```
Filtered = True.
```

Помимо перечисленных, класс TDataSet содержит еще много свойств и методов, которые обеспечивают работоспособность многих полезных в практическом программировании приложений баз данных функций.

6.2. Типы данных

В среде разработки Delphi можно создавать приложения для работы с самыми разными базами данных. Такая универсальность подразумевает

необходимость применения средств, которые бы обеспечили возможность работы со многими типами данных, используемыми в этих базах данных.

Естественно, что существует большая группа типов данных, конкретная реализация которых практически не отличается от платформы к платформе. Это, например, строки, символы, целые и вещественные числа и т. д.

Есть типы данных, которые реализованы далеко не на каждой платформе. Есть, наконец, просто уникальные типы данных.

Для удовлетворения потребностей разработчиков в Delphi применен следующий способ работы с типами данных.

Тип данных однозначно связан с конкретным полем таблицы базы данных. Без этого поля само понятие типа данных не имеет практического смысла. В Delphi свойства абстрактного поля инкапсулирует класс TField, который не имеет заранее определенного типа данных. Уже от этого класса порождено целое семейство классов для типизированных полей, каждый из которых умеет обращаться со своим типом данных.

Весь список доступных типов данных содержится в типе `TFileldType`:

```
type TFieldType = (ftUnknown, ftString, ftSmallint,
   ftlnteger, ftWord, ftBoolean, ftFloat, ftCurrency,
   ftBCD, ftDate, ftTime, ftDateTime, ftBytes, ftVarBytes,
   ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
   ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor,
   ftFixedChar, ftWideString, ftLargeint, ftADT, ftArray,
   ftReference, ftDataSet, ftOraBlob, ftOraClob,
   ftVariant, ftlnterface, ftlDispatch, ftGuid,
   ftTimeStamp, ftFMTBcd);
```

6.2.1. О типах полей в Paradox

Мы применяем обозначение типов полей в таблицах Paradox, как принято в Delphi. Хотя изначально в самой СУБД Paradox обозначения типов отличалось. Сейчас нет смысла использовать прежние обозначения типов.

Таблица прежних названий и весь набор типов полей в Paradox

Название типа в Delphi	Объём памяти	Название в Paradox	Буквенное обозначение в Paradox	Код типа в Paradox
ftString	Задаётся (1–255)	Alpha	A	$01
ftSmallInt	2	Short	S	$03
ftInteger	4	Long Integer	I	$04
ftFloat	8	Number	N	$06
ftCurrency	8	Money	$	$05
ftBoolean	1	Logical	L	$09

ftDate	4	Date	D	$02
ftTime	4	Time	T	$14
ftDateTime	8	Timestamp	@	$15
ftBCD	15	BCD	#	$17
ftAutoInc	4	Autoincrement	+	$16
ftBytes	Задаётся (1–255)	Bytes	Y	$18
ftMemo	1–240 в DB *	Memo	M	$0C
ftFmtMemo	1–240 в DB *	Formatted Memo	F	$0E
ftBLOB	1–240 в DB *	Binary	B	$0D
ftGraphic	1–240 в DB *	Graphic	G	$10
ftParadoxOLE	1–240 в DB *	OLE	O	$0F

6.3. Стандартные компоненты

Наверное, вы заметили, что на рис. 6.1 набор компонентов для каждой из представленных технологий доступа к данным примерно одинаков. Везде есть компонент, инкапсулирующий табличные функции, компонент запроса SQL и компонент хранимой процедуры. И хотя все они имеют разных ближайших предков, тем не менее, функциональность подобных компонентов в различных технологиях почти одинакова. Поэтому имеет смысл рассмотреть общие для компонентов свойства и методы, представив, что существуют некие виртуальные общие предки для таблицы, запроса и хранимой процедуры.

Примечание

Некоторые из описываемых ниже свойств и методов присутствуют не в каждой реализации компонентов.

6.3.1. Компонент таблицы *TTable*

Компонент *TTable* обеспечивает доступ к таблице базы данных целиком, создавая набор данных, структура полей которого полностью повторяет таблицу БД. За счет этого компонент прост в настройке и обладает многими

* Если размер не указан, то по умолчанию принимается 1. Причём размер в DB-файле от указанного увеличивается на 10. Всё, что не поместится в DB-файле (если реальный размер больше указанного), идёт в MB-файл.

дополнительными функциями, которые обеспечивают применение табличных индексов.

Но в практике программирования работа с таблицами целиком используется не так часто. А при работе с серверами баз данных промежуточное ПО, используемых технологий доступа к данным, все равно транслирует запрос на получение табличного набора данных в простейший запрос SQL, например:

```
SELECT * FROM Orders
```

В такой ситуации применение табличных компонентов становится менее эффективным, чем использование запросов.

Создание таблицы с помощью компонента TTable

Однако, компонент удобен для создания таблиц базы данных типа Paradox (тип dBase этим способом в Delphi создаётся с нарушением стандарта формата dBase, но если такую базу использовать только в Delphi-приложении, то проблем не возникнет). Суть данного способа заключается в предварительном создании объектов-полей в редакторе полей компонента *TTable*. Это также означает, что еще на этапе проектирования можно настроить формат объектов-полей по своему усмотрению.

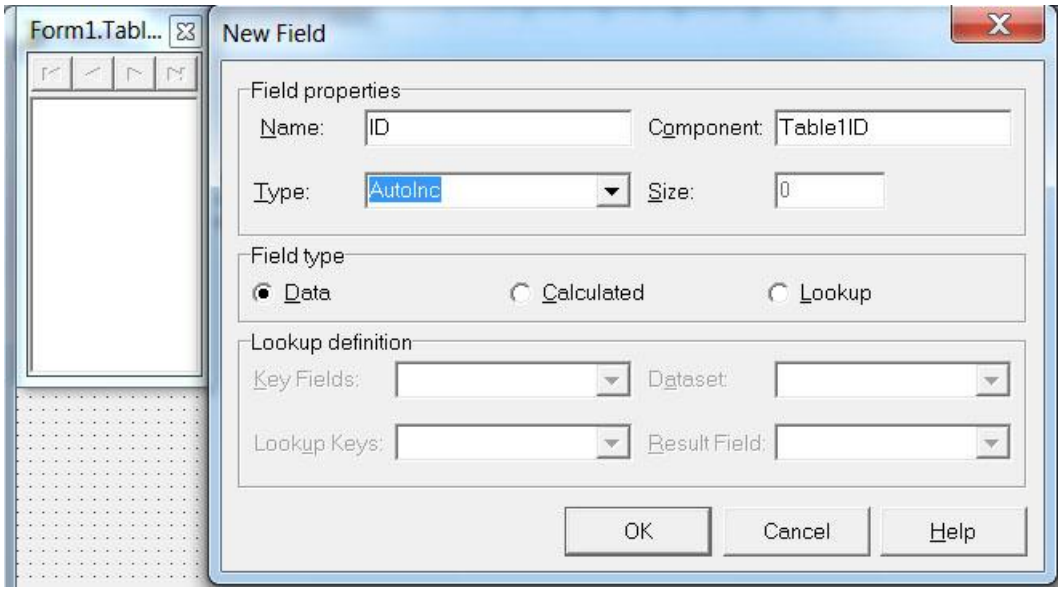
Рассмотрим этот способ на примере. Допустим, мы создали новое приложение. Разместим в нём три компонента: кнопку *Button1* для запуска процедуры, *TTable1* (с вкладки *BDE*) и *SaveDialog1 (*с вкладки *Dialogs).*

Подготовим диалоговый компонент. Присвоим его свойству *Filter* строку *Таблицы Paradox|*.db.* Таким образом, мы указали, что диалог будет работать только с таблицами типа *Paradox*. Кроме того, у компонента зададим свойство *DefaultExt = db.* Это свойство указывает расширение файла по умолчанию, если пользователь не назначит расширения сам.

Табличный компонент *TTable* имеет одно важное свойство **TableType.** Это свойство указывает на тип используемой или создаваемой таблицы. Свойство может иметь следующие значения:

- **ttASCI –** Таблица содержится в формате обычного текстового файла. Строки и поля разделяются специальными символами – разделителями. Имя файла таблицы имеет расширение *.TXT
- **ttDBase –** Таблица содержится в формате dBase, файл по умолчанию имеет расширение *.DBF
- **ttDefault –** Компонент определяет тип таблицы по расширению имени файла таблицы. При создании таблицы, если не указано расширение имени файла, принимается тип Paradox.
- **ttFoxPro –** Таблица содержится в формате FoxPro, файл по умолчанию также имеет расширение *.DBF
- **ttParadox –** Таблица содержится в формате Paradox, файл по умолчанию имеет расширение *.DB

В свойстве *TableType* компонента Table1 выберем значение *ttParadox*, то есть, таблица будет работать только с типом Paradox. Далее дважды щёлкнем по компоненту, открыв редактор полей. Нам нужно будет добавить запланированные поля таблицы. Щёлкнем по редактору правой кнопкой, выберем команду New Field (Новое поле). В поле *Name* впишем имя поля, например, ID. В поле *Type* выберем тип поля *AutoInc*. В поле *Size* нужно указывать размер поля, но это справедливо только для текстовых полей. Убедимся, что переключатель *Field type* установлен на *Data*, это создаст пустое поле указанного типа.



Нажав кнопку «ОК» добавим объект-поле в редактор полей.

Таким же образом создаём еще несколько разнотипных полей. Каждому полю присваиваем уникальное имя (ведь в таблице не может быть двух полей с одинаковым именем!). Важно, чтобы вы добавляли только те типы полей, которые поддерживаются выбранным типом таблиц, в нашем случае это Paradox (перечень типов полей для него рассмотрен выше).

Делать табличный компонент активным на этапе проектирования не нужно. Итак, не имея базы данных, не имея физической таблицы, мы заранее установили тип таблицы и нужные нам поля. Как вы, наверное, догадываетесь, мы также имеем возможность сразу настроить нужные нам форматы для каждого поля, изменяя такие его свойства, как *DisplayFormat*, *EditMask*, *DisplayLabel* и др.

Далее нам осталось непосредственно создать и открыть таблицу. Дважды щёлкнем по кнопке «Создать таблицу», сгенерировав для нее событие. В процедуру этого события впишем код:

```
//если пользователь не выбрал таблицу, выходим:
if not SaveDialog1.Execute then Exit;
//закроем таблицу, если вдруг уже есть открытая:
Table1.Close;
//вначале устанавливаем адрес базы данных – т.е. папку с БД:
Table1.DatabaseName := ExtractFilePath(SaveDialog1.FileName);
```

```
//теперь устанавливаем имя таблицы:
Table1.TableName := ExtractFileName(SaveDialog1.FileName);
//физически создаем таблицу:
Table1.CreateTable;
ShowMessage('OK');
```

Метод `CreateTable` создает новую таблицу в базе данных, используя заданное имя и описание полей из свойств `TFieldDefs`. Если таблица с таким именем уже имеется в базе данных, то она будет уничтожена и создана заново с новой структурой.

Можно создать таблицу программным путём – без использования Object Inspector. То есть, поля таблицы задать через соответствующие методы объекта *Table*. Для этого задействуем свойство

```
property FieldDefs: TFieldDefs;
```

Это свойство содержит список полей с их полным описанием. Работать с ним можно, как со списком объектов типа `TFieldDef` – пополняя этот список, указывая каждому объекту `FieldDef` его свойства: `Name`, `DataType`, `Size`.

Пример с использованием метода

```
function AddFieldDef: TFieldDef;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
   FD: TFieldDef;
begin
   Table1.TableName:= 'Primer1.db';
   Table1.FieldDefs.Clear;
   FD:= Table1.FieldDefs.AddFieldDef;
   FD.Name:= 'ID';
   FD.DataType:= ftAutoInc;
   FD:= Table1.FieldDefs.AddFieldDef;
   FD.Name:= 'Name';
   FD.DataType:= ftString;
   FD.Size:= 20;
   Table1.CreateTable;
   ShowMessage('OK');
end;
```

Используя метод

```
procedure Add(const Name: string; DataType: TFieldType;
      [Size: Integer=0]; [Required: Boolean = False]);
```

можно сделать то же самое короче – например, фрагмент

```
FD:= Table1.FieldDefs.AddFieldDef;
FD.Name:= 'Name';
FD.DataType:= ftString;
FD.Size:= 20;
```

заменить на строку

```
Table1.FieldDefs.Add('Name', ftString, 20)
```

Использование индексов в компоненте Table

Продолжаем рассматривать компонент *Table* – теперь уже с готовой таблицей (файл физически существует).

После соединения с источником данных (обычно это путь к папке с базой данных) необходимо задать имя таблицы в свойстве

```
property TableName: String;
```

В свойстве `TableType можно` дополнительно задать тип таблицы.

Если соединение с источником данных настроено правильно, имя таблицы можно выбрать из выпадающего списка свойства `TableName`.

Преимуществом табличного компонента является использование индексов, которые ускоряют работу с таблицей. Все индексы, созданные в базе данных для таблицы, автоматически загружаются в компонент. Их параметры доступны через свойство

```
property IndexDefs: TIndexDefs;
```

Подробно класс `TIndexDefs` будет рассмотрен ниже.

При работе с компонентом разработчик имеет возможность управлять индексами. Существующий индекс можно выбрать в Инспекторе объектов в списке свойств

```
property IndexName: String;
```

или использовать свойство

```
property IndexFieldNames: String;
```

в котором можно задать произвольное сочетание имен индексированных полей таблицы. Имена полей разделяются символом точкой с запятой. Таким образом, при помощи свойства `IndexFieldNames` можно создавать составные индексы.

Свойства `IndexName` и `IndexFieldNames` нельзя использовать одновременно.

Число полей, используемых в текущем индексе табличного компонента, возвращает свойство

```
property IndexFieldCount: Integer;
```

а свойство

```
property IndexFields: [Index: Integer]: TField;
```

представляет собой индексированный список полей, входящих в текущий индекс:

```
for i := 0 to MyTable.IndexFieldCount – 1 do
    MyTable.IndexFields[i].Enabled := False;
```

Для выполнения операций с таблицами и индексами целиком в табличных компонентах реализовано несколько методов.

Метод

```
procedure EmptyTable;
```

удаляет из набора данных и таблицы базы данных все записи.

Метод

```
procedure DeleteTable;
```

уничтожает таблицу базы данных, связанную с компонентом. Набор данных должен быть закрыт.

Метод

```
type
   TIndexOption = (ixPrimary, ixUnique, ixDescending,
     ixCaseInsensitive,ixExpression, ixNonMaintained);
TIndexOptions = set of TIndexOption;
procedure AddIndex(const Name, Fields: String; Options:
      TIndexOptions, const DescFields: String='');
```

добавляет к таблице БД новый индекс. Параметр `Name` задает имя индекса.

В параметре `Fields` через точку с запятой определяются имена полей, входящих в индекс; Параметр `DescFields` задает описание индекса из констант, объявленных в типе `TIndexOption`.

Метод

```
procedure DeleteIndex(const Name: string);
```

уничтожает индекс.

Кроме этого, табличные компоненты содержат и другие свойства и методы, которые будут описаны позже.

6.3.2. Компонент запроса *TQuery*

Компонент запроса предназначен для создания запроса SQL, подготовки его параметров, передачи запроса на сервер БД и представления результата запроса в наборе данных. При этом набор данных может быть редактируемым или нет.

Любой компонент запроса, каждая строка набора данных которого однозначно связывается с одной строкой таблицы БД, может редактироваться.

Например, запрос

```
SELECT * FROM Country
```

редактировать можно. Если же приведенное правило не выполняется, то набор данных можно использовать только для просмотра, и, конечно, возможности компонентов здесь ни при чем. Куда, к примеру, записывать результаты редактирования записей следующего запроса:

```
SELECT CustNo, SUM(AmountPaid)
FROM Orders
GROUP BY CustNo
```

Ведь в таком запросе каждая запись есть результат суммирования неизвестного заранее числа других записей.

Тем не менее компоненты запросов предоставляют разработчику мощный и гибкий механизм работы с данными. С помощью компонентов запросов можно решать гораздо более сложные задачи, чем с табличными компонентами.

Рассмотрим общие свойства и методы компонентов запросов.

Текст запроса определяется свойством

```
property SQL: TStrings;
```

В свойстве

```
property Text: PChar;
```

содержится окончательно подготовленный текст запроса перед пересылкой его на сервер.

Выполнение запроса возможно тремя способами.

Если запрос возвращает результат в набор данных, то применяется метод

```
procedure Open;
```

или свойство

```
property Active: Boolean;
```

которому присваивается значение True. После выполнения запроса открывается набор данных компонента. Закрывается такой запрос методом

```
procedure Close;
```

или тем же свойством Active.

Если запрос не возвращает результат в набор данных (например, использует операторы `INSERT, DELETE, UPDATE`), то используется метод

```
procedure ExecSQL;
```

и после выполнения запроса набор данных компонента не открывается. Попытка использовать для такого запроса метод `Open` или свойство `Active` приведет к ошибке создания указателя на курсор данных.

После выполнения запроса в свойстве

```
property RowsAffected: Integer;
```

возвращается число обработанных при выполнении запроса записей.

Для того чтобы разрешить редактирование набора данных запроса, необходимо свойству

```
property RequestLive: Boolean;
```

присвоить значение True. Это свойство устанавливается, но не работает для запроса, результат которого не модифицируется из-за самого запроса.

Для подготовки запроса к выполнению предназначен метод

```
procedure Prepare;
```

который обеспечивает выделение необходимых ресурсов на сервере и проведение оптимизации.

Метод

```
procedure UnPrepare;
```

освобождает занятые при подготовке запроса ресурсы.

Результат выполнения этих двух операций отражается в свойстве

```
property Prepared: Boolean;
```

Значение True данного свойства говорит о том, что запрос подготовлен для выполнения.

Вызов методов `Prepare` и `UnPrepare` не является обязательным, т. к. компонент делает это автоматически. Однако если запрос будет выполняться несколько раз подряд, то подготовку необходимо провести перед первым выполнением запроса вручную. Тогда при последующих выполнениях сервер не будет тратить время на проведение бесполезной операции – ведь ресурсы под запрос уже были выделены.

Часто запросы имеют настраиваемые параметры, значения которых определяются непосредственно перед выполнением запроса.

Свойство

```
property Params: TParams;
```

представляет собой список объектов `TParams`, каждый из которых содержит настройки одного параметра. Свойство `Params` обновляется автоматически при изменении текста запроса. Подробнее о классе `TParams` рассказывается ниже в этой главе.

Примечание

В компоненте TADOQuery свойство, аналогичное описанному свойству `Params`, называется `Parameters`.

Свойство

```
property ParamCount: Word;
```

возвращает число параметров запроса.

Свойство

```
property ParamCheck: Boolean;
```

определяет, необходимо ли обновлять свойство `Params` при изменении текста запроса во время выполнения. При значении True обновление осуществляется.

Кроме этого, компоненты запросов содержат некоторые другие свойства и методы, которые будут описаны позже.

6.3.3. Компонент хранимой процедуры *TStoredProc*

Компонент хранимой процедуры предназначен для определения процедуры, установки ее параметров, выполнения процедуры и возвращения результатов в компонент.

В зависимости от выбранной технологии доступа к данным, каждый компонент хранимой процедуры имеет собственный способ соединения с сервером. После подключения к источнику данных имя хранимой процедуры можно выбрать из списка свойства

```
property StoredProcName: String;
```

После этого свойство

```
property Params: TParams;
```

предназначенное для хранения параметров процедуры, автоматически заполняется.

Для хранимых процедур важно деление параметров на входные и выходные. Первые содержат исходные данные, а вторые передают результаты выполнения процедуры.

Детально класс `TParams` описывается ниже.

Общее число параметров возвращает свойство

```
property ParamCount: Word;
```

Для подготовки хранимой процедуры используется метод

```
procedure Prepare;
```

или свойство

```
property Prepared: Boolean;
```

которое должно получить значение True.

Метод

```
procedure UnPrepare;
```

или свойство `Prepared := False` выполняют обратное действие.

Кроме того, проверка значения свойства `Prepared` позволяет установить, осуществлялась ли подготовка процедуры к выполнению или нет.

После выполнения хранимой процедуры исходный порядок следования параметров в списке `Params` может измениться. Поэтому для доступа к конкретному параметру рекомендуется использовать метод

```
function ParamByName(const Value: String): TParam;
```

Если хранимая процедура возвращает набор данных, компонент можно открывать методом

```
procedure Open;
```

или свойством

```
property Active: Boolean;
```

В противном случае для выполнения процедуры используется метод

```
procedure ExecProc;
```

и после этого выходные параметры получат вычисленные значения.

6.4. Индексы в наборе данных

Важнейшей проблемой для любой БД является достижение максимальной производительности и ее сохранение при дальнейшем увеличении объемов хранимых данных. Использование индексов позволяет решить эту задачу.

Индекс представляет собой часть базы данных, в которой содержится информация об организации данных в таблицах БД.

В отличие от ключей, которые просто идентифицируют отдельные записи, индексы занимают дополнительные объемы памяти (довольно значительные) и могут храниться как совместно с таблицами, так и в виде отдельных файлов. Индексы создаются вместе со своей таблицей и обновляются при модификации данных. При этом работа по обновлению индекса для большой таблицы может отнимать много ресурсов, поэтому имеет смысл ограничить число индексов для таких таблиц, где происходит частое обновление данных.

Индекс содержит в себе уникальные идентификаторы записей и дополнительную информацию об организации данных. Поэтому если при выполнении запроса сервер или локальная СУБД обращается для отбора записи к индексу, то это занимает значительно меньше времени, т. к. понятно, что идентификатор гораздо меньше самой записи. Кроме этого, индекс "знает", как организованы данные и может ускорять обработку за счет группирования записей по сходным значениям параметров.

Создание для БД эффективного набора индексов является нетривиальной задачей.

Во-первых, нужно верно определить оптимальное число индексов для каждой таблицы. Во-вторых, каждый индекс должен содержать только необходимые поля, при этом большую роль играет их упорядочивание.

В большинстве СУБД при создании индексов требуется только задать поля и название индекса, вся остальная работа выполняется автоматически.

Естественно, что в компонентах доступа к данным VCL Delphi используются все возможности такого мощного инструмента, как индексы. Причем свойства и методы для работы с индексами присутствуют только в

табличных компонентах, т. к. в компонентах запросов работа с индексами осуществляется средствами SQL.

Набор данных может работать и без применения индексов, но для этого соответствующая таблица БД не должна иметь первичного ключа – случай довольно редкий. Поэтому по умолчанию в наборе данных используется первичный индекс. При открытии набора данных все записи отсортированы в соответствии с первичным ключом.

6.4.1. Механизм подключения индексов

Для того чтобы подключить к набору данных вторичный индекс, необходимо присвоить свойству `IndexName` название индекса. Если свойство не имеет значения, то в наборе данных используется первичный индекс.

Альтернативный способ задания индекса заключается в использовании свойства `IndexFieldNames`, в котором задается перечень имен полей необходимого индекса, разделенных точкой с запятой. При этом в Инспекторе объектов для данного свойства список полей существующих индексов создается автоматически, разработчику остается сделать выбор. При помощи свойства `IndexFieldNames` можно создавать и составные индексы. Для этого необходимо, чтобы все входящие в список поля были индексированы.

Список имен всех индексов можно получить при помощи метода `GetIndexNames`.

Примечание

Изменение текущего индекса можно осуществлять без отключения набора данных, поэтому в приложениях очень удобно делать сортировку данных по индексам. Такой метод смены индексов называется индексацией "на лету".

После установки индекса количество полей в индексе передается в свойство `IndexFieldCount`.

6.4.2. Список описаний индексов

Информация об индексах набора данных содержится в свойстве класса TDataSet

```
property IndexDefs: TIndexDefs;
```

В нем для каждого индекса создается структура TIndexDef. Доступ к информации об индексах осуществляется через свойство

```
property Items [Index: Integer]: TIndexDef; default;
```

являющееся списком объектов TIndexDef.

Объекты типа TIndexDef можно добавлять в список при помощи метода

```
function AddIndexDef: TIndexDef;
```

или

```
procedure AddIndex(const Name: string; const Fields:
```

```
    string; Options: TIndexOptions;
  [const DescFields: string=''])
```

В первом случае добавляется пустой объект (свойства после этого надо заполнить), во втором – свойства добавляемого объекта указываются в параметрах. Первый метод работает только вместе с созданием таблицы (*подробности будут ниже*).

Поиск объекта описания индекса осуществляет метод

```
function Find(const Name: String): TIndexDef;
```

который возвращает найденный объект по заданному в параметре `Name` имени индекса.

Пара методов

```
function FindIndexForFields(const Fields: string):
     TIndexDef;
function GetIndexForFields(const Fields: String;
     CaseInsensitive: Boolean): TIndexDef;
```

находит объект описания индекса по списку полей, входящих в индекс. Если индекс не найден, ищется первый индекс, начинающийся с указанных полей. Первый из этих двух методов в случае неудачного поиска генерирует исключительную ситуацию EDatabaseError, а второй возвращает **`nil`**.

Список IndexDefs обновляется автоматически при открытии набора данных.

Но метод

```
procedure Update; reintroduce;
```

обновляет список описаний индексов без открытия набора данных.

6.4.3. Описание индекса

Параметры каждого индекса набора данных представлены в классе TIndexDef, а их совокупность для набора данных содержится в свойстве IndexDefs класса TDataSet.

Свойство

```
property Name: String;
```

определяет название индекса.

Список всех полей индекса содержится в свойстве

```
property Fields: String;
```

Поля разделяются точкой с запятой.

Свойство

```
property CaseInsFields: String;
```

содержит список полей, регистр символов в которых при сортировке не учитывается. Поля разделяются точкой с запятой. Все поля из этого списка

должны входить в свойство `Fields`. В наборе данных по умолчанию используется сортировка записей с учетом регистра символов. Но некоторые серверы БД допускают комбинированную сортировку по полям с учетом регистра и без.

Свойство

```
property DescFields: String;
```

содержит список полей через точку с запятой, которые сортируются в обратном порядке. Все поля из этого списка должны входить в свойство `Fields`. По умолчанию все поля сортируются в прямом порядке. Некоторые серверы БД поддерживают одновременную сортировку полей в прямом и обратном порядке.

Параметры индекса определяются свойством

```
property Options: TIndexOptions;
```

Для индекса возможны сочетания следующих параметров:

- ixPrimary – первичный индекс;
- ixUnique – значения индекса уникальны;
- ixDescending – индекс сортирует записи в обратном порядке;
- ixCaseInsensitive – индекс сортирует записи без учета регистра символов;
- ixExpression – в индексе используется выражение (для индексов таблиц формата dBase);
- ixNonMaintained – индекс не обновляется при открытии таблицы.

6.4.4. Использование описаний индексов

Описания индексов наряду с описаниями полей *(рассмотрено выше)* также используются при создании новых таблиц БД. Для каждого планируемого индекса необходимо создать соответствующее описание.

Метод `AddIndexDef` работает только в момент создания новой таблицы:

```
with Table1 do begin
  TableName:= 'Primer1.db';
  with FieldDefs do begin
    Clear;
    Add('ID', ftInteger);
    Add('Name1', ftString, 20);
    Add('Name2', ftString, 20);
  end;
   { Создание описаний полей }
  with IndexDefs do begin
    Clear;
    AddIndexDef;
    with Items[0] do begin
```

```
      Name:= '';
      Fields:= 'ID';
      Options:= [ixPrimary, ixUnique];
    end;
    AddIndexDef;
    with Items[1] do begin
      Name:= 'iNames';
      Fields:= 'Name1; Name2';
      Options:= [ixCaseInsensitive];
    end;
    CreateTable;
      { только в такой последовательности -
        иначе AddIndexDef не работает }
  end;
end;
ShowMessage('OK')
```

При создании описаний индексов метод `AddIndexDef` при каждом вызове добавляет к списку Items (свойство) объекта TIndexDefs новый объект TIndexDef. То есть, объект TIndexDefs – это список объектов TIndexDef, обращаться к которым можно через Items с указанием индекса элемента.

Здесь сначала создается первичный индекс (в таблицах Paradox он не имеет имени), затем вторичный индекс – с именем `iNames`. Для каждого описания обязательно определяются составляющие индекс поля и параметры индекса (свойства Fields и Options).

Метод `AddIndex` более удобен: им можно пользоваться и при создании таблицы, и для создания новых индексов в готовой таблице:

```
with Table1 do begin
  TableName:= 'Primer2.db';
   { Создание таблицы - если её нет }
  with FieldDefs do begin
    Clear;
    Add('ID', ftInteger);
    Add('Name1', ftString, 20);
    Add('Name2', ftString, 20);
  end;
  CreateTable;
   { Создание индексов }
  with IndexDefs do begin
    Clear;
    AddIndex('', 'ID', [ixPrimary, ixUnique]);
    AddIndex('iNames', 'Name1; Name2', [ixCaseInsensitive]);
  end;
end;
```

6.5. Состояния набора данных

В процессе своего функционирования (от открытия методом `Open` и до закрытия методом `Close`) набор данных может выполнять самые разнообразные операции. Можно просто перемещаться по записям, можно редактировать данные и удалять записи, можно проводить поиск по различным параметрам и т. д.

Набор данных в любой момент времени находится в некотором состоянии, т. е. подготовлен к выполнению действий строго определенного рода. И для каждой группы операций набор данных выполняет ряд подготовительных действий.

Все состояния набора данных делятся на две группы.

- К первой группе относятся состояния, в которые набор данных переходит автоматически, а также непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 6.1).
- Во вторую группу входят состояния, которыми можно управлять из приложения, например, перевод набора данных в режим редактирования (табл. 6.2).

Базовый класс TDataSet, инкапсулирующий свойства набора данных, позволяет изменять состояние, а также проверять текущее состояние набора данных.

Текущее состояние набора данных передается в свойство State, имеющее тип TDataSetState:

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit,
dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue,
dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);
```

Для управления состояниями набора данных используются методы `Open`, `Close`, `Edit`, `Insert`.

Таблица 6.1. *Автоматические состояния набора данных*

Константа состояния	Описание
`dsNewValue`	Включается при обращении к свойству `NewValue` поля набора данных
`dsOldValue`	Включается при обращении к свойству `OldValue` поля набора данных
`dsCurValue`	Включается при обращении к свойству `CurValue` поля набора данных
`dsInternalCalc`	Включается при расчете значений полей, для которых `FindKind = fkInternalCalc`
`dsCalcFields`	Включается при выполнении метода `OnCalcFields`
`dsBlockRead`	Включается механизм ускоренного перемещения по

	набору данных
dsOpening	Существует при открытии набора данных методом Open или свойством Active
dsFilter	Включается при выполнении метода OnFilterRecord

Таблица 6.2. *Управляемые состояния набора данных*

Константа состояния	Метод	Описание
dsInactive	Close	Набор данных закрыт
dsBrowse	Open	Данные доступны для просмотра, но недоступны для редактирования
dsEdit	Edit	Данные можно редактировать
dsInsert	Insert	К набору данных можно добавлять новые записи
dsSetKey	SetKey	Включается механизм поиска по ключу. Также могут использоваться диапазоны

Рассмотрим, как изменяется состояние набора данных при выполнении стандартных операций.

Закрытый набор данных всегда имеет неактивное состояние dsInactive.

При открытии набор данных переходит в состояние просмотра данных dsBrowse. В этом состоянии по записям набора данных можно перемещаться и просматривать их содержимое, но редактировать данные нельзя. Это основное состояние открытого набора данных, из него можно перейти в другие состояния, но любое изменение состояния происходит через просмотр данных (рис. 6.2).

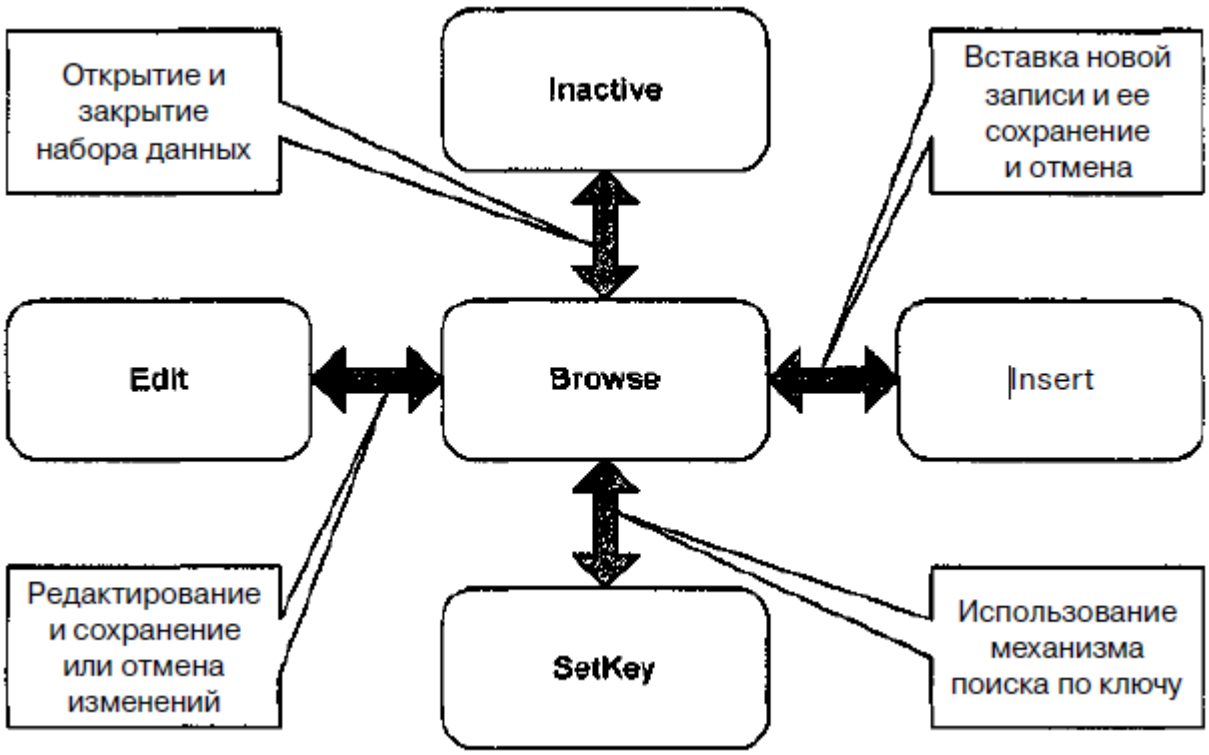


Рис. 6.2. Схема изменения состояний набора данных

При необходимости редактирования данных набор должен быть переведен в состояние редактирования `dsEdit`, для этого используется метод `Edit`. После выполнения метода можно изменять значения полей для текущей записи. При перемещении на следующую запись набор данных автоматически переходит в состояние просмотра.

Для того чтобы вставить в набор данных новую запись, необходимо использовать состояние вставки `dsInsert`. Метод `Insert` переводит набор данных в это состояние и добавляет на месте текущего курсора новую пустую запись. При переходе на другую запись, после проверки на уникальность первичного ключа (если он есть) набор данных возвращается в состояние просмотра.

Состояние установки ключа `dsSetKey` используется только в табличных компонентах при необходимости поиска методами `FindKey` и `FindNext`, а также при использовании диапазонов (метод `SetRange`). Это состояние сохраняется до момента вызова одного из методов поиска по ключу или метода отмены диапазона. После этого набор данных возвращается в состояние просмотра.

Состояние просмотра по блокам `dsBlockRead` используется набором данных при реализации быстрого перемещения по большим массивам записей без показа промежуточных записей в компонентах отображения данных и без вызова обработчика события перемещения по записям. Для реализации быстрого перемещения по набору данных можно использовать методы `DisableControls` и `EnableControls`. Первый отключает связь между визуальными компонентами и невизуальными, второй – включает.

Резюме

Набор данных является образом таблицы базы данных в приложении. Он содержит группу записей и обеспечивает их использование.

Класс TDataSet, инкапсулирующий функциональность набора данных, является базовым классом для всех технологий доступа к данным. На его основе созданы все основные компоненты, применяемые при разработке приложений баз данных. Условно их можно разделить на три группы:

- компоненты таблиц;
- компоненты запросов;
- компоненты хранимых процедур.

В этой главе рассмотрены важнейшие свойства, методы и структуры, реализованные в компонентах, инкапсулирующих набор данных.

7. Механизмы управления данными

Наряду с описываемыми в предыдущих главах свойствами и методами, стандартный набор данных Delphi инкапсулирует ряд дополнительных механизмов, облегчающих управление записями и полями.

К ним относятся такие полезные функции, как быстрое перемещение по записям, поиск нужной записи по значениям полей, дополнительная фильтрация записей набора данных без использования возможностей СУБД и т. д. Большинство этих механизмов применяют в своей работе индексы таблиц БД.

Абстрактные методы, обеспечивающие управление данными, реализованы в базовом классе TDataSet. А классы-потомки, в свою очередь, реализуют механизмы управления данными в соответствии с возможностями технологий доступа к данным.

Все рассматриваемые в этой главе методы управления данными в полном объеме доступны только в компонентах, инкапсулирующих таблицу БД. Это связано с тем, что компоненты запросов SQL и хранимых процедур не обеспечивают полноценное использование индексов.

В этой главе рассматриваются следующие вопросы:

- связанные таблицы;
- методы поиска данных;
- диапазоны;
- быстрая навигация по набору данных;
- фильтрация записей в наборе данных.

7.1. Связанные таблицы

В рамках одного проекта таблицы БД можно связывать отношениями "один-ко-многим" и "многие-ко-многим", при этом отношения обязательно устанавливаются между индексированными полями двух таблиц.

При создании отношений в качестве главной таблицы можно использовать любой компонент, инкапсулирующий набор данных. Для задания подчиненной таблицы можно использовать только табличные компоненты *(см. гл. 6)*.

7.1.1. Отношение "один-ко-многим"

Для установления отношения "один-ко-многим" в наборе данных предназначены два свойства – MasterSource и MasterFields, которые задаются для подчиненной таблицы. Набор данных главной таблицы не требует никаких дополнительных настроек, и заданная связь будет работать только при перемещениях по записям главной таблицы.

Свойство MasterSource определяет компонент TDataSource, который связан с главной таблицей.

Затем при помощи свойства MasterFields необходимо установить отношения между полями главной и подчиненной таблицы. В нем содержится имя индексированного поля, по которому устанавливается связь. Если таких полей несколько, их имена разделяются точкой с запятой. При этом не все поля, входящие в индекс, обязаны участвовать в создании отношения.

Для задания свойства MasterFields можно использовать Редактор связей полей (Field Link Designer), который вызывается щелчком на кнопке в поле редактирования этого свойства в Инспекторе объектов (рис. 7.1).

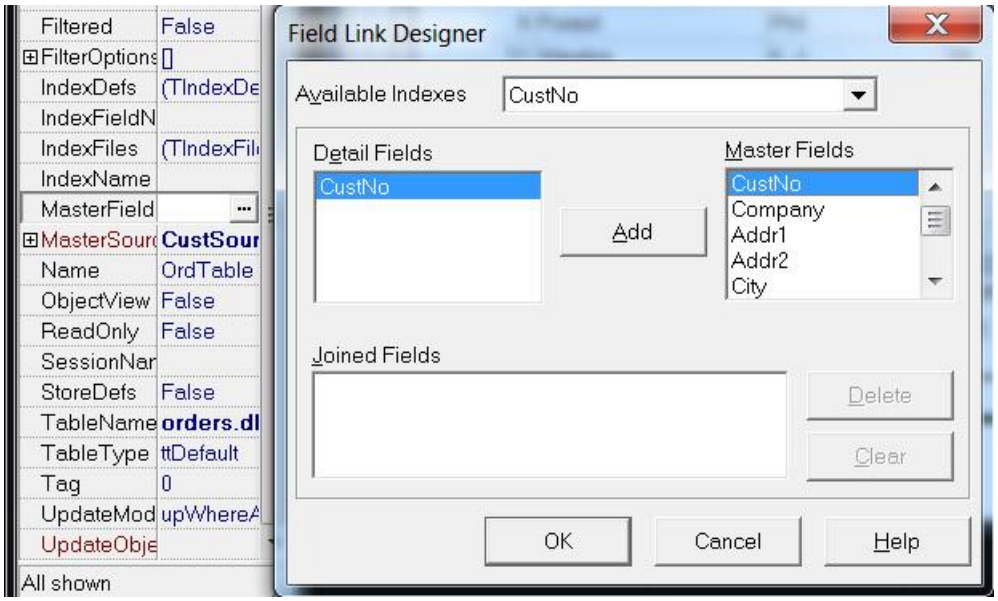


Рис. 7.1. Редактор связей полей

Здесь в разворачивающемся списке **Available Indexes** выбирается требуемый индекс для подчиненной таблицы. После этого в списке **Detail Fields** появляются имена всех полей, входящих в этот индекс. В списке **Master Fields** отображаются все поля главной таблицы.

Теперь требуется создать связи между полями. Для этого в левом списке выбирается поле подчиненной таблицы, а затем соответствующее ему поле главной таблицы в правом списке. После этого активизируется кнопка **Add,** щелчок на которой создает отношение по двум полям главной и подчиненной таблиц. Созданная связь отображается в списке **Joined Fields**

```
CustNo->CustNo
```

Примечание

После создания связи по индексированным полям данный индекс становится текущим для набора данных. При этом в зависимости от типа СУБД автоматически заполняется свойство IndexName или IndexFieldNames.

Уже созданные связи можно удалить. Кнопка **Delete** удаляет выбранную связь, кнопка **Clear** – все связи.

После создания связей между полями отношение "один-ко-многим" считается установленным. Теперь достаточно открыть оба набора данных, чтобы увидеть работу отношения.

В качестве примера рассмотрим проект DemoJoins, в котором связываются таблицы из демонстрационной базы данных DBDEMOS.

Таблица Customers представлена в наборе данных компонента `CustTable`, она содержит данные о покупателях. Таблица Orders представлена в наборе данных компонента `OrdTable`, она содержит данные о заказах. Таблица Employee представлена в наборе данных компонента `EmpTable`, она содержат данные о продавцах (рис. 7.2).

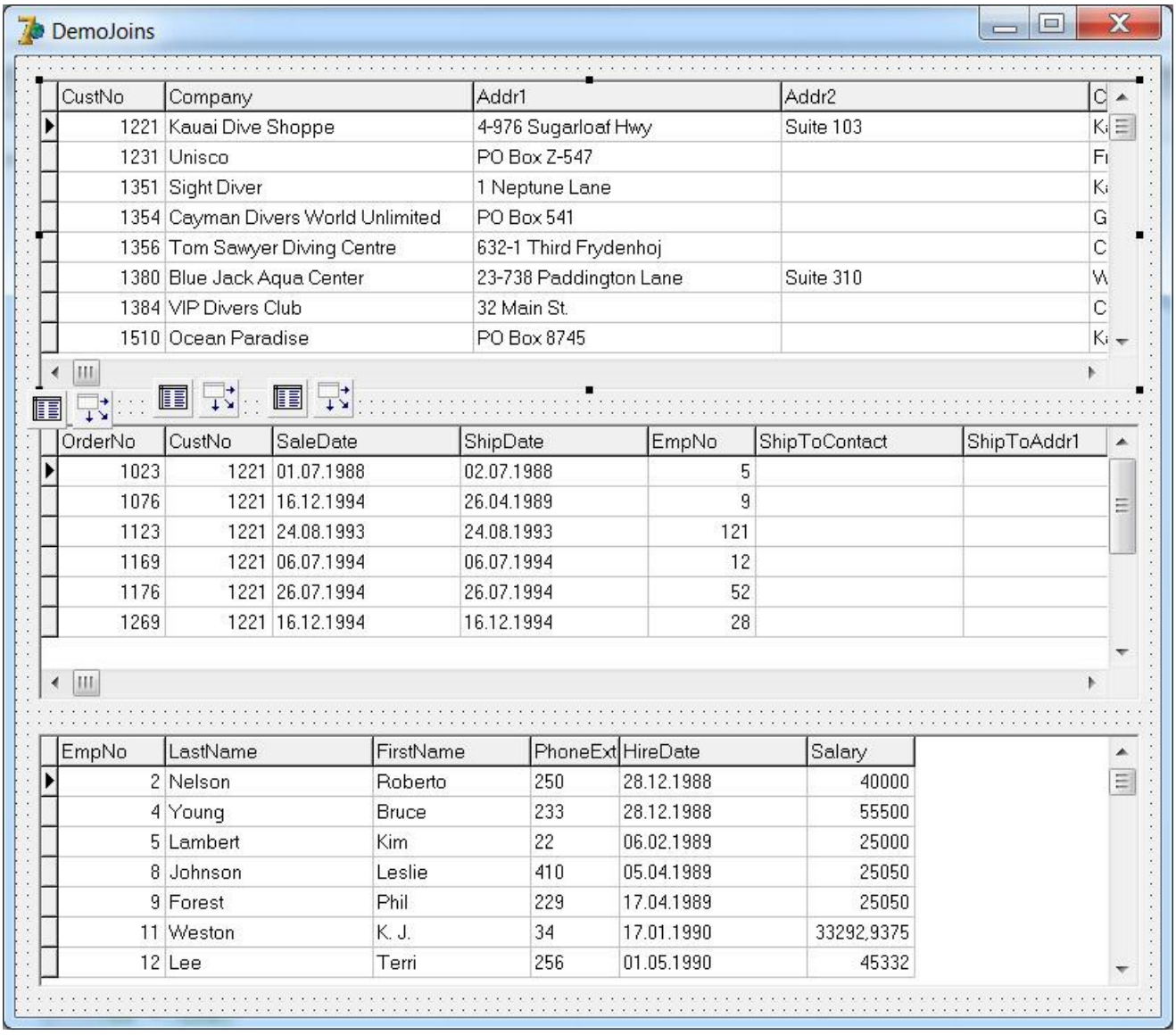


Рис. 14.2. Главная форма проекта DemoJoins

Примечание

Приложение DemoJoins не содержит дополнительного исходного кода. Все отношения между таблицами заданы при помощи Инспектора объектов.

Отношение "один-ко-многим" задано между таблицами покупателей (Customers) и заказов (Orders). Таблица покупателей является главной. Для создания отношения установлены следующие значения свойств компонента `OrdTable` (подчиненная таблица).

Свойство `MasterSource` должно указывать на компонент `CustSource`, связанный с набором данных `CustTable`.

Свойство `MasterFields` указывает на поле CustNo таблицы Customers.

В наборе данных `OrdTable` включен вторичный индекс на основе поля CustNo (IndexName = 'CustNo').

Таким образом, две таблицы связаны отношением "один-ко-многим" по индексированным полям CustNo (номер покупателя). В результате, при перемещении по записям таблицы покупателей, в таблице заказов будут показаны только те заказы, которые относятся к текущему покупателю.

7.1.2. Отношение "многие-ко-многим"

Отношение "многие-ко-многим" отличается тем, что подчиненная таблица еще раз связывается в качестве главной с другой подчиненной таблицей аналогичной последовательностью действий, как и в отношении "один-ко- многим".

В приложении DemoJoins отношением "многие-ко-многим" связаны таблицы заказов (Orders) и продавцов (Employee). Таблица заказов уже работает в отношении "один-ко-многим" в качестве подчиненной.

В наборе данных EmpTable заданы следующие свойства:

- свойство MasterSource указывает на компонент OrdSource;
- свойство MasterFields содержит имя поля EmpNo, по которому осуществляется связь между таблицами. Для подчиненной таблицы поле EmpNo является первичным.

Результат работы связанных трёх таблиц представлен на рисунке 7.3

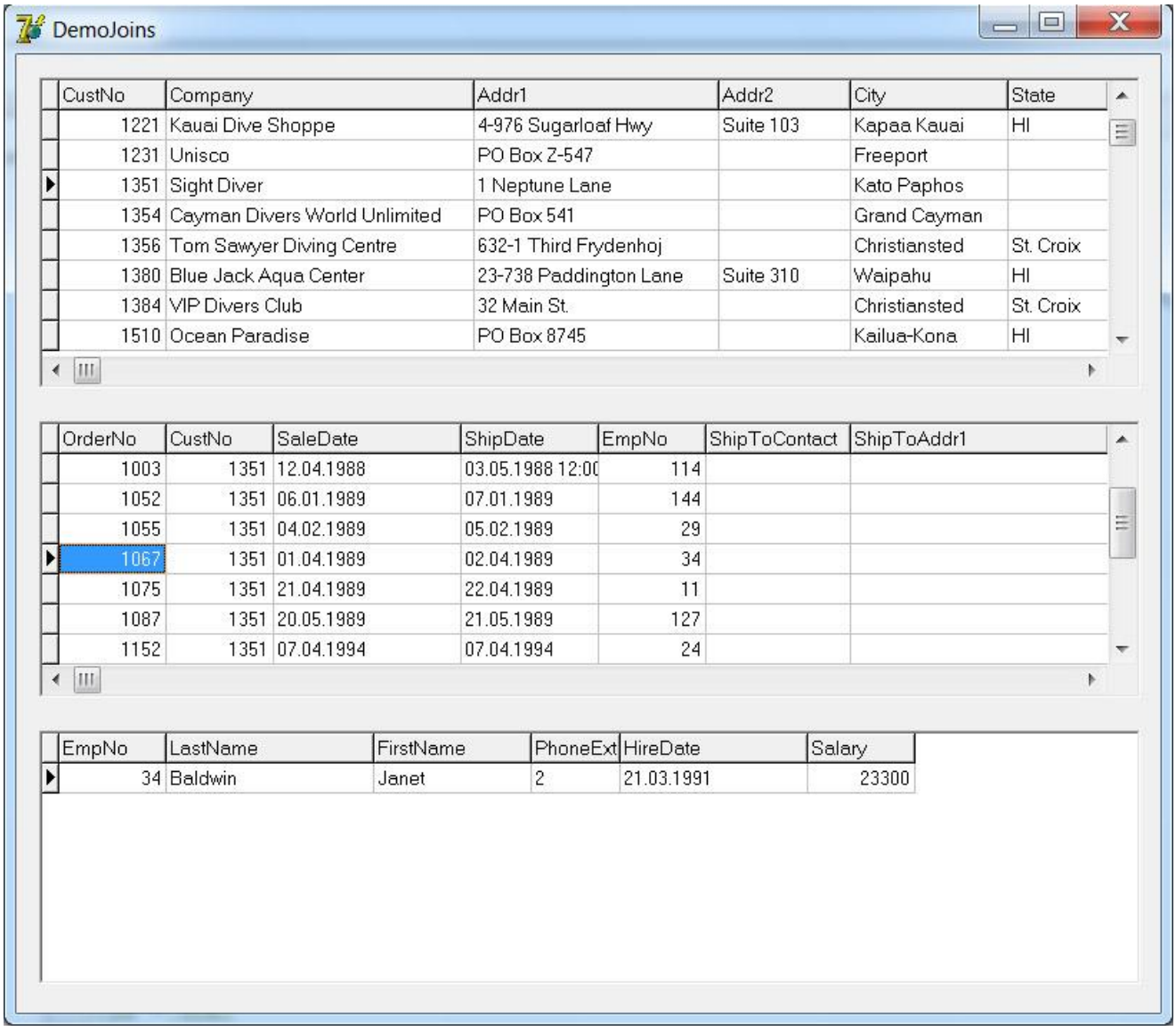


Рис.7.3. Готовое приложение DemoJoins

7.2. Поиск данных

В наборе данных Table реализованы два способа поиска записей по заданным значениям полей. Один способ основан на использовании индексов и является более быстрым, но поиск проводится только по индексированным полям. Второй способ применяет специальные методы классов наборов данных и позволяет проводить поиск по любому сочетанию полей, но он более медленный.

7.2.1. Поиск по индексам

Для организации индексного поиска к набору данных должен быть подключен индекс (свойства IndexName или IndexFieldNames).

Метод **`FindKey(const KeyValues: array of const): Boolean`** проводит поиск записи по заданным в параметре **`KeyValues`** значениям ключевых полей текущего индекса набора данных. В случае успеха курсор

набора данных устанавливается на найденной записи, а метод возвращает значение True, в противном случае – False.

Если индекс состоит из нескольких полей, значения для поиска записываются в виде множества, причем отсутствующие значения приравниваются к Null.

Рассмотрим простейший пример, в котором реализован поиск по вторичному индексу в таблице CUSTOLY.DB демонстрационной базы данных DBDEMOS. Индекс основан на полях `Last_Name` и `First_Name` (рис. 7.4).

В компоненте `Table1`, помимо стандартных настроек на таблицу, при помощи свойства IndexName задан и вторичный индекс (его имя Names). Значения для поиска задаются в компонентах Edit1 и Edit2.

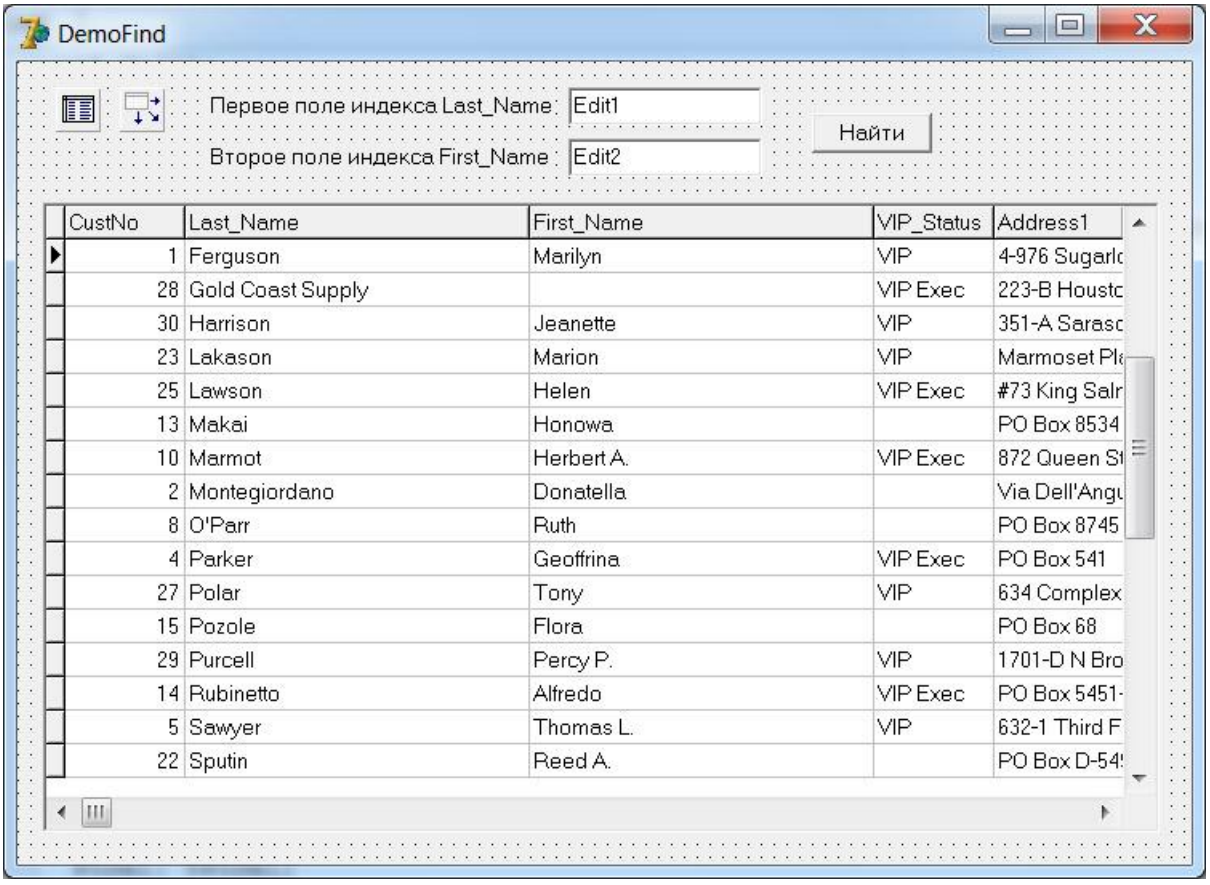


Рис. 7.4. Главная форма проекта DemoFind

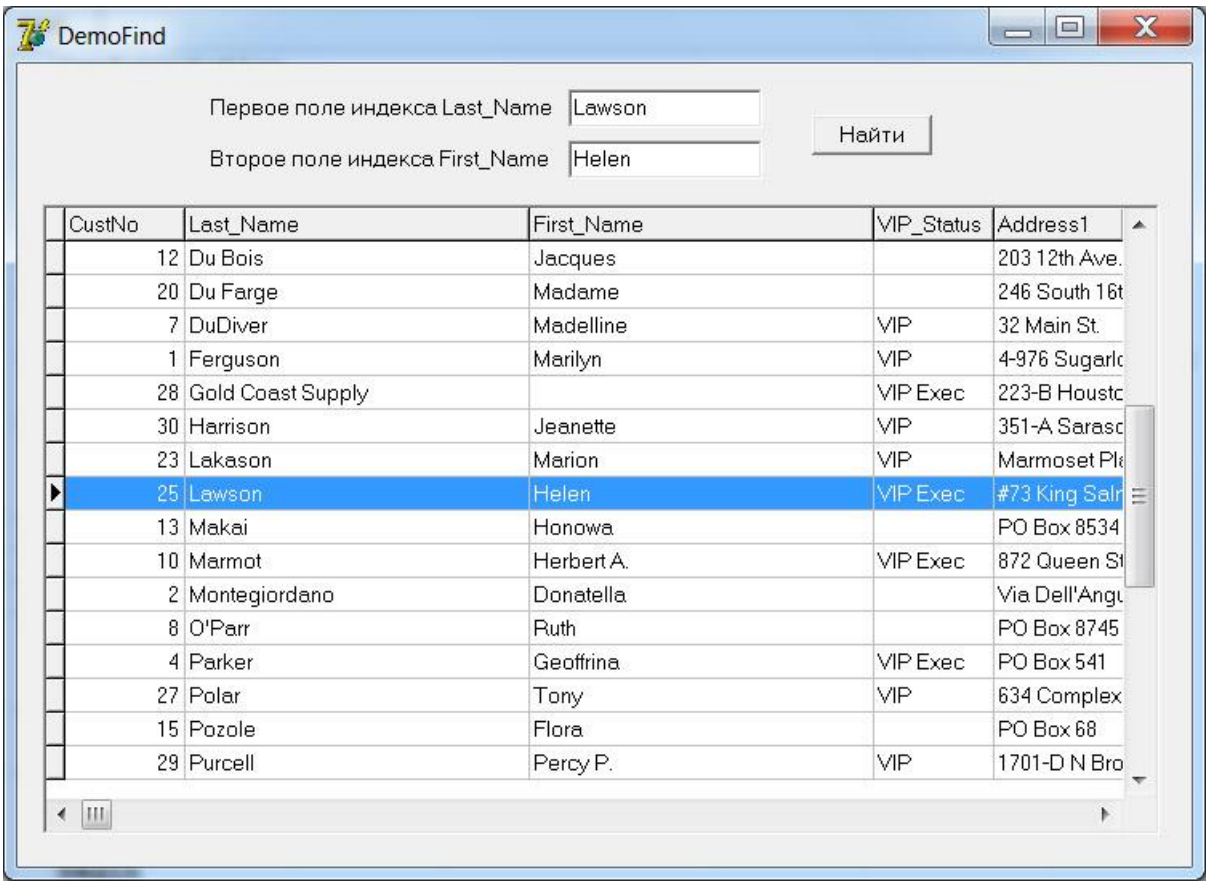


Рис. 7.5. Приложение DemoFind в работе

Секция implementation главного модуля Main проекта DemoFind

```
implementation

{$R *.dfm}

procedure TForm1.FormShow(Sender: TObject);
begin
  try
    Table1.Open;
  except
    on E: EDBEngineError do ShowMessage('Ошибка при открытии таблицы');
  end;
end;

procedure TForm1.FindBtnClick(Sender: TObject);
begin
  try
    if not Table1.FindKey([Edit1.Text, Edit2.Text]) then
      ShowMessage('Запись не найдена');
  except
    on E: EDatabaseError do
      ShowMessage('Ошибка поиска');
  end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Table1.Close;
end;
```

Набор данных открывается в методе-обработчике FormShow при открытии формы и закрывается в методе-обработчике FormClose. При щелчке на кнопке FindBtn в метод FindKey передаются значения для поиска из компонентов Editl и Edit2.

Вместо **FindKey** индексный поиск можно организовать группой методов. Метод **SetKey** переводит набор данных в состояние dsSetKey, затем должно следовать присваивание ключевым полям значений для поиска. Сам поиск осуществляется методом **GotoKey**:

```
with Table1 do
  begin
    SetKey;
    FieldByName('Last_Name').Value := 'Sputin';
    FieldByName('First_Name').Value := 'Reed A.';
    if GotoKey then
      ShowMessage('Reed A. Sputin')
    else
      ShowMessage('Запись не найдена: Reed A. Sputin');
    EditKey;
    FieldByName('Last_Name').Value := 'Harrison';
    FieldByName('First_Name').Value := 'Jeanettz';
    GotoNearest;
    ShowMessage('Jeanettz Harrison');
  end;
```

В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение True. Вместо этого метода можно применять метод **GotoNearest**, который в случае неудачного поиска ищет запись, минимально отличающуюся от критерия поиска – точнее, курсор устанавливается на то место, где должна была бы находиться искомая запись согласно условиям сортировки.

Изменение параметров поиска осуществляется методом **EditKey**.

Вместо точного поиска FindKey можно аналогично выполнять неточный поиск методом **FindNearest**. Параметры в нём такие же, как и в FindKey, но результат всегда будет положительным – аналогично методу GotoNearest из предыдущей комбинации.

Пример:

```
Table1.IndexName:= 'Names';
Table1.FindNearest(['Q','Z']);
```

Курсор установится на первую запись, где имя начинается на букву ’Q’. Если такого имени нет, то будет найдено имя, начинающееся с ближайшей к ‘Q’ следующей буквы (например, ‘R’).

Замечание

*С форматом БД dBase в случае составного индекса метод **FindKey** не работает – только комбинация **SetKey**, **GotoKey** (или **GotoNearest**).*

Диапазоны

В наборе данных имеется также быстрое средство отбора записей на основе использования индексов. Группа методов позволяет отбирать в набор данных только те записи, значения индексированных полей которых (для текущего индекса) соответствуют *диапазону* заданных величин.

При использовании диапазонов набор данных обязательно должен находиться в состоянии dsSetKey.

Для того чтобы включить диапазон, необходимо задать стартовое и конечное значение диапазона для ключевых полей, затем применить созданный диапазон к набору данных. Работающий диапазон можно модифицировать.

Примечание

Все методы работы с диапазонами используют те поля, которые заданы в текущем индексе. Для таблиц Paradox и dBase это свойство IndexName. Для таблиц серверов SQL это свойство IndexFieldNames.

Метод `SetRangeStart` переводит набор данных в режим dsSetKey, следующее за этим присваивание ключевым полям значений означает задание начальной границы диапазона.

Метод `SetRangeEnd` переводит набор данных в режим dsSetKey, следующее за этим присваивание ключевым полям значений означает задание конечной границы диапазона.

После этого необходимо использовать метод `ApplyRange`, который применяет созданный диапазон к набору данных:

```
with Table1 do begin
   SetRangeStart;
   Fields[0].Value := '439';
   SetRangeEnd;
   Fields[0].Value := '522';
   ApplyRange;
end;
```

Работающий диапазон можно модифицировать аналогичным образом: после вызова методов `EditRangeStart` и `EditRangeEnd` необходимо задать новые границы для ключевых полей и снова вызвать метод `ApplyRange`:

```
with Table1 do begin
   EditRangeStart;
   Fields[0].Value := '500';
   EditRangeEnd;
   Fields[0].Value := '522';
   ApplyRange;
end;
```

Отмена диапазона осуществляется методом `CancelRange`.

Если индекс содержит несколько полей, то перед вызовом метода `ApplyRange` необходимо задать значения для всех ключевых полей.

Для одновременного задания верхней и нижней границы диапазона можно использовать метод

```
procedure SetRange(const ValStart: array of const;
                   const ValEnd: array of const).
```

Пример:

```
with Table1 do begin
   SetRange(['500'], ['522']);
   ApplyRange; { не обязательно }
end;
```

Тем, какая граница будет у диапазона – открытая или закрытая, управляет свойство `KeyExclusive`. Если оно имеет значение True, граничные значения в диапазон не включаются, в противном случае – включаются.

Замечание

С форматом БД dBase в случае составного индекса метод `SetRange` *не работает – только комбинация* `SetRangeStart – SetRangeEnd (EditRangeStart – EditRangeEnd), ApplyRange`.

7.2.2. Поиск по произвольным полям

Для поиска по произвольной выборке полей можно использовать методы `Locate` и `Lookup`.

```
function Locate(const KeyFields: string;
                const KeyValues: Variant;
                Options: TLocateOptions): Boolean;
function Lookup(const KeyFields: string;
                const KeyValues: Variant;
                const ResultFields: string): Variant;
```

В метод `Locate` необходимо передать список полей, по которым будет идти поиск (параметр `KeyFields`, имена полей разделяются точкой с запятой), их требуемые значения (параметр `KeyValues`, значения разделяются запятой) и настройки поиска (параметр `Options`). В настройках можно задать опцию `loCaseInsensitive`, которая отключает проверку на регистр символов (но с русскими «Я» и «Ч» будут проблемы), и опцию `loPartialKey`, которая включает поиск с минимальными отличиями (совпадение начального фрагмента).

В случае успеха поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение True.

```
Table1.Locate('Last_Name; First_Name',
      VarArrayOf([Edit1.Text, Edit2.Text]), []);
```

В метод `Lookup` передается список полей для поиска (параметр `KeyFields`, имена полей разделяются точкой с запятой) и их требуемые значения (параметр `KeyValues`, значения разделяются запятой). В случае успешного поиска функция возвращает массив значений типа `Variant` для полей, названия которых содержатся в параметре `ResultFields`. Курсор набора данных не переходит к найденной записи.

```
Table1.Lookup('Last_Name; First_Name',
      VarArrayOf([Edit1.Text, Edit2.Text]),
     'Last_Name; First_Name');
```

Оба эти метода автоматически используют быстрый индексный поиск в случае, если в параметре KeyFields задать поля индекса.

Замечание о типе Variant

Для описания переменной вариантного типа служит ключевое слово `Variant`. Такой переменной можно присваивать значения целочисленных (кроме Int64), вещественных, символьных, строковых и логических типов. Вариантную переменную можно интерпретировать так же, как массив вариантной переменной. При этом нельзя назначить обычный статический массив вариантной переменной. Вместо этого следует создать массив `Variant` с помощью одной из двух стандартных функций

```
VarArrayCreate(const Bounds: array of integer;
   VarType: integer): Variant
```

или

```
VarArrayOf(const Values:array of Variant):Variant.
```

Пример применения этих функции:

```
var
   V: Variant;
   s: string;
begin
   V:= VarArrayCreate([0,3], varVariant);
   V[0]:= 1;
   V[1]:= 'Hello, world';
   V[2]:= True;
   V[3]:= VarArrayOf([5,7,11,13,17]);
   s:= VarToStr(V[3][0]);
   // s:= IntToStr(V[3][0]);  здесь можно и так
   Label1.Caption:= V[1];
   ShowMessage(s);
end;
```

Примеры использования метода `Lookup`.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  myResult: Variant; //для получения результата
  s: string;
begin
  Table1.Open;
  // получаем результат:
  myResult := Table1.Lookup('FullName', Edit1.Text,
     'FullName;BYear;DYear;State');
  // проверяем, не Null ли это:
  if VarType(myResult) = varNull then
    ShowMessage('Сотрудник с такой фамилией не найден!')

  // если это массив, то из его элементов собираем строку
  else if VarIsArray(myResult) then begin
    s := myResult[0] + ' ' + VarToStr(myResult[1]) + '-' +
    VarToStr(myResult[2]) + ' ' + myResult[3];
    ShowMessage(s);
  end;
  Table1.Close
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  myResult: Variant; //для получения результата
begin
  Table1.Open;
  myResult := Table1.Lookup('FullName', Edit1.Text, 'BYear');
  if VarType(myResult) = varNull then
    ShowMessage('Сотрудник с такой фамилией не найден!')
  // если это одно число, то показываем его
  else
    ShowMessage(VarToStr(myResult));
  Table1.Close
end;
```

7.2.3. Фильтры

Наиболее эффективным способом отбора записей в набор данных (особенно из больших таблиц) является создание и выполнение соответствующего запроса SQL. Но что делать, если набор данных функционирует на базе табличного компонента? В этом случае на помощь приходит встроенный в набор данных механизм фильтрации данных.

Применение фильтра основано всего на двух основных свойствах и одном вспомогательном. Текст фильтра должен содержаться в свойстве `Filter`, а свойство `Filtered` включает и выключает фильтр. Параметры фильтра определяются свойством `FilterOptions`.

Примечание

Компонент TQuery также может использовать фильтры. Эта возможность подчас позволяет легко и изящно решать довольно сложные проблемы, которые иначе требуют изменения текста запроса или создания нового компонента запроса.

При использовании фильтра его текст транслируется в синтаксис SQL и передается для выполнения на сервер или через соответствующий драйвер в локальную СУБД.

Фильтры можно создавать двумя способами:

- при помощи свойства `Filter` создаются довольно простые фильтры, для которых достаточно предоставляемого механизмом фильтрации синтаксиса;
- для создания более сложных фильтров с применением всех возможных средств языка программирования применяется метод-обработчик набора данных `OnFilterRecord`.

При создании текста фильтра для свойства `Filter` используются имена полей соответствующей таблицы БД, а для задания отношений применяются все операторы сравнения (>, >=, <, <=, =, <>) и логические операторы (AND, OR, NOT):

```
Field1>100 AND Field2=20
```

Сравнивать между собой два поля нельзя. Следующий фильтр вызовет ошибку при попытке использования:

```
ItemCount=Balance AND InputPrice>OutputPrice
```

При создании динамических фильтров можно изменять как выражение фильтра целиком, так и его части. Например, ограничивающее значение для поля можно задавать при помощи элементов управления формы, что позволяет пользователю приложения управлять фильтрацией набора данных:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
   with Table1 do begin
      Filtered := False;
      Filter := 'Field1>=' + TEdit(Sender).Text;
      Filtered := True;
   end;
end;
```

В фильтрах можно производить отбор по частям строк для строковых полей, для этого используется символ звездочка:

```
ItemName='A*'
```

Фильтр начинает работать только после того, как свойству `Filtered` присваивается истинное значение. Перед изменением текста динамического фильтра или для отключения фильтра свойству `Filtered` присваивается значение False.

Параметры фильтра определяются свойством `FilterOptions`:

```
property FilterOptions: TFilterOptions;
TFilterOption = (foCaseInsensitive, foNoPartialCompare);
```

```
TFilterOptions = set of TFilterOption;
```

Параметр foCaseInsensitive, будучи включенным в свойстве, отключает сравнение строковых значений с учетом регистра символов.

Параметр foNoPartialCompare отключает отбор строковых значений по *части* строки.

Метод-обработчик `onFilterRecord` имеет следующее объявление:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
                        var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Если этот метод создан для набора данных, то он вызывается для каждой его записи. Программный код метода должен присваивать параметру `Accept` истинное или ложное значение (по умолчанию True). В результате запись передается в набор данных или отсекается:

```
procedure TForml.Table1FilterRecord(DataSet: TDataSet;
    var Accept: Boolean);
begin
   Accept := ArchOrdersArchDat.AsString >= DateEditl.Text;
end;
```

Метод включается / выключается так же через значение `Filtered`.

Важнейшее преимущество метода `onFilterRecord`, по сравнению со свойством `Filter`, заключается в том, что в этом методе-обработчике можно сравнивать поля и производить вычисления над их значениями.

Недостатком метода является недостаточная гибкость, хотя такой фильтр можно модифицировать путем присвоения методу процедурной переменной, содержащей ссылку на новый метод.

7.2.4. Быстрый переход к помеченным записям

Закладки, как инструмент работы с записями набора данных, позволяют осуществлять быстрое перемещение на нужную запись. Набор данных может содержать неограниченное число закладок, каждая из которых представляет собой указатель. Закладку можно создать только для текущей записи набора данных.

При работе с закладками используются три основных метода:

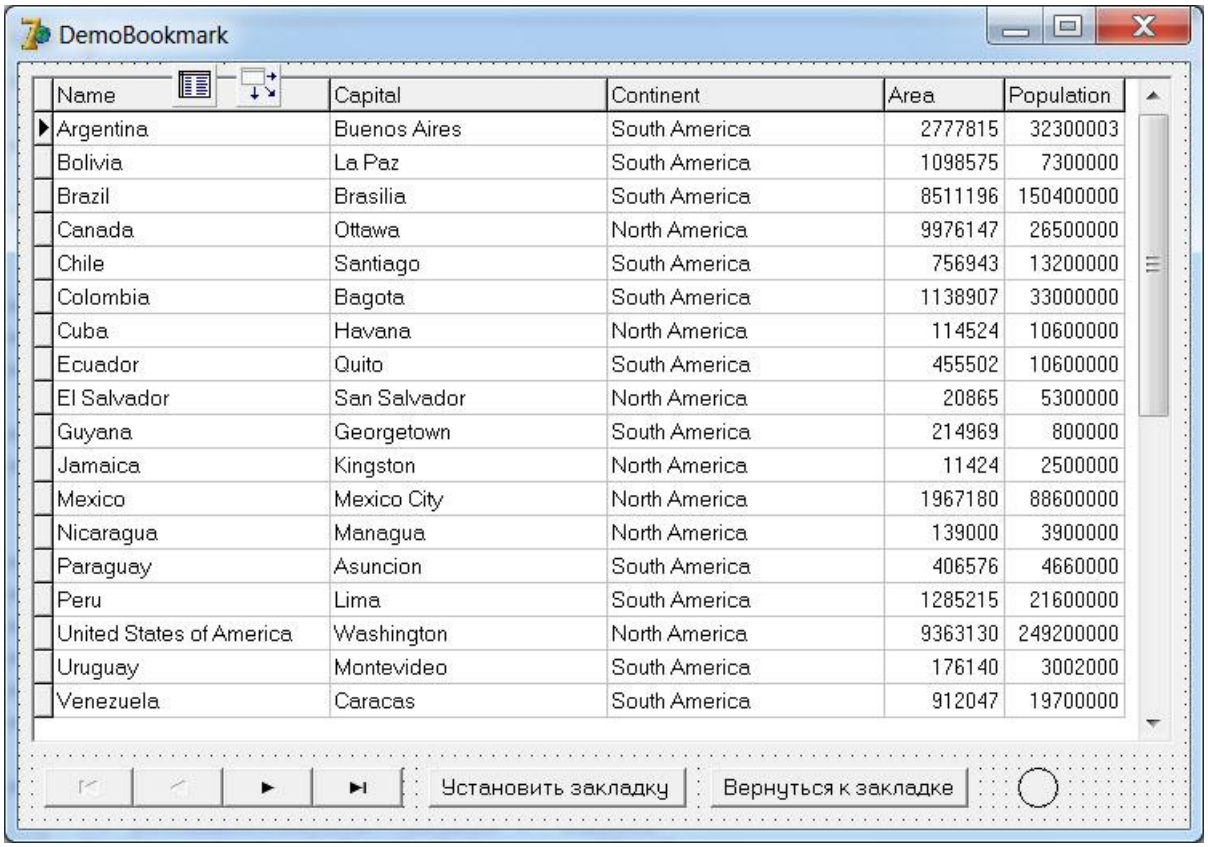
- метод `GetBookmark` создает новую закладку для текущей записи;
- метод `GotoBookmark` осуществляет переход к закладке, переданной в параметре;
- метод `FreeBookmark` удаляет закладку, переданную в параметре.

Кроме этого, можно использовать метод `BookmarkValid`, который проверяет, указывает ли закладка на реально существующую запись. Метод `CompareBookmark` позволяет сравнить между собой две закладки:

```
var Bookmark1, Bookmark2: TBookmark;
if Table1.CompareBookmark(Bookmark1, Bookmark2) = 1
then ShowMessage ('Закладки одинаковы');
```

В наборе данных имеется свойство `Bookmark`, которое содержит название текущей закладки.

Рассмотрим небольшой пример, где право управлять закладками предоставлено пользователю (рис. 7.6). На форме, помимо других элементов управления (среди которых есть компонент TDBGrid), имеются две кнопки. Кнопка StartBookmark помечает текущую запись, кнопка StopBookmark переходит к закладке, а затем уничтожает ее.



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. 7.6. Главная форма проекта DemoBookmark

```
implementation

{$R *.dfm}

var SaveRecPos: TBookMark;

procedure TForm1.FormShow(Sender: TObject);
begin
  try
    Table1.Open;
    BookmarkControl.Brush.Color := clBtnFace;
  except
```

```
    ShowMessage('Ошибка открытия набора данных');
  end;
end;

procedure TForm1.StartBookmarkClick(Sender: TObject);
begin
  if Not Table1.BookmarkValid(SaveRecPos) then
    SaveRecPos := Table1.GetBookmark;
  BookmarkControl.Brush.Color := clLime;
end;

procedure TForm1.StopBookmarkClick(Sender: TObject);
begin
  with Table1 do begin
    if Not BookmarkValid(SaveRecPos) then Exit;
    GotoBookmark(SaveRecPos);
    FreeBookmark(SaveRecPos);
    SaveRecPos := Nil;
  end;
  DBGrid1.SetFocus;
  BookmarkControl.Brush.Color := clBtnFace;
end;

procedure TForm1.FormClose(Sender: TObject; var Action:
     TCloseAction);
begin
  Table1.Close
end;
```

Использование метода `BookmarkValid` позволяет корректно переопределять закладку, если она уже установлена, и избежать ошибок при произвольных нажатиях кнопок. Компонент BookmarkControl типа TShape сигнализирует о том, что закладка установлена или удалена.

Примечание

Закладки также используются в компоненте TDBGrid. Он имеет свойство SelectedRows типа TBookmarkList, которое представляет собой список закладок, указывающих на одновременно выделенные записи.

Резюме

Разработчик приложений БД в Delphi может использовать ряд полезных механизмов набора данных, которые реализованы для компонентов всех технологий доступа к данным.

К этим механизмам относятся методы быстрого поиска и перехода к найденным записям; связывания наборов данных по индексированным полям; метод дополнительной фильтрации записей набора данных.

8. Компоненты отображения данных

До этого момента мы рассмотрели аспекты создания приложений баз данных, касающиеся организации доступа к данным и создания в приложениях наборов данных. Теперь более подробно остановимся на вопросах отображения данных в приложениях (интерфейс приложений).

Отображение данных обеспечивает достаточно представительный набор компонентов VCL Delphi. Многие из них унаследованы от компонентов, инкапсулирующих стандартные элементы управления. Для связи с набором данных эти компоненты используют компонент TDataSource.

Механизмы управления данными реализованы в компонентах наборов данных и активно взаимодействуют с компонентами отображения данных.

В этой главе рассматриваются следующие вопросы:

- использование стандартных компонентов отображения данных;
- навигация по данным.

Классификация компонентов отображения данных

Все компоненты отображения данных можно разделить на группы по нескольким критериям (рис. 8.1).

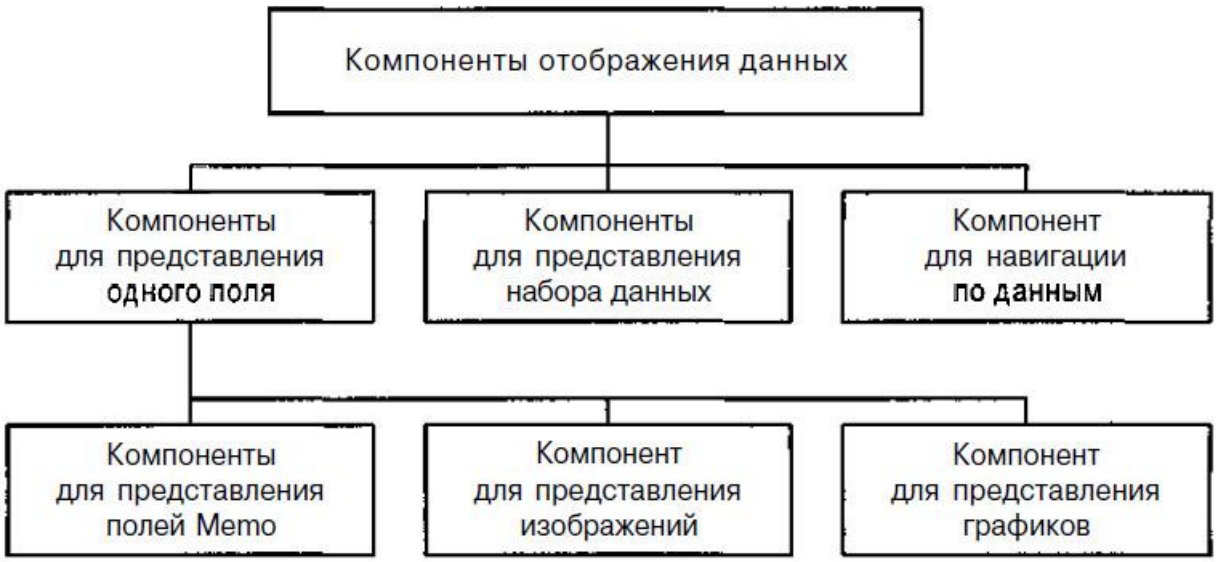


Рис. 8.1. Классификация компонентов отображения данных

Большинство компонентов предназначены для работы с отдельным полем, т. е. при перемещении по записям набора данных такие компоненты показывают текущие значения только одного поля. Для соединения с набором данных через компонент TDataSource предназначено свойство DataSource. Поле задается свойством DataField.

Компонент TDBGrid обеспечивают просмотр наборов данных целиком или в произвольном сочетании полей. В нём присутствует только свойство DataSource.

Особенную роль среди компонентов отображения данных играет компонент TDBNavigator. Он не показывает данные и не предназначен для их редактирования, зато обеспечивает навигацию по набору данных.

Наиболее часто в практике программирования используются компоненты TDBGrid, TDBEdit и TDBNavigator.

Для представления и редактирования информации, содержащейся в полях типа Memo, используются специальные компоненты TDBMemo и TDBRichEdit. Для просмотра (без редактирования) изображений предназначен компонент TDBImage.

Отдельную группу составляют компоненты синхронного просмотра данных. Они обеспечивают показ значений поля из одной таблицы в соответствии со значениями поля из другой таблицы.

Наконец, данные можно представить в виде графика. Для этого предназначен компонент TDBChart.

Как видите, набор компонентов отображения данных весьма разнообразен и позволяет решать задачи по созданию любых интерфейсов для приложений баз данных.

Ввиду общности решаемых задач, компоненты отображения данных имеют несколько важных общих свойств, которые представлены в табл. 8.1 и в дальнейшем изложении опущены.

Таблица 8.1. Общие свойства компонентов отображения данных

Объявление	Описание
property DataField: string;	Поле связанного с компонентом набора данных
property DataSource: TDataSource;	Связываемый с компонентом компонент TDataSource
property Field: Tfield;	Обеспечивает доступ к классу TField, который соответствует полю набора данных, заданному свойством DataField
property Readonly: Boolean;	Управляет работой режима "только для чтения"

8.1. Табличное представление данных. Компонент *TDBGrid*

Этот компонент инкапсулирует двумерную таблицу, в которой строки представляют собой записи, а столбцы – поля набора данных.

Компонент TDBGrid является потомком Классов TDBCustomGrid и TCustomGrid.

От класса TCustomGrid наследуются все функции отображения и управления работой двумерной структуры данных. Класс TDBCustomGrid обеспечивает визуализацию и редактирование полей из набора данных, причем TDBGrid только публикует свойства и методы класса TDBCustomGrid, не добавляя собственных.

Настройка параметров компонента TDBGrid, от которых зависит его внешний вид и некоторые функции, осуществляется при помощи свойства Options. Текущая позиция в двумерной структуре данных может быть определена свойствами SelectedField, SelectedRows, SelectedIndex. При необходимости разработчик может использовать разнообразные методы-обработчики событий. Среди них есть как стандартные методы, присущие всем элементам управления, так и специфические.

В компоненте TDBGrid можно отображать произвольное подмножество полей используемого набора данных, но число записей ограничить нельзя – в компоненте всегда присутствуют все записи связанного набора данных. Требуемый набор полей можно составить при помощи специального Редактора столбцов, который открывается при двойном щелчке на компоненте, перенесенном на форму, или кнопкой свойства Columns в Инспекторе объектов.

Новая колонка добавляется при помощи кнопки **Add New,** после этого ее название появляется в списке колонок (рис. 8.2). Для выбранной в списке колонки доступные для редактирования свойства появляются в Инспекторе объектов. Колонки в списке можно редактировать, удалять, менять местами.

Каждая колонка компонента TDBGrid описывается специальным классом TColumn, а совокупность колонок доступна через свойство Columns компонента, оно имеет тип TDBGridColumns и представляет собой индексированный список объектов колонок. Поле набора данных связывается с конкретной колонкой при помощи свойства FieldName класса TColumn. При этом в колонку автоматически переносятся все необходимые параметры поля, в частности заголовок поля, настройки шрифтов, ширина поля. После ручного изменения параметров первоначальные значения восстанавливаются методами соответствующих объектов TColumn.

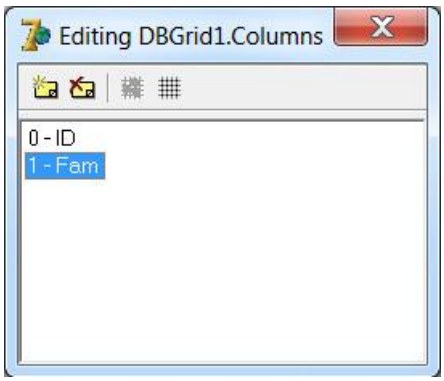


Рис. 8.2. Редактор колонок компонента TDBGrid

Если не пользоваться редактором столбцов самой сетки, *DBGrid* будет выводить значения по умолчанию – будут выведены все поля набора данных, а заголовки столбцов будут соответствовать именам полей. Но стоит только добавить в редактор столбцов хоть один столбец, и сетка *DBGrid* будет отображать только его. Таким образом, мы можем показывать только те столбцы, которые действительно необходимы.

Изменить параметры заголовка столбца можно в раскрывающемся свойстве **Title**, которое имеет ряд собственных свойств:

Alignment – свойство устанавливает выравнивание заголовка и может быть *taCenter* (по центру), *taLeftJustify* (по левому краю) и *taRightJustify* (по правому краю). По умолчанию, заголовок выровнен по левому краю.

Caption – свойство содержит текст, который отображается в заголовке столбца. Если поле НД имеет имя латинскими буквами, именно здесь можно отобразить его кириллицей.

Color – цвет заголовка, по умолчанию это свойство равно *clBtnFace*, что обеспечивает стандартный серый цвет. Если вы желаете украсить программу, можете выбрать другой цвет.

Font – шрифт заголовка. Если дважды щелкнуть по этому свойству, откроется диалоговое окно, в котором можно изменить шрифт, начертание, размер и цвет шрифта. То же самое можно сделать, если раскрыть это свойство и непосредственно изменять нужные свойства.

Надо заметить, что свойства *Font*, *Alignment* и *Color* внутри свойства *Title* меняют шрифт, выравнивание и цвет фона только заголовка столбца, а не его содержимого. Но у столбца имеются эти же свойства, они меняют шрифт, выравнивание и цвет фона выводимых в столбце данных.

Свойство *Visible* разрешает или запрещает отображение столбца, а свойство *Width* позволяет изменить его ширину.

Рис. 8.3. Свойства DBGrid.Columns в Object Inspector

Пустые столбцы

Если добавить в редактор столбцов сетки *DBGrid* новый столбец, но в свойстве *FieldName* не выбирать поле БД, а оставить его пустым, мы получим пустой столбец. Для чего нужны пустые столбцы в сетке? Во-первых, в них можно выводить обработанные данные из других столбцов. К примеру, пользователю неудобно просматривать три столбца «Фамилия», «Имя» и «Отчество». Ему было бы удобней просмотреть один сборный столбец в формате «Фамилия И.О.». Во-вторых, пустое поле может выводить информацию по требованию.

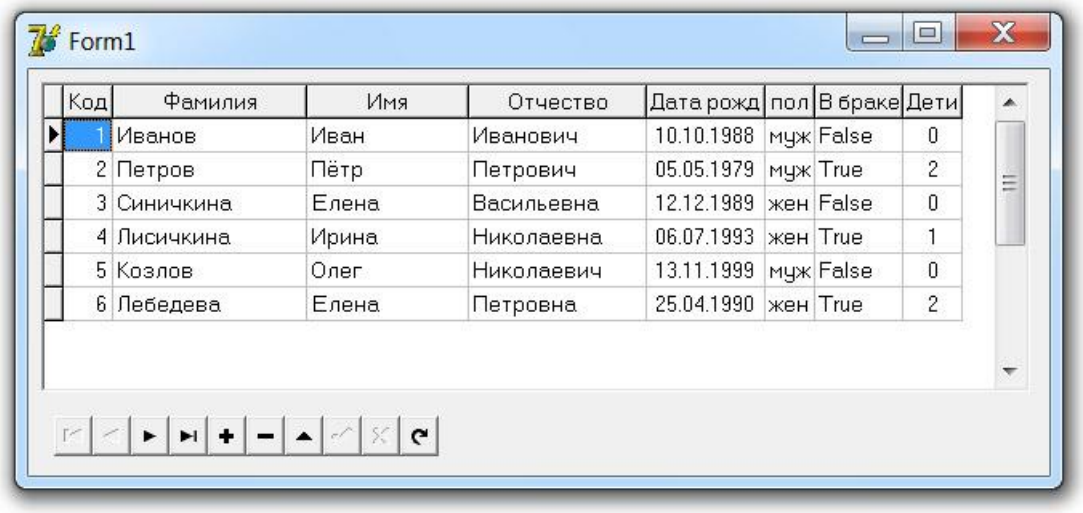


Рис. 8.4. Начальный проект

Рассмотрим эти случаи.

Создадим новый столбец, но не будем назначать ему поле из НД. Выделим этот столбец, и в его свойстве *Title.Caption* впишем «Фамилия И.О.», а в свойстве *Width* укажем ширину в 150 пикселей.

Столбцы «Фамилия», «Имя» и «Отчество» нам уже не нужны, скроем их, установив их свойство *Visible* в *False*. А новый столбец перетащим мышью наверх, его индекс будет равен 1 – вместо прежних Ф.И.О.

Нам придется написать код, который нужно вписать в событие **OnDrawColumnCell** сетки. Это событие наступает при прорисовке каждой ячейки столбца. Также имеется событие **OnDrawDataCell**, которое выполняет схожие функции, но оно оставлено для поддержки старых версий, и использовать его не желательно. Итак, выделяем сетку, генерируем событие *OnDrawColumnCell* и вписываем код:

```
    { ----- Прорисовка таблицы ----- }
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
     const Rect: TRect; DataCol: Integer; Column: TColumn;
     State: TGridDrawState);
var
   s: String;
begin
   //если это наш новый (пустой) столбец
```

```
    if Column.Index = 1 then begin
       if not Table1.FieldByName('Fam').IsNull then
          s:= Table1['Fam'] + ' ';
       if not Table1.FieldByName('Im').IsNull then
          s:= s + Copy(Table1['Im'], 1, 1) + '.';
       if not Table1.FieldByName('Ot').IsNull then
          s:= s + Copy(Table1['Ot'], 1, 1)+ '.';
       DBGrid1.Canvas.TextOut(Rect.Left + 2, Rect.Top + 2, s);
    end;
end;
```

Здесь мы вначале проверяем – наш ли это столбец (равен ли индекс 1)? Если наш, то в переменную s начинаем собирать нужный текст. При этом имеем в виду, что пользователь мог и не заполнить некоторые поля. Чтобы у нас не произошло ошибки, вначале убеждаемся, что поле не равно **Null** (то есть, текст есть). Если текст есть, добавляем его в переменную s. Причем если это имя или отчество, с помощью функции *Copy()* получаем только первую букву и добавляем к ней точку. Когда s сформирована, добавляем этот текст в наш столбец с помощью метода **TextOut()** свойства **Canvas** сетки. В метод передаются три параметра: координаты левой позиции, верхней позиции и сам текст.

Эти координаты мы берем из параметра события *OnDrawColumnCell* – **Rect**, который имеет такие свойства, как *Left* и *Top*, показывающие, соответственно, левую и верхнюю позиции текущей ячейки.

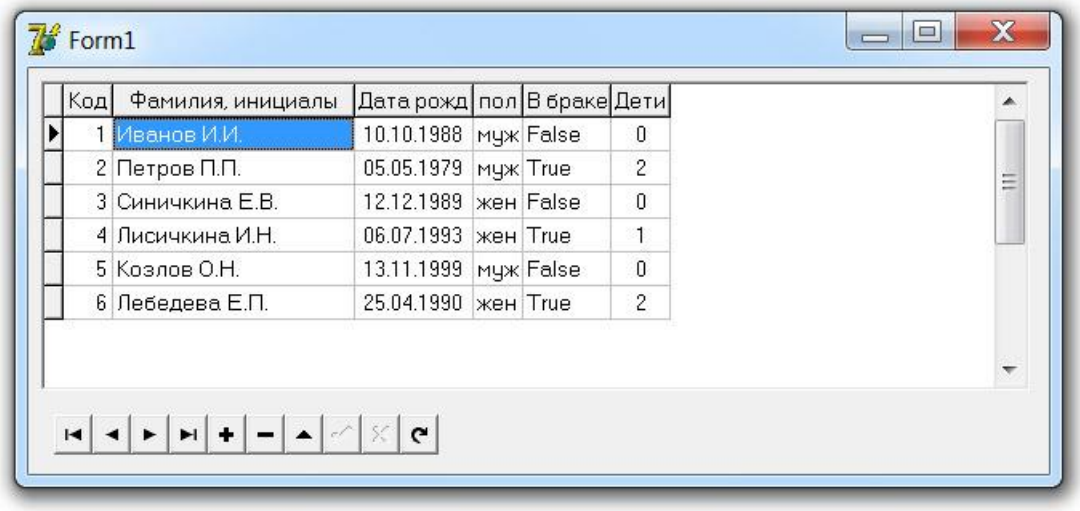


Рис. 8.5. Таблица с добавленным столбцом

Теперь о том, как использовать пустой столбец для вывода информации по требованию пользователя. Допустим, в сетке *DBGrid* нам нужна кнопка, нажатие на которую привело бы к выводу будет пустым. В свойстве **ButtonStyle** выберите значение **cbsEllipsis**. Это приведет к тому, что при попытке редактировать этот столбец образуется кнопка с тремя точками: сообщения об образовании сотрудника.

Создайте новый пустой столбец. Перетаскивать его не нужно, пусть будет последним. Свойство *Width* (ширина) установите в 20 пикселей. Название столбца (*Title.Caption*) вписывать не нужно, пусть будет пустым. В свойстве **ButtonStyle** выберите значение **cbsEllipsis**. Это приведет к тому, что при попытке редактировать этот столбец образуется кнопка с тремя точками:

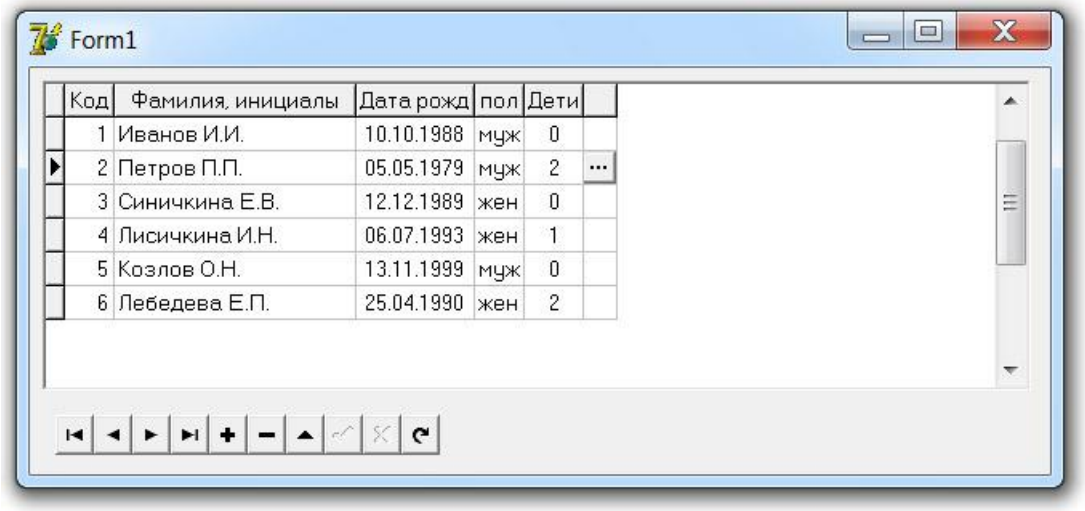


Рис. 8.6. Кнопка в пустом столбце

Нужный код пишется в событии **OnEditButtonClick()** сетки *DBGrid*, которое происходит всякий раз, когда пользователь нажимает на кнопку «…». Сгенерируем это событие и впишем только следующий код:

```
if Table1.FieldByName('Married').AsBoolean then begin
    if Table1.FieldByName('Pol').Value = 'муж' then
      ShowMessage('Женат')
    else
      ShowMessage('Замужем')
  end
  else
    if Table1.FieldByName('Pol').Value = 'муж' then
      ShowMessage('Холост')
    else
      ShowMessage('Незамужем')
```

Теперь, когда пользователь нажмёт на эту кнопку, ему будет выведено сообщение с текстом о семейном положении текущего сотрудника.

Список выбора в столбце

Для организации списков выбора служит компонент *ComboBox*. Однако сетка *DBGrid* позволяет устроить такой же список в одном из своих столбцов без использования каких-либо других компонентов. В нашем примере есть поле «Пол». Это текстовое поле из трех символов.

Откроем редактор столбцов сетки и выделим столбец «Пол». Обратите внимание на свойство **PickList** в Инспекторе объектов. Это свойство имеет

тип *TStrings*, то есть представляет собой набор строк, так же как и свойство *Items* у компонента *ComboBox*. Щелкнем дважды по *PickList*, чтобы открыть редактор, и впишем туда

муж

жен

– именно так, каждое значение на своей строке. Сохраним проект, скомпилируем его и попробуем редактировать этот столбец. При попытке редактирования в ячейке покажется похожий на *ComboBox* список, в котором можно будет выбрать одно из указанных значений:

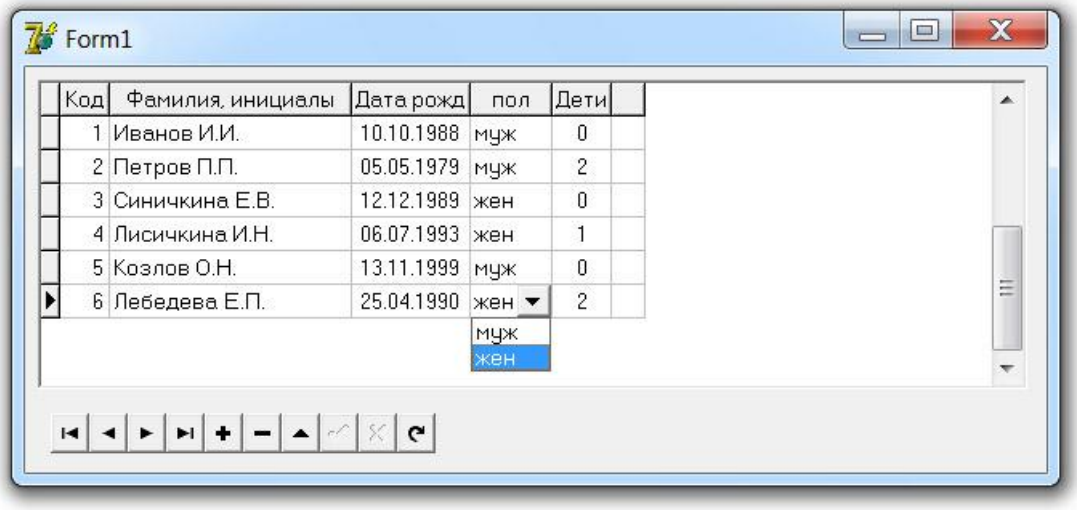


Рис. 8.7. Список выбора в столбце

Следует иметь в виду, что наличие такого списка не препятствует пользователю ввести какое-то иное значение. Этот список нужен не для контроля, а только для облегчения пользователю ввода данных. Если же вы не желаете, чтобы пользователь имел возможность вводить другие данные, контроль следует организовать иным способом.

Еще нужно заметить, что в практике программирования список чаще формируется во время работы программы, а строки списка берутся, как правило, из другой связанной таблицы. Добавить в список новую строку очень просто:

```
DBGrid1.Columns.Items[4].PickList.Add('абв');
```

Выделение отдельных строк

Очень часто в практике приходится выделять какие-то строки, изменяя их фон или цвет шрифта. Например, в бухгалтерии обычно выделяют строки, в которых значение меньше нуля. Допустим, ваша программа показывает клиентов, а какой-то столбец содержит их баланс на счету вашей компании.

Если этот баланс меньше 0, значит, клиент имеет задолженность перед вашей фирмой. Бухгалтеру будет очень удобно, если дебиторы (должники) будут выделяться в общем списке красным цветом.

Способ прорисовки данных в сетке *DBGrid* зависит от значения ее свойства **DefaultDrawing**. По умолчанию свойство равно *True*, то есть

данные прорисовываются автоматически. Если свойство содержит *False*, то прорисовку придется кодировать самостоятельно в свойствах *OnDrawColumnCell* или *OnDrawDataCell*, о которых уже упоминалось в этой лекции.

Если мы написали алгоритм прорисовки, но свойство *DefaultDrawing* содержит *True*, то вначале сетка заполнится данными автоматически, а затем будет выполнен наш алгоритм. Другими словами, прорисовка некоторых частей сетки будет выполнена дважды. Это не очень хорошо для быстродействия программы, однако нам придется поступать именно так: ведь мы не все строки и столбцы собираемся выводить другим способом, а только некоторые. Остальные будут заполнены данными по умолчанию.

Разберем этот метод подробней. Если найти его в справке Delphi, то увидим:

```
property OnDrawColumnCell: TDrawColumnCellEvent;
```

То есть, этот метод имеет тип *TDrawColumnCellEvent*. Описание типа такое:

```
type TDrawColumnCellEvent = procedure (Sender:TObject;
     const Rect:TRect; DataCol:Integer; Column:TColumn;
     State:TGridDrawState) of object;
```

Разберемся с параметрами.

Rect – координаты прорисовки.

DataCol – порядковый номер текущего столбца (начиная с 0).

Column – данные текущего столбца.

State – состояние ячейки. Может быть:

- **gdSelected** – ячейка выделена
- **gdFocused** – фокус ввода в ячейке
- **gdFixed** – ячейка – заголовок столбца.

У нас есть свой пример. Допустим, в нашем примере требуется, чтобы строки с холостыми сотрудниками выделялись красным цветом:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
    const Rect: TRect; DataCol: Integer; Column: TColumn;
    State: TGridDrawState);
var
  s: string;
  k: integer;
begin
  with DBGrid1.Canvas do begin
      // поле "Married" содержит истину?
    if (Table1['Married']) = False then begin
      if not (gdSelected in State) then begin
        // выводим все ячейки строки красным по белому:
        Font.Color:= clRed;
        FillRect(Rect);
```

```
      end
      else begin  // выделенная строка: жёлтым по синему
        Font.Color:= clYellow;
        Brush.Color := clBlue;
        FillRect(Rect);
      end;
    end;
    // если это наш новый (сборный) столбец
    if Column.Index = 1 then begin
      if not Table1.FieldByName('Fam').IsNull then
         s:= Table1['Fam'] + ' ';
      if not Table1.FieldByName('Im').IsNull then
         s:= s + Copy(Table1['Im'], 1, 1) + '.';
      if not Table1.FieldByName('Ot').IsNull then
         s:= s + Copy(Table1['Ot'], 1, 1)+ '.';
      DBGrid1.Canvas.TextOut(Rect.Left + 2, Rect.Top + 2, s);
    end
    else // все остальные столбцы, кроме пустого с кнопкой
      if ((Table1['Married']) = False) and (Column.Index < 9)
      then begin
        s:= Column.Field.AsString;
        case Column.Alignment of
           taLeftJustify: k:= Rect.Left + 2;
          taRightJustify: k:= Rect.Right - TextWidth(s) - 3;
            else { taCenter }
               k:= Rect.Left + (Rect.Right - Rect.Left) div 2
               - (TextWidth(s) div 2);
        end;
        // TextRect(Rect, k, Rect.Top + 2, s); { иногда это имеет
значение: Rect/Out}
        TextOut(k, Rect.Top + 2, s);
      end;
  end;
end;
```

8.2. Навигация по набору данных

Перемещение или навигация по записям набора данных может осуществляться несколькими путями. Например, в компоненте TDBGrid, который отображает сразу несколько записей набора данных, можно использовать клавиши вертикального перемещения курсора или вертикальную полосу прокрутки.

Но что делать, если на форме находятся только компоненты, отображающие одно поле только текущей записи набора данных (TDBEdit, TDBComboBox и т. д.)? Очевидно, что в этом случае на форме должны быть расположены дополнительные элементы управления, отвечающие за

перемещение по записям. Аналогично, ни один из таких компонентов не имеет встроенных средств для создания и удаления записей целиком.

Для решения указанных задач и предназначен компонент TDBNavigator, который представляет собой совокупность управляющих кнопок, выполняет операции навигации по набору данных и модификации записей целиком.

Компонент TDBNavigator при помощи свойства `DataSource` связывается с компонентом TDataSource и через него с набором данных. Такая схема позволяет обеспечить изменение текущих значений полей сразу во всех связанных с TDataSource компонентах отображения данных. Таким образом, TDBNavigator только дает команду на выполнение перемещения по набору данных или другой управляющей операции, а всю реальную работу выполняют компонент набора данных и компонент TDataSource. Компонентам отображения данных остается только принять новые данные от своих полей.

Компонент TDBNavigator содержит набор кнопок, каждая из которых отвечает за выполнение одной операции над набором данных. Всего имеется 10 кнопок, разработчик может оставить в наборе любое количество кнопок в любом сочетании. Видимостью кнопок управляет свойство `VisibleButtons`:

```
type
   TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast,
     nbInsert, nbDelete,nbEdit, nbPost, nbCancel, nbRefresh);
TButtonSet = set of TNavigateBtn;
property VisibleButtons: TButtonSet;
```

Каждый элемент типа TNavigateBtn представляет одну кнопку, их назначение описывается ниже:

`nbFirst` – перемещение на первую запись набора данных;

`nbPrior` – перемещение на предыдущую запись набора данных;

`nbNext` – перемещение на следующую запись набора данных;

`nbLast` – перемещение на последнюю запись набора данных;

`nbInsert` – вставка новой записи в текущей позиции набора данных;

`nbDelete` – удаление текущей записи, курсор перемешается на следующую запись;

`nbEdit` – набор данных переводится в режим редактирования;

`nbPost` – в базу данных переносятся все изменения в текущей записи;

`nbCancel` – все изменения в текущей записи отменяются;

`nbRefresh` – восстанавливаются первоначальные значения текущей записи, сделанные после последнего переноса изменений в базу данных.

Самой критичной к возможной потере данных вследствие ошибки является Операция удаления записи, поэтому при помощи свойства `ConfirmDelete` можно включить механизм контроля удаления. При каждом удалении записи нужно будет дать подтверждение выполняемой операции.

Нажатие любой кнопки можно эмулировать программно при помощи метода `BtnClick`.

В случае необходимости выполнения дополнительных действий при щелчке на любой кнопке можно воспользоваться обработчиками событий `BeforeAction` и `OnClick`, в которых параметр `Button` определяет нажатую кнопку.

8.3. Представление отдельных полей

Большинство компонентов отображения данных предназначено для представления данных из отдельных полей. Для этого все они имеют свойство `DataField`, которое указывает на требуемое поле набора данных.

В зависимости от типа данных поля могут использовать различные компоненты. Для большинства стандартных полей используются компоненты TDBText, TDBEdit, TDBComboBox, TDBListBox.

Данные в формате Memo отображаются компонентами TDBMemo и TDBRichEdit.

Для показа изображений предназначен компонент TDBImage.

Компонент *TDBText*

Этот компонент представляет собой статический текст, который отображает текущее значение некоторого поля связанного набора данных. При этом данные можно просматривать в режиме "только для чтения".

Непосредственным предком компонента является класс TCustomLabel, поэтому он очень похож на компонент TLabel.

При использовании компонента следует обратить внимание на возможную длину отображаемых данных. Для предотвращения обрезания текста можно использовать свойства `AutoSize` и `WordWrap`.

Компонент *TDBEdit*

Компонент представляет собой стандартный однострочный текстовый редактор, в котором отображаются и изменяются данные из поля связанного набора данных.

Прямой предок компонента – класс TCustomMaskEdit, который также является прямым предком компонента TEdit.

Компонент может осуществлять проверку редактируемых данных по заданной для поля маске. Непосредственно для редактора задать маску нельзя, т. к. содержащее маску свойство `EditMask` в классе TCustomMaskEdit является защищенным, а в TDBEdit не перекрыто. Тем не менее механизм контроля полностью унаследован. Саму же маску можно задать в связанном с редактором поле. Объект TField имеет собственное свойство `EditMask`, которое и используется при проверке данных в редакторе.

Проверка редактируемого текста на соответствие маске осуществляется методом `ValidateEdit` после каждого введенного или измененного символа. В случае ошибки генерируется исключение `ValidateError` и курсор устанавливается на первый ошибочный символ.

В компоненте можно использовать буфер обмена. Это делается средствами операционной системы пользователем или программно при помощи методов `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard`.

Компонент *TDBCheckBox*

Компонент представляет собой почти полный аналог обычного флажка (компонент TCheckBox) и предназначен для отображения и редактирования любых данных, которые могут иметь только два значения. Это может быть логический тип данных или любые строковые значения, но поле может принимать значения только из двух строк.

Предопределенные значения задаются свойствами `ValueChecked` и `ValueUnchecked`. По умолчанию они имеют значения True и False. Этим свойствам можно также присваивать любые строковые значения, причем одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой.

Включение флажка происходит, если значение поля набора данных совпадает со значением свойства `ValueChecked` (единственным или любым из списка). Если же флажок включил пользователь, то значение поля данных приравнивается к единственному или первому в списке значению свойства `ValueChecked`.

Аналогичные действия происходят и со свойством `ValueUnchecked`.

Компонент *TDBRadioGroup*

Компонент представляет собой стандартную группу переключателей, состояние которых зависит от значений поля связанного набора данных. В поле можно передавать фиксированные значения, связанные с отдельными переключателями в группе.

Если текущее значение связанного поля соответствует значению какого-либо переключателя, то он включается. Если пользователь включает другой переключатель, то связанное с переключателем значение заносится в поле.

Возможные значения, на которые должны реагировать переключатели в группе, заносятся в свойство `Values` при помощи специального редактора в Инспекторе объектов или программно посредством методов класса TStrings. Каждому элементу свойства `Values` соответствует один переключатель (порядок следования сохраняется).

Свойство `Items` содержит список поясняющих надписей для переключателей группы. Если для какого-либо переключателя нет заданного

значения, но есть поясняющий текст, то такой переключатель включается при совпадении значения связанного поля с поясняющим текстом.

Текущее значение связанного поля содержится в поле `Value`.

Компонент *TDBListBox*

Компонент отображает текущее значение связанного с ним поля набора данных и позволяет изменить его на любое фиксированное из списка. Функционально компонент ничем не отличается от компонента TListBox. Значение поля должно совпадать с одним из элементов списка.

Специальных методов компонент не содержит.

Компонент *TDBComboBox*

Компонент отображает текущее значение связанного с ним поля набора данных в строке редактирования, при этом значение поля должно совпадать с одним из элементов разворачивающегося списка. Текущее значение можно изменить на любое фиксированное из списка компонента. Функционально компонент ничем не отличается от компонента TComboBox представляющего собой комбинированный список.

Компонент может работать в пяти различных стилях, которые определяются свойством `Style`.

Специальных методов компонент не содержит.

Компонент *TDBMemo*

Компонент представляет собой обычное поле редактирования, к которому подключается поле с типом данных Memo или BLOB. Основное его преимущество – возможность одновременного просмотра и редактирования нескольких строк переменной длины. Компонент может отображать только строки, которые целиком видны по высоте.

В компоненте можно использовать буфер обмена при помощи стандартных средств операционной системы или унаследованными от предка TCustomMemo методами `CopyToClipBoard`, `CutToClipBoard`, `PasteFromClipBoard`.

Для ускорения навигации по набору данных при отображении полей типа BLOB можно использовать свойство `AutoDisplay`. При значении True любое новое значение поля автоматически отображается в компоненте. При значении False новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Метод `LoadMemo` используется автоматически при загрузке значения поля, если свойство `AutoDisplay = False`.

Поведением компонента при работе со слишком длинными строками можно управлять при помощи свойства `WordWrap`. При значении True слишком длинная строка сдвигается влево при перемещении текстового

курсора за правую границу компонента. При значении False остаток длинной строки переносится на новую строку, при этом реально новая строка в данных не создается.

Компонент *TDBImage*

Компонент предназначен для просмотра изображений, хранящихся в базах данных в графическом формате, работает только с форматом BMP.

Редактировать изображения можно только в каком-либо графическом редакторе, перенося исходное и измененное изображение при помощи буфера обмена. Это делается средствами операционной системы пользователем или программно при помощи методов `CopyToClipboard`, `CutToClipboard`, `PasteFromClipboard`.

Можно использовать универсальный способ загрузки файла в поле типа BLOB с помощью метода `LoadFromFile`, например:

```
(Form1.Table1.FieldByName('Photo') as
      TBLOBField).LoadFromFile(OpenDialog1.FileName)
```

Аналогично можно выгрузить в файл содержимое поля BLOB методом `SaveToFile`:

```
(Form1.Table1.FieldByName('Photo') as
      TBLOBField).SaveToFile(SaveDialog1.FileName)
```

Визуализация изображения осуществляется при помощи свойства `Picture`, которое представляет собой экземпляр класса TPicture.

Также можно полностью заменить существующее изображение или сохранить новое в новой записи набора данных. Для этого используются методы свойства `Picture`.

Свойство `AutoDisplay` позволяет управлять процессом загрузки новых изображений из набора данных в компонент. При значении True любое новое значение поля автоматически отображается в компоненте. При значении False новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Для ускорения просмотра изображений можно применять свойство `QuickDraw`, которое задает используемую изображением палитру. При значении True применяется стандартная системная палитра. В результате уменьшается время загрузки изображения, но может ухудшиться и качество изображения, в некоторых случаях до полного искажения. При значении False используется собственная палитра изображения, и процесс загрузки замедляется.

Компонент *TDBRichEdit*

Компонент предоставляет возможности полноценного текстового редактора для просмотра и изменения текстовых данных, хранящихся в

связанном поле набора данных. Поле должно содержать информацию о форматировании текста.

Внешне компонент ничем не отличается от поля редактирования, поэтому о реализации доступа к гораздо более богатым возможностям редактора через интерфейс пользователя должен позаботиться разработчик. Для этого можно использовать дополнительные элементы управления.

8.4. Синхронный просмотр данных

При разработке приложений для работы с базами данных часто возникает необходимость в связывании двух наборов данных по ключевому полю. Например, в таблице Orders (содержит данные о заказах) демонстрационной базы данных DBDEMOS имеется поле `CustNo`, которое содержит идентификационный номер покупателя. Под этим же номером в таблице Customers хранится информация о покупателе (адрес, телефон, реквизиты и т. д.). При разработке пользовательского интерфейса приложения баз данных необходимо, чтобы при просмотре перечня заказов в форме приложения отображались не идентификационные номера покупателей, а их параметры.

Таким образом, в наборе данных заказов вместо поля номера покупателя должно появиться поле имени покупателя из таблицы Customers. Механизм связывания полей из различных наборов данных по ключевому полю называется *синхронным просмотром.* В рассмотренном примере ключевым является поле `CustNo` из таблицы Customers, а выбор конкретного наименования производится по совпадению значений ключевого поля и заменяемого поля из исходного набора данных – Orders. Причем необходимо, чтобы в таблице Customers поле `CustNo` было уникальным (составляло первичный или вторичный ключ).

Таблицу, в которой расположено поле, значения которого замещаются на синхронные, будем называть *исходной таблицей* (это таблица Orders).

Таблицу, содержащую ключевое поле и поле данных для синхронного просмотра, будем называть *таблицей синхронного просмотра* (таблица Customers).

В Delphi механизм синхронного просмотра реализован на уровне отдельных полей и компонентов.

В наборе данных динамически можно создать специальное поле синхронного просмотра, которое будет автоматически замещать одно значение другим в зависимости от значения ключевого поля. Такое поле можно связать с любым рассмотренным выше компонентом отображения данных.

Помимо простого синхронного просмотра данных может возникнуть задача редактирования данных в аналогичной ситуации. Для этого предназначены специальные компоненты синхронного просмотра данных, которые позволяют, например, выбирать покупателя из списка, а изменится

при этом номер покупателя в наборе данных заказов. Использование таких компонентов делает пользовательский интерфейс значительно более удобным и наглядным. В VLC Delphi есть два таких компонента: TDBLookupListBox и TDBLookupComboBox.

8.4.1. Механизм синхронного просмотра

Непосредственным предком компонентов синхронного просмотра данных является класс TDBLookupControl, который инкапсулирует список значений для просмотра и сам механизм синхронного просмотра.

Как и в любом другом компоненте отображения данных, в компонентах синхронного просмотра должны присутствовать средства связывания с требуемым полем некоторого набора данных (табл. 8.2). Это уже известные свойства: `DataSource` – применяется для задания набора данных через компонент TDataSource и `DataField` – для определения требуемого поля набора данных. Для синхронного просмотра следует выбирать такое поле, значения которого не дают пользователю полной информации об объекте и совпадают с ключевым полем в таблице синхронного просмотра. Название этого поля может не совпадать с названием ключевого поля, но типы данных должны быть одинаковыми.

Теперь необходимо задать таблицу синхронного просмотра, ключевое поле и поле синхронного просмотра.

Набор данных, содержащий указанные поля, определяется через соответствующий компонент TDataSource в свойстве `ListSource`.

Ключевое поле задается свойством `KeyField`. Во время работы компонента в свойстве `KeyValue` содержится текущее значение, которое связывает между собой два набора данных.

Поле синхронного просмотра определяется свойством `ListField`. Здесь можно задавать сразу несколько полей, которые будут отображаться в компоненте синхронного просмотра. Названия полей разделяются точкой с запятой. Если свойство не определено, то в компоненте будут отображаться значения ключевого поля.

Таблица 8.2. Основные свойства, включающие механизм синхронного просмотра

Объявление	Описание
`property KeyField: string;`	Ключевое поле таблицы синхронного просмотра
`property KeyValue: Variant;`	Текущее значение ключевого поля
`property ListField: string;`	Поле или список полей синхронного про смотра в таблице синхронного просмотра
`property ListFieldIndex: Integer;`	Номер основного поля синхронного просмотра (используется, когда свойство `ListField` содержит список полей)
`property ListSource:`	Указывает на компонент TDataSource,

`TDataSource;`	связанный с таблицей синхронного просмотра
`property NullValueKey: TShortCut;`	Определяет комбинацию клавиш, нажатие которых задает нулевое значение поля

В качестве примера рассмотрим приложение DemoLookup (рис. 8.8), в котором с набором данных таблицы Orders из базы данных DBDEMOS связаны компоненты TDBGrid и TDBLookupComboBox. Во втором компоненте при перемещении по записям набора данных отображается имя покупателя, оформившего текущий заказ. Параллельно для ознакомления задействован и компонент LookupListBox – с такими же настройками.

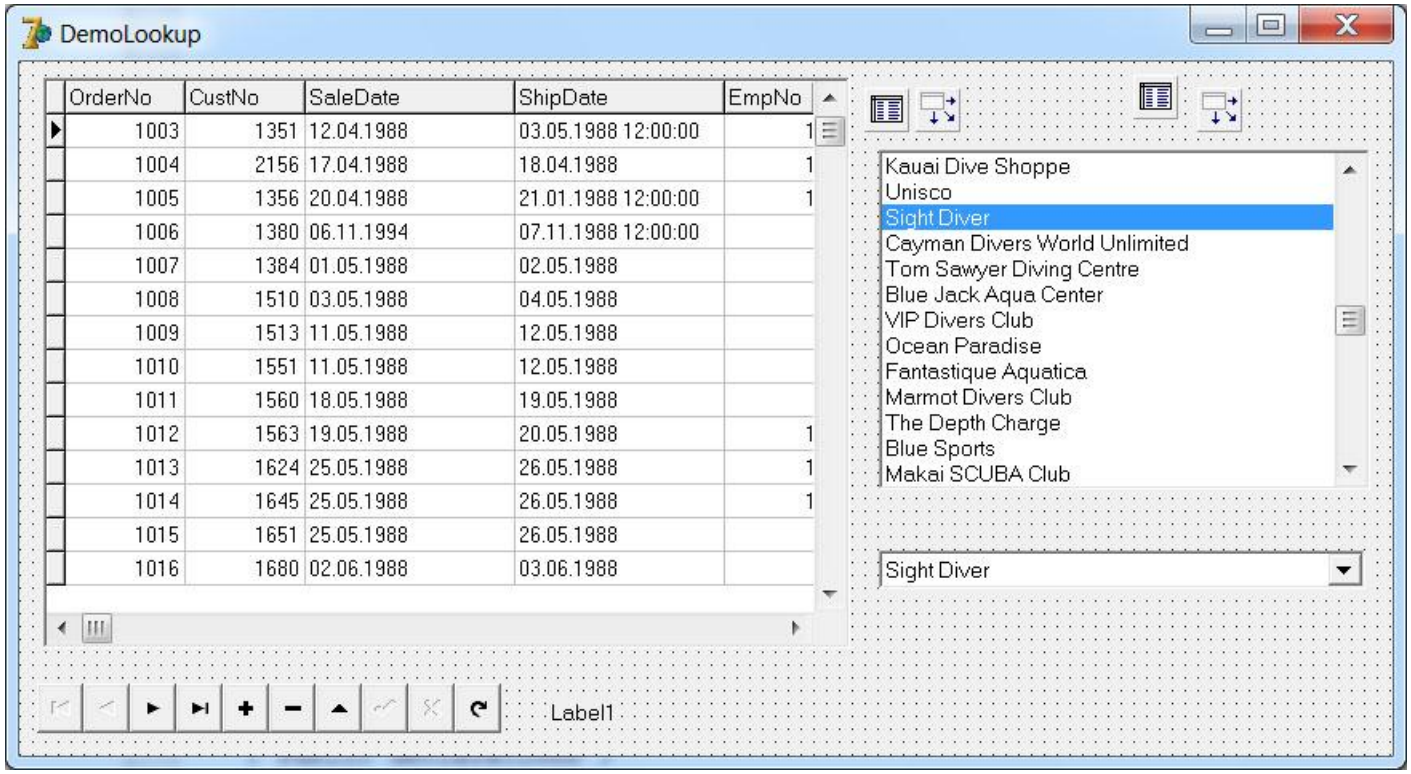


Рис. 8.8. Главная форма проекта DemoLookup

Настройки ключевых свойств компонентов `DBLookupComboBox` и `DBLookupListBox` следующие.

Свойство `ListSource` указывает на компонент `CustSource` типа TDataSource, который связан с набором данных синхронного просмотра `CustTable`. Свойство `ListField` указывает на поле `Company`, все значения которого доступны в списке компонента.

Свойство `KeyField` указывает на внешний ключ – поле `CustNo`, по которому осуществляется связь с основной таблицей.

Окончательно для связи с основной таблицей:

```
DataSource = OrdSurce,
DataField = CustNo.
```

Рассмотрим основные свойства и методы самих компонентов отображения данных, за исключением тех, которые представлены в табл. 8.2 и полностью идентичны для двух компонентов.

Компонент *TDBLookupListBox*

Компонент представляет собой список значений поля синхронного просмотра для поля, заданного свойством `DataField`, из набора данных `DataSource`. Его основное назначение – автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом TDBListBox.

Компонент *TDBLookupComboBox*

Компонент представляет собой комбинированный список значений поля синхронного просмотра для поля, заданного свойством `DataField`, из набора данных `DataSource`. Его основное назначение – автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом TDBComboBox.

Резюме

Компоненты отображения данных играют важную роль при создании интерфейсов приложений баз данных. Разнообразие предлагаемых элементов управления позволяет решать любые задачи по организации взаимодействия пользователя с базой данных. Все они взаимодействуют с набором данных через компонент TDataSource.

9. Введение в язык SQL

Язык SQL был разработан фирмой IBM для своей системы управления базами данных DB/2. Потом этот язык стал общепризнанным стандартом при работе с базами данных. Фактически, взаимодействие пользователя с современными СУБД осуществляется только при помощи этого языка. На данный момент существует несколько стандартов, описывающих этот язык.

По своей структуре язык делится на три части. В разделе DDL (Data Definition Language) собраны команды, которые задают структуру тех или иных объектов базы данных. К ним относятся таблицы, представления, индексы, домены и прочие структурные сущности. Раздел DML (Data Manipulation Language) предоставляет разработчику набор команд, позволяющих манипулировать данными. С их помощью можно производить выборки данных, удалять, добавлять и изменять записи. Раздел DCL (Data Control Language) состоит из средств, которые определяют права доступа к объектам базы данных. Например, разрешают доступ к данным или запрещают его.

Рис. 9.1. Взаимодействие с базой данных при помощи SQL

Этот язык предоставил пользователю простой и универсальный аппарат для доступа к данным и осуществления с ними различных операций. На рис. 9.1 показана схема, отражающая принцип использования SQL. Пользователь передает запрос интерпретатору, который, в свою очередь, возвращает представление, таблицу или курсор. Эти объекты находятся на так называемом виртуальном уровне и формируются только по запросу. Но они взаимодействуют с реальным уровнем, то есть с таблицами баз данных, на основе которых происходит формирование виртуальных объектов. Конечно, данное разделение является условным, но оно хорошо отражает идеологию SQL.

9.1. Типы данных

Каждое значение, хранящееся в базе данных, имеет некий тип. Язык SQL, конечно, поддерживает типизацию значений. Основные типы данных приведены в табл. 9.1.

Таблица 9.1. Типы данных в SQL

- **TEXT** – Поля этого типа могут хранить строку символов неограниченной длины
- **CHAR** – В полях этого типа хранится строка текста ограниченной длины. В аргументе указывается максимальный размер строки
- **DEC** – Тип задает десятичное число. Аргумент состоит из двух частей – разрядности и точности. Разрядность определяет количество значащих цифр, из которых состоит число. Точность определяет число цифр после запятой. Точность не может быть больше разрядности. Если точность равна нулю, то число является целым
- **NUMERIC** – Тип является двойником типа DEC
- **INT** – Тип предназначен для хранения целых чисел. Возможные значения находятся в промежутке от –2 147 483 648 до 2 147 483 648
- **SMALLINT** – Тип предназначен для хранения целых чисел. Возможные значения находятся в промежутке от –32 768 до 32 768
- **FLOAT** – Тип предназначен для хранения чисел с плавающей точкой. В аргументе указывается число, определяющее минимальную точность
- **REAL** – Тип предназначен для хранения чисел с плавающей точкой. Этот тип отличается от типа FLOAT только точностью
- **DOUBLE** – Тип отличается от REAL повышенной точностью
- **DATE** – Тип предназначен для хранения даты
- **TIME** – Тип предназначен для хранения времени

Кроме этого, большинство реализаций также имеют нестандартный тип **VARCHAR** (n). Он описывает текстовую строку, которая может иметь произвольную длину до определенного конкретной реализацией SQL максимума (в Oracle – до 2000 символов). В отличие от типа CHAR, в этом случае при вводе текстовой константы, фактическая длина которой меньше

заданной, не производится ее дополнение пробелами до заданного максимального значения.

9.2. Манипуляции данными с помощью запросов на языке SQL

9.2.1. Оператор *SELECT*

Оператор SELECT является фактически основным и самым сложным оператором SQL. Он предназначен для выборки данных из таблиц, именно он и реализует одно их основных назначений базы данных – предоставлять по запросу информацию из базы данных пользователю. Поскольку язык SQL является не процедурным, а декларативным, оператор SELECT предназначен для того, чтобы описать, какие данные должны быть получены из базы данных в результате выполнения запроса.

Полное описание синтаксиса оператора SELECT является достаточно сложным. Для начального ознакомления рассмотрим упрощенную форму оператора:

```
SELECT  [DISTINCT] (<список полей> | *)
    FROM  <список таблиц>
    [WHERE  <условие отбора строк>]
    [ORDER BY  <список полей для сортировки>]
    [GROUP BY  <список полей для  группировки>
    [HAVING  <условия отбора групп>]
    [UNION  <вложенная инструкция SELECT>]
```

В таком результирующем наборе данных могут быть разрешены или запрещены повторяющиеся записи. Этим режимом управляет описатель DISTINCT. Если он отсутствует, то в наборе данных разрешаются повторяющиеся записи.

В инструкцию SELECT обязательно включается список полей и операнд FROM, остальные операнды могут отсутствовать. В списке операнда FROM перечисляются имена таблиц, для которых отбираются записи. Список должен содержать как минимум одну таблицу.

Список полей определяет состав полей результирующего набора данных, эти поля могут принадлежать разным таблицам. В списке должно быть задано хотя бы одно поле. Если в набор данных требуется включить все поля таблицы (таблиц), то вместо перечисления имён можно указать символ *. Если список содержит поля нескольких таблиц, то для указания принадлежности поля к таблице используют составное имя, включающее в себя имя таблицы и имя поля, разделённые точкой.

Операнд WHERE задаёт условия отбора, которым должны удовлетворять записи в результирующем наборе данных. Выражение, описывающее условия отбора, является логическим. Его элементами могут быть имена

полей, операции сравнения, арифметические и логические операции, скобки, специальные функции LIKE, NULL, IN и др.

Операнд GROUP BY позволяет выделять группы записей в результирующем наборе данных. Группой являются записи с одинаковыми значениями в полях, перечисленных за операндом GROUP BY. Выделение групп требуется для выполнения групповых операций над записями, например, для определения количества какого-либо товара на складе.

Операнд HAVING действует совместно с операндом GROUP BY и используется для отбора записей внутри группы. Правила записи условий группирования аналогичны правилам формирования условия отбора в операнде WHERE.

Операнд ORDER BY содержит список полей, определяющих порядок сортировки записей результирующего набора данных. По умолчанию сортировка по каждому полю выполняется в порядке возрастания значений; если необходимо задать для поля сортировку по убыванию, то после имени этого поля указывается описатель DESC.

Инструкции SELECT могут иметь сложную структуру и быть вложенными друг в друга. Для объединения инструкций используется операнд UNION, в котором располагается вложенная инструкция SELECT. Результирующий набор данных представляют записи, отобранные с учётом выполнения условий отбора операндами WHERE обеих инструкций.

Инструкция SELECT используется также внутри других инструкций, например, инструкций модификации записей, обеспечивая для их выполнения требуемый отбор записей.

Использование оператора IN

Оператор **IN** определяет массив значений, в который может входить или не входить значение поля данной записи. Например, необходимо выбрать сотрудников с заработной платой 40 000, 55 500 и 25 000. Запрос потребуется переработать:

```
SELECT LastName, FirstName, Salary FROM Employee
WHERE Salary = 40000 or Salary = 55500 or Salary = 25000
```

Однако этот же запрос можно написать в более короткой и красивой форме при помощи оператора **IN**:

```
SELECT LastName, FirstName, Salary FROM Employee
WHERE Salary IN (40000, 55500, 25000)
```

В качестве аргументов оператору **IN** были переданы значения полей, по которым производился отбор записей.

Оператор **IN** может использоваться и для поиска символьных значений. Предположим, нам необходимо выяснить названия компаний, находящихся в городах Christiansted, Grand Cayman и St.Thomas. Эти данные содержаться в таблице Customer. Запрос снова понадобится немного изменить:

```
SELECT Company, City FROM Customer
WHERE City IN ('Christiansted', 'Grand Cayman', 'St.Thomas')
```

Значения поля City являются текстовыми, поэтому они были заключены в апострофы.

Использование оператора BETWEEN

Оператор **BETWEEN** используется для указания диапазона значений, которые используются для установки условия отбора записей. Этот оператор чувствителен к порядку перечисления параметров, определяющих границы диапазона. В качестве примера можно привести простой запрос:

```
SELECT CustomerID, EmployeeID, ShipName FROM Orders
WHERE EmployeeID BETWEEN 3 AND 5
```

В результате выполнения запроса были выбраны записи, значения поля EmployeeID, которых находятся в промежутке от трех до пяти включительно.

Следующий пример показывает, как можно выбрать номера заказов, сделанных в 2018 году:

```
SELECT OrderID, OrderDate, ShipName FROM Orders
WHERE OrderDate BETWEEN '01.01.2018' AND '31.12.2018'
```

Допустим, требования изменились. Теперь необходимо выбрать те номера заказов, которые не попадают в указанный промежуток времени и вес груза в которых составляет более ста единиц. В этом случае запрос будет выглядеть иначе:

```
SELECT OrderID, OrderDate, ShipName, Freight FROM Orders
WHERE OrderDate NOT BETWEEN '01.01.2018' AND '31.12.2018'
      AND Freight > 100
```

Использование оператора LIKE

Оператор **LIKE** используется для выбора всех записей, в которые входит подстрока, указанная в качестве параметра. В качестве условия оператор также принимает специальные символы. Символ подчеркивания заменяет любой одиночный символ, а знак процента обозначает любое количество символов.

Предположим, необходимо выбрать компанию, в названии которой не хватает нескольких букв. В этом случае название можно обозначить как S?mons?bistro.

Соответствующий запрос будет использовать указанный оператор **LIKE**:

```
SELECT CompanyName, ContactName FROM Customers
WHERE CompanyName LIKE 'S_mons_bistro'
```

Можно составить запрос, в котором будет производиться поиск некоей подстроки, входящей в запись. Предположим, что необходимо найти все компании, в названиях которых встречается последовательность символов ‘ric’.

Задачу решает несложный запрос

```
SELECT CompanyName, ContactName FROM Customers
WHERE CompanyName LIKE '%ric%'
```

Можно расширить условия отбора данных. Предположим, что необходимо найти все компании, в названии которых встречается сочетание символов ‘r?c’, то есть символ в середине подстроки неизвестен.

```
SELECT CompanyName, ContactName FROM Customers
WHERE CompanyName LIKE '%r_c%'
```

Вычисляемые поля

Автоматическое вычисление значений полей довольно часто применяется в самых разнообразных запросах. Пример соответствующего запроса выглядит довольно просто:

```
SELECT OnHand, OnOrder, (OnHand*OnOrder) AS Произведение,
      (OnHand+OnOrder) AS Сумма
FROM Parts
```

Результат выполнения этого запроса приведен в табл. 9.2. В данном примере производится перемножение и суммирование значений полей OnHand и OnOrder.

Те же действия с полями могут быть произведены с использованием числовых констант. Оператор AS присваивает данному полю другое имя, которое будет использовано в результирующем наборе.

Таблица 9.2. Выборка с вычисляемыми полями

OnHand	OnOrder	Произведение	Сумма
24	16	384	40
5	3	15	8
165	216	35 640	381
98	88	8624	186
75	70	5250	145

Запрос SELECT может также включать в себя числовые и текстовые константы.

В качестве примера можно привести следующий запрос:

```
SELECT OnHand, OnOrder, 'MUL', (OnHand + 1) AS Плюс,
      'SUB', (OnHand – 1) AS Минус
FROM Parts
```

В кавычках указаны текстовые константы, которые будут включены в результирующую таблицу в качестве значений соответствующих полей.

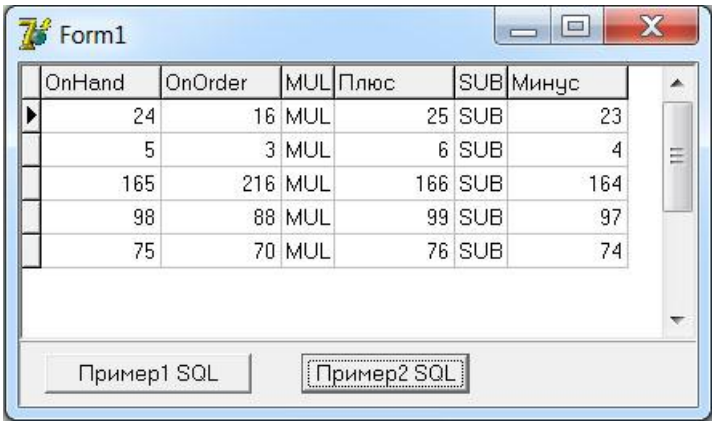


Рис. 9.2. Вычисляемые поля с текстом

Агрегатные функции

В некоторых случаях требуется в самом запросе произвести вычисление значений полей, получить количество найденных записей, произвести поиск максимального значения поля или выполнить иную вычислительную работу. Функции, реализующие эти возможности, называются *агрегатными.* Агрегатные функции возвращают одно значение для всего поля таблицы. Список агрегатных функций приведен ниже:

- Оператор **COUNT** возвращает количество записей, удовлетворяющих условию запроса.
- Оператор **SUM** суммирует значения записей поля.
- Оператор **AVG** вычисляет среднее значение записей поля.
- Оператор **MAX** возвращает наибольшее значение данного поля.
- Оператор **MIN** возвращает наименьшее значение данного ноля.

Агрегатные функции используются подобно именам полей в запросе, а настоящие имена полей передаются им как аргументы. С операторами SUM и AVG могут использоваться только числовые поля. С операторами COUNT, MAX и MIN могут использоваться числовые и символьные поля. В случае применения функций MAX и MIN к символьным полям их значения будут транслированы в ASCII-код.

Минимальному значению функции будет соответствовать символ алфавита, находящийся ближе к его началу, максимальному – находящийся ближе к концу.

Ниже приведен запрос, выбирающий из таблицы Orders среднее значение веса груза из поля Freight, минимальное значение веса груза, максимальное значение веса груза, его суммарное значение и количество грузов, вес которых составляет более трехсот единиц.

```
SELECT AVG(Freight) AS Srednee, MIN(Freight) AS Minimum,
   MAX(Freight) AS Maksimum, SUM(Freight) AS Summarnoe,
   COUNT(Freight) AS Kolichestvo
FROM Orders
WHERE Freight > 300
```

Функция COUNT производит подсчет всех записей. Для того чтобы исключить повторы, следует использовать оператор DISTINCT. Этот оператор располагается перед названием поля, внутри функции COUNT. Запрос, демонстрирующий этот механизм, показан ниже:

```
SELECT COUNT(DISTINCT City) AS KolvoCity FROM Customers
```

Для исключения повторов при использовании функций AVG и SUM тоже может быть использован этот оператор.

Оператор GROUP BY используется для определения полей, к которым могут применяться агрегатные функции. В случае, если этот оператор явно не указан, все поля, указанные в выражении SELECT, трактуются как аргументы агрегатных функций. Поля, указанные в качестве параметров оператора GROUP BY, становятся группирующими. Все записи результирующего набора, имеющие одинаковые значения группирующих полей, образуют единую группу. Далее к каждой такой группе будет применена агрегатная функция. Фактически, оператор GROUP BY дает возможность объединять поля и агрегатные функции в едином запросе.

Иллюстрирует вышесказанное запрос, отыскивающий города, в которых расположены фирмы, количество этих городов и максимальное значение почтового индекса для фирмы, расположенной в данном городе:

```
SELECT City, COUNT (*) AS Количество, MAX (PostalCode)
AS Почтовый_индекс FROM Customers
GROUP BY City
```

Легко заметить, что поле City не входит в агрегатную функцию в качестве параметра, поэтому оно было объявлено с использованием оператора GROUP BY.

В ходе выполнения запроса были выбраны города, и для каждого города было подсчитано количество вхождений. Результат выполнения запроса приведен в табл. 9.3.

Таблица 9.3. Использование оператора GROUP BY

City	Количество	Почтовый_индекс
Mexico D.F.	5	5033
Rio de Janeiro	3	05454-876
Sao Paulo	4	05634-030
Buenos Aires	3	1010
Leipzig	1	4179

Этот пример можно усложнить. Можно создать запрос, который получает только те города, которые повторяются в таблице больше двух раз, и при этом в конечный результат не должен включаться город Buenos Aires. Оператор WHERE в данном случае использовать не получится, так как он работает только с отдельными записями, а не с массивами. Придется использовать оператор HAVING, который является аналогом оператора

WHERE, но может работать с агрегатными функциями. Сам запрос будет довольно сильно изменен:

```
SELECT City, COUNT (*) AS Количество, MAX (PostalCode)
        AS Почтовый_индекс
FROM Customers Where City <> 'Buenos Aires'
GROUP BY City
HAVING COUNT (*) >=3
```

Результат выполнения запроса приведен в табл. 9.4.

Таблица 9.4. Использование оператора HAVING

Город	Количество	Почтовый_индекс
London	6	WX3 6FW
Madrid	3	28034
Mexico D.F.	5	5033
Rio de Janeiro	3	05454-876
Sao Paulo	4	05634-030

Упорядочивание записей

Оператор ORDER BY используется для упорядочивания записей результирующего набора данных. Записи сортируются в соответствии с порядком следования полей и их значений. Если сортировка будет производиться по возрастанию, то следует использовать параметр ASC. Для сортировки по убыванию используется параметр DESC. В качестве примера можно привести несложный SQL-запрос

```
SELECT CompanyName, ContactName, City FROM Customers
ORDER BY City
```

Сортировка записей производится по полю City.

К созданному запросу можно добавить сортировку по количеству городов в порядке убывания записей:

```
SELECT City, COUNT(*) AS Количество, MAX (PostalCode)
       AS Почтовый_индекс FROM
Customers Where City <> 'Buenos Aires'
GROUP BY City
HAVING COUNT(*)>=3
ORDER BY Количество DESC
```

Нужно обратить внимание на то, что в качестве аргумента параметра ORDER BY было использовано название поля, так как его значения являются результатом агрегатной функции COUNT. Для включения сортировки по убыванию был указан параметр DESC, расположенный после названия поля. В табл. 9.5 приведен результат выполнения запроса.

Таблица 9.5. Использование оператора ORDER BY с указанием порядка сортировки записей

City	Количество	Почтовый_индекс
London	6	WX3 6FW
Mexico D.F.	5	5033
Sao Paulo	4	05634-030
Rio de Janeiro	3	05454-876
Madrid	3	28034

Многотабличные запросы

Как правило, при проектировании таблиц в них стараются включать только те поля, которые однозначно связаны с данной сущностью. Это делается для того, чтобы было проще модифицировать базу данных и поддерживать ее целостность. В связи с этим возникает необходимость создания многотабличных запросов, то есть запросов, использующих для формирования результата данных из нескольких таблиц. В этой главе будут рассмотрены структура подобных запросов и методы их создания.

Объединение таблиц

Во многих случаях требуется получать данные из нескольких таблиц и сводить их в одну результирующую таблицу. Такая операция называется *объединением таблиц.* При объединении производится связывание полей разных таблиц. При этом между полями устанавливаются связи за счет использования соответствующих справочных значений.

После оператора FROM таблицы перечисляются через запятую. Полное имя поля фактически состоит из имени таблицы и самого поля, разделенных точкой. Если все столбцы объединяемых таблиц имеют разные имена, то к ним можно обращаться напрямую, не указывая имя таблицы, которой они принадлежат.

Для рассмотрения принципов работы многотабличных запросов следует создать простой пример. Предположим, необходимо узнать названия судов с грузом, которые отправила каждая компания, вес отправленного груза, дату его отправки, контактное лицо и его телефон

```
SELECT Orders.ShipName AS Судно, Orders.Freight
    AS Вес_груза, Orders.OrderDate AS Дата_Отправки,
    Customers.ContactName, Customers.Phone
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
```

При выполнении запроса были выбраны поля только тех записей, у которых значения поля CustomerID совпадали. При помощи этого поля были объединены и связаны две таблицы. Результат выполнения запроса приведен в табл. 9.6.

Таблица 9.6. Объединение таблиц Orders и Customers

Судно	Вес_груза	Дата_Отправки	ContactName	Phone
Vins et alcools Chevalier	32,38	04.07.1996	Paul Henriot	26.47.15.10
Toms Spezialitaten	11,61	05.07.1996	Karin Josephs	0251-031259
Hanari Carnes	65,83	08.07.1996	Mario Pontes	(21) 555-0091
Victuailles en stock	41,34	08.07.1996	Mary Saveley	78.32.54.86
Supremes delices	51,3	09.07.1996	Pascale Cartrain	(071) 23 67 22 20

Этот запрос можно усложнить. Предположим, что необходимо получить информацию именно о тех судах, груз которых весил более 500 тонн и был отправлен в период с 17.03.1998 по 17.04.1998:

```
SELECT Orders.ShipName AS Судно, Orders.Freight
     AS Вес_груза,  Orders.OrderDate AS Дата_Отправки,
     Customers.ContactName,  Customers.Phone
FROM Customers, Orders
WHERE Customers.CustomerID = Orders.CustomerID
     AND Freight > 500 AND Orders.OrderDate BETWEEN
     '17.03.1998' AND '17.04.1998'
```

Результат выполнения запроса представлен в табл. 9.7.

Таблица 9.7. Объединение таблиц Orders и Customers с использованием реляционного оператора

Судно	Вес_груза	Дата_Отправки	ContactName	Phone
Save-a-lot Markets	657,54	27.03.1998	Jose Pavarotti	(208) 555-8097
Ernst Handel	754,26	13.04.1998	Roland Mendel	7675-3425
Save-a-lot Markets	830,75	17.04.1998	Jose Pavarotti	(208) 555-8097
White Clover Markets	606,19	17.04.1998	Karl Jablonski	(206) 555-4112

При помощи этого механизма можно объединять более двух таблиц, указывая связующие поля и условия отбора записей.

Интересной возможностью является объединение полей таблицы с самой собой. Такое объединение может помочь выявить несогласованность данных и повторы записей. Механизм объедения полей одной таблицы аналогичен механизму объединения разных таблиц. Но в рассматриваемом случае необходимо использовать временные имена, присваиваемые таблице. Эти имена называют псевдонимами:

```
SELECT Tab1.ID AS ID1, Tab1.Name AS Name1, Tab2.Name AS
       Name2, Tab2.ID AS ID2
FROM Subjects.db Tab1, Subjects.db Tab2
WHERE Tab1.Name = Tab2.Name
      AND Tab1.ID < Tab2.ID
```

Следует обратить внимание на то, что было создано два псевдонима для одной таблицы. В запросе использовались псевдонимы `Tab1` и `Tab2`.

Результат выполнения запроса приведен на рис. 9.3.

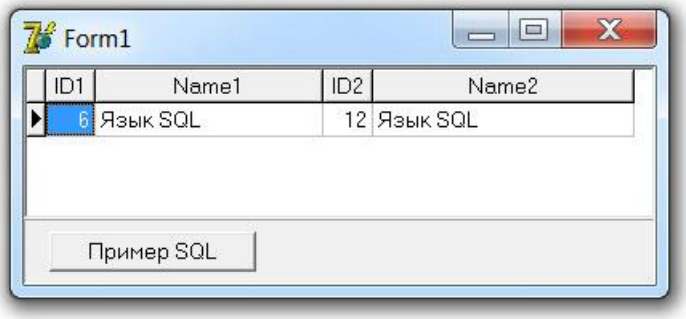


Рис. 9.3. Объединение одной таблицы

В ходе выполнения запроса для каждого поля CompanyName первой таблицы были отобраны все соответствующие поля второй таблицы. При помощи псевдонимов можно отследить различные зависимости распределений данных.

Вложенные подзапросы

Вложенные запросы могут использоваться в качестве дополнительных условий отбора записей. Для того чтобы понять механизм работы этого условия, следует рассмотреть простой запрос, в котором выводится список названий судов, которые обслужил сотрудник по имени Steven Buchanan и даты их отправки:

```
SELECT ShipName AS Название_судна, OrderDate AS
     Дата_отправки FROM Orders
WHERE EmployeeID IN
   (SELECT EmployeeID FROM Employees
   WHERE FirstName = 'Steven' AND LastName = 'Buchanan')
```

В этом запросе оператор IN может быть заменен оператором равенства. Однако следует учитывать, что оператор IN работает с массивом значений, а скалярный оператор равенства – только с одним.

Выполнение запросов с вложенными подзапросами всегда начинается с подзапроса, располагающегося на самом нижнем уровне. В рассматриваемом подзапросе отыскивается индивидуальный номер сотрудника EmployeeID по его имени и фамилии. Основной запрос принимает найденное значение в качестве параметра. В табл. 9.8 представлен результат выполнения запроса.

Таблица 9.8. SQL-запрос с вложенным подзапросом

Название_судна	Дата_отправки
Vins et alcools Chevalier	04.07.1996
Chop-suey Chinese	11.07.1996
White Clover Markets	31.07.1996
Blondel pere et fils	04.09.1996
Wartian Herkku	03.10.1996

Можно усложнить вложенный запрос. В примере будет приведен запрос, отображающий список сотрудников, обслуживших более девяноста судов:

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE EmployeeID IN
  (SELECT EmployeeID FROM Orders GROUP BY EmployeeID
   HAVING COUNT(EmployeeID) > 90)
ORDER BY FirstName, LastName
```

Во вложенном запросе производится отбор идентификаторов работников, встречающихся в таблице более 90 раз. В табл. 9.9 приведен результирующий набор данных.

Таблица 9.9. Использование агрегатных функций во вложенных запросах

TitleOfCourtesy	FirstName	LastName
Mrs	Margaret	Peacock
Ms	Janet	Leverling
Dr	Andrew	Fuller
Ms	Nancy	Davolio
Ms	Laura	Callahan

Использование оператора EXISTS

Логические операторы EXISTS и NOT EXISTS возвращают значение True или False в зависимости от наличия записей, удовлетворяющих условию поиска. Как правило, оператор EXISTS используется с вложенными запросами. Для иллюстрации принципов его применения можно использовать довольно простой запрос

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE EXISTS
  (SELECT * FROM Orders
   WHERE Freight > 1000)
ORDER BY LastName
```

В подзапросе выбираются строки, значение которых больше 1000. Так как подобные строки существуют, то оператору WHERE передается значение

True и выражение SELECT выбирает соответствующие записи. В табл. 9.10 приведен результат выполнения запроса.

Таблица 9.10. Использование оператора EXISTS

TitleOfCourtesy	FirstName	LastName
Mr	Steven	Buchanan
Ms	Laura	Callahan
Ms	Nancy	Davolio
Ms	Anne	Dodsworth
Dr	Andrew	Fuller

Можно изменить условие, накладываемое на поле Freight, и использовать вместо оператора **EXISTS** оператор **NOT EXISTS**

```
SELECT TitleOfCourtesy, FirstName, LastName
FROM Employees
WHERE NOT EXISTS
  (SELECT * FROM Orders
   WHERE Freight > 2000 )
ORDER BY LastName
```

Результат выполнения запроса будет аналогичен предыдущему. Нужно разобраться, почему так произошло. Оператор NOT EXISTS вернет значение True только в том случае, если ни одна запись не будет удовлетворять заданному условию. Так как ни одно судно не перевезло больше чем 2 тысячи тонн груза, то ни одна запись не будет выбрана.

Использование объединения UNION

Оператор UNION используется для объединения результатов двух и более запросов в единый набор полей и записей. Когда результаты запросов подвергаются объединению, их столбцы вывода должны быть совместимы. Это означает, что все запросы должны указывать одинаковое число столбцов в одном и том же порядке. И все совпадающие поля должны иметь один и тот же тип. Это иллюстрируется простым запросом

```
SELECT CustomerID FROM Customers
UNION
SELECT CustomerID FROM Orders
```

В таблице 9.11 представлен результат выполнения запроса.

В ходе выполнения запроса в результирующую таблицу были включены записи из двух таблиц.

Таблица 9.11. Использование оператора UNION

CustomerID
ALFKI
ANATR
ANTON
AROUT
BERGS
BLAUS

9.2.2. Модификация данных *(UPDATE, INSERT, DELETE)*

Изменение данных производится при помощи трех команд языка DML:

- Оператор INSERT позволяет добавить новую запись.
- Оператор UPDATE отвечает за обновление записи.
- Оператор DELETE производит удаление записи.

Модификация может производиться как в отношении одной записи, так и в отношении целой группы записей.

Использование оператора UPDATE

Редактирование записей представляет собой изменение значений полей в группе записей. Оно выполняется инструкцией `UPDATE` следующего формата

```
UPDATE <Имя таблицы>
   SET <Имя поля1> = <Выражение1>,
       <Имя поля2> = <Выражение2>, …
   [WHERE <Условие отбора>]
```

`<Имя поля>` указывает модифицируемое поле всей совокупности записей, а `<Выражение>` определяет значение, которое будет присвоено этому полю. Например:

```
UPDATE Personnel
   SET Salary = Salary + 2000
   WHERE Salary < 15000
```

Если сотрудник имеет оклад менее 15000 (рублей), то оклад увеличивается на 2000 (рублей).

Критерий отбора, указанный в операнде `WHERE`, не отличается от критерия, задаваемого в инструкции `SELECT`. Если он не задан, то изменяются значения всех указанных полей.

В одной инструкции UPDATE можно изменить значения нескольких полей, в этом случае для каждого из них указывается соответствующее значение. Например:

```
UPDATE Store
   SET Quantity = 0, Note = 'Обнулено'
   WHERE Quantity BETWEEN -0.05 AND 0.05
```

Вставка записей – INSERT

Вставка записей осуществляется с помощью инструкции INSERT, которая позволяет добавлять к таблицам одну или несколько записей. При добавлении одной записи инструкция INSERT имеет формат:

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

В результате выполнения этой инструкции к таблице, имя которой указано после слова INTO, добавляется одна запись. Для добавленной записи заполняются поля, перечисленные в списке. Значения полей берутся из списка, расположенного после слова VALUES. Список полей и список значений должны соответствовать друг другу по числу элементов и типу. При этом порядок полей в запросе может отличаться от порядка полей в таблице.

Пример запроса на добавление записи:

```
INSERT INTO Store
   (Name, Price, Quantity)
   VALUES ('Торшер', 4995.5, 10)
```

Список полей в инструкции INSERT может отсутствовать, в этом случае необходимо указать значения всех полей таблицы. Порядок и тип этих значений должны соответствовать порядку и типу полей таблицы.

При добавлении к таблице сразу нескольких записей инструкция INSERT имеет формат:

```
INSERT INTO <Имя таблицы>
   [(<Список полей>)]
   Инструкция SELECT
```

В данном случае значения полей новых записей определяются через значения полей записей, отобранных с помощью инструкции SELECT. Число добавленных записей равно числу отобранных записей. Список значений полей, возвращаемых инструкцией SELECT, должен соответствовать списку инструкции INSERT по числу и типу полей.

С помощью вставки группы записей можно скопировать данные из одной таблицы в другую, например, при резервном копировании или архивировании записей. При этом обе таблицы обычно имеют одинаковую структуру или их структуру частично совпадают.

Вот запрос на добавление нескольких записей:

```
INSERT INTO CardsArchive
   (ACode, AMove, ADate)
   SELECT CCode, CMove, CDate
   FROM Cards
   WHERE CDate = '31.01.2018'
```

В архивную таблицу `CardsArchive` добавляется группа записей из таблицы `Cards` движения товара для записей, сделанных в 31 января 2018 года.

Удаление записей – DELETE

Для удаления группы записей используется инструкция `DELETE`, имеющая формат:

```
DELETE FROM <Имя таблицы>
   [WHERE <Условие отбора>]
```

В результате выполнения инструкции из таблицы, имя которой указано после слова `FROM`, удаляются все записи, которые удовлетворяют условию отбора.

Пример:

```
DELETE FROM Store
   WHERE Quantity = 0
```

Отметим, что если с записями это таблицы связаны записи другой таблицы, например, движения товара, то может потребоваться их предварительное удаление, что связано с действием бизнес-правил и налагаемыми ограничениями.

9.3. Создание и удаление таблиц баз данных

Таблицы создаются командой **CREATE TABLE.** Эта команда создает пустую таблицу без записей. Но при этом формируется ее структура, объявляются названия полей, указываются их типы и размер. Ниже приведен синтаксис команды **CREATE TABLE:**

```
CREATE TABLE <table_name>
(<column_name1 > <data type>,
 <column_name2 > <data type>, … )
```

В качестве примера можно спроектировать таблицу, объявить ее и заполнить значениями. Предположим, что она нужна для проведения учета стройматериалов на складе. Запрос на создание таблицы приведен ниже:

```
CREATE TABLE Sclad
  (Tovar Char (20),
   Firma Char (20),
   Col Integer,
   DataPostup DATETIME)
```

В созданную таблицу надо занести несколько записей.

```
INSERT INTO Sclad VALUES ('Болты', 'Завод Космос', 20,
                          '11.11.2004')
INSERT INTO Sclad VALUES ('Винты', 'Уралмаш', 100,
                          '07.11.2004')
INSERT INTO Sclad VALUES (Гайки', 'Озерский комбинат', 5,
                          '01.19.2004')
INSERT INTO Sclad VALUES ('Отвертки',
          'Камышловский комбинат', 55, '04.21.2004')
INSERT INTO Sclad VALUES ('Плоскогубцы',
          'Камышловский комбинат', 20, '12.01.2004')
```

После этого данные можно извлечь из таблицы

```
SELECT * FROM Sclad
```

Результат выполнения запроса представлен в табл. 9.12.

Таблица 9.12. Содержимое новой таблицы

Tovar	Firma	Col	DataPostup
Болты	Завод Космос	20	11.11.2004
Винты	Уралмаш	100	11.07.2004
Гайки	Озерский комбинат	5	19.01.2004
Отвертки	Камышловский комбинат	55	21.04.2004
Плоскогубцы	Камышловский комбинат	20	01.12.2004

Для удаления таблицы можно воспользоваться командой DROP TABLE. Ее синтаксис чрезвычайно прост:

```
DROP TABLE <table_name>
```

Создание и удаление индексов

Команда создания индекса имеет следующий синтаксис:

```
CREATE INDEX <index_name> ON <table_name>
(<field1>, <field2> . . .)
```

Команда довольно проста. В ней указываются имя индекса, таблица, для которой он создается, и список полей, по которым будет вестись индексирование. В качестве примера можно построить индекс по полю Firma для созданной таблицы:

```
CREATE INDEX ScladIndex ON Sclad (Firma)
```

Для того чтобы сделать индекс уникальным, необходимо перед ключевым словом INDEX использовать директиву UNIQUE. С учетом этого запрос несколько изменится:

```
CREATE UNIQUE INDEX ScladIndex ON Sclad (Firma)
```

Перед тем как индекс будет создан в качестве уникального, будет произведена проверка на неповторяемость значений ключевого поля Firma.

Так как поле Firma содержит два одинаковых значения, индекс создан не будет. Для удаления индекса следует использовать команду DROP INDEX:

```
DROP INDEX Sclad.ScladIndex
```

После команды DROP INDEX указываются имя таблицы и имя индекса.

На этом можно завершить обзор возможностей языка SQL.