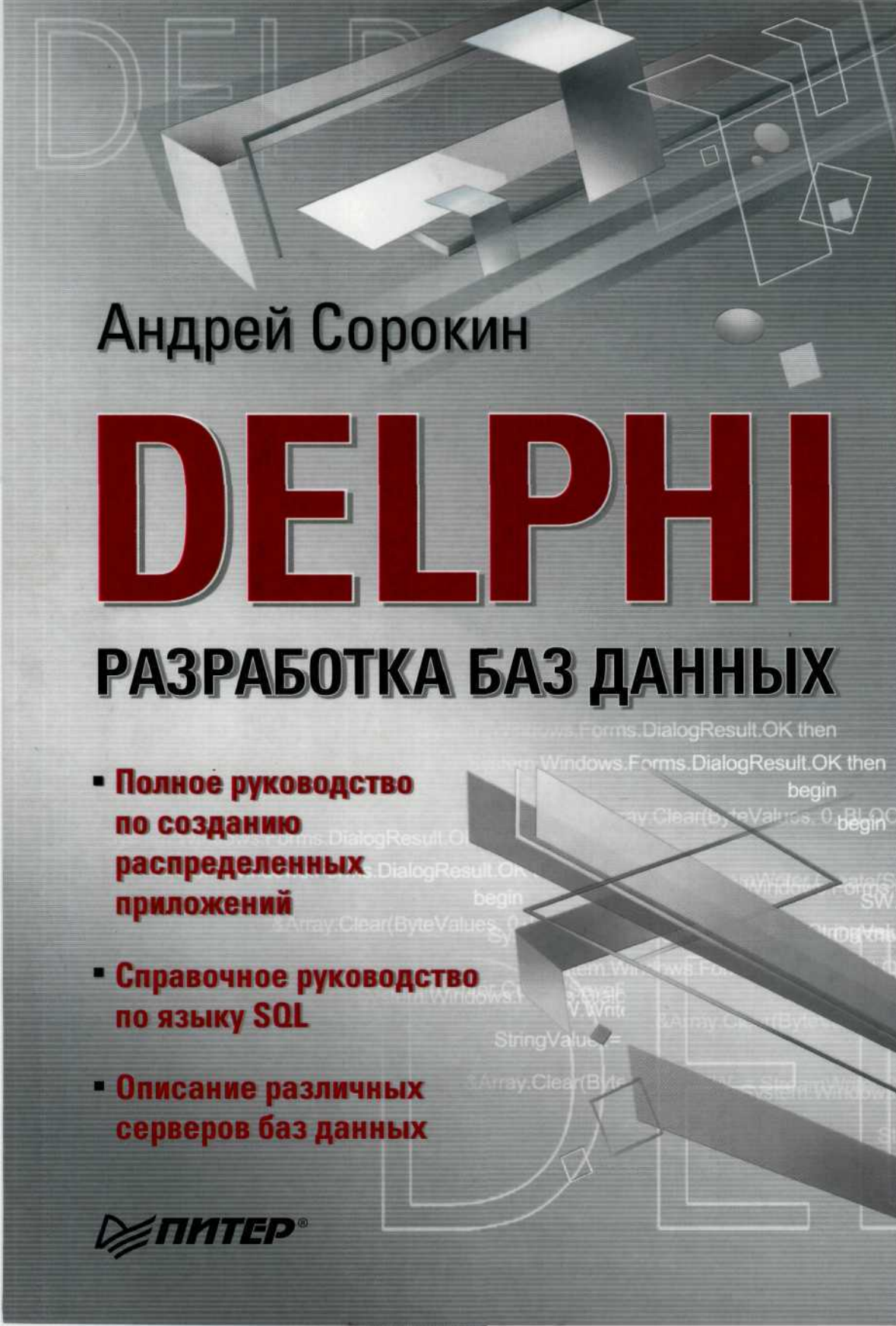


Андрей Сорокин

DELPHI

РАЗРАБОТКА БАЗ ДАННЫХ

- Полное руководство по созданию распределенных приложений
- Справочное руководство по языку SQL
- Описание различных серверов баз данных

ПИТЕР®

Андрей Сорокин

DELPHI

РАЗРАБОТКА БАЗ ДАННЫХ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2005

ББК 32.973.233-018
УДК 004.65
С65

Сорокин А. В.
С65 Delphi. Разработка баз данных. — СПб.: Питер, 2005. — 477 с.: ил.

ISBN 5-469-00927-0

Разработка приложений баз данных является одной из наиболее востребованных возможностей среды программирования Delphi. Эта среда программирования предоставляет разработчику поистине великолепный набор простых в использовании инструментов, позволяющих быстро разрабатывать сложные проекты.

В этой книге подробно рассматриваются все наиболее распространенные серверы баз данных и приводятся примеры взаимодействия с каждым из них. Также уделено внимание вопросам проектирования эффективных баз данных, что позволит читателю, не обладающему необходимым опытом, незамедлительно приступить к работе.

Книга в первую очередь ориентирована на начинающих программистов, работающих с Delphi, и на тех, кто хочет углубить свои знания в сфере разработки баз данных.

ББК 32.973.233-018
УДК 004.65

ISBN 5-469-00927-0

Краткое содержание

Содержание

Введение

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.

В какой-то период жизни подавляющее число разработчиков понимает — пришло время разобраться с этой областью. Кто-то встречает это время со спокойным осознанием того, что способен выполнить поставленную задачу. А кто-то с неуверенностью, так как не знаком с методами разработки распределенных систем. Рано думать, что настали темные времена и опять придется перелопачивать гору непонятной документации. Эта книга поможет разобраться в вопросах создания распределенных систем обработки данных.

К сожалению, на рынке ощущается недостаток литературы для начинающих по данной теме. Аспекты разработки распределенных приложений затрагиваются в отдельных главах, но не являются ключевыми. Задача книги заключается в том, чтобы максимально просто объяснить, как разрабатывать распределенные СУБД, излагая максимум сопутствующего материала.

На сегодняшний день на рынке представлено множество технологий доступа к данным и серверов баз данных. Современные приложения обработки данных ориентированы на работу с большим количеством пользователей, на их удаленность от места расположения основного сервера БД. В этой книге не раз будет акцентироваться вопрос общности работы с различными серверами БД. Именно поэтому будут рассматриваться методы работы с разными серверами.

Среда разработки Delphi предоставляет разработчику поистине великолепный набор простых в использовании инструментов, позволяющих быстро разрабатывать сложные проекты, создавая приятный и удобный пользовательский интерфейс. В этой среде очень просто работать с любым современным сервером баз данных, для которого есть соответствующий драйвер. Благодаря сумме технологий DataSnap, ADO и COM очень легко разрабатывать распределенные двух- и трехзвенные приложения баз данных. Связь с базой данных в Delphi устанавливается настройкой всего нескольких свойств и заданием пары-тройки дополнительных параметров.

Задача разработчика — написать стабильную систему управления БД. Грамотно настроить сервер БД, чтобы свести к минимуму возможные потери данных. Выбрать аппаратную платформу, чтобы обеспечить приемлемую производительность СУБД при пиковых нагрузках на сервер. Ведь зачастую в небольших организациях программист выполняет также функции администратора сервера баз данных. После прочтения этой книги эти вопросы уже не будут казаться сложными и непонятными.

Благодарности

Благодарю Моих Родителей, мою любимую — Таню, тётю Люду, чету Тойчей, Владимира Владимировича Шахиджаняна (автора книги «СОЛО на клавиатуре»), Серёгу (Serg999), Бабушку, Бориса Борисовича, Игоря Шапошникова (моего редактора) и еще тех немногих, кто мне помогал, и тех, кто мне мешал. Спасибо вам всем.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.

1 УРОК Введение в базы данных

Во все времена перед человечеством стояла задача сохранения информации и получения своевременного доступа к ней. С развитием информационных систем получили свое развитие базы данных (БД) — хранилища разнородной информации с определенной структурой. Сама база данных хранится и обрабатывается при помощи соответствующего программного обеспечения, которое обычно называется сервером баз данных. Сервер БД — информационная система, осуществляющая работу с данными, регламентирующая доступ к ним и призванная обеспечить их сохранность при помощи резервирования. С сервером БД взаимодействуют программы, написанные сторонним разработчиком. Подобные приложения называются системами управления базой данных (СУБД). СУБД, взаимодействуя с сервером, получает возможность оперировать данными: добавлять, удалять, изменять и получать их по запросу пользователя. СУБД имеет интерфейс, регламентирующий в той или иной степени действия пользователя, выполняет обработку данных и создает на их основе различные отчеты.

Терминология

В базе данные хранятся в табличном виде. Таблица представляет собой двухмерный массив, состоящий из набора строк и столбцов. Каждая строка представляет собой единицу хранения информации — *запись*. Каждая запись содержит *поля данных*, которые и являются столбцами таблицы. Поле данных является *атрибутом* записи. Базы данных, между отдельными таблицами которых существуют связи, называются реляционными. В табл. 1.1 приведен пример простой таблицы БД.

Каждое поле имеет свой заголовок. В примере это «Наименование CD», «Количество», «Дата поступления». Также каждое поле имеет свой тип данных. Подобных типов может быть довольно много. Чаще всего используются текстовый тип данных, числовой или временные отметки и промежутки.

Таблица 1.1. Пример таблицы БД

Наименование CD	Количество	Дата поступления
CD 1	60	06.08.2004
CD 2	35	07.08.2004
CD 3	12	07.08.2004

В зависимости от расположения СУБД различают локальные и распределенные системы. Все компоненты локальной СУБД, то есть сам сервер и таблицы с данными, расположены на машине конечного пользователя. В случае распределенной системы на машине конечного пользователя располагается только клиентская программа, которая взаимодействует с сервером БД по сети. Базы данных могут иметь многозвенную архитектуру. Чаще всего встречаются двух- и трехзвенные СУБД. На рис. 1.1 показана схема двухзвенной СУБД.

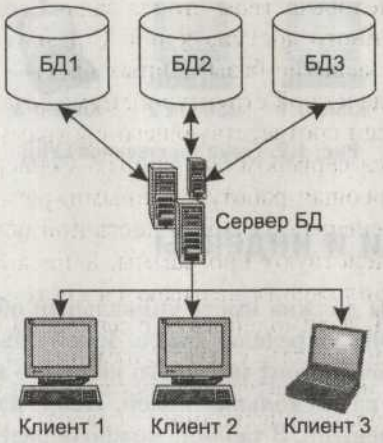


Рис. 1.1. Схема двухзвенной СУБД

При использовании двухзвенной системы происходит непосредственное взаимодействие клиентского приложения с сервером БД. На рис. 1.2 показана схема трехзвенной СУБД.

При использовании трехзвенной СУБД клиентское приложение взаимодействует с промежуточной программой — сервером приложения. Сервер приложения осуществляет обмен данными с сервером БД, получает от него данные, обрабатывает их и передает клиенту. Таким образом, вся вычислительная нагрузка ложиться на сервер приложения, а клиент становиться очень «легким», так как получает только запрошенные данные. Примером легкого клиента может служить интернет-браузер. При работе с веб-приложениями пользователь выбирает ссылку в браузере и получает связанный с ней документ.

Распределенные СУБД в общем случае могут быть как двухзвенными, так и трехзвенными.

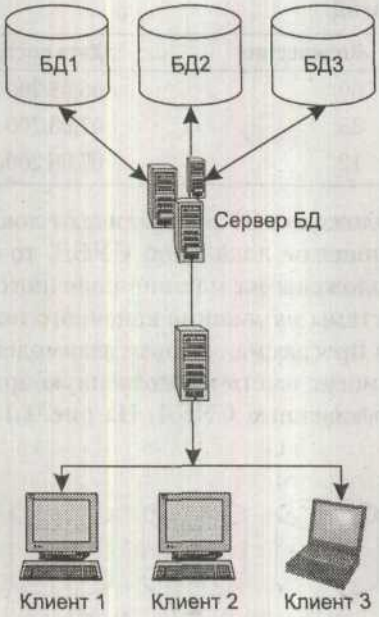


Рис. 1.2. Схема трехзвенной СУБД

Первичные ключи и индексы

Каждая запись таблицы должна иметь уникальный объект (поле, группу полей), который однозначно определял бы ее. Такой объект называется *первичным ключом*. Когда ключ состоит из одного поля, его называют простым. Составной ключ состоит из нескольких полей. Поля, из которых составляется ключ, называются ключевыми. В таблице можно определить только один ключ. В соответствии с ним будут производиться сортировка и сохранение данных. Для примера, в табл. 1.1 можно определить поле «Наименование CD» как ключевое. Соответственно, упорядочивание строк будет производиться по этому полю.

Ключ обеспечивает:

- однозначную идентификацию данных таблицы;
- предотвращение повторения значений ключа (контроль за выполнением этого условия осуществляет сервер БД);
- ускорение выполнения запросов к БД;
- использование ограничений ссылочной целостности.

Поскольку первичный ключ должен быть уникальным, для него могут использоваться не все поля таблицы. Например, поле «Заказанный товар» табл. 1.2 не может быть ключевым, так как один и тот же товар может быть заказан несколькими покупателями. В качестве ключевого можно взять поле «Телефон».

Таблица 1.2. «Слепок» накладной на доставку товаров

Заказанный товар	Количество	Предприятие	Телефон
Микроконтроллер	2	Какое-то НИИ	455-333-222
Датчик уровня	10	Другое НИИ	112-789-456
Микроконтроллер	5	Третье НИИ	587-982-563

Физически ключ является системной таблицей (каждый сервер БД имеет свой формат таблицы и по-своему работает с ней), в которой в определенном порядке хранятся значения, составляющие ключ. Для каждого значения ключа имеется уникальная ссылка, указывающая на расположение соответствующей записи в таблице БД. При запросе сервер производит поиск по значению ключевого поля и, таким образом, быстро получает нужные данные. Таблица, в которой содержаться значения ключа, может располагаться как в отдельном файле, так и вместе с самой базой данных. Например, в БД Paradox значения ключа содержаться в одноименном файле с расширением *.px*, а сервер MS SQL 2000 хранит значения ключа в том же файле, где располагается сама БД.

Каждое значение ключа занимает определенный размер в системной таблице, следовательно, на это же значение увеличивается общий «вес» базы. Размер таблицы зависит не только от количества записей, но и от полей составляющих ключ. Текстовое поле, состоящее в ключе, может значительно увеличить его размер.

В связи с этим к построению ключей предъявляются определенные требования:

- Ключ должен быть уникальным.
- Ключ должен быть достаточным и неизбыточным.
- Ключ не должен содержать поля неоднозначного содержания.

В качестве ключевого поля удобно использовать автоинкрементное поле, которое автоматические увеличивает максимальное значение числа при вводе новой записи. Данный тип поля поддерживается многими серверами БД, такими как, например, Access или SQL Server 2000. В иных случаях разработчик обязан сам предусмотреть реализацию данного механизма. Реализовать его легко, достаточно лишь создать триггер с соответствующим содержанием.

Помимо ключей в базах данных очень широко используются индексы. *Индекс* является самостоятельным объектом базы данных, но он связан с определенными колонками таблицы. Индекс, так же как и ключ, содержит отсортированные в том или ином порядке значения таблицы и ссылку на положение нужной записи в таблице. Использование индекса повышает скорость доступа к данным за счет того, что доступ осуществляется не последовательным (путем перебора всех данных), а упорядоченным способом, по ключевым полям индекса. Индекс может создаваться по одному или по нескольким полям. Значения индекса, так же как и значения ключа, содержатся в системных таблицах.

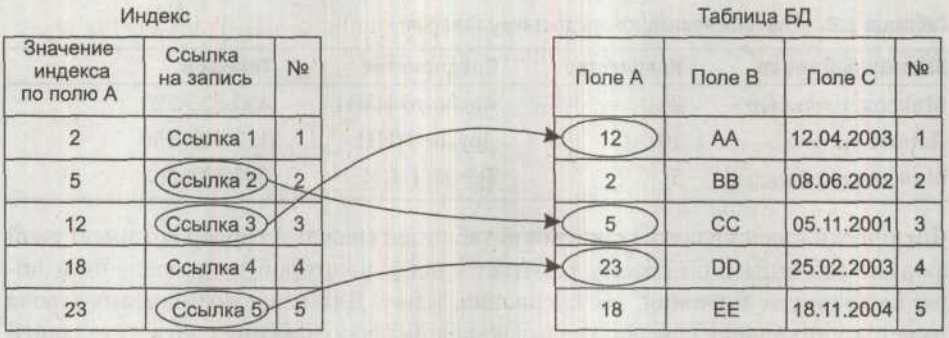


Рис. 1.3. Использование индекса

На рис. 1.3 приведен обобщенный пример взаимодействия индекса с таблицей БД. Как было сказано ранее, индекс содержит в себе упорядоченные значения одного или нескольких полей. Также он содержит ссылки на записи, расположенные в таблице. При осуществлении поиска по индексированному полю сервер БД быстро находит нужный элемент, если он существует, получает ссылку на него и сразу переходит к нему в базе и возвращает данные клиентскому приложению. В приведенном примере есть таблица, состоящая из трех полей: A, B, C. Как можно заметить, поля в таблице А не упорядочены. Однако в индексе, созданном по этому полю, данные хранятся в отсортированном виде. При осуществлении операций с данными сервер БД проверяет, индексировано ли поле, по которому производится запрос. Если поле индексировано, сервер БД осуществляет быстрый поиск по индексу. По такому же принципу осуществляется работа с ключом.

Но во всем важно знать меру. Можно индексировать всю таблицу и получить медленно работающую БД гигантского размера. При создании индекса важно продумать, по каким полям чаще всего будет вестись поиск. Грамотно созданный индекс существенно увеличивает производительность БД. Но необходимо помнить о балансе. Индексировать нужно лишь те поля, по которым будет чаще всего производиться поиск.

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. Связывать можно одну или несколько таблиц. 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 индексирова-

ны, так как это ускоряет доступ к связанным записям. В общем случае поле связи и внешний ключ представляют собой индексы.

Название города	Код города	Поле связи
Екатеринбург	XXXX	1
Санкт-Петербург	XXXX	2
Воронеж	XXXX	3

Абонент	Длительность разговора	Внешний ключ
Алексеев	3 мин	1
Костин	5 мин	1

Рис. 1.4. Пример связи между таблицами

В качестве примера на рис. 1.4 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. 1.5.

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	24.12.2004	5
2	18.05.2004	5
3	15.06.2003	4

Рис. 1.5. 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. 1.6. Связь «один-ко-многим»

Таблица «Отпуск товаров со склада»

Товар	кол-во ед.	Код покупателя
Макароны	2000	1A
Минеральная вода	1000	2C
Консервы	2000	1A
Макароны	5000	2C
Сок томатный	550	2C
Рожки	300	1B
Сок томатный	150	1A

Таблица «Покупатель»

Код покупателя	ФИО	Адрес
1A	Иванов И. А.	XXX
2C	Кузнецов А. Г.	XXX
1A	Перминов И. В.	XXX

Рис. 1.7. 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. 1.6, одному студенту в главной таблице со-

ответствует 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. 1.7, 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.

Ссылочная целостность

На рис. 1.8 представлена таблица, находящаяся в отношении «один-ко-многим». Связь производиться по полю «Номер».

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	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. 1.8. 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. 1.10. 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. 1.9. 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. 1.10. 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.

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.

Физически транзакция представляет собой последовательность команд, производящих с базой данных те или иные действия. Главная особенность транзакций заключается в том, что все действия должны выполниться, иначе будет отменена вся транзакция.

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.

Параллельная обработка транзакций

Когда несколько транзакций в одно и то же время работают с базой данных, такие транзакции называются *параллельными транзакциями*. За работой транзакций следит объект сервера БД, который условно можно назвать «менеджер обработки транзакций». Схематично он изображен на рис. 1.11. Несколько транзакций отправляют свои запросы данному менеджеру. Он в определенной последовательности, зависящей от сервера, выбирает транзакцию из списка и обрабатывает ее.

При параллельной обработке часто возникает ситуация, когда несколько транзакций обращаются к одной записи или к одному блоку записей и вносят в них

изменения, не дождавшись освобождения ресурсов. Таким образом, получается, что транзакция, часть которой была выполнена, изменила блок данных и была «временно приостановлена», а другая транзакция, стоящая следующей в очереди менеджера обработки транзакций, также обратилась к этому блоку записей и также изменила его. Когда первая транзакция снова обратилась к данному блоку записей, он уже был изменен, и она продолжила работать уже с измененными данными. Одно из средств борьбы, против несогласованностей, вызванных параллельной обработкой транзакций, называется *блокировкой ресурсов*.

Рис. 1.11. Схематичный пример работы нескольких транзакций с БД

Блокировка ресурсов

Один из способов предотвратить проблемы при параллельной обработке — запретить совместное использование ресурсов путем блокировки данных, которые считываются для обновления. В этом случае, если блок записей или одна запись «заняты» какой-либо транзакцией, то другая транзакция не сможет их изменить.

Блокировки могут налагаться либо автоматически, по требованию СУБД, либо по запросу пользователя. Некоторые СУБД предусматривают блокировку ресурсов на уровне строк, другие — на уровне страницы, третьи — на уровне таблицы, четвертые — на уровне всей базы данных. Размер блокируемого ресурса называется *глубиной детализации*. В данном случае действует простое правило — чем глубже детализация, тем медленнее идет работа. При блокировке на уровне строк и большой «пользовательской нагрузке» СУБД может работать очень медленно, так как на блокировку будут производиться очень

большие затраты ресурсов и времени. Наиболее часто применяется постраничная блокировка данных. При блокировке на уровне таблицы базы данных будут неизбежно возникать простои, особенно в тех случаях, когда пользователь запрашивает большие объемы данных.

Блокировки различаются также по типу. При *монопольной блокировке* полностью блокируется доступ к блоку данных. *Коллективная блокировка* блокирует блок данных от изменения, но не от чтения.

Сериализуемые транзакции и виды блокировок

Когда две или более транзакции обрабатываются параллельно, их результаты, сохраняемые в базе данных, должны быть логически согласованы с результатами, которые получились бы, если бы данные транзакции обрабатывались каким-нибудь последовательным способом. Такая схема обработки параллельных транзакций называется сериализуемой (serializable).

Существует теорема, что сериализуемость транзакций заведомо гарантируется, если блокировки, относящиеся к одновременно выполняемым транзакциям, удовлетворяют правилу: «Ни одна блокировка от имени какой-либо транзакции не должна устанавливаться, пока не будет снята ранее установленная блокировка». Это правило известно под названием двухфазового блокирования. Транзакция сначала устанавливает блокировки, а потом сама же блокировки снимает.

При двухфазной блокировке строки данных блокируются по мере необходимости. Изменения производятся, но не записываются в базу данных, пока транзакция не будет полностью обработана. После этого изменения, внесенные в ходе выполнения транзакции, записываются и все блокировки снимаются.

Для распределенных СУБД те же задачи переносятся на распределенную среду. Транзакции могут выполняться на нескольких узлах, где располагаются необходимые данные. Выполнение множества распределенных транзакций сериализуемо (свойство *глобальной сериализуемости*) тогда и только тогда, когда:

- выполнение этого множества транзакций сериализуемо на каждом узле;
- упорядочение транзакций на всех узлах одинаково.

В алгоритмах, основанных на блокировании, для обеспечения свойства глобальной сериализуемости применяется один из четырех протоколов:

- централизованный протокол двухфазной блокировки;
- протокол двухфазной блокировки первичной копии;
- распределенный протокол двухфазной блокировки;
- протокол блокирования большинства.

При централизованном блокировании для всей распределенной базы данных поддерживается единая таблица блокировок. Эта таблица, располагаемая на

одном из узлов, находится под управлением единого менеджера блокировок. Менеджер блокировок отвечает за установку и снятие блокировок от имени всех транзакций. Поскольку управление блокировками сосредоточено на од-

Каждой транзакции приписывается некоторый уровень изоляции: READUNCOMMITTED (незавершенное чтение), READCOMMITTED (завершенное чтение), REPEATABLEREAD (воспроизводимое чтение) или SERIALIZABLE (сериализуемость). Уровень изоляции транзакции определяет степень, в которой на операции этой транзакции влияют операции параллельно выполняющихся транзакций и в которой операции данной транзакции влияют на операции других транзакций. Уровень изоляции может быть явно установлен оператором SETTRANSACTION. По умолчанию устанавливается уровень изоляции SERIALIZABLE. При выполнении конкурирующих транзакций на этом уровне изоляции гарантируется их сериализуемость.

На рис. 1.12 показаны уровни изоляции транзакции и ошибки, которые могут на них возникать. Отсутствие на уровне SERIALIZABLE (сериализуемость) ошибок является следствием того, что такие транзакции являются сериализуемыми. Чем выше уровень транзакции, тем больше ресурсов требуется для ее обработки. Программист должен найти компромисс «производительность–достоверность». Например, уровень READUNCOMMITTED является наиболее быстродействующим, но в то же время на этом уровне возможны искажения данных. Вряд ли имеет смысл ставить уровень SERIALIZABLE на сервер, с которым работают 2–3 клиента, и вероятность того, что они обратятся к одной записи, весьма мала.

Уровень изоляции

Уровень изоляции	Грязное чтение	Невоспроизводимое чтение	Фантомное чтение
Незавершенное чтение	Возможно	Возможно	Возможно
Завершенное чтение	Невозможно	Возможно	Возможно
Воспроизводимое чтение	Невозможно	Невозможно	Возможно
Сериализуемость	Невозможно	Невозможно	Невозможно

Рис. 1.12. Уровни изоляции транзакции

Курсор

Курсором называется именованный указатель на блок данных (набор строк). Обычно курсоры определяются с помощью оператора SELECT:

```
EXEC SQL BEGIN DECLARE SECTION;
char szLastName[] = "White";
```

В данном запросе определяется курсор author_cursor, указывающий на набор полей au_fname, в которых поле au_lname содержит значение White. Такой курсор называется статическим. Динамический курсор получает значения параметров в ходе выполнения запроса. Транзакция может открывать несколько курсоров как последовательно, так и одновременно. Так как курсоры занимают значительное количество памяти, то открытие большого их числа может сильно замедлить работу системы. Один из способов снизить накладные расходы — использовать курсоры разного типа, особенно в тех случаях, когда полнофункциональный курсор не требуется. Всего существует четыре типа курсора:

1. Последовательный (forward only cursor).
2. Статический (static cursor).
3. Ключевой (keyset cursor).
4. Динамический (dynamic cursor).

Последовательный курсор позволяет приложениям передвигаться по полученному набору строк только вперед. Изменения, произведенные другими транзакциями, будут видны только в том случае, если затронутые ими строки не были запрошены ранее, то есть находятся впереди.

Статический курсор обрабатывает слепок базы данных, сделанный на момент открытия данного экземпляра курсора. Изменения, сделанные таким курсором, являются видимыми для самого курсора. Изменения, сделанные другими транзакциями, являются невидимыми.

Ключевой курсор при открытии для каждой строки базы данных сохраняет значение первичного ключа. Когда приложение устанавливает курсор на некоторую строку, СУБД по ключу считывает текущее значение этой строки. Если строка базы данных, к которой обращается данный курсор, была удалена, а текущая транзакция ввела в нее какие-либо данные, то строка будет создана вновь по значению ключа и в нее будут занесены измененные данные. Данный курсор видит лишь сохраненные обновления и удаления.

Динамический курсор является полнофункциональным курсором. Любые изменения данных являются видимыми для данного курсора. Если уровень изоляции транзакции не допускает «грязное чтение», то курсор может видеть только сохраненные изменения.

Курсор может располагаться как на сервере, так и на стороне клиентского приложения.

Серверный курсор используется в том случае, если база данных имеет большой размер и пересылать ее целиком на клиентскую систему нецелесообразно. Скорость работы клиентского приложения при этом несколько снижается.

Клиентский курсор обеспечивает передачу базы данных клиенту от сервера целиком. Такой режим работы подходит, если база данных имеет небольшой размер.

ПРИМЕЧАНИЕ От выбора типа курсора и места его расположения зависит производительность СУБД. В случае многозвенных СУБД часто применяется расположение курсора на стороне сервера. Если курсор располагается на стороне клиента, то ему автоматически присваивается ключевой курсор.

Хранимые процедуры

Хранимая процедура — последовательность команд SQL, хранящаяся на сервере БД в скомпилированном виде. Хранимые процедуры составляют для часто выполняемых операций. Например, каждый месяц сотрудникам надо пересчитывать зарплату. Можно создать хранимую процедуру, производящую перерасчет, и таким образом сэкономить время. Хранимая процедура может принимать параметры и возвращать результаты работы. Когда приложение использует процедуру, оно передает параметры, если они есть, а сервер СУБД затем выполняет хранимую процедуру без повторной компиляции (рис. 1.13).

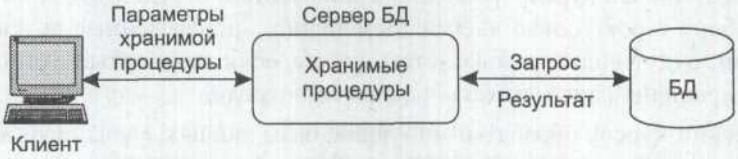


Рис. 1.13. Выполнение хранимой процедуры

Использование хранимых процедур имеет несколько преимуществ с точки зрения производительности:

1. Так как хранимые процедуры уже скомпилированы и записаны, то серверу требуется меньше времени для их выполнения.
2. Снижается нагрузка на сеть, так как передается меньшее число операторов.

Создать хранимую процедуру довольно просто. Следующий фрагмент кода иллюстрирует этот процесс:

```
CREATE PROC give_raise AS
UPDATE EMPLOYEES SET salary = salary * 1.10
```

В данном примере создается хранимая процедура `give_raise`, модифицирующая таблицу `EMPLOYEES`. Значение поля `salary` увеличивается на 10%.

Понятие триггера

Триггер — хранимая процедура, вызов которой происходит автоматически при выполнении с базой данных определенных действий: удаление, изменение, добавление записей. В зависимости от того, какой оператор модификации данных активизирует триггер, он называется триггером вставки (insert trigger), триггером удаления (delete trigger) или триггером обновления (update trigger).

Триггеры часто используются для обеспечения целостности на уровне ссылок, для выполнения каскадных удалений. Так как триггеры вызываются самой СУБД, нет возможности напрямую вызвать их из клиентского приложения. Триггер может быть вызван неявно другим триггером при совершении некоторых действий с базой данных:

```
USE pubs
IF EXISTS (SELECT name FROM sysobjects
WHERE name = 'reminder' AND type = 'TR')
DROP TRIGGER reminder
GO
CREATE TRIGGER reminder
ON titles
FOR INSERT, UPDATE, DELETE
AS
EXEC master..xp_sendmail 'MaryM',
'Don''t forget to print a report for the distributors.'
GO
```

В данном примере приведено определение триггера `reminder`. Он выполняется при удалении, добавлении или изменении записи. В теле триггера вызывается хранимая процедура `xp_sendmail`, отправляющая некой Мэри М. по электронной почте соответствующее напоминание.

Представление

Представление является виртуальной таблицей, записи в которую собраны с помощью оператора `Select` из одной и более таблиц. Работа с представлением осуществляется, как с обычной таблицей:

```
CREATE VIEW Example AS
SELECT Field1, Field2
FROM EXTABLE
WHERE Field3 = 3
SELECT * FROM Example
```

В приведенном примере создается представление `Example`, в которое включаются поля `Field1` и `Field2`. Выбираются записи, в которых значение поля `Field2` равно трем. Последний оператор вызывает просмотр. Само по себе представление не содержит данных, оно их получает каждый раз при вызове.

Представления подобны окнам, через которые разработчик или пользователь просматривает информацию, которая фактически хранится в базовой таблице.

Преимущество использования представления заключается в том, что представление будет модифицировано автоматически всякий раз, когда таблица, ле-

жащая в его основе, изменяется. Содержание представления не фиксировано и переназначается каждый раз, когда оно вызывается тем или иным оператором SQL. В большинстве случаев представления могут использоваться как механизм защиты данных, поскольку даются права доступа к представлению базы данных, а не к самим таблицам, содержащим данные.

Представления могут быть модифицируемыми или позволяющими только чтение данных. Модифицируемое представление позволяет изменять данные в таблице.

Для того чтобы представление было модифицируемым, оно должно удовлетворять списку условий:

- В представлении должны быть собраны записи только из одной таблицы.
- В операторе SELECT не должны использоваться статистические функции, описатель DISTINCT, операнд HAVING, а также соединения таблиц.
- Представление не должно содержать полей, которые являлись бы агрегатными функциями.
- Для команды INSERT представление должно содержать любые поля таблицы, имеющие ограничение NOT NULL, если не задано иное ограничение.

В качестве примера можно взять модифицированное представление Example и показать, как в него добавляются записи при помощи небольшого фрагмента кода:

```
INSERT INTO Example
VALUES (333, 'fds')
```

Модели данных

Под моделью базы данных обычно понимаются структура базы и методы работы с ней. В общем случае понятиями, на основе которых строится модель, являются объекты и отношения между ними. Подобную модель данных, функционирующую на сервере, и можно называть базой данных.

Существует несколько видов моделей, и постоянно развиваются новые модели, отвечающие конкретным требованиям, предъявляемым новыми концепциями.

Иерархическая модель

Иерархическая модель (ИМ) представляется связанным графом типа дерева. Вершины деревьев располагаются на разных иерархических уровнях. Иерархическая БД состоит из упорядоченного набора деревьев. Тип дерева в целом представляет собой иерархически организованный набор типов записей. На иерархическую модель налагается целый список ограничений и правил:

- Все типы связей должны быть функциональными (1:1, 1:M, M:1).
- Дерево представляет собой неориентированный граф, не содержащий циклов.
- Тип дерева состоит из одного «корневого» типа записи и упорядоченного набора из нуля или более типов поддеревьев (каждое из которых является некоторым типом дерева).
- В дереве каждый узел связан только с одним родительским узлом (любой сын может иметь не более родного отца, а любой отец — множество сыновей).
- Ветвь дерева соответствует типу связи «исходный—порожденный».
- Доступ к каждому порожденному узлу выполняется непосредственно через исходный узел.
- Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для БД определен полный порядок обхода — сверху вниз, слева направо.
- Существует единственный линейный иерархический путь доступа к любому узлу, начиная с корня дерева.

На рис. 1.14 приведен пример базы данных с иерархической моделью. База данных имеет два уровня. Узел содержит 2 потомка.

Рис. 1.14. Пример модели иерархической БД

У иерархической модели есть и определенные недостатки:

- Операции манипулирования данными в иерархических системах ориентированы прежде всего на поиск информации сверху вниз, то есть по конкретному экземпляру сегмента-отца можно найти все экземпляры сегментов-сыновей. Обратный поиск затруднен, а часто и невозможен.
- Происходит дублирование данных на логическом уровне.
- Для представления связи M:N необходимо дублирование деревьев.
- В ИМ автоматически поддерживается целостность ссылок между предками и потомками. Никакой потомок не может существовать без своего родителя. Целостность по ссылкам между записями, не входящими в одну иерар-

хию, не поддерживается. Поэтому невозможно хранение в БД порожденного узла без соответствующего исходного. Аналогично, удаление исходного узла влечет удаление всех порожденных узлов (деревьев), связанных с ним.

Сетевая модель

Сетевой подход к организации данных является расширением иерархического. Цель разработчиков заключалась в создании модели, позволяющей описывать связи M:N и одновременно в исправлении недостатков иерархической модели. Сетевая модель данных базируется также на использовании представления данных в виде графа. С точки зрения теории графов сетевой модели соответствует произвольный граф. В иерархических структурах запись-потомок должна иметь только одного предка. В сетевой структуре данных потомок может иметь любое число предков. На рис. 1.15 представлен схематический вид сетевой модели.

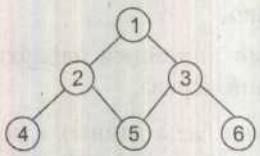


Рис. 1.15. Схематическое изображение сетевой модели

Сетевая БД состоит из набора записей и набора связей между этими записями. Тип связи определяется для двух типов записей: предка и потомка. Экземпляр типа связи состоит из одного экземпляра типа записи предка и упорядоченного набора экземпляров типа записи потомка. Набор — поименованная совокупность записей, в котором записи одного типа объявляются владельцами набора, а записи других типов — членами этого набора.

У сетевой модели есть свои особенности:

- Все типы связей должны быть функциональными (1:1, 1:M, M:1). В модели это внутреннее ограничение выражается тем утверждением, что для данного типа связи L с типом записи предка P и типом записи потомка C (набора) должны выполняться следующие два условия:
  - каждый экземпляр типа P является предком только в одном экземпляре L;
  - каждый экземпляр C является потомком не более чем одного экземпляре L.
- Для представления связи M:N вводятся дополнительный тип записи и две функциональные связи типа 1:M и M:1. При необходимости запись-связка может содержать дополнительную информацию.
- Экземпляр записи может быть членом только одного экземпляра набора среди всех экземпляров набора одного типа (он может входить в состав двух и более экземпляров наборов, но разных типов).

На рис. 1.16 показана сетевая модель базы данных «Учет товаров на складе». Она была получена путем преобразования иерархической модели, изображенной на 1.15. В базу данных были введены дополнительные записи-связки, на рисунке они обозначены эллипсами. Такая структура модели, в отличие от иерархической, позволяет быстро получить те места на складе (номера боксов), где расположены продовольственные товары.

Любая сетевая структура может быть приведена к более простому виду введением избыточности. В некоторых случаях возникающая при этом избыточность мала и является допустимой, в других случаях она может быть чрезмерной.

Рис. 1.16. Пример модели сетевой БД

Реляционная модель

Отношение можно представить как двухмерную таблицу. Каждая строка в таблице содержит данные, относящиеся к некоторой вещи или к ее части. Каждый столбец описывает какой-либо атрибут этой вещи. Строки отношения называются *сущностями*, а столбцы — *атрибутами*.

Чтобы таблицу можно было считать отношением, она должна удовлетворять определенным требованиям:

- Значения в ячейках должны быть одиночными.
- Все записи в столбце должны быть одного типа.
- Каждый столбец должен иметь уникальное имя.
- В отношении не может быть двух одинаковых строк.
- Порядок строк не имеет значения.

На рис. 1.17 представлено отношение «Товар». Данное отношение содержит в себе сущность «товар». Эта сущность обладает атрибутами «Наименование

товара», «Единица измерения», «Цена, руб.», «Код товара», полностью описывающими и характеризующими ее как уникальный объект.

	Атрибут 1	•	•	Атрибут 4
	Наименование товара	Единица измерения	Цена, руб.	Код товара
Сущность 1	Макароны	пачка	25	1А
•	Устрицы	кг	600	1Б
•	Колбаса	кг	100	1С
Сущность 4	Хлеб	булка	12	1Д

Рис. 1.17. Отношение «Товар»

Функциональная зависимость является важным термином, который необходимо знать, чтобы понять, что такое нормализация. *Функциональная зависимость (functional dependency)* — это связь между атрибутами. Предположим, что нам известен какой-либо атрибут сущности. Имея известный атрибут (или их группу), можно вычислить неизвестный атрибут. Такая зависимость называется функциональной. Функциональные зависимости в отношениях обычно выражаются не уравнениями, но смысловыми зависимостями.

Реляционная модель данных, несмотря на ее достоинства, совсем не идеальна. В ряде случаев она не позволяет ясно отразить особенности предметной области. Одной из иллюстраций этого заявления служит отсутствие прямых средств выражения иерархии. На смену ей были разработаны другие модели.

Объектно-ориентированная модель

Объектно-ориентированная модель опирается на несколько базовых концепций:

- Объекты, обладающие внутренней структурой и однозначно идентифицируемые уникальным внутрисистемным ключом.
- Классы, являющиеся типами объектов.
- Операции над объектами одного или разных типов, называемые методами.
- Наследуемость внешних свойств объектов на основе соотношения «класс—подкласс».

К достоинствам объектно-ориентированной модели обычно относят возможность для пользователя системы определять свои сколь угодно сложные типы данных (используя имеющийся синтаксис и свойства наследуемости и инкапсуляции). Также в качестве бонусов можно отметить наследуемость свойств объектов и повторное использование программного описания типов объектов при обращении к другим типам, на них ссылающимся.

Но помимо достоинств у объектно-ориентированной модели есть и недостатки:

- Отсутствие строгих определений; разное понимание терминов и различия в терминологии.
- Отсутствие общеупотребимых стандартов, позволяющих связывать конкретные объектно-ориентированные системы с другими системами работы с данными.

Теоретически предполагается, что в любом случае данные из одной модели можно перенести в другую, соответствующим образом преобразовав их. Наиболее распространенной является реляционная модель базы данных, хотя она имеет ряд серьезных недостатков.

Нормализация таблиц при проектировании баз данных

Реляционные базы данных во многих случаях требуют нормализации. Под нормализацией следует понимать процесс преобразования отношения, имеющего некоторые аномалии, к отношению с меньшим их количеством. Аномалией можно считать любые несогласованности структуры базы данных, приводящие к появлению чрезмерной избыточности, нарушению данных при удалении записей или невозможности ввода данных до наступления каких-либо дополнительных условий. Различают несколько видов нормальных форм. В этом разделе будут рассмотрены первая, вторая и третья нормальные формы, так как они являются наиболее распространенными.

Первая нормальная форма

Первая нормальная форма требует, чтобы каждое поле таблицы базы данных было атомарным, то есть было неделимым и не содержало повторяющихся групп.

Неделимость поля означает, что содержащиеся в нем значения не должны делиться на более мелкие. Для примера, поле «Ф.И.О.» следует разделить на поля «Фамилия», «Имя» и «Отчество», тогда будет соблюдаться атомарность значений.

Повторяющимися группами являются поля, содержащие одинаковые по смыслу значения. Например, рис. 1.18 показывает список сотрудников, работающих в какой-либо конторе в данный момент.

Таблица «Список сотрудников»

Дата	Сотрудник 1	Сотрудник 2	Сотрудник 3

Рис. 1.18. Пример дублирующихся полей

Каждый раз, когда штат расширяется или сокращается, придется перестраивать таблицу, добавляя отдельное поле. А если список сотрудников будет

большим. То работать с ней будет просто неудобно. В этом случае таблицу следует преобразовать к виду, представленному на рис. 1.19.

Таблица «Список сотрудников»

Дата	Сотрудник

Рис. 1.19. Преобразованная структура таблицы

В качестве примера можно привести некоторую ненормализованную таблицу, представленную на рис. 1.20, к первой нормальной форме. Предположим, руководству организации захочется получить список всех сотрудников, которые проживают в том или ином городе. В приведенной таблице содержится поле «Домашний адрес». Из него нужно вынести значение «Город» в отдельное поле. «Адрес и улицу» оставить в поле «Адрес». Далее необходимо разделить поле «Ф.И.О.» на составляющие его поля «Фамилия», «Имя», «Отчество». Таким образом будет получена нормализованная таблица, приведенная к первой нормальной форме.

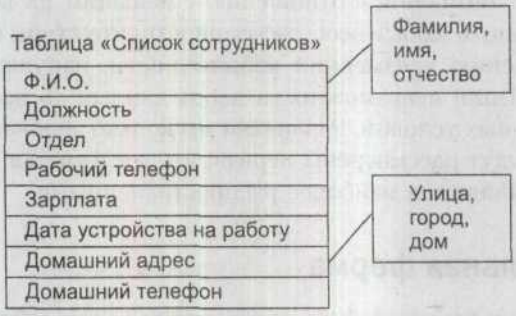


Рис. 1.20. Ненормализованная таблица

Полученная таблица представлена на рис 1.21.

Таблица «Список сотрудников»

Фамилия
Имя
Должность
Отдел
Рабочий телефон
Зарплата
Дата устройства на работу
Город
Адрес
Домашний телефон

Рис. 1.21. Таблица, приведенная к первой нормальной форме

Вторая нормальная форма

Вторая нормальная форма требует, чтобы все неключевые поля (атрибуты) таблицы зависели от первичного ключа, то есть чтобы первичный ключ однозначно определял запись и не был избыточен. Те поля, которые зависят только от части первичного ключа, должны быть выделены в составе отдельных таблиц. Также отношение должно удовлетворять первой нормальной форме.

Можно продолжить приведение таблицы к второй нормальной форме. Сначала потребуется выделить поля, которые будут входить в первичный ключ. Фамилия, имя, отчество и должность в общем случае будут однозначно идентифицировать конкретного сотрудника, их можно выделить в первичный ключ. Полученную таблицу можно увидеть на рис. 1.22.

Таблица «Список сотрудников»

Фамилия
Имя
Отчество
Должность
Отдел
Рабочий телефон
Зарплата
Дата устройства на работу
Город
Адрес
Домашний телефон

Рис. 1.22. Таблица с первичным ключом

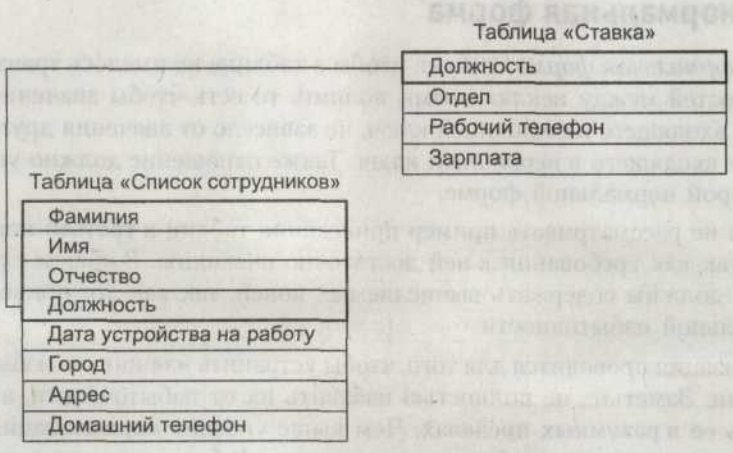


Рис. 1.23. Выделение таблицы «Ставка»

Поля «Отдел», «Рабочий телефон», «Зарплата» зависят от поля «Должность», входящего в первичный ключ, но не от всего ключа. Поэтому их нужно выне-

сти в отдельную таблицу «Ставка», связанную с таблицей «Список сотрудников», отношением «один-к-одному». Полученный результат изображен на рис. 1.23.

Рассматривая структуру базы, можно увидеть, что поля «Город», «Адрес», «Телефон» зависят только от полей «Фамилия», «Имя», «Отчество» первичного ключа, но не от всего ключа. Поэтому их нужно выделить в отдельную таблицу «Дом» и связать ее с основной таблицей в соотношении «один-к-одному». В результате будет получена структура базы данных во второй нормальной форме (рис. 1.24).

Рис. 1.24. Таблица во второй нормальной форме

Третья нормальная форма

Третья нормальная форма 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 значения другого поля, также не входящего в первичный ключ. Также отношение должно удовлетворять второй нормальной форме.

Я решил не рассматривать пример приведения таблиц к третьей нормальной форме, так как требования к ней достаточно очевидны. В общем случае таблицы не должны содержать вычисляемых полей, так как это приводит к дополнительной избыточности.

Нормализация проводится для того, чтобы устранить излишнюю избыточность из таблиц. Заметьте, не полностью избавить их от избыточности, а минимизировать ее в разумных пределах. Чем выше уровень нормализации, тем на большее число таблиц «разбиваются» сущности. В больших организациях базы данных могут состоять из сотен связанных таблиц, что отнюдь не упрощает их понимание. Другим недостатком нормализованной базы данных является то, что при обращении к ней запрос должен считывать связанные данные из

нескольких таблиц. Что, само собой, не добавляет производительности. Таким образом, при нормализации баз данных большого объема приходится идти на компромисс между избыточностью таблиц и удобством работы. Существуют и другие нормальные формы, но чаще всего пользуются именно перечисленными тремя.

Пользователи и роли

Безопасность данных, особенно предотвращение к ним доступа тех, у кого нет соответствующих прав, является серьезной проблемой. В клиент-серверных СУБД для этой цели используются понятия пользователя и его роли. *Пользователь* характеризуется учетной записью конкретного человека, которая включает в себя набор ограничений на доступ к данным, логин пользователя и пароль. Роль объединяет группу пользователей и позволяет задавать им те или иные общие правила доступа к данным. Используя списки пользователей, можно для конкретной таблицы назначить видимые поля. Можно разрешить той или иной роли или отдельному пользователю осуществлять с базой данных те или иные действия.

Системный каталог

Любая реляционная база данных, в которой поддерживаются такие объекты, как пользователи и роли, должна сохранять их списки. Кроме того, обычно с базой данных хранятся системные таблицы, триггеры, транзакции, хранимые процедуры — соответствующая часть памяти СУБД называется системным каталогом.

2 УРОК Архитектура приложений баз данных

Приложения, работающие с базами данных, обычно состоят из интерфейса пользователя, компонентов, предоставляющих доступ к базе данных, и компонентов, соединяющих их друг с другом и с источником данных. Составляя эти компоненты в определенной последовательности, можно достаточно легко разработать приложение, взаимодействующее с базой данных. Общая схема приведена на рис. 2.1.

Рис. 2.1. Обобщенная схема БД

Как видно из рисунка, база данных представляет собой соединение пользовательского интерфейса и модуля данных. Модуль данных предназначен для хранения соответствующих компонентов. Одним из них является источник данных, предоставляющий данные другим частям приложения. Вторым компонентом является набор данных, содержащий в себе базу данных. Дополняет картину компонент, реализующий соединение с базой данных.

Общая структура приложения баз данных

Как было написано во вводной части, приложение обычно состоит из пользовательского интерфейса и средств доступа к данным. Интерфейс пользователя обычно разрабатывается из стандартных компонентов и располагается на базовой форме приложения.

Приложение может содержать произвольное число форм и использовать любую парадигму работы с множественными документами (MDI или SDI). Обычно одна форма отвечает за выполнение группы однородных операций, объединенных общим названием.

Визуальные компоненты отображения данных расположены на вкладке Data Controls. Они по большей части являются стандартными компонентами отображения данных с небольшими изменениями и дополнениями.

В основе любого приложения баз данных лежат *наборы данных*, представляющие собой массивы записей, полученные из баз данных. Наборы данных хранятся в специальных компонентах, построенных на базе класса TDataSet. Для обеспечения связи набора данных с визуальными компонентами отображения данных предназначен специальный компонент — источник данных, реализуемый классом TDataSource. Компонент TDataSource обеспечивает передачу данных в визуальные компоненты, отслеживает синхронное изменение их состояния, если изменяется состояние связанного с ними набора данных, передачу измененных данных обратно в набор. Этот компонент располагается на вкладке Data Access.

Базовый механизм доступа к данным основывается на трех «китах»:

- компонентах-потомках класса TDataSet (наборах данных);
- компонентах TDataSource;
- визуальных компонентах отображения данных, формирующих интерфейс пользователя.

Эта схема показана на рис. 2.2.

При открытии набора данных в него автоматически передаются записи из таблицы базы данных, и курсор устанавливается на первую запись. При перемещении по набору данных значения записей автоматически обновляются (или не обновляются, в зависимости от расположения и типа курсора базы данных).

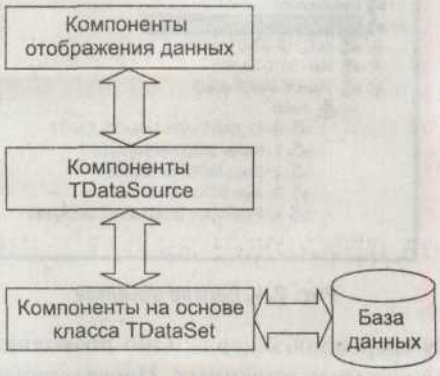


Рис. 2.2. Механизм доступа к данным

Модуль данных

Модуль данных (Data Module) представляет собой невизуальный контейнер, в котором размещаются компоненты доступа к данным. В модуле данных можно размещать только невизуальные компоненты. Модуль данных доступен программисту на этапе разработки в виде формы, в которую он может положить компоненты и настроить их свойства. Модуль данных отличается от обычной формы, так как берет свое начало от класса `TComponent`.

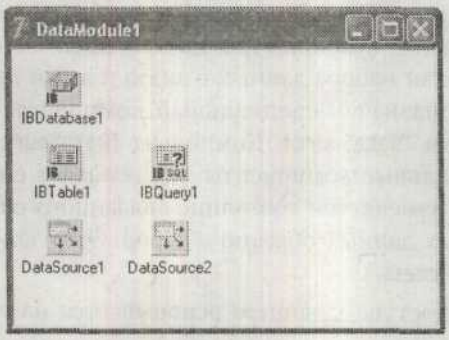


Рис. 2.3. Модуль данных

Для создания модуля данных можно воспользоваться репозиторием объектов, который активируется при выполнении команды меню New ▸ Data Module.

Вкладка Diagram открывает окно формирования моделей данных (диаграмм, схем). Модели данных позволяют наглядно отобразить связи, существующие между наборами данных. Это особенно удобно, если приходиться работать с большим количеством таблиц.

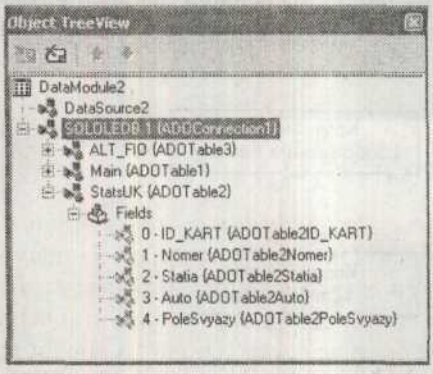


Рис. 2.4. Дерево объектов

На рис. 2.4 показано «дерево объектов». Оно позволяет отображать в списке объекты, размещенные в модуле данных. Перетаскивая объекты из «дерева объектов» в рабочую область Diagram, можно легко настраивать их свойства,

буквально соединяя объекты линиями связи. На рис. 2.5 показана модель некой базы данных. Для того чтобы создать подобную модель, потребуется совсем немного времени. При помощи кнопки Property connector легко связать таблицы с компонентом-провайдером. А кнопка Master\Detail connector позволяет установить связь «один-ко-многим» в диалоговом окне Field Link Designer.

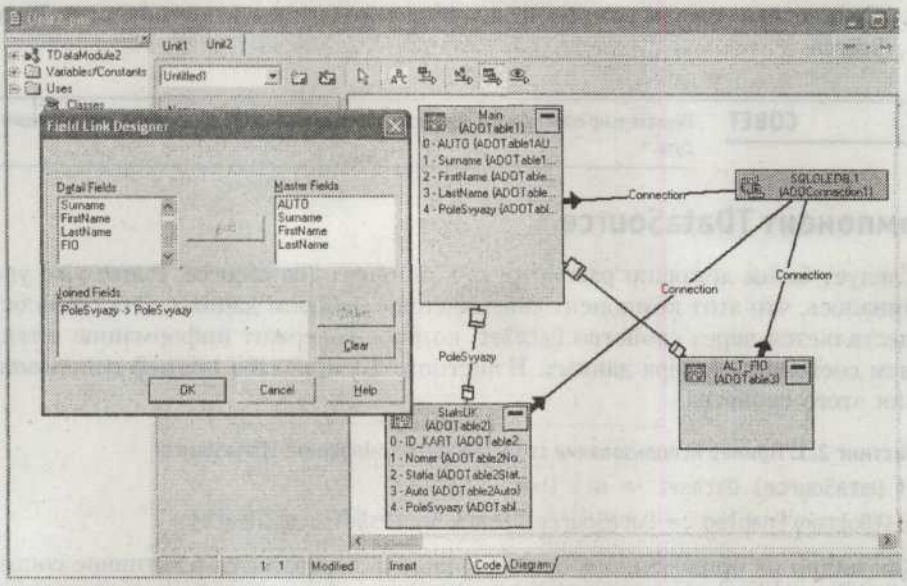


Рис. 2.5. Модель данных БД

Использование модуля данных при разработке приложения дает программисту массу преимуществ. Например, при изменении какого-либо свойства объекта в модуле данных изменения автоматически отобразятся во всех связанных с ним компонентах. Также немаловажным преимуществом является то, что все компоненты, обеспечивающие доступ к базе данных, собраны в одном месте. Это облегчает восприятие базы в целом.

Подключение данных

Компонент набора данных является основой приложения, взаимодействующего с базами данных. Он содержит набор данных из выбранных таблиц и позволяет эффективно управлять им. В процессе работы данный компонент использует функции соответствующей технологии через совокупность интерфейсов (методов, предоставляемых данной технологией).

Рассмотрим основные шаги, которые потребуются для подключения набора данных и получения информации из него:

1. Поместить компонент в модуль набора данных и связать его с базой данных.
2. Подключить к компоненту таблицу, используя свойство `TableName`.

3. Активизировать связь, установить для компонента значение свойства Active как True.
4. Разместить на форме компонент TDataSource. Он обеспечивает взаимодействие визуальных компонентов отображения данных с набором данных.
5. Связать набор данных и компонент TDataSource.
6. Связать визуальные компоненты отображения данных с компонентом TDataSource.

СОВЕТ Желательно присваивать объектам простые и осмысленные имена, отражающие их суть.

Компонент TDataSource

Следует более детально рассмотреть компонент TDataSource. Ранее уже упоминалось, что этот компонент связывается с набором данных. Эта связь осуществляется через свойство DataSet, которое содержит информацию о текущем состоянии набора данных. В листинге 2.1 приведен пример использования этого свойства.

Листинг 2.1. Пример использования свойства State компонента TDataSource

```
if DataSource1.Dataset <> nil then
PostButton.Enabled := DataSource1.State in [dsEdit, dsInsert];
```

Как видно из приведенного примера, кнопка переводится в активное состояние в том случае, если связанный набор данных в текущий момент доступен для редактирования. У этого компонента существует набор свойств и методов, которые облегчают работу с ним.

Свойство AutoEdit автоматически переводит набор данных в состояние редактирования, если имеет значение True, когда связанный элемент ввода получает фокус.

Метод Edit переводит связанный набор данных в состояние редактирования.

Метод-обработчик OnDataChange вызывается при редактировании данных в связанном визуальном компоненте.

Метод-обработчик события OnUpdateData вызывается перед тем, как измененные данные будут сохранены в наборе данных. Обработчик вызывается перед выполнением метода Post.

Метод-обработчик события OnStateChange вызывается, когда изменяется состояние связанного набора данных.

Набор данных

Массив записей, полученный приложением по собственному запросу, называется набором данных. Набор данных как объект ведет свое начало от класса TDataSet и наследует его свойства.

Класс TDataSet является базовым классом иерархии, он инкапсулирует *абстрактный набор данных* и реализует общие методы работы с ним. На основе базового класса реализованы специальные компоненты, предоставляющие разработчику доступ к той или иной технологии.

Для чтения или записи в набор данных его необходимо для начала открыть. Открыть набор данных можно двумя способами.

Можно присвоить свойству Active значение True либо вызвать метод Open. Завершение работы с набором данных производиться сходным образом. Либо свойству Active присваивается значение False, либо вызывается метод Close. Перед открытием набора данных автоматически инициируется событие BeforeOpen, а после открытия, когда в набор будут переданы данные, активируется событие AfterOpen.

Перед закрытием набора данных вызываются методы BeforeClose и AfterClose.

Используя данные методы-обработчики, можно выполнить в приложении некоторые действия, как это показано в листинге 2.2.

Листинг 2.2. Обработка события, возникающего при закрытии набора данных

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Желаете сохранить?', mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { Сохранение изменений}
      mrNo:     CustTable.Cancel; { Отмена изменений}
      mrCancel: Abort;            { Прекращение закрытия набора данных }
    end;
  end;
end;
```

В приведенном примере производится обработка события, возникающего перед тем, как набор данных будет закрыт.

Навигация по набору данных

Запись, выбранная в какой-либо момент времени в наборе данных, называется курсором. После открытия набора данных курсор автоматически устанавливается на первую запись.

Для перемещения по набору данных используются методы Next, Prior, First и Last.

Методы Next и Prior перемещают курсор на следующую и предыдущую запись соответственно, методы First и Last — на первую и последнюю.

Свойство Eof булева типа показывает, достиг курсор набора данных последней записи или нет.

Номер текущей записи и перемещение на определенную запись по ее номеру можно выполнить при помощи целочисленного свойства RecNo, присваивая или получая его значение.

Для перемещения по набору данных обычно используется метод MoveBy. Параметр метода Distance указывает число записей, на которое будет осуществлен переход. Если параметр имеет отрицательное значение, курсор смещается назад.

Для того чтобы выяснить размер записи в байтах, следует обратиться к свойству RecordSize.

В тех случаях, когда требуется перемещение по большому набору данных, для ускорения можно отключать отображение информации из набора данных в связанных элементах отображения и редактирования. Обычно эту возможность используют для ускорения работы приложения во время массовой обработки данных. В этот момент отображать все изменения на форме совершенно не нужно, поэтому отображение данных отключается на время. Для этого используются методы DisableControls и EnableControls. Первый из них отключает отображение информации в органах управления формы, а второй — возвращает им эту способность

Для получения числа записей, содержащихся в наборе, следует использовать свойство RecordCount.

Редактирование набора данных

Перед тем как изменять набор данных, следует узнать, возможно ли его изменение при помощи свойства CanModify, которое принимает значение True, если набор данных может быть изменен.

Метод Edit переводит набор данных в состояние редактирования. В некоторых случаях набор данных переводится в состояние редактирования автоматически, например при его изменении через связанные элементы редактирования или при использовании некоторых методов, таких как Insert или Append.

Для сохранения измененных данных вызывается метод Post. Этот метод может вызываться как разработчиком, так и самим набором данных при переходе на другую запись.

Для различных типов баз данных действие метода Post несколько различается:

- Для наборов данных, связанных с базой данных напрямую, изменения сохраняются сразу на диск.
- При использовании клиентских наборов изменения сохраняются в локальном кэше базы данных. Для их сохранения на сервере необходимо вызвать метод ApplyUpdates.

В некоторых ситуациях бывает необходимо отменить произведенные действия. В этом случае вызывается метод Cancel. Метод возвращает набор данных в состояние, которое было при последнем вызове метода Post.

Для добавления новой записи по месту расположения курсора используется метод Insert. А если необходимо добавить запись в конец набора данных, следует вызывать метод Append.

Выбранная запись удаляется методом Delete. А метод ClearFields очищает поля выбранной записи. Набор данных должен находиться в режиме ввода новой записи либо в режиме редактирования.

Поиск записей и фильтрация в наборах данных

Метод Locate используется для поиска информации в базе данных. Он ищет первую запись, удовлетворяющую критерию поиска, и, если такая запись найдена, делает ее текущей. В случае «удачного» поиска метод возвращает True, в противном случае — False.

В состав параметров этого метода входит список KeyFields. В этом параметре указывается список полей (минимум одно), по которым будет производиться поиск. В том случае, если поиск будет производиться по нескольким полям, их названия перечисляются через точку с запятой. Значения полей, по которым производиться поиск, задаются в вариантном массиве KeyValues.

В свойстве TLocateOption задаются необязательные дополнительные опции поиска:

- Значение loCaseInsensitive указывает, что поиск ведется без учета регистра символов.
- Значение loPartialKey используется в том случае, если ключевые значения полей поиска могут включать только часть значения поля. Например, если в ключе поиска задано значение NN, а поле содержится значение NNOR, то метод остановит поиск на этой записи, так как она содержит в своем поле часть ключевого значения.

Метод Locate позволяет производить поиск по любому полю. Поля, по которым будет производиться поиск, могут не состоять в первичном ключе и могут не индексироваться. Но если поле, по которому производится поиск, включено в тот или иной индекс, метод использует его.

Листинг 2.3. Пример использования Locate

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company, Vendor', VarArrayOf(['Sight
Diver','Point']), SearchOptions);
end;
```

В примере, приведенном в листинге 2.3, производится поиск в наборе данных по полям Company и Vendor.

Впрочем, поиск можно производить с более жесткими условиями. Метод Lookup находит запись, точно удовлетворяющую условиям поиска. Этот метод не переводит курсор на найденную запись, но возвращает значения некоторых полей найденной записи в виде вариантного массива.

В списке KeyFields указывается список полей, по которым будет производиться поиск. Если поиск ведется по нескольким полям, то они разделяются точкой с запятой. В массиве KeyValues указываются ключевые значения, по которым будет производиться поиск. В списке ResultFields указываются поля, которые будут возвращены в вариантном массиве в случае, если поиск окажется успешным.

В листинге 2.4 приведен пример, в котором производится поиск по полю Company значения «Professional Divers, Ltd.». В результате возвращается вариантный массив, содержащий значения полей Company, Contact и Phone.

Листинг 2.4. Использование Lookup. Поиск по одному полю

```
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
'Company;Contact; Phone');
end;
```

В листинге 2.5 приведен пример, аналогичный примеру, приведенному в листинге 2.4. Отличие заключается в том, что в данном примере поиск производится по нескольким полям.

Листинг 2.5. Использование Lookup. Поиск по нескольким полям

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver',
'Christiansted']),    'Company; Addr1; Addr2; State; Zip');
end;
```

В случае, если метод ничего не нашел, он возвращает значение null. Проверить данное условие можно при помощи условного оператора if VarType(LookupResults) = varNull.

Свойство Filter позволяет задать критерии фильтрации. Набор данных будет отфильтрован, как только его свойство Filtered примет значение True. В условие фильтрации можно включать логические операторы and, or, not, <>, <, >, <=, >=.

Например:

1. ([size] > 20) and ([size] <= 40).

2. ([size] > 20) and ([weight] < 30).

3. ([name] <> 'Tetras') and ([size] < 10).

Как видно из приведенных примеров, сравнение заданного условия производится со значением поля.

При помощи свойства FilterOptions можно задать необязательные параметры фильтрации, как и для метода Locate.

Событие OnFilterRecord возникает, когда в свойстве Filtered устанавливается значение True. Метод-обработчик события TFilterRecordEvent имеет два параметра. В параметре DataSet передается фильтруемый набор данных, а в параметре Accept — переменная, в которую помещается значение True в случае, если текущая запись соответствует условиям фильтрации. Пример приведен в листинге 2.6.

Листинг 2.6. Пример использования события OnFilterRecord

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept:=((DataSet.FieldByName('Size').AsInteger+2)<10)
  and ((DataSet.FieldByName('Weight').AsFloat/2)<2);
end;
```

В современных условиях практически все операции выборки, сортировки, поиска осуществляются средствами SQL.

Состояния набора данных

Набор данных может находиться в нескольких состояниях, в которые он переходит в процессе своего функционирования. Состояния меняются в результате *перемещения по набору данных, редактирования, ввода новых записей* и т. д. Отслеживая текущее состояние набора данных, программист может предпринять какие-либо действия. Текущее состояние набора данных содержится в свойстве State, значение которого принадлежит перечислимому типу TDataSetState. Возможные состояния указаны в табл. 2.1.

Таблица 2.1. Состояния набора данных

Значение свойства	Состояние набора данных
dsInactive	Набор данных закрыт
dsBrowse	Набор данных находится в состоянии просмотра. *Редактирование невозможно. Данное состояние является* «обычным», устанавливается по умолчанию
dsEdit	Выбранная запись может быть отредактирована
dsInsert	Запись «помещена» в набор, но не утверждена в нем методом Post. Запись может быть изменена, принята или отменена

продолжение

Таблица 2.1 *(продолжение)*

Значение свойства	Состояние набора данных
dsSetKey	Используется механизм поиска по ключу или по диапазонам
dsCalcFields	Устанавливается при возникновении события OnCalcFields
dsFilter	Устанавливается при возникновении события OnFilterRecord
dsNewValue	Устанавливается при обращении к свойству NewValue поля набора данных
dsOldValue	Устанавливается при обращении к свойству OldValue поля набора данных
dsCurValue	Устанавливается при обращении к свойству CurValue поля набора данных
dsBlockRead	Устанавливается в случае «ускоренного перемещения по набору данных». Переводится в данный режим вызовом метода DisableControls
dsInternalCalc	Устанавливается при расчете значений полей, для которых FieldKind имеет значение fkInternalCalc
dsOpening	Устанавливается в момент открытия набора данных

Набор данных может переходить в состояния `dsNewValue`, `dsOldValue`, `dsCurValue`, `dsInternalCalc` только в том случае, когда доступ к базе данных осуществляется через компонент `TClientDataSet` или используется кэш данных.

Когда набор открывается, он находится в состоянии `dsOpening`. После того как он открылся, активируется состояние просмотра `dsBrowse`. Это состояние является основным для набора данных.

В листинге 2.7 приведен пример, в котором производится проверка того, находится ли набор данных в состоянии редактирования или вставки новой записи, после этого выполняется сохранение изменений.

Листинг 2.7. Демонстрация использования сведений о состоянии набора данных

```
with DataSet do begin
  if state in [dsInsert, dsEdit] then
    Post;
end;
```

Работа с полями

Любая запись набора данных представляет собой совокупность полей. Поля представляют собой объекты, производные от типа `TField`. Каждое поле имеет определенный тип данных и соответствующий ему объект. Свойство `Fields` содержит коллекцию полей набора данных.

Использование объектов-полей

К полю набора данных можно обратиться по его имени, используя метод FieldByName, или через свойство Fields. В первом случае используется конструкция DataSet.Fields[i].AsInteger, а во втором — DataSet.FieldByName(«SomeField»).AsInteger.

Каждый объект имеет ряд свойств, определяющих структуру поля. Список этих свойств не так уж велик:

- название поля;
- тип данных поля;
- вид поля;
- размер поля;
- иные атрибуты, присущие полю конкретного типа.

Свойство FieldDefs содержит список параметров каждого поля из набора данных. В примере, приведенном в листинге 2.8, демонстрируется использование свойства FieldDefs. При помощи метода AddFieldDef добавляется несколько объектов полей с определенными свойствами.

Листинг 2.8. Использование FieldDefs

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with ClientDataSet1 do
  begin
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := 'Field1';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 10;
      Name := 'Field2';
    end;
  end;
end;
```

Как видно из примера, каждое поле имеет некоторый тип данных (DataType), размер (DataSize) и название (FieldName). Также каждое поле имеет какой-либо вид (FieldKind), определяющий его функциональное назначение.

Свойство FieldCount возвращает число полей, содержащихся в данном наборе данных. Конечно, одним количеством полей не обойтись. Чаще всего требу-

ется получить еще и дополнительную информацию о полях. Для этой цели используется метод `GetFieldNames`. В качестве параметра этому методу передается список. После отработки метода в переданном списке будут размещены названия всех полей данного набора данных.

Название столбца содержится в свойстве `DisplayLabel`. Оно отображается в компоненте таблицы в качестве заголовка столбца. Следует отметить, что название столбца отображаемой таблицы не обязано совпадать с именем соответствующего поля.

Статические и динамические поля

Существует два способа задания состава полей в наборе данных. По умолчанию используется динамический способ. Но разработчик может принудительно использовать статический способ установки списка полей.

Динамические поля создаются автоматически при каждом открытии набора данных, если ранее не были созданы объекты полей. Любой объект поля является наследником класса `TField`, а его конкретный тип зависит от типа данных, содержащихся в таблице.

Статические поля создаются на этапе разработки, их свойства доступны в инспекторе объектов. Также на этапе разработки объекту статического поля можно присвоить имя.

Компонент набора данных после подключения к таблице базы данных без дополнительных настроек использует только динамические поля. Если на этапе разработки задано хоть одно статическое поле, динамические поля создаваться уже не будут.

Добавить к списку статических полей новое поле очень просто, для этого необходимо выбрать команду Fields Editor контекстного меню набора данных либо щелкнуть на нем два раза.

На рис. 2.6 представлен вид редактора полей набора данных. В наборе определено несколько статических полей.

Рис. 2.6. Редактор полей

Типы и виды полей

Функциональное назначение поля определяется свойством `FieldKind`. В общем случае его назначение определяется автоматически на этапе создания поля. Вид поля можно изменить в процессе работы приложения. Если новый тип данных поля не совместим со значением поля, то будет выведено сообщение об ошибке.

Значение свойства `TFieldKind` относится к перечислимому типу. Соответственно, существует ограниченное количество значений этого свойства:

- `fkData` — поле данных, имеющее то же тип, что и поле базы данных;
- `fkCalculated` — вычисляемое поле;
- `fkLookup` — поле синхронного просмотра;
- `fkInternalCalc` — внутреннее вычисляемое поле;
- `fkAggregate` — агрегатное поле.

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.

Тип данных поля содержится в свойстве `DataType`. На основе приведенных типов полей и класса `TField` порождаются объекты полей определенного типа. Список этих классов приведен в табл. 2.2.

Таблица 2.2. Классы полей

Класс	Описание поля
TADTField	Поле абстрактного типа данных. Данное поле может содержать массивы полей, наборы данных и любые иные типы данных
TAggregateField	Агрегатное поле
TArrayField	Поле содержит массив
TDateField	Поле даты
TDateTimeField	Поле даты и времени
TSmallIntField	Целочисленное поле. Под хранение данных отводится по два байта
TFloatField	Поле для хранения вещественных чисел
TSQLTimeStampField	Поле даты и времени набора DBExpress
TAutoIncField	Автоинкрементное поле
TFMTBCDField	Форматированное двоично-десятичное поле
TStringField	Строковое поле

продолжение

Таблица 2.2 *(продолжение)*

Класс	Описание поля
TBCDField	Двоично-десятичное поле
TGraphicField	Графическое поле
TTimeField	Поле времени
TBinaryField	Двоичное поле
TGuidField	Поле, хранящее уникальный идентификатор GUID
TVarBytesField	Байтовое поле переменной длины
TBlobField	BLOB-поле
TIDispatchField	Поле, хранящее указатель на интерфейс диспетчеризации
TVariantField	Вариантное поле
TBooleanField	Логическое поле
TIntegerField	Поле для хранения целочисленных данных длиной 32 бита
TWideStringField	Большое строковое поле
TBytesField	Байтовое поле фиксированной длины
TInterfaceField	Поле, хранящее указатель на интерфейс
TWordField	Поле для 16-разрядных целочисленных значений без знака
TCurrencyField	Поле для хранения информации в виде денежных сумм
TLargeintField	Поле для хранения целочисленных данных длиной 64 бита
TDataSetField	Поле, содержащее вложенный набор данных
TMemoField	Memo-поле

Delphi поддерживает большое количество полей. Есть поля для работы с COM, специальные типы данных для работы с объектно-ориентированными БД, что позволяет разработчику проявлять потрясающую гибкость в вопросах планирования собственной работы.

Стандартные компоненты, связываемые с набором данных

Наиболее распространенным компонентом, без которого не обходится практически ни одна СУБД, является компонент, реализующий табличное представление данных `TDBGrid` (рис. 2.7).

NAME	SIZE	WEIGHT	AREA
Angel Fish	2	2	Computer Aquariums
Boa	10	8	South America
Critters	30	20	Screen Savers
House Cat	10	5	New Orleans
Ocelot	40	35	Africa and Asia
Parrot	5	5	South America
Tetras	2	2	Fish Bowls

Рис. 2.7. Компонент `TDBGrid`

Компонент TDBGrid используется для отображения содержимого набора данных в табличном виде, когда строки соответствуют записям набора данных, а столбцы — полям записи.

Объект DBGrid связывается с источником данных через свойство DataSource, которое в свою очередь ссылается на набор данных. В процессе выполнения приложения можно менять источники данных, используя одну таблицу для отображения данных от нескольких источников.

Для определения состава столбцов можно использовать редактор столбцов, реализованный диалоговым окном Column Editor. Он показан на рис. 2.8.

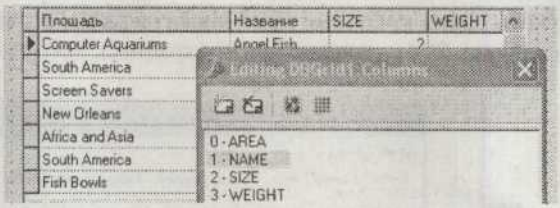


Рис. 2.8. Редактор столбцов

Если редактор столбцов не использовался, то используются поля, объявленные при помощи редактора полей набора данных. При этом все свойства столбцов и порядок их следования наследуются из редактора набора данных. В случае, когда производится динамическое формирование объектов TField, порядок следования полей и их характеристики соответствуют тем, что были заданы при определении структуры записи таблицы базы данных при создании.

Редактор столбцов вызывается двойным щелчком на компоненте. В диалоговом окне редактора можно установить различные свойства объектов-столбцов, такие как цвет фона, шрифт заголовка, основного поля таблицы и выборочного столбца.

Обратиться к конкретному столбцу компонента TDBGrid можно как к элементу индексированного массива TDBGrid.Columns.Items[i].

Также для отображения и редактирования данных используются элементы TDBEdit. Они представляют собой модифицированные компоненты TEdit. Связь с источником данных осуществляется при помощи свойства DataSource. Поле, с которым связывается данный компонент, указывается в свойстве DataField.

Компонент TDBMemo представляет собой многострочный редактор. Обычно связывается с memo-полями. С его помощью часто сохраняют обширные текстовые массивы в соответствующих полях.

Компонент DBComboBox представляет собой список, связываемый с определенным полем набора данных. Значения списка хранятся в свойстве Items. Значения в список можно добавлять, удалять из него, сохранять и загружать с помощью методов Delete, Insert, LoadFromFile и SaveToFile.

Параметр Index указывает на позицию строки в списке, а параметр FileName содержит полный путь к файлу, откуда загружались значения списка. Данный компонент удобно использовать при организации пользовательских словарей.

Компонент TDBImage предназначен для отображения графической информации, хранящейся в связанном с ним BLOB-поле (binary large object). Связывается этот объект с набором данных через те же свойства, что и остальные компоненты отображения данных. Изображение, получаемое из набора данных, хранится в свойстве Picture.

Компонент TDBRichEdit представляет собой многострочный редактор с возможностью отображать форматированный текст. Данный компонент идеально подходит для отображения различных примечаний или хранения отчетов.

Компонент TDBChart служит для построения графиков на основе данных, получаемых из набора данных. Иногда при помощи всего нескольких щелчков мышью можно получать очень красивые объемные графики, отражающие различные зависимости. На рис. 2.11 приведен пример графика, построенного в TDBChart.

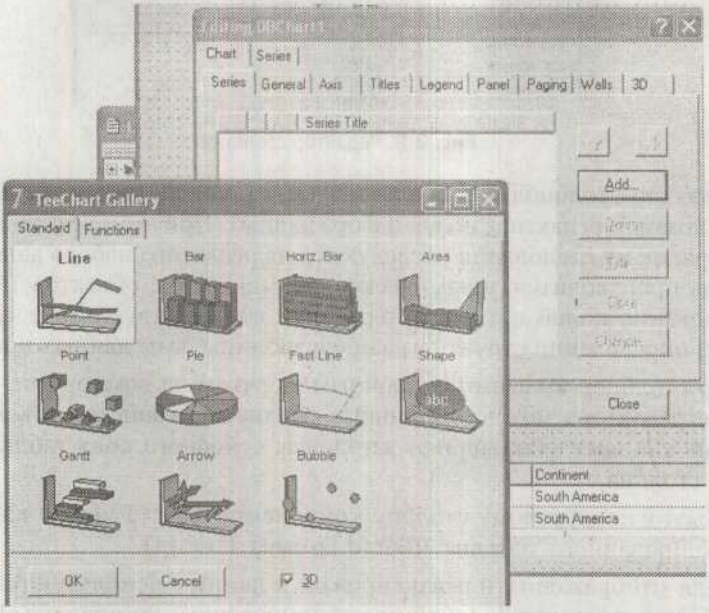


Рис. 2.9. Создание графика

Для того чтобы построить график, надо выполнить команду контекстного меню Edit Chart. В результате будет активировано диалоговое окно Editing DBChart, показанное на рис. 2.9. Для создания новой серии данных необходимо нажать кнопку Add. В результате будет активировано следующее диалоговое окно TeeChart Gallery, в котором можно будет выбрать тип графика. Перейдя на вкладку Series, можно получить доступ к настройкам данного графика. На этой странице расположены вкладки Format, General, Mark и Data Source. На вкладке Format можно задать различные параметры отображения графика. Вкладка General позволяет задать несколько свойств общего характера. На вкладке Mark можно определить режим подписи графика — легенду, размещение пояснений к нему и другие параметры. А вкладка Data Source, показанная на рис. 2.10, по-

зволяет задать источник данных и определить поля, по которым будет строиться график.

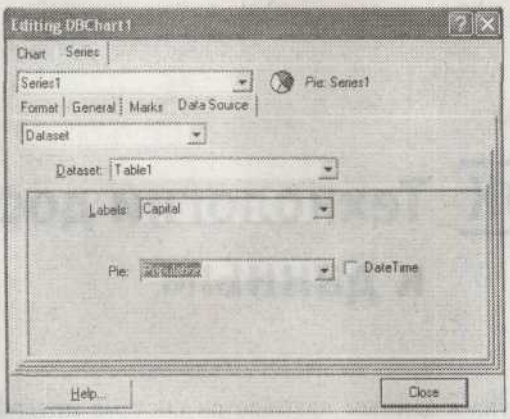


Рис. 2.10. Выбор источника данных

Из раскрывающегося списка можно выбрать тип источника данных строящегося графика. В данном примере стоит остановиться на значении Dataset. Далее из списка Dataset нужно выбрать конкретный набор данных. В списке Labels необходимо выбрать поле, значения которого будут подписываться под значениями графика. Поле, на основании которого будет построен график, выбирается в списке Pie из доступных полей. Полученный график показан на рис. 2.11.

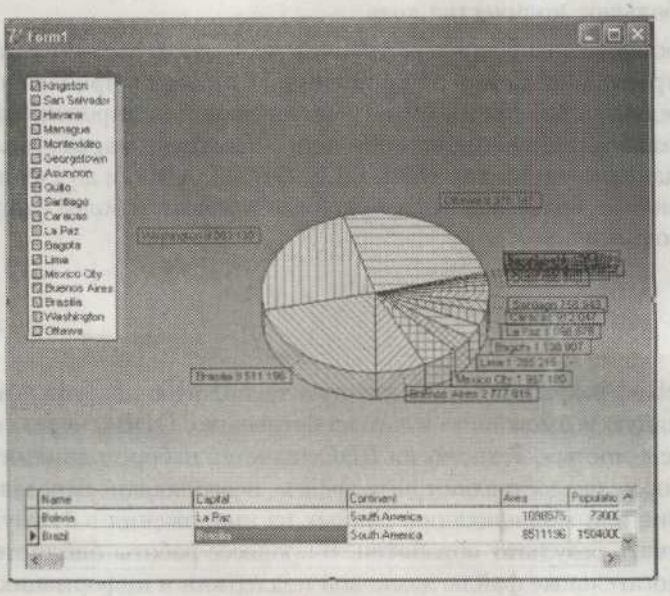
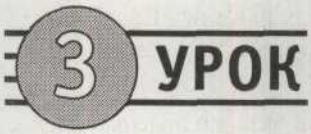


Рис. 2.11. График построенный в TDBChart



УРОК 3 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.

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 Data-Base Connectivity), например к файлам электронных таблиц (Excel, Lotus 1-2-3), и серверам баз данных, не имеющим драйверов SQL Links (например, Gupta/Centura);
- 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 рис. 3.1 представлена схема, на которой показана связь приложений и BDE.

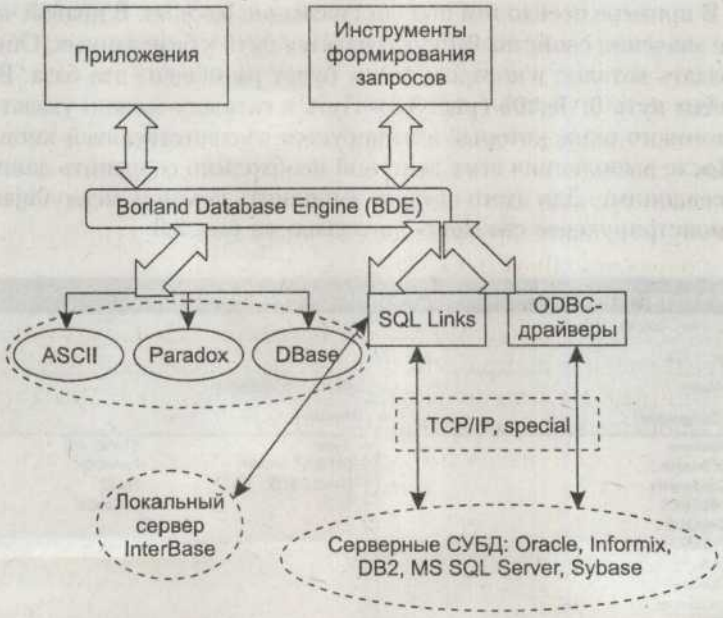


Рис. 3.1. Связь приложений с источниками данных при помощи BDE

Создание псевдонима базы данных

При работе с таблицами локальных БД или СУБД сама база размещается либо в каталоге на диске и хранится в виде отдельного набора файлов, либо на удаленном сервере. Обращение к базе данных происходит по ее псевдониму (*Database Alias*). Псевдоним должен быть зарегистрирован на конкретной

машине, с которой будет производиться доступ к базе данных. Псевдонимы баз данных и другие настройки BDE хранятся в файле idapi32.cfg, расположенном в том же каталоге, что и файлы BDE.

Создание и работа с псевдонимами баз данных производится из утилиты BDE Administrator. Для того чтобы создать псевдоним, в главном меню утилиты надо выбрать пункт Object ▸ New. В появившемся окне, показанном на рис. 3.2, будет предложено выбрать драйвер для доступа к базе данных.

Рис. 3.2. Окно выбора драйвера

В рассматриваемом случае надо оставить выбранным пункт STANDARD и нажать кнопку OK. Созданный псевдоним базы данных можно переименовать по своему вкусу. В примере псевдоним получил название TestAlias. В правой части окна в качестве значения свойства Path указывается путь к базе данных. Обычно требуется создать каталог, в котором позже будет размещена эта база. В примере используется путь D:\TestDB (рис. 3.3). Путь к каталогу можно указать с помощью диалогового окна, который активируется соответствующей кнопкой в поле Path. После выполнения этих действий необходимо сохранить данные о созданном псевдониме. Для этого следует выполнить команду меню Object ▸ Apply. Окно, демонстрирующее сказанное, показано на рис. 3.3.

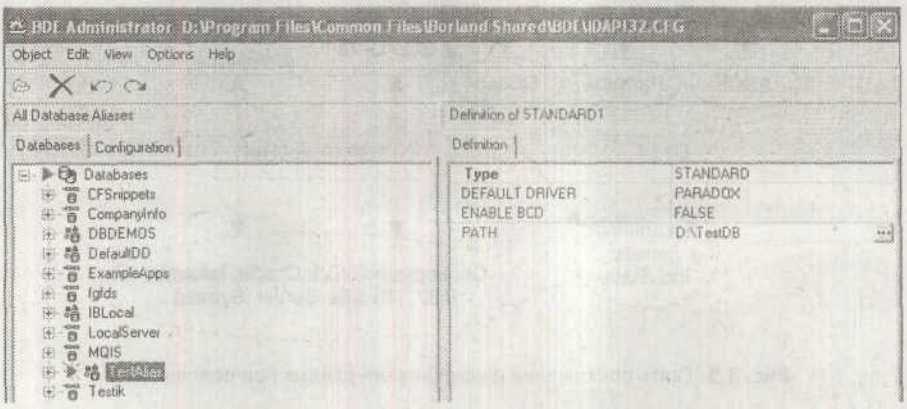


Рис. 3.3. Создание и настройка псевдонима

Создание таблиц базы данных

Для создания таблиц базы данных можно использовать утилиту Database Desktop. Сначала нужно задать рабочий каталог. Для этого следует выполнить

команду меню Object ▸ Working Directory и из списка Aliases выбрать только что созданный псевдоним. Таким образом, при открытии диалога выбора таблиц (для локальных баз данных) будет открываться именно тот каталог, в котором они располагаются.

Для создания таблицы необходимо выбрать пункт меню Object ▸ New ▸ Table. В появившемся окне необходимо выбрать тип таблицы. Для реализации тестового примера нужно выбрать значение Paradox 7. В результате будет активировано диалоговое окно, показанное на рис. 3.4. Каждая строка соответствует определенному полю в создаваемой таблице.

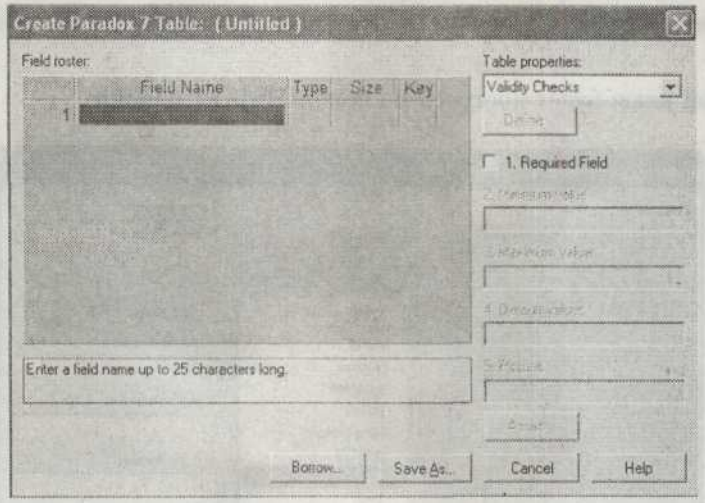


Рис. 3.4. Определение структуры таблицы

В столбце FieldName указываются имена полей создаваемой таблицы. Столбец Type задает их типы. Для строковых полей в столбце Size указывается их размер. А в столбце Key помечаются символом звездочки те поля, которые будут входить в состав первичного ключа.

Для тестового примера нужно создать таблицу, содержащую информацию о студентах. В ней будут содержаться поля STUDENT, GROUP и DATAPOSTUP. Чтобы выбрать тип поля при его определении, необходимо перейти в поле Type и нажать клавишу пробела. Точно таким же образом отмечается вхождение поля в ключ.

Описание полей созданной таблицы базы данных приведено в табл. 3.1.

Таблица 3.1. Описание полей таблицы «Студенты»

Field Name	Type	Size	Key	Примечание
STUDENT	A	30	*	Поле типа Alfa
GROUP	S			Поле типа Short
DATAPOSTUP	D			Поле типа Data

Любому полю таблицы можно задать те или иные ограничения:

- Модификатор Required Field указывает, что поле должно быть обязательно заполнено данными.
- Ограничение Minimum Value определяет минимальное значение, которое может содержаться в поле.
- Maximum Value задает максимальное значение, которое может содержаться в поле.
- Параметр Default Value определяет значение, которое поле будет принимать по умолчанию.
- Параметр Picture определяет шаблон значений, которые будут храниться в поле.

Первичный ключ будет построен по полям STUDENT и GROUP.

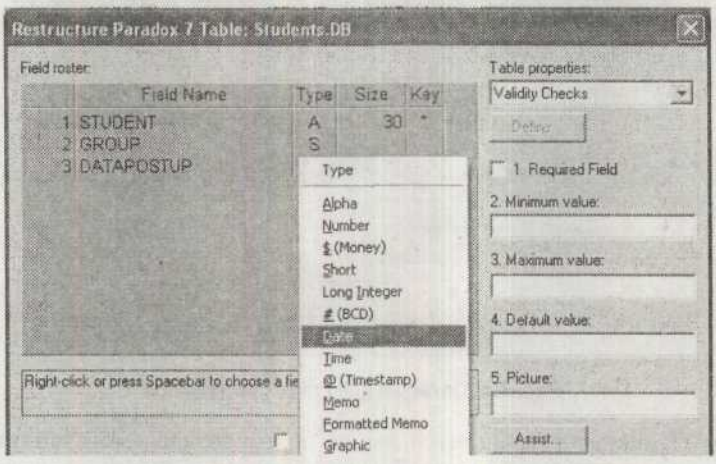


Рис. 3.5. Таблица «Студенты»

Для полей Student и Group следует установить ограничение Required Field. На рис. 3.5 показан один из этапов формирования таблицы.

Далее необходимо выбрать языковый драйвер для того, чтобы русский текст отображался в базе данных без искажений. Для этого в списке Table properties необходимо выбрать пункт Table Language, затем нажать кнопку Modify и выбрать в появившемся списке значение Pdox ANSY Cyrillic. Теперь таблицу можно сохранить под именем Students при помощи кнопки Save/Save As.

Определение индексов и ссылочной целостности

Для продолжения работы следует создать таблицу, содержащую данные об оценках студентов. Структура этой таблицы показана в табл. 3.2. Помимо информативных полей таблица содержит автоинкрементное поле, которое будет использоваться в качестве ключа.

Таблица 3.2. Таблица «Оценки»

Field Name	Type	Size	Key	Примечание
AUTO	+		*	Автоинкрементное поле
STUDENT	A	30		Поле типа Alfa
EXAM	A	30		Поле типа Alfa
GRADE	S			Поле типа Short
DATAEXAM	D			Поле типа Data

Для поля Grade, в котором будут храниться оценки, необходимо задать определенные ограничения. Следует выбрать параметр Required Field, для параметра Minimum Value установить значение, равное трем, а для Maximum Value — пяти.

Созданную таблицу нужно сохранить под именем GradeTab. В ней также потребуется создать индекс по полям STUDENT, GROUP и GRADE. Для этого в списке Table properties необходимо выбрать значение Secondary Indexes и нажать кнопку Define. В результате будет активировано окно, показанное на рис. 3.6.

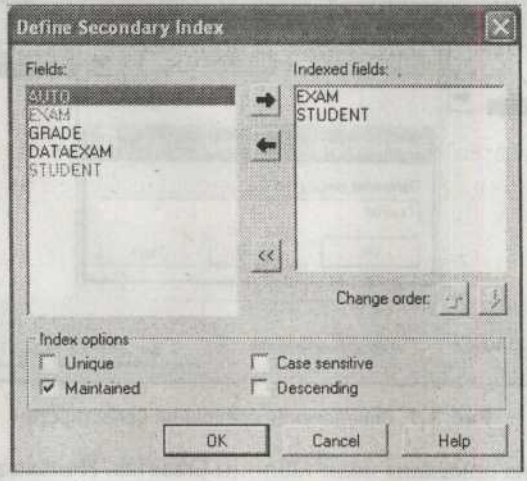


Рис. 3.6. Создание индекса

Вторичный индекс может обладать дополнительными возможностями и ограничениями, которые налагаются на него при помощи флажков в диалоговом окне:

- Параметр Unique обязывает все значения, входящие в состав индекса, быть уникальными.
- Case sensitive указывает, что индекс будет учитывать регистр символов.
- Параметр Maintained указывает, что индекс будет обновляться каждый раз, когда будет производиться модификация набора данных. Если этот флажок не установлен, индекс обновляется только при непосредственной работе с ним.

- Параметр Descending задает обратное направление сортировки значений, входящих в индекс. Если флажок не установлен, то сортировка производится в прямом направлении.

Созданный индекс нужно сохранить. Следует нажать кнопку OK и ввести имя индекса SomeIndex. Впоследствии этот индекс можно изменить. Для этого необходимо выбрать его в списке и нажать кнопку Modify.

Теперь нужно определить ссылочную целостность. Таблицы Students и GradeTab находятся в отношении «один-ко-многим». Один студент может сдать несколько экзаменов на разные оценки. Теперь эту связь надо жестко прописать в проекте.

Нужно открыть таблицу GradeTab и выбрать пункт меню Table ▸ Restructure. Таблица откроется для внесения изменений. В списке Table properties нужно выбрать элемент Referential Integrity и нажать кнопку Define. В результате будет активировано диалоговое окно, показанное на рис. 3.7.

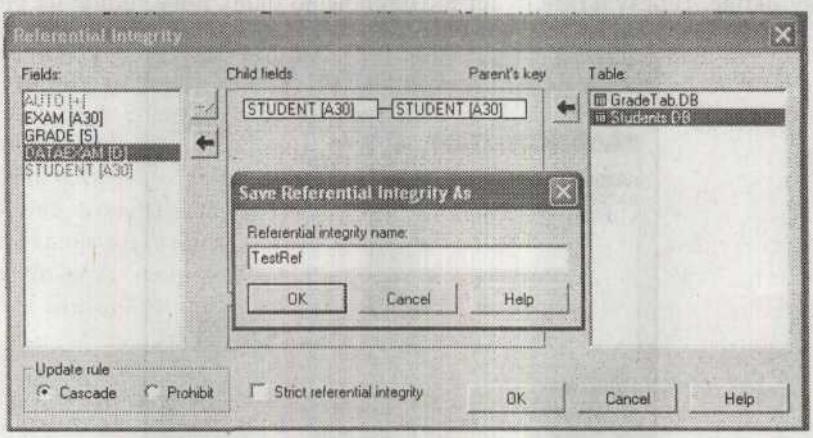


Рис. 3.7. Определение ссылочной целостности

В левой части окна показаны поля таблицы GradeTab. Необходимо создать связь по полям STUDENT и GROUP. Соответственно, их необходимо выбрать из дочерней таблицы двойным щелчком мыши. Поля будут помещены в список Child fields. Далее необходимо выбрать таблицу Students, находящуюся в списке Table, который находится в правой части окна. Подходящие поля будут загружены в таблицу Parent's key и будет автоматически установлена связь.

Для созданной ссылочной целостности можно определить дополнительные параметры:

- Группа Update rule позволяет задавать правила модификации записей:
  - переключатель Cascade указывает, что обновления данных производятся каскадно;
  - переключатель Prohibit запрещает изменение значений, входящих в первичный ключ, пока есть связанные записи.

- Флажок Strict Referential Integrity в активизированном состоянии запрещает работать с таблицей Paradox из пакета Paradox for DOS.

Осталось лишь присвоить отношению имя TestRef и сохранить его, нажав на кнопку OK.

Разработка простого приложения БД

Теперь, когда база данных создана, можно перейти к созданию первого приложения для работы с ней. Прежде всего в Delphi необходимо создать новый проект. Затем в него нужно добавить модуль данных командой главного меню File ▸ New ▸ Data Module. Модуль нужно сохранить под именем DataModule. В нем нужно расположить два компонента TTable, которые находятся на вкладке BDE, и два компонента TDataSource. Для обоих компонентов TTable в свойстве Database Name надо выбрать из списка значение TestAliace. Компонент Table1 надо переименовать в StudentsTbl при помощи свойства Name, а компонент Table2 — в GradeTbl. Также компоненту DataSource1 надо дать имя StudentsSrc, а DataSource2 — GradeSrc.

Теперь надо связать попарно компоненты таблиц и источников данных. При помощи свойства Dataset надо увязать StudentsTbl с StudentsSrc, а GradeTbl — с GradeSrc. На этом этапе уже можно определить отношения между таблицами. Для этого в свойстве MasterSource компонента GradeTbl необходимо указать родительский источник данных, который будет связывать компонент с родительской таблицей. В данном случае это будет StudentSrc. Далее в свойстве MasterFields этого же компонента необходимо указать поля, по которым будет установлена связь. В правой части свойства необходимо нажать на кнопку, и появится диалоговое окно, показанное на рис. 3.8.

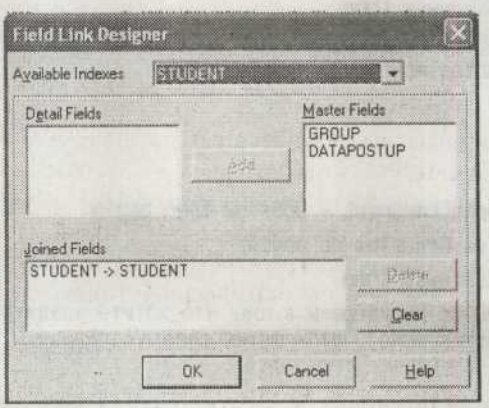


Рис. 3.8. Окно редактора связей

Оно содержит два списка — Master Fields и Detail Fields. В них перечислены поля, по которым можно создать связь. В обоих списках надо выбрать поля STUDENT, а потом нажать кнопку Add. Будет создана связь между таблицами. В списке

Available Indexes перечислены доступные индексы, по которым тоже можно создать связь.

Затем на основной форме приложения нужно разместить два компонента TDBGrid, располагающихся на вкладке Data Controls палитры компонентов. Модуль данных надо подключить к главному модулю приложения одной строкой кода:

```
uses DataModule;
```

Теперь необходимо связать компоненты TDBGrid с компонентами TDatasource, расположенными в модуле данных, при помощи одноименного свойства DataSource. Также надо разместить на форме компонент TPopupMenu. Его надо связать с компонентами TDBGrid через их свойство PopupMenu.

Двойным щелчком на компоненте TPopupMenu нужно активировать редактор меню и определить несколько новых пунктов. Потребуется создать пункт Новая, для которого в свойстве Name указать значение NewRec. Также нужно будет создать пункты Принять (PostRec) и Удалить (DeleteRec). Реализация этих методов приведена в листинге 3.1.

Листинг 3.1. Реализация методов контекстного меню

```
uses DataModule;
{$R *.dfm}

{Удаление выбранной записи}
procedure TForm1.DeleteRecClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataModule1.StudentsTbl do begin
      if State = dsBrowse then
        if MessageDlg('Вы уверены в том, что хотите удалить запись?',
mtConfirmation, [mbYes,mbNo], 0) = mrYes then
          Delete;
    end;
  end;
  if PopupMenu1.PopupComponent = DBGrid2 then begin
    with DataModule1.GradeTbl do begin
      if State = dsBrowse then
        if MessageDlg('Вы уверены в том, что хотите удалить запись?',
mtConfirmation, [mbYes,mbNo], 0) = mrYes then
          Delete;
    end;
  end;
end;

{Сохранение изменений}
```

```
procedure TForm1.PostRecClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataModule1.StudentsTbl do begin
      if State in [dsInsert, dsEdit] then
        Post;
    end;
  end;
  if PopupMenu1.PopupComponent = DBGrid2 then begin
    with DataModule1.GradeTbl do begin
      if State in [dsInsert, dsEdit] then
        Post;
    end;
  end;
end;

{Добавление новой записи}
procedure TForm1.NewRecClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataModule1.StudentsTbl do begin
      if State = dsBrowse then
        Insert;
    end;
  end;
  if PopupMenu1.PopupComponent = DBGrid2 then begin
    with DataModule1.GradeTbl do begin
      if State = dsBrowse then
        Insert;
    end;
  end;
end;
```

Двойным щелчком на каком-либо из компонентов TDBGrid нужно вызвать редактор полей. Командой Add All Fields контекстного меню можно добавить все поля. Лишние поля потом можно будет просто стереть. Данной командой определяются поля, которые будут отображаться в таблице. Вид окна представлен на рис. 3.9.

Для выбранного поля можно указать тип шрифта столбца, тип шрифта заголовка, цвет заголовка, название заголовка. Для изменения названия заго-

ловков достаточно выбрать поле и в свойстве Title ▸ Caption указать новый заголовок. В свойстве PickList можно указать список значений, которые будут доступны из таблицы в данном поле. Основное окно программы показано на рис. 3.10.

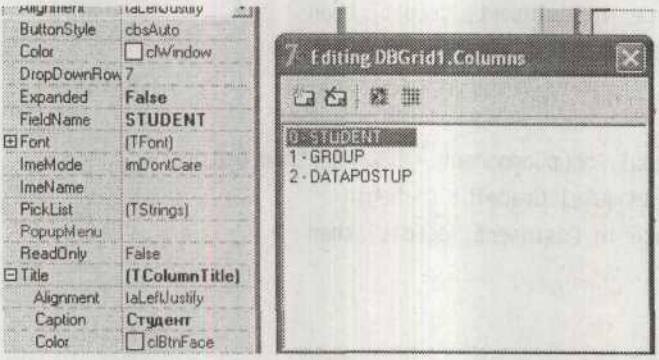


Рис. 3.9. Окно редактора полей

Рис. 3.10. Окно программы

Настройка BDE

Для доступа к параметрам драйвера в левой части окна Администратора нужно выбрать вкладку Configuration и требуемый драйвер. Система BDE взаимодействует с двумя типами драйверов. Это драйверы Native, являющиеся «родными» драйверами SQL Links, и драйверы ODBC.

При выборе нужного драйвера в правой части окна появляется окно с информацией о его свойствах:

- Свойство TYPE указывает тип сервера баз данных. Значение FILE указывает, что используется локальный сервер Interbase или таблицы Paradox,

Dbase. Значение SERVER свидетельствует об использовании удаленного SQL-сервера.

- Свойство LANGDRIVER определяет драйвер языка, используемого для кодировки символов.
- Свойство ENABLE BCD указывает на необходимость перевода чисел в двоично-десятичный формат.
- Свойство OPEN MODE определяет режим доступа к данным.
- Параметр MAX ROWS определяет количество записей, включаемых в пакет данных. Используется при работе с удаленными серверами.
- Свойство SERVER NAME указывает имя удаленной базы данных и содержит путь к ней.
- Параметр SQLPASSTHRU MODE регламентирует режим взаимодействия BDE с СУБД на уровне транзакций:
  - значение SHARED AUTOCOMMIT означает, что транзакции выполняются автоматически, без соответствующей команды от клиентского приложения;
  - значение SHARED NOAUTOCOMMIT указывает, что транзакция не выполняется автоматически. За их выполнением следит клиентское приложение;
  - значение NOT SHARED применяется, когда для каждого компонента, связанного с БД, создается свое соединение с базой данных.
- Параметр SQLQRYMODE определяет политику выполнения запросов:
  - значение LOCAL означает, что запрос выполняется локально;
  - значение NULL указывает, что будет применяться смешанный механизм выполнения запросов. Сначала BDE отправляет запрос на сервер; если результат не получен, то запрос повторяется в режиме *LOCAL;*
  - значение SERVER указывает, что запрос выполняется на сервере.
- Свойство DLL32 содержит имя динамической библиотеки драйвера.

Системные установки делятся на два типа. Первым типом можно назвать *установки по умолчанию*, а ко второму относятся разнообразные форматы данных. Установки по умолчанию доступны в разделе Configuration ▸ System ▸ Init:

- Свойство DEFAULT DRIVER содержит название драйвера, устанавливаемого по умолчанию для локальных баз данных.
- Свойство LANGDRIVER задает используемый по умолчанию драйвер языка.
- Параметр LOCAL SHARE определяет возможность разделения доступа к локальным базам данных между BDE и другими приложениями при одновременном обращении к ним.

Как уже отмечалось ранее, настройки BDE хранятся в файле idapi32.cfg.

Компонент TDatabase

Обычно при разработке приложений, использующих базы данных, с помощью утилит конфигурации BDE создаются псевдонимы, указывающие на тип и расположение данных. Компоненты `TTable`, `TQuery`, `TStoredProc` обладают свойством `DatabaseName`, при установке которого на этапе проектирования можно выбрать необходимый псевдоним из выпадающего списка или явно указать каталог, в котором располагаются таблицы локальных баз данных. Однако иногда бывает необходимо создать псевдоним динамически или переопределить какие-либо параметры настройки драйвера базы. В этом случае обычно используется компонент `TDatabase`, помещаемый явно на форму или в модуль данных. Если определить свойство `DatabaseName` этого компонента, оно появится в списке псевдонимов при установке одноименного свойства `DatabaseName` компонентов `TTable`, `TQuery` и `TStoredProc`. Если компонент не был размещен на форме, то он будет создан динамически, с настройками по умолчанию. Так как компонент отвечает за взаимодействие с BDE, его создание будет инициироваться компонентами `TTable`, `TQuery` и `TStoredProc`.

Переопределить параметры псевдонима базы данных можно с помощью инспектора объектов. В свойстве `TTransIsolation` можно определить уровень изоляции транзакции.

Также параметры псевдонима базы данных можно изменять в редакторе свойств компонента `TDatabase`. Окно редактора показано на рис. 3.11. Вызывается редактор двойным щелчком мыши на компоненте.

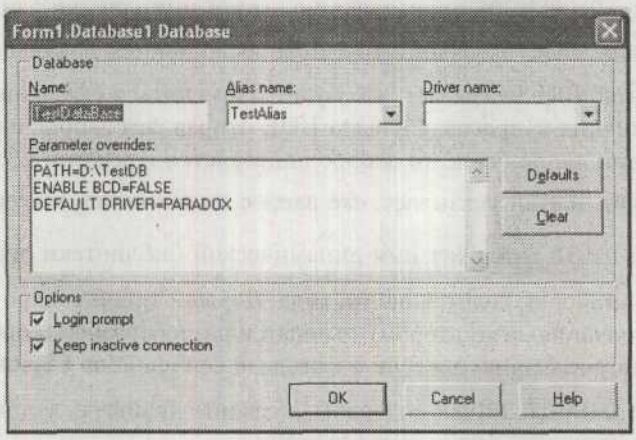


Рис. 3.11. Редактор свойств компонента TDatabase

Нажав кнопку `Defaults`, можно установить стандартные настройки данного псевдонима. Обратиться к параметрам псевдонима можно через свойство `Params`.

Также компонент `TDatabase` используется для сокращения числа обращений к базе данных. К примеру, если на форме расположено несколько компонен-

тов, взаимодействующих с базой данных, то каждый из них обращается к серверу и передает ему параметры, идентифицирующие пользователя. На сервере производится проверка и принимается решение о допуске пользователя к работе. Для того чтобы минимизировать число обращений к серверу, все компоненты, предназначенные для работы с какой-либо базой данных, связываются через свойство Database с компонентом TDatabase, а тот, в свою очередь, связывается с базой данных. Если приложение должно работать с несколькими базами данных, то в модуле данных размещается несколько компонентов TDatabase.

Повлиять на выполнение серверных транзакций можно путем кэширования внесенных пользователем изменений вместо попытки немедленного сохранения их в базе данных. В этом случае изменения, внесенные пользователем, сохраняются в кэш данных методом Post и отсылаются на сервер методом ApplyUpdates. Для включения механизма кэширования измененных данных свойству CachedUpdates компонентов TTable и TQuery присваивается значение True.

Кэширование изменений удобно по многим причинам. Во-первых, значительно снижается нагрузка на сеть, так как пересылаются данные не нескольких транзакций, а только одной. А во-вторых, если по каким-либо причинам транзакция не была завершена, то метод вернет значение False, но при этом несохраненные изменения останутся в кэше базы данных. Потом можно попытаться сохранить их еще раз или изменить их.

При выполнении метода ApplyUpdates автоматически запускается транзакция, которая завершается после внесения всех изменений на сервере. В случае успешного завершения транзакции кэш очищается.

Компонент TSession

Компонент TSession реализует текущий сеанс работы с базой данных и предназначен для управления всеми соединениями с базами данных. Компонент создается автоматически при запуске приложения. Явное использование компонента может быть продиктовано необходимостью создания многопоточных приложений баз данных. Компонент также может быть применен для управления настройками псевдонимов и драйверов.

Свойство SessionName определяет имя сеанса. Компоненты TTable, TQuery и TDatabase связываются с компонентом TSession через свойство Session, в котором указывается название сессии, определенной программистом. Свойства сессии, заданной компонентом TSession, действуют на все связанные с ним компоненты TDatabase.

Свойство Databases содержит массив связанных компонентов TDatabase. При помощи этого свойства можно получить доступ к активным базам данных, с которыми установлено соединение. А свойство DatabaseCount показывает количество связанных баз данных.

Методы GetDatabaseNames и GetAliasNames возвращают списки драйверов и псевдонимов баз данных. Метод GetAliasParams позволяет разработчику получить

параметры данного псевдонима. А метод GetTableNames возвращает имена таблиц, содержащихся в обрабатываемой базе данных.

Также компонент имеет методы, позволяющие удалять псевдонимы баз данных и их драйверы, добавлять их и совершать иные действия.

Компонент TStoredProc

Компонент TStoredProc используется для выполнения из приложений хранимых процедур, содержащихся на серверах баз данных. Хранимые процедуры могут возвращать наборы данных, отдельные записи или ничего не возвращать. Так как хранимая процедура скомпилирована заранее, то ее выполнение занимает меньше времени по сравнению с обычным запросом. Хранимые процедуры могут принимать входные параметры, передаваемые им из клиентского приложения.

Имя базы данных, с которой связывается хранимая процедура, содержится в свойстве DatabaseName. А имя хранимой процедуры содержится в свойстве StoredProcName. При разработке в инспекторе объектов значение свойства выбирается из списка хранимых процедур, получаемого с сервера при соединении с базой данных.

В свойстве Params содержится массив входных и выходных параметров данной хранимой процедуры. Метод ParamByName возвращает параметр процедуры, если методу передано наименование этого параметра. А метод ExecProc инициирует выполнение хранимой процедуры на сервере.

Для подготовки хранимой процедуры к выполнению следует вызывать метод Prepare. Если необходимо освободить ресурсы, занятые подготовленной к выполнению хранимой процедурой, следует использовать метод UnPrepare.

Стоит рассмотреть небольшой пример использования хранимых процедур. В качестве сервера будет использован InterBase, поставляющийся вместе с Delphi. На форму нужно поместить компоненты TDatabase, TStoredProc, TMemo и кнопку TBitBtn. При установке InterBase создает псевдоним для доступа к локальному серверу — IBLocal. Он указывает на демонстрационную базу данных EMPLOYEE.GDB. Этот псевдоним надо выбрать в списке AliasName компонента TDatabase. Свойству DatabaseName компонента TDatabase нужно присвоить значение MyTestDB. В свойстве Params потребуется ввести два параметра, как указано в табл. 3.3.

Таблица 3.3. Параметры, содержащие логин и пароль для доступа к серверу Interbase

Key	Value
USER NAME	SYSDBA
PASSWORD	masterkey

Указанные имя пользователя и пароль устанавливаются сервером по умолчанию и в дальнейшем могут быть изменены. Для свойства LoginPrompt нужно установить значение False. В результате сервер не будет заправшивать логин и пароль при каждом соединении. Затем необходимо установить соединение

с базой данных. Для этого достаточно свойству Connected присвоить значение True.

Пришло время связать компоненты TDatabase и TStoredProc. Для этого в свойстве DatabaseName компонента TStoredProc нужно выбрать из списка значение MyTestDB. Теперь в свойстве StoredProcName надо выбрать из списка хранимую процедуру ORG_CHART. В методе-обработчике кнопки останется ввести код, представленный в листинге 3.2.

Листинг 3.2. Использование компонента TStoredProc

```
procedure TForm1.ExecBtnClick(Sender: TObject);
var I : Integer;
begin
  with StoredProc1 do begin
    StoredProc1.ExecProc;
    Memo1.Lines.Clear;
    for I:= 0 to ParamCount - 1 do begin
      Memo1.Lines.Add(Params[I].AsString);
    end;
  end;
end;
```

Окно примера с результатами показано на рис. 3.12. В компонент TMemo загружаются значения параметров, в которые хранимая процедура поместила результаты своей работы.

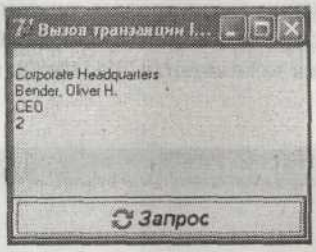


Рис. 3.12. Пример вызова хранимой процедуры

Компонент TUpdateSQL

Компонент TUpdateSQL может быть использован для модификации (добавления, изменения, удаления) данных на сервере с помощью операторов SQL. Компонент содержит методы-обработчики, в которых можно определить набор команд, выполняющихся при вызове методов Insert, Delete, Update компонентов TQuery и TTable.

Компонент UpdateSQL может быть связан с компонентами TQuery и TTable через их свойство UpdateObject, в котором указывается имя компонента. Если исполь-

зуется кэширование данных, то в процессе выполнения транзакции, инициированной методом ApplyUpdates при выполнении вставки, удаления или изменения записи, выполняется заранее определенная последовательность SQL-операторов. Если кэширование не используется, то выполняется немедленная модификация данных при вызове метода Post.

Компонент содержит старые значения полей, которые имело поле до внесения в него изменений, в полях с приставкой «OLD». Например:

```
UPDATE "Country.db"
(Name, Capital, Continent)
VALUES (:Name, :Capital, :Continent)
WHERE :OLD_Name = "Rangoon"
```

Параметр OLD_Name содержит старое значение поля, по которому запись будет обновлена.

Для управления каждым отдельным обновлением внутри транзакции, переносящей данные из кэша на сервер, можно использовать событие OnUpdateRecord соответствующего компонента с помощью параметров UpdateKind и UpdateAction.

Рассмотрим пример использования данного компонента. Для начала потребуется создать новый проект. В него нужно будет добавить модуль данных, и в модуле разместить компоненты TQuery, TUpdateSQL и TDataSource. Созданный модуль данных нужно сохранить с именем UpdateModule. Компонент TQuery должен получить имя ModifyQuery, а TDataSource — ModifySource. В примере будет использоваться созданная ранее база данных, содержащая информацию о студентах. В свойстве DatabaseName компонента TQuery нужно выбрать из списка псевдоним TestAlias. В свойстве SQL нужно ввести строку SELECT * FROM Students.DB.

Затем для компонента нужно активировать режим кэширования. Компонент TQuery потребуется связать с компонентом TUpdateSQL через свойство UpdateObject. Двойной щелчок на компоненте TUpdateSQL активирует редактор SQL-запросов (рис. 3.13).

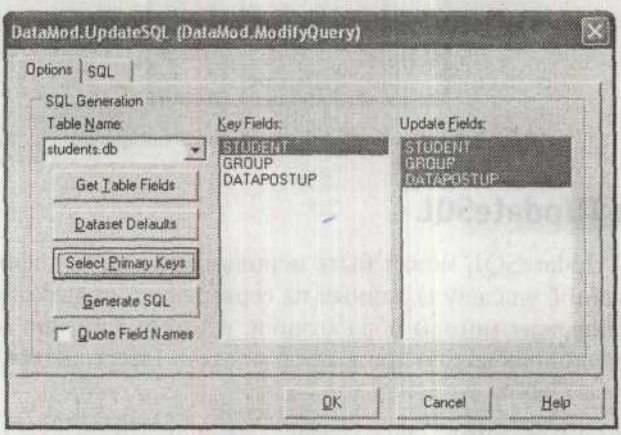


Рис. 3.13. Редактор запросов

Редактор имеет две вкладки — Options и SQL. На вкладке Options можно определить общие критерии и сгенерировать модифицирующий запрос. Кнопкой Select Primary Keys в поле Key Fields будут выбраны ключевые поля, по которым будут выбираться записи для внесения изменений. В списке Update Fields указаны поля, изменения которых будут внесены в набор данных или в которые будут добавлены данные. При нажатии на кнопку Dataset Defaults будут выбраны все поля в обоих списках. Кнопкой Generate SQL будут сгенерированы соответствующие SQL-операторы. После генерации SQL-операторов нужно связать компонент TDataSource с компонентом TQuery через свойство DataSet. Модуль данных надо подключить к форме. Для этого в секции Implementation надо добавить строку подключения

```
uses UpdateModule;
```

Теперь на форме надо разместить компонент TDBGrid и связать его с компонентом TDataSource. Потребуется еще настроить заголовки таблицы и разместить на форме три кнопки. В них будут вызываться методы Delete, Post и ApplyUpdates. В листинге 3.3 приведен код, содержащийся в методах-обработчиках.

Листинг 3.3. Код методов-обработчиков

```
procedure TForm1.DeleteBtnClick(Sender: TObject);
begin
  DataMod.ModifyQuery.Delete;
end;
procedure TForm1.PostBtnClick(Sender: TObject);
begin
  DataMod.ModifyQuery.Post;
end;
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  DataMod.ModifyQuery.ApplyUpdates;
end;
```

На рис. 3.14 показано окно демонстрационного приложения.

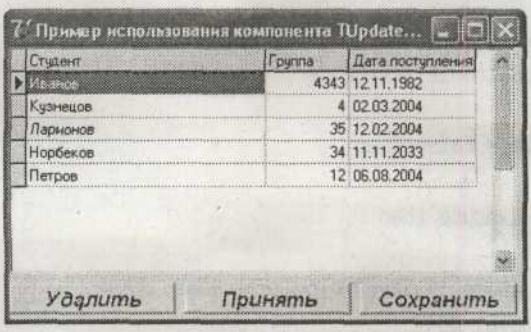


Рис. 3.14. Использование компонента TUpdateSQL

Компонент TBatchMove

Этот компонент обеспечивает копирование данных из одной таблицы в другую. Таблица-источник указывается в свойстве Source. Таблица, в которую копируются данные, указывается в свойстве Destination. Свойство Mode определяет тип перемещения данных:

- Значение batAppend указывает, что новые строки просто добавляются в результирующую таблицу.
- Значение batUpdate заставляет метод заменять строки таблицы-преемника соответствующими строками таблицы-источника. Соответствие строк определяется по индексному полю.
- Использование значения batAppendUpdate приводит к тому, что метод заменяет строки преемника соответствующими строками таблицы-источника. Соответствие строк определяется по индексному полю. Если соответствующие строки не обнаружены, то происходит их добавление в таблицу.
- Значение batCopy указывает, что создается таблица с той же структурой, что и исходная. Если таблица существует, то метод удаляет ее и создает новую.
- Значение batDelete заставляет метод удалять строки преемника, соответствующие строкам исходной таблицы. Соответствие строк определяется по индексному полю.

Свойство Mappings определяет соответствие полей исходной таблицы полям принимающей таблицы. Для таблиц формата Paradox можно использовать свойство KeyViolTableName. В этом свойстве хранится название таблицы, в которую помещаются записи, которые не удалось скопировать из-за ограничений ссылочной целостности или нарушений ключа. А в свойстве ProblemTableName хранится таблица Paradox, в которую помещаются записи, не скопированные по причине несоответствия типов полей или по иной причине.

Для начала копирования данных следует вызвать метод Execute. Свойства AbortOnKeyViol и AbortOnProblem в случае установки их значений в True прервут операцию копирования при возникновении ошибки. В листинге 3.4 приведен пример использования компонента.

Листинг 3.4. Пример использования компонента TBatchMove

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Query1.Active = False then
    Exit;
  if SaveDialog1.Execute then
    begin
      Table1.TableName := SaveDialog1.FileName;
      with BatchMove1 do
        begin
```

```
        Source := Query1;
        Destination := Table1;
        Mode := batCopy;
        Execute;
        ShowMessage(IntToStr(MovedCount) + ' records copied');
      end;
    end;
end;
```

Пример связи с Excel через BDE

В данном разделе будет рассмотрен интересный пример — связь с таблицей Excel, получение из нее данных и изменение их.

Для начала необходимо подготовить файл Excel и настроить доступ к нему через драйвер ODBC. В табл. 3.4. приведено содержимое электронной таблицы в формате Excel, которая будет использоваться в рассматриваемом примере.

Таблица 3.4. Таблица «Метеоусловия»

Дата	Погода	Температура
12.11.1995	Солнечная	–25
12.07.1995	Облачно	22
12.05.1995	Дождь	12

Затем нужно выделить диапазон ячеек (рис. 3.15) и задать ему имя, например TestTable.

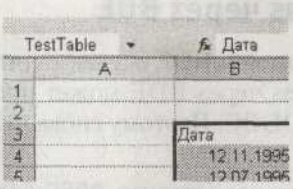


Рис. 3.15. Именованный диапазон ячеек

Созданный файл надо сохранить с именем Test.xls. Далее необходимо разрешить многопользовательский доступ к файлу. Для этого достаточно выполнить команду меню Сервис ▸ Доступ к книге. В открывшемся диалоговом окне нужно взвести флажок Разрешить совместный доступ, а затем снова сохранить файл.

На этой стадии уже можно создать ODBC-соединение с рабочей книгой. Для этого следует запустить утилиту ODBC Administrator. В системе Windows XP она называется Data Sources и находится в разделе Administrative Tools. В основном окне утилиты нужно нажать кнопку Add и в появившемся диалоговом окне выбрать значение Driver do Microsoft Excel. После этого — нажать кнопку Finish. В появившемся окне потребуется дать имя TestExcelDS источнику данных в поле Data Source Name. Нажмите кнопку Select WorkBook и в диалоговом окне выбо-

ра укажите ранее созданный файл. А затем нужно при помощи кнопки Options открыть диалоговое окно настроек и снять флажок Read Only. Созданное соединение появится в списке источников данных ODBC и автоматически появится в списке доступных псевдонимов баз данных BDE.

Теперь можно перейти к разработке соответствующего приложения. На форме следует разместить компоненты TTable, TDataSource, TDBGrid и две кнопки. В свойстве DatabaseName компонента TTable надо выбрать из списка псевдоним TestExcelDS. В свойстве TableName указать значение TestTable, которое определяет заданный ранее диапазон ячеек.

Необходимо связать компоненты TTable и TDataSource, а потом TDataSource и TDBGrid. В кнопках, расположенных на форме, потребуется описать вызовы методов Post и ApplyUpdates из предыдущего примера.

После этого можно скомпилировать приложение, закрыть Delphi и запустить скомпилированный модуль автономно. На рис. 3.16 показано окно программы. Она позволяет изменять данные и вводить новые. При вводе новых данных границы именованной области ячеек будут увеличиваться.

Рис. 3.16. Пример работы с Excel через BDE

Пример связи с Access через BDE

Точно так же можно при помощи BDE осуществлять доступ к таблицам в формате Access. Работа с Access через BDE может производиться только за счет использования драйверов ODBC. Поэтому снова нужно связать драйвер ODBC с источником данных.

Как и прежде, нужно запустить утилиту ODBC Administrator. В список нужно добавить драйвер Driver do Microsoft Access. Созданную связь с источником данных надо назвать TestAccessDS. Затем нужно нажать кнопку Select и в диалоговом окне выбора базы данных указать имя файла dbdemos.mdb. Эта тестовая база данных поставляется с BDE и обычно расположена в каталоге X:\Program Files\Common Files\Borland Shared\Data. После нажатия кнопки OK в BDE автоматически появится псевдоним TestAccessDS, настроенный на соединение с базой данных.

Теперь нужно создать новое приложение и добавить в него модуль данных. Модуль данных надо сохранить с именем AccessDM. На форме потребуется разместить компоненты TDataSource, TDatabase и TTable. Компонент TDatabase нужно связать с базой данных при помощи псевдонима TestAccessDS. Свойство LoginPrompt должен получить значение False, а свойство Connected — значение True.

Теперь свойству Database нужно присвоить значение AccessDB. Для компонента TTable — задать имя EmployeeTbl. В качестве источника данных для нее можно использовать таблицу employee. Свойство CachedUpdates должно получить значение True.

На этой стадии можно активировать набор данных. Компонент TDataSource должен получить имя EmployeeDS, а затем потребуется связать его с таблицей EmployeeTbl. Модуль данных нужно подключить к главной форме командой uses AccessDM. На форме нужно также разместить компонент TDBGrid и три кнопки. Компонент TDBGrid потребуется связать с TDataSource. А в обработчиках нажатия кнопок нужно указать код, приведенный в листинге 3.5.

Листинг 3.5. Код методов-обработчиков

```
procedure TForm1.PostBtnClick(Sender: TObject);
begin
  with AccessDataModule.EmployeeTbl do begin
    if state In [dsInsert, dsEdit] then
      Post;
  end;
end;
procedure TForm1.DeleteBtnClick(Sender: TObject);
begin
  with AccessDataModule.EmployeeTbl do begin
    if state = dsbrowse then
      Delete;
  end;
end;
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  AccessDataModule.EmployeeTbl.ApplyUpdates;
end;
```

Форма приложения показана на рис. 3.17.

Рис. 3.17. Работа с Access через BDE

Так как таблица связана с дочерней таблицей, то в силу ограничений ссылочной целостности данные из таблицы нельзя удалить. Можно заносить и удалять свои данные. Приложение можно расширить, добавив в модуль данных остальные таблицы базы и определив для них отношения ссылочной целостности.

Пример связи с InterBase через BDE

Как уже говорилось ранее, Borland поставляет с BDE набор драйверов SQL Links с настроенным псевдонимом IBLocal для доступа к базе данных EMPLOYEE.GDB. В этом разделе будет создано простое приложение, которое будет обращаться к базе, читать из нее информацию и вносить изменения. На форме потребуется разместить компоненты TDatabase, TQuery, TUpdateSQL, TDataSource, TDBGrid, два компонента TDBEdit и три кнопки.

В свойстве AliasName компонента TDatabase надо выбрать псевдоним IBLocal. В свойстве DatabaseName — установить имя IbaseDemo. Свойство LoginPrompt должно получить значение False, а для свойства Params следует использовать значения из табл. 3.3. Компонент нужно связать с базой данных, присвоив значение True свойству Connected. Также необходимо установить связь между компонентами TQuery и TDatabase. В свойстве SQL компонента TQuery потребуется указать текст SQL-запроса, который получает все строки таблицы:

```
SELECT * FROM Country
```

Теперь необходимо настроить компонент TUpdateSQL и связать его с компонентом TQuery. Затем связать TQuery с таблицей TDBGrid. Компоненты TDBEdit должны быть связаны с источником данных через свойство DataSource. А в свойствах DataField для каждого поля ввода надо указать соответствующие поля таблицы. Далее в обработчиках кнопок останется прописать код для методов Insert, Delete и ApplyUpdates. Вид основного окна приложения показан на рис. 3.18.

Рис. 3.18. Работа с Interbase через BDE

Для демонстрации была взята довольно маленькая таблица. Пример можно расширить, если добавить в него остальные таблицы базы и настроить между ними отношения ссылочной целостности.

В конце этого раздела следует отметить, что Borland перестала развивать технологию BDE с 2002 года. Разрабатывать проекты с этой технологией все еще можно, но уже не рекомендуется. В данном разделе примеры работы с этой технологией были приведены, чтобы продемонстрировать легкость работы с базами данных из Delphi.

Стандарт ODBC

ODBC — это стандарт, описывающий систему интерфейсов, с помощью которых прикладные программы могут обращаться к базам данных и обрабатывать их независимым от СУБД способом. ODBC предоставляет интерфейсы для доступа к реляционным базам данных и базам данных с табличной организацией. Широкое распространение стандарт получил благодаря поддержке Майкрософт.

Архитектура ODBC

На рис. 3.19 схематично изображена архитектура ODBC.

Рис. 3.19. Архитектура ODBC

Как видно из рисунка, приложение обращается к диспетчеру драйверов, а диспетчер, в свою очередь, обращается к источнику данных и производит с ними какие-либо действия. Источник данных — это база данных, приложение, операционная система или даже некая аппаратная платформа. Приложение инициирует запросы на установление соединения с источником данных, на выполнение каких-либо действий с базой данных. Стандарт ODBC предоставляет разработчику набор интерфейсов для выполнения каждого из этих запросов и регламентирует стандартные коды ошибок, которые будут возвращены приложению в случае неудачного выполнения запроса.

Диспетчер драйверов (driver manager) служит связующим звеном между приложениями и драйверами СУБД. Когда приложение инициирует соединение с базой данных, диспетчер определяет тип СУБД и загружает соответствующий драйвер в память (или использует ранее загруженный). Диспетчер драйверов обрабатывает запросы на инициализацию соединения и контролирует формат запросов и порядок их поступления от приложения. Диспетчер драйвера является частью Windows.

Драйвер (driver) обрабатывает запросы, поступающие от приложения, преобразует их в набор команд API СУБД и, таким образом, производит какие-либо действия с базой данных. Драйвер отвечает за то, чтобы стандартные команды ODBC выполнялись корректно. В некоторых случаях источник данных не поддерживает некоторые функции, таким образом, их приходится выполнять драйверу. Если источник имеет полную поддержку SQL, то драйвер всего лишь передает запрос на обработку и получает результаты. На драйвере также лежит функция приведения кодов ошибок, поступающих от источника, к стандартным, определенным в ODBC.

Определяют два типа драйверов — одноуровневые и многоуровневые. Одноуровневые обрабатывают вызовы ODBC и операторы SQL. Многоуровневые обрабатывают только вызовы ODBC, оставляя СУБД осуществлять обработку SQL-запросов.

Уровни соответствия

Уровень соответствия ODBC (ODBC conformance level) описывает то, какие возможности и функции доступны через API (интерфейс прикладных программ) драйвера. API драйвера содержит набор функций, которые может вызывать приложение для обращения к интерфейсам источника данных. Различают несколько уровней соответствия ODBC, обеспечивающих разные наборы возможностей:

- Базовый уровень (Core API):
  - соединение с источником данных;
  - подготовка и выполнение SQL-запросов;
  - получение результирующего набора данных;
  - фиксация и откат транзакций;
  - получение информации об ошибках.
- Первый уровень (Level 1 API):
  - соответствие ODBC на базовом уровне;
  - получение информации о параметрах, возможностях и функциях драйвера;
  - соединение с источниками данных, содержащих информацию, специфичную для драйвера;
  - получение информации из каталога.

- Второй уровень (Level 2 API):
  - соответствие ODBC на первом и базовом уровнях;
  - обзор возможных соединений и источников данных;
  - использование диалекта SQL данной СУБД;
  - вызов библиотеки преобразований;
  - обработка двунаправленных курсоров.

Если какой-либо драйвер не поддерживает требуемый приложению уровень, то программист может реализовать нужные функции самостоятельно, обрабатывая данные, получаемые из источника соответствующим образом.

Уровни соответствия SQL определяют, какие SQL-операторы, выражения и типы данных доступны драйверу на данном уровне. Стандартом определены три уровня соответствия SQL:

- Минимальный синтаксис (Minimum SQL grammar):
  - CREATE TABLE, DROP TABLE;
  - оператор SELECT (без вложенных подзапросов);
  - INSERT, UPDATE, DELETE;
  - простые выражения (сравнения, арифметические операции);
  - типы данных CHAR, VARCHAR, LONGCHAR.
- Базовый синтаксис (Core SQL grammar):
  - минимальный синтаксис;
  - ALTER TABLE, TREATE INDEX, DROP INDEX;
  - CREATE VIEW, DROP VIEW;
  - GRANT, REVOKE;
  - полный синтаксис оператора SELECT;
  - встроенные функции SUM, COUNT, MAX, MIN, AVG.
- Расширенный синтаксис (Extended SQL grammar):
  - базовый синтаксис;
  - UPDATE и DELETE с использованием позиции курсора;
  - скалярные функции SUBSTRING и ABS;
  - переменные даты, времени и временная метка;
  - пакетная обработка операторов SQL;
  - хранимые процедуры.

Приложение может вызвать драйвер и определить, какой уровень соответствия SQL он поддерживает.

Драйверы ODBC могут поддерживать многопоточность (multithreaded driver), то есть с одним драйвером могут одновременно работать несколько приложе-

ний в синхронном режиме, внося какие-либо изменения в источник данных. В случае, если драйвер не является многопоточным, он работает только в асинхронном режиме.

Определение имен источников данных

Для того чтобы сделать источник данных доступным для приложения, необходимо указать соответствующие *имена источников данных DSN* (Data Source Name). Существует три типа пользовательских источников данных:

- User DSN (пользовательский). Драйвер этого типа виден пользователю, определившему его. Соответственно, приложение будет работать только в том случае, если в системе зарегистрирован (в данный момент) именно тот пользователь, который настроил данный источник;
- System DSN (системный). Драйвер этого типа виден всем приложениям, работающим на данной машине;
- File DSN (файловый). Драйвер подобного типа может быть сделан общим ресурсом (совместное пользование) для всех пользователей, у которых установлен такой же драйвер и у которых есть соответствующие права доступа.

Для того чтобы создать источник данных, сначала нужно запустить утилиту ODBC Data Source Administrator при помощи команды меню Administrative Tools ▸ Data Sources. Окно Администратора ODBC показано на рис. 3.20.

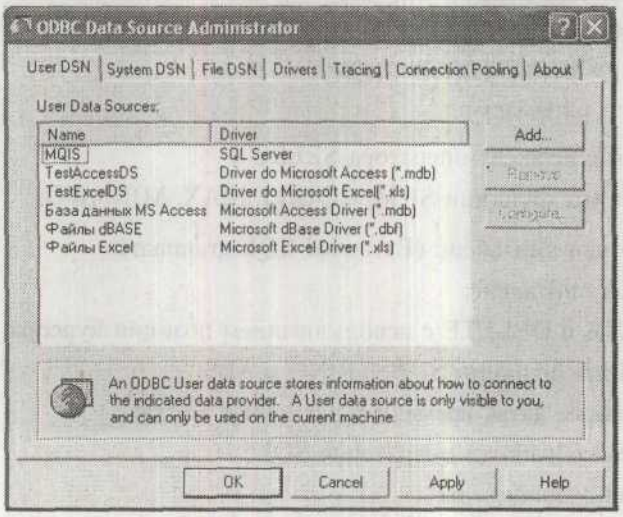


Рис. 3.20. Окно ODBC Data Source Administrator

Для примера можно создать соединение с файлом в формате Access. Нужно нажать кнопку Add, в результате чего будет активировано диалоговое окно,

показанное на рис. 3.21. В нем надо выбрать драйвер Driver do Microsoft Access (*mdb).

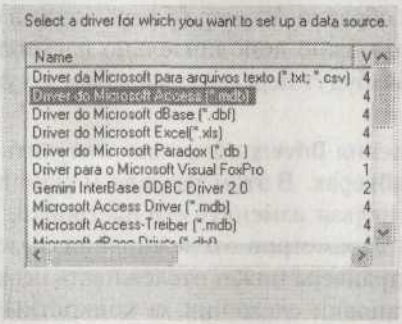


Рис. 3.21. Окно создания нового источника данных

Выбор подтверждается нажатием кнопки Finish. Будет активировано диалоговое окно, в котором нужно задать имя источника, ввести путь к базе и настроить некоторые параметры, как показано на рис. 3.22.

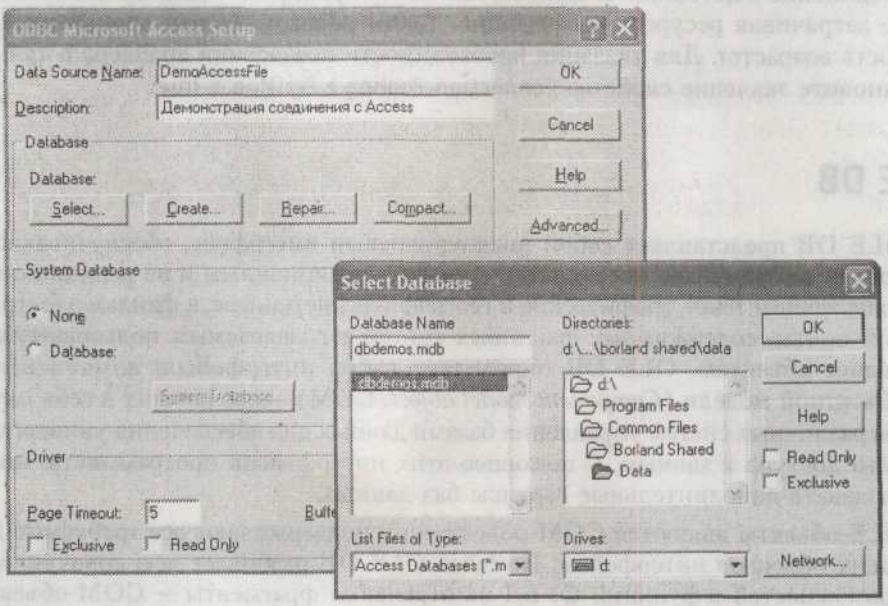


Рис. 3.22. Настройка источника данных

Теперь необходимо указать используемую базу данных. Для этого нужно нажать кнопку Select. Будет отображено окно Select Database, в котором указывается путь к базе данных. При помощи кнопки Create можно создать собственную базу. Кнопка Repair позволяет восстановить поврежденный файл. Для того чтобы очистить файл от «мусора», используется кнопка Compact.

В поле Page Timeout задается интервал времени, в течение которого неиспользуемая страница данных будет содержаться в буфере. В свойстве Buffer Size указывается размер буфера, используя который, драйвер передает данные от приложения на диск и обратно. Размер буфера должен быть кратен 256. В окне Set Advanced Options можно дополнительно настроить соединение, задать логин и пароль, которые будут передаваться в базу по умолчанию, и иные специфичные настройки.

Группа органов управления Drivers позволяет получить информацию об установленных ODBC-драйверах. В этом разделе можно получить информацию о версии драйвера, но нельзя изменить его настройки. Раздел Tracing используется для настройки параметров отслеживания вызовов функций ODBC-драйверов. Менеджер драйвера может отслеживать несколько соединений или только одно. Для установки слежения за конкретным драйвером его надо выбрать в диалоговом окне Select DLL.

Вкладка Connection Pooling используется для того, чтобы указать, может или нет драйвер ODBC повторно создавать метки соединения для сервера базы данных. Когда соединение с источником данных освободится, менеджер драйверов поместит его в пул на определенный период. Когда поступит запрос на соединение с источником данных, менеджер драйверов назначит его из пула, не затрачивая ресурсов на создание. Таким образом, общая производительность возрастет. Для указания необходимости помещения драйвера в пул установите значение свойства Connection Pooling ▸ PerfMon в True.

OLE DB

OLE DB представляет собой низкоуровневый интерфейс, обеспечивающий доступ к различным источникам данных — реляционным и не реляционным, содержащим текст, графические и географические данные, к файлам электронной почты, содержимому файловых систем и создаваемым пользователями бизнес-объектам. OLE DB определяет набор интерфейсов компонентной объектной модели (*Component model object*, COM), включающих в себя службы различных систем управления базами данных для обеспечения универсального доступа к данным. С помощью этих интерфейсов программисты могут создавать дополнительные сервисы баз данных.

OLE-объекты являются COM-объектами и поддерживают все требуемые для таких объектов интерфейсы. По сути, OLE DB разбивает всю совокупность возможностей и функций СУБД на отдельные фрагменты — COM-объекты, выполняющие определенные функции. Некоторые объекты отвечают за выполнение запросов, другие — за обновление данных и т. д. Это свойство OLE DB позволяет преодолеть огромный недостаток ODBC. Чтобы драйвер ODBC считался законченным, производитель должен обеспечить в нем вызов всех интерфейсов, предоставляемых СУБД. В случае OLE DB производитель может выпустить драйвер с частично реализованной функциональностью, а позже добавлять в него новые интерфейсы.

Основные конструкции OLE DB

На верхнем уровне абстракции можно выделить три главных компонента:

- потребители;
- провайдеры данных;
- провайдеры сервисов.

Любое приложение, использующее интерфейсы OLE DB, является *потребителем*. В роли потребителя может выступать прикладная программа базы данных, средство разработки, средство создания отчетов или же объектная модель *ActiveX Data Object* (ADO).

Провайдер данных (Data provider) представляет собой объект, «владеющий» данными, то есть связаный с ними. Он находится между потребителем и массивом данных. В OLE DB все провайдеры представляют данные в виде виртуальных таблиц. Провайдер выполняет несколько задач:

- Принятие запросов на доступ к данным, поступающих от потребителя.
- Выполнение выборки и обновления данных.
- Передача результатов потребителю (данные или коды ошибок).

Провайдер сервисов (Service provider) реализует расширенные возможности, которые не поддерживаются обычными провайдерами данных, и сам не «владеет» данными. Этот провайдер, к примеру, обеспечивает сортировку, фильтрацию, управление транзакциями, обработку SQL-запросов и т. д. Провайдер сервисов может работать с массивами данных напрямую либо через провайдер данных. На рис. 3.23 представлена схема, отражающая суть сказанного.

Рис. 3.23. Компоненты OLE DB

Как видно из рисунка, потребитель может получать данные как непосредственно от провайдера данных, так и используя службы, которые предоставляет провайдер сервисов.

Ядро объектной модели OLE DB составляют четыре объекта:

- DataSource;
- Session;

- Command;
- Rowset.

Схема модели представлена на рис. 3.24.

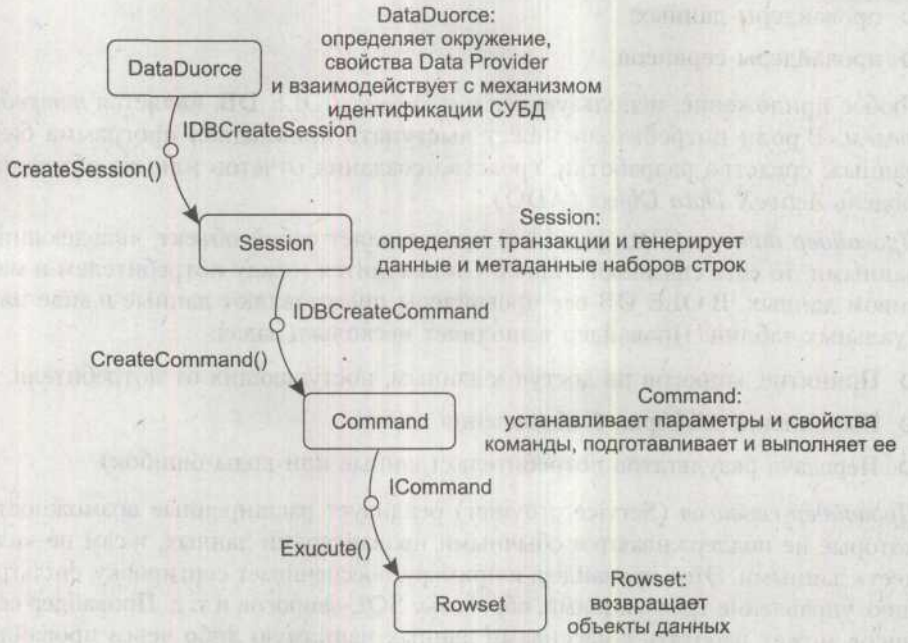


Рис. 3.24. Ядро объектной модели OLE DB

Как видно из рисунка, объект DataSource, вызвав интерфейс IDBCreateSession, создает новую сессию. В свою очередь объект Session вызывает интерфейс IDBCreateCommand и создает объект Command, содержащий определенную команду. Далее объект Command вызывает интерфейс ICommand и создает объект Rowset, который содержит полученные результаты.

Объект DataSource используется потребителем данных OLE DB для связи с провайдером данных. Этот объект может быть создан различными способами: порожден от объекта Enumerator и связан с моникером[1] файла или с моникером источника данных. Также объект может быть создан с помощью вызова COM-функции CoCreateInstance. Каждый провайдер OLE DB присваивает свой собственный идентификатор класса объекту, являющемуся источником данных. Объект DataSource содержит информацию о параметрах соединения, включая имя пользователя и пароль. Основная задача объекта — предоставить информацию из массива данных. Он поддерживает несколько интерфейсов, необходимых для взаимодействия с SQL-сервером:

[1] Моникер является объектом операционной системы Windows и представляет собой именованную область памяти, которую могут разделять несколько процессов.

- Интерфейс IDBInitialize инициализирует и устанавливает среды данных и безопасности (аутентификация и другие действия).
- Интерфейс IDBCreateSession создает объект Session.
- Интерфейс IDBProperties позволяет получать информацию о возможностях провайдера и производить инициализацию необходимых свойств.

Объект Session содержит данные транзакций и генерирует наборы строк из источника данных, а также команды для запросов к источнику и манипулирования им. Если провайдер поддерживает команды, объект Session выступает в роли фабрики классов объекта Command. Для создания сессии из объекта DataSource следует вызвать интерфейс IDBCreateSession::CreateSession. С одним объектом DataSource может быть ассоциировано сразу несколько сессий.

Объекта Session тоже поддерживает несколько интерфейсов:

- Интерфейс IOpenRowset позволяет открывать набор строк из таблицы, индекса или диаграммы.
- Интерфейс IGetDataSource возвращает объект DataSource из объекта Session.
- Интерфейс IDBCreateCommand создает объект Command.

Объект Command служит для обработки команд, которые представляют собой строки, передаваемые от потребителя к провайдеру для исполнения. В большинстве случаев такая команда включает в себя SQL-оператор SELECT, но может содержать и другие операторы. Один сеанс (объект Session) может содержать несколько связанных с ним команд. Результатом выполнения команды является новый объект Rowset.

Объект Command поддерживает свой набор интерфейсов:

- Интерфейс ICommand используется для выполнения запросов или команд.
- Интерфейс ICommandText используется для определения текста запроса.
- Интерфейс ICommandProperties определяет требуемые свойства набора строк, возвращаемого по команде.
- Интерфейс ICommandWithParameters позволяет выполнять запросы с параметрами.

Объект Rowset позволяет провайдерам данных представлять данные из источников в табличном формате. Этот объект представляется как некоторое множество строк, в каждой из которых содержится один или несколько столбцов. Такой объект может быть создан в результате вызова интерфейса IOpenRowset::OpenRowset или сгенерирован провайдером данных при наступлении определенного события. Этот объект может применяться для обновления, вставки и удаления строк, если провайдер имеет соответствующие интерфейсы.

Наиболее часто используемые интерфейсы объекта Rowset перечислены ниже:

- Интерфейс IRowset позволяет объекту сканировать ячейки.
- Интерфейс IAccessor присваивает столбцы наборам строк.

- Интерфейс IColumnsInfo позволяет получать информацию о столбцах набора строк.
- Интерфейс IRowsetInfo отвечает за получение информации о свойствах набора строк.
- Интерфейс IrowsetIndex требуется для индексированных наборов строк. Используется функциями, работающими с индексами.
- Интерфейс IConvertType производит проверку на возможность преобразования столбца набора строк одного типа к другому.

В состав OLE DB входят и другие объекты, помимо рассмотренных выше. Объект Enumerator используется для перечисления доступных объектов источников данных (провайдеров OLE DB), а также указания других перечислителей, имеющихся в системе. В большинстве случаев информация, возвращаемая объектом, извлекается из реестра. Объект Enumerator предоставляет интерфейс ISourceRowset и возвращает объект Rowset с описанием всех источников данных и перечислителей, которые он включает в себя. Получить эту информацию можно с помощью метода GetSourcesRowset интерфейса ISourceRowset. Существует корневой объект Enumerator, возвращающий источники данных верхнего уровня, и другие перечислители, которые могут использоваться для извлечения информации, к которой имеет доступ конкретный провайдер.

Объект Transaction поддерживает транзакции с источниками данных. Различают локальные и распределенные транзакции. *Локальными транзакциями* называются такие транзакции, которые выполняются с участием локального провайдера данных. Провайдер данных, который поддерживает локальные транзакции, предоставляет интерфейс ITransactionLocal посредством объекта Session.

Свойства транзакций провайдера можно выяснить через интерфейс IDBProperties. Для выполнения распределенных транзакций, то есть транзакций, использующих несколько распределенных провайдеров данных, потребители используют интерфейс ITransactionJoin. Этот интерфейс может применяться лишь в том случае, если провайдер поддерживает распределенные транзакции. Приложение вызывает метод JoinTransaction для включения в сеанс распределенной транзакции. После включения в распределенную транзакцию используется интерфейс ITransaction для завершения или отмены транзакции.

Помимо кодов возврата и информации о состоянии, указывающей на успешный или же имеющий неблагоприятный исход вызов каждого метода OLE DB, провайдеры могут предоставлять расширенную информацию об ошибке с помощью объекта Error. Разработчику следует использовать ISupportErrorInfo для того, чтобы установить, может ли данный объект возвратить объекты ISupportErrorInfo и получить интерфейсы, которые возвращают эти объекты.

Стандартные провайдеры OLE DB

Некоторые производители СУБД поставляют со своими продуктами драйверы OLE DB для доступа к базам данных. Майкрософт поставляет стандартные

провайдеры OLE DB в пакете Microsoft Data Access Components (MDAC). На текущий момент доступна версия 2.8. Основные провайдеры перечислены ниже:

- Провайдер Microsoft OLE DB Provider for ODBC предоставляет возможность получить доступ к источнику данных ODBC. Но использовать данное решение не рекомендуется, так как драйверы ODBC сами по себе не отличаются быстродействием, а дополнительный уровень скорости работы не прибавит.
- Провайдер Microsoft OLE DB Provider for Microsoft Indexing Service позволяет получить доступ (в режиме только для чтения) к файловым системам и интернет-ресурсам, проиндексированным с помощью Microsoft Indexing Service.
- Провайдер OLE DB Provider for Microsoft Directory Services позволяет получить доступ к ресурсам службы каталогов (Active Directory Services).
- Провайдер Microsoft Jet4.0 OLE DB Provider используется для работы с базами данных MS Access.
- Провайдер Microsoft OLE DB Simple Provider предназначен для соединения с источниками данных, поддерживающими только базисные возможности технологии OLE DB.
- Провайдер Microsoft OLE DB Provider for Internet Publishing предназначен для обеспечения доступа к веб-серверам и получения информации от ресурсов, построенных на базе Microsoft FrontPage и Microsoft Internet Information Server.
- Провайдер Microsoft OLE DB Provider Oracle обеспечивает соединение с сервером Oracle.
- Провайдер Microsoft OLE DB Provider SQL Server обеспечивает соединение с сервером Microsoft SQL Server.
- Провайдер Microsoft OLE DB Provider for OLAP Services используется для обеспечения доступа к службам OLAP сервера MS SQL.

ADO

Технология *Microsoft ActiveX Data Objects (ADO)* представляет собой высокоуровневую объектную надстройку над OLE DB. Несмотря на то что OLE DB предоставляет полный набор интерфейсов для манипулирования данными, большинство разработчиков не нуждается в низкоуровневом контроле за процессом соединения с данными и управления ими, который предоставляет OLE DB. В то же время разработчики часто используют высокоуровневые языки, которые не поддерживают указатели на функции и другие механизмы C++. ADO может использоваться для работы с любыми провайдерами OLE DB. Схема приведена на рис. 3.25.

Рис. 3.25. Доступ к данным при помощи ADO

В качестве источников данных могут выступать различные хранилища информации, например таблицы, файлы и базы данных.

Основы ADO

Объектная модель ADO состоит из семи объектов, иерархия которых показана на рис. 3.26.

Объект Connection инкапсулирует в себе объекты OLE DB DataSource и Session. Он содержит единственную сессию с источником данных. Объект Connection определяет свойства соединения, определяет возможности локальных транзакций, предоставляет централизованный объект для получения информации об ошибках Error и указатели для использования схем запросов.

Объект Command инкапсулирует одноименный объект OLE DB Command. Объект используется для выполнения команд определения и манипуляции данными. Если в качестве источника данных выступает реляционная СУБД, объект может выполнить некоторые SQL-операторы. Объект Command позволяет определить параметры и установить порядок выполнения запросов. Коллекция объектов Parameter предоставляет доступ к параметрам.

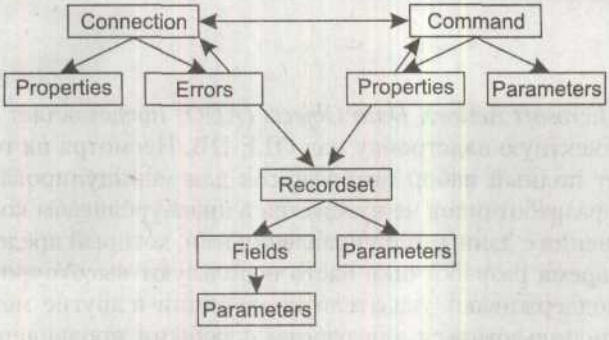


Рис. 3.26. Объектная модель ADO

Объект Recordset инкапсулирует функциональность объекта OLE DB Rowset. Объект Recordset является текущим интерфейсом доступа к данным, который может быть получен в результате обработки запроса или каким-либо другим способом. Объект позволяет контролировать используемый механизм блокировок, тип используемого курсора, число строк, возвращаемых в одном пакете, и т. д. Объект Recordset предоставляет доступ к коллекции объектов Field, которые содержат метаданные о свойствах столбцов набора данных, таких как имя, тип, длина и точность. Также содержатся текущие значения записей. Объект Recordset также используется для перемещения по набору данных и их модификации.

Каждый высокоуровневый объект ADO содержит коллекцию объектов Property. Объект Property позволяет ADO динамически публиковать возможности любого провайдера данных. Так как не все провайдеры поддерживают некоторые функции, очень важной особенностью объектной модели ADO является возможность предоставления динамического доступа к специфичным функциям.

Компоненты библиотеки VCL, предназначенные для работы с ADO, строятся на базе рассмотренных объектов.

Компонент TADOConnection

Компонент TADOConnection инкапсулирует объект ADO Connection. Данный компонент предназначен для соединения с хранилищами данных. С одним компонентом TADOConnection может быть связанно несколько компонентов TADOTable и TADOQuery.

Соединение с хранилищем данных открывается и закрывается при помощи свойства Connected или метода Open. Методу Open можно передать параметры UserID и Password, в которых хранятся логин и пароль. Закрыть соединение можно, вызвав метод Close.

Свойство ConnectOptions определяет тип соединения. Можно создавать синхронное и асинхронное соединения. По умолчанию соединение определяется как синхронное. В том случае, когда сервер работает довольно медленно, выбирается асинхронное соединение.

Свойство CursorLocation определяет порядок функционирования курсоров. Если для свойства задать значение clUseServer, то обработка строк будет производиться на сервере. Клиентское приложение будет получать лишь готовые результаты запросов. Если использовать значение clUseClient, то приложению пересылается весь набор данных, и курсор будет обрабатываться на клиенте. Однако использовать клиентский курсор для больших наборов данных невыгодно. Серверный курсор является чуть более медленным по сравнению с клиентским, но снимает обязанности по обработке данных с клиента и значительно снижает нагрузку на сеть.

Свойство IsolationLevel определяет уровень изоляции транзакции. Этот уровень определяет, как транзакции взаимодействуют с другими соединениями,

одновременно обращающимися к таблице, и определяет область видимости транзакции.

В списке, приведенном ниже, описаны основные типы изоляции транзакций и дополнительные эффекты, возникающие при их использовании:

- Значение `ilUnspecified` указывает, что сервер использует уровень изоляции транзакции, отличный от того, который был запрошен. Также это значение используется, если уровень изоляции транзакции не был определен заранее.
- Значение `ilChaos` говорит, что изменения, внесенные транзакциями, имеющими более высокий уровень, не могут быть перезаписаны в данной сессии.
- Значение `ilReadUncommitted` указывает, что несохраненные изменения, внесенные другими транзакциями, являются видимыми.
- Значение `ilBrowse` показывает, что несохраненные изменения, внесенные другими транзакциями, являются видимыми.
- Значение `ilCursorStability` указывает, что изменения, внесенные другими транзакциями, будут видны только после их сохранения в базе данных.
- Значение `ilReadCommitted`, как и в предыдущем случае, свидетельствует, что изменения, внесенные другими транзакциями, будут видны только после их сохранения в базе данных.
- Значение `ilRepeatableRead` показывает, что изменения, произведенные другими транзакциями, изначально не будут видны, но повторный запрос (обновление) набора данных может возвратить новые записи.
- Значение `ilSerializable` заставляет сервер изолировать транзакции друг от друга.

Свойство `Provider` содержит имя провайдера, который в данный момент используется объектом `Connection`. Права доступа, приписанные соединению, определяются при помощи свойства `Mode`. Значение свойства `Mode` указывает на то, какие операции могут быть выполнены в данном соединении. Это свойство напрямую связано со свойством `ConnectModeEnum` объекта ADO `Connection`.

- Значение `cmUnknown` указывает, что ограничения не были установлены или не могут быть определены.
- Значение `cmRead` указывает, что данное соединение может читать данные, но не изменять их.
- Значение `cmWrite` позволяет соединению только изменять данные, но не читать их.
- Значение `cmReadWrite` позволяет соединению читать и изменять данные.
- Значение `cmShareDenyRead` указывает, что другие пользователи не могут открыть соединение с разрешением на чтение.
- Значение `cmShareDenyWrite` указывает, что пользователи не могут открыть соединение с разрешением на запись.

- Значение cmShareExclusive запрещает другим пользователям открывать соединение.
- Значение cmShareDenyNone указывает, что другие пользователи не могут открыть соединение, независимо от их полномочий.

Свойство KeepConnection определяет, может ли данное приложение поддерживать связь с базой данных, если нет открытых наборов данных. Когда свойство имеет значение True, соединение будет удерживаться в открытом состоянии. Для соединений с удаленными СУБД или для приложений, которые часто открывают и закрывают наборы данных, установка значения свойства в True значительно уменьшает сетевой трафик и увеличивает скорость работы приложения, так как не требуется каждый раз проходить аутентификацию на сервере.

Для получения прямого доступа к объекту ошибок ADO следует обратиться к свойству Errors. Свойство DataSets содержит массив активных наборов данных, связанных с компонентом. А свойство DataSetCount позволяет получить число активных наборов данных, связанных с компонентом. Пример его использования показан в листинге 3.6.

Листинг 3.6. Обращение к наборам данных по их индексам

```
var
  i: Integer;
begin
  for i := 0 to (ADOConnection1.DataSetCount - 1) do
    ListBox1.Items.Add(ADOConnection1.DataSets[i].Name)
```

Свойство State позволяет узнать, в каком состоянии находится набор данных. А Свойство ConnectionString содержит строку, в которой указывается информация, необходимая для установки соединения с источником данных.

До и после открытия и закрытия соединения возникают методы-обработчики событий AfterConnect, BeforeConnect, AfterDisconnect и BeforeDisconnect соответственно. Методы BeginTrans, CommitTrans и RollbackTrans инициируют транзакцию, подтверждают ее и производят откат транзакции соответственно. Эти методы следует поместить в блоки секций try-finally и try-except.

Механизм соединения с хранилищем данных ADO

Перед созданием соединения необходимо определить его параметры. Для этого, как уже говорилось, предназначено свойство ConnectionString.

Набор параметров изменяется в зависимости от типа используемого провайдера и может настраиваться как вручную, так и с помощью редактора. Для того чтобы вызывать редактор соединений, необходимо дважды щелкнуть на компоненте TADOConnection. В результате будет активировано диалоговое окно, показанное на рис. 3.27.

В этом окне можно настроить соединение, используя поле Use Connection String, или загрузить параметры соединения из файла в разделе Use Data Link File.

Параметры соединения хранятся в файлах UDL, представляющих собой обычные текстовые файлы, содержащие параметры соединения. В листинге 3.7 приведен текст, содержащийся в демонстрационном файле.

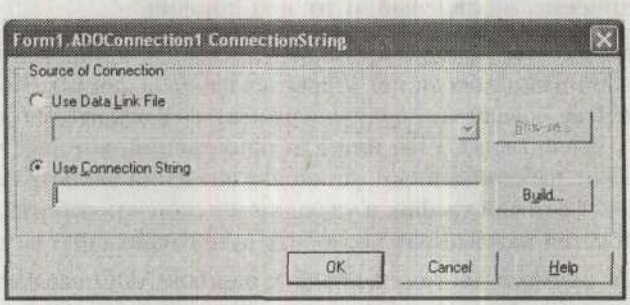


Рис. 3.27. Окно редактора строки соединения

Листинг 3.7. DBDEMOS.udl

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=D:\Program Files\Common
Files\Borland Shared\Data\DBDEMOS.mdb
```

Для того чтобы настроить соединение с данным провайдером, необходимо нажать на кнопку Build. Появится окно, изображенное на рис. 3.28, в котором будет опубликован список доступных провайдеров.

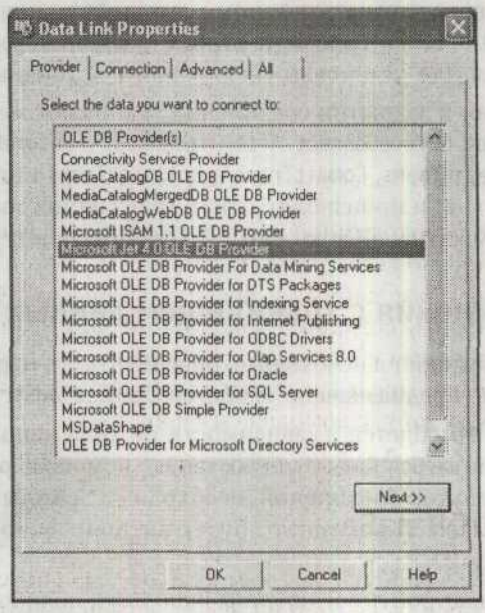


Рис. 3.28. Настройка параметров соединения

На вкладке Provider можно выбрать подходящий провайдер данных OLE DB для конкретного источника данных. В списке провайдеров также присутствуют провайдеры, предназначенные для доступа к конкретным службам операционной системы. На вкладке Connection необходимо указать путь к базе данных или сервер. Вкладка Advanced предназначена для указания режима доступа, аналогично свойству Mode. Вкладка All предназначена для более «тонкой» настройки специфичных свойств провайдера. Для дальнейшей работы нужно выбрать провайдер Microsoft Jet 4.0 OLE DB Provider. Затем нужно перейти на вкладку Connection. Появится окно, показанное на рис. 3.29.

Рис. 3.29. Настройка параметров источника данных

В появившемся окне необходимо указать путь к базе данных. В поле Select or enter a database name нужно указать путь к демонстрационной базе dbdemos.mdb. После указания пути к базе данных и задания остальных необходимых параметров нужно проверить созданное соединение при помощи кнопки Test Connection. Если параметры соединения указаны верно, появится сообщение Test connection succeeded. После закрытия этого окна в строке соединения будет отображена информация, с помощью которой провайдер сможет получить доступ к данным.

Класс TCustomADODataSet

Класс TCustomADODataSet является базовым классом, на основе которого построены компоненты наборов данных ADO. Класс инкапсулирует свойства, методы и сообщения для работы с провайдером данных через провайдер ADO. При-

ложения не могут непосредственно работать с объектами класса TCustomADODataSet, но могут взаимодействовать с компонентами, порожденными от данного класса, такими как ADODataSet, TADOTable, TADOQuery или TADOStoredProc. Класс TCustomADODataSet является прямым потомком класса TDataSet. Следовательно, он наследует его методы и свойства и добавляет собственные. Поэтому повторно рассматривать унаследованные возможности нет необходимости.

При помощи свойства Connection набор данных соединяется с хранилищем данных. В этом свойстве можно выбрать экземпляр компонента TADOConnection или указать параметры соединения при помощи обычной строки.

Свойство LockType позволяет определить блокировки, налагаемые на набор данных при его открытии:

- Значение ltUnspecified указывает, что тип блокировки не определен, поэтому она будет установлена источником данных.
- Значение ltReadOnly используется для открытия набора данных в режиме «только для чтения».
- Значение ltPessimistic указывает, что другие записи не могут редактировать запись до тех пор, пока она не будет записана в хранилище данных.
- Значение ltOptimistic указывает, что блокировка налагается на запись только в момент внесения изменений.
- Значение ltBatchOptimistic налагает блокировку на пакет записей во время записи в хранилище данных.

Для того чтобы получить сведения о состоянии набора данных, следует обратиться к свойству RecordsetState. Данное свойство напрямую связано со свойством State объектов ADO Connection, Command и Recordset.

Свойство CursorLocation определяет порядок обработки курсоров. А свойство CursorType определяет тип курсора набора данных ADO. Тип курсора набора данных указывает направление, в котором будет производиться перемещение по набору данных и, в соответствии с этим, отображение видимых записей. Тип курсора должен быть установлен перед активацией компонента набора данных.

Для этого свойства предусмотрен свой жестко заданный набор значений:

- Значение ctUnspecified указывает, что тип курсора не был определен, поэтому он будет выбран в соответствии с возможностями источника данных.
- Значение ctOpenForwardOnly задает использование однонаправленного курсора, допускающего перемещение по записям только вперед.
- Значение ctKeyset позволяет использовать двунаправленный курсор, не обеспечивающий возможность просмотра записей, добавленных и удаленных другими пользователями.
- Значение ctDynamic указывает, что данный тип курсора является двунаправленным, и обеспечивает просмотр всех изменений, внесенных в набор данных.

- Значение ctStatic позволяет использовать двунаправленный курсор, получающий слепок набора данных и, соответственно, игнорирующий изменения, внесенные другими пользователями. Используется в основном при создании отчетов.

В случае расположения курсора на стороне клиента поддерживается только статический курсор, задаваемый значением ctStatic. Если заданный тип курсора не поддерживается провайдером данных, то последний сам выбирает подходящий тип курсора.

После обновления набора данных вызывается метод-обработчик события OnFetchComplete. В качестве параметра метод может получать объект Error, который является ссылкой на одноименный объект ADO Error. Через параметр EventStatus набору данных возвращается сообщение, извещающее об успешном или неуспешном выполнении данной операции.

Перед перемещением записи вызывается метод-обработчик события OnWillMove. Параметр этого метода Reason содержит метод, вызвавший данный метод-обработчик. После перемещения записи вызывается метод-обработчик события OnMoveComplete.

Целочисленное свойство CacheSize определяет размер кэша набора данных. После передачи клиенту пакета записей они размещаются в буфере локальной памяти. Когда приложение перемещается по набору данных, ему пересылаются строки, расположенные в буфере.

Для получения размера записи следует обратиться к свойству RecordSize. А свойство BlockReadSize позволяет определить число записей, помещаемых в буфер при пакетной передаче. По умолчанию свойство имеет нулевое значение и является неактивным. Если значение свойства содержит ненулевое значение, то набор данных переводится в режим dsBlockRead.

Свойство RecordStatus определяет статус текущей записи. Это свойство содержит информацию о том, была ли запись добавлена, изменена, удалена или вообще не подвергалась изменениям. Данное свойство может также предоставить информацию о том, почему строка не была сохранена после ее модификации, удаления или добавления. Количество значений этого свойства строго регламентировано:

- Значение rsOK указывает, что запись была успешно сохранена.
- Значение rsNew свидетельствует, что запись была успешно добавлена.
- Значение rsModified указывает, что запись была изменена.
- Значение rsDeleted указывает, что запись была удалена.
- Значение rsUnmodified указывает, что запись не была изменена.
- Значение rsInvalid указывает, что запись не была сохранена.
- Значение rsMultipleChanges сигнализирует, что запись не может быть сохранена из-за множественных изменений.
- Значение rsPendingChanges указывает, что запись не может быть сохранена из-за ссылки на несохраненные изменения.

- Значение rsCanceled указывает, что операция с записью была отменена.
- Значение rsCantRelease указывает, что запись была заблокирована.
- Значение rsConcurrencyViolation сигнализирует, что запись не была сохранена из-за того, что использовалась оптимистическая блокировка.
- Значение rsIntegrityViolation указывает на нарушение ссылочной целостности.
- Значение rsMaxChangesExceeded указывает, что все изменения не могли быть сохранены из-за слишком большого их количества.
- Значение rsObjectOpen указывает на конфликт с открытым объектом хранилища данных.
- Значение rsOutOfMemory свидетельствует о недостатке оперативной памяти.
- Значение rsPermissionDenied указывает, что данный пользователь не имеет прав на совершение данной операции.
- Значение rsSchemaViolation указывает на нарушение структуры базы данных.
- Значение rsDBDeleted указывает, что запись была ранее удалена.

Метод UpdateStatus возвращает текущее состояние записи. Этот метод сигнализирует об изменениях в кэше, если они произошли. Значения, возвращаемые данным методом, приведены в списке:

- Значение usUnmodified указывает, что данная запись не содержит несохраненных изменений.
- Значение usModified свидетельствует о том, что запись содержит несохраненные изменения.
- Значение usInserted указывает, что текущая запись была добавлена в таблицу, но не была сохранена в хранилище данных.
- Значение usDeleted указывает, что запись была удалена, но изменения не были сохранены.

Метод-обработчик события OnWillChangeRecord вызывается перед внесением изменений в запись набора данных. А событие OnRecordChangeComplete инициируется сразу после внесения изменений. Параметр RecordCount метода, обрабатывающего это событие, содержит число измененных записей.

Свойство CommandType определяет тип выполняемой команды. В нем определяется тип команды, содержащейся в свойстве CommandText. Значение свойства CommandType должно согласовываться со значением свойства CommandText. Например, если свойство CommandText содержит название таблицы, то свойство CommandType должно иметь значение cmdTable или cmdTableDirect. По умолчанию свойству CommandType присваивается значение cmdUnknown. Если тип команды определен на этапе разработки, то существенно экономится время, требуемое для определения типа команды по ее значению. Данное свойство напрямую соединяется со свойством ADO CommandTypeEnum. Значения этого свойства указаны в списке:

- Значение cmdUnknown указывает, что тип команды не определен.
- Значение cmdText указывает, что выполняется текстовая команда, интерпретируемая источником данных (SQL-запрос или хранимая процедура).
- Значение cmdTable указывает, что данная команда возвращает набор данных из хранилища по имени таблицы.
- Значение cmdStoredProc указывает, что используется хранимая процедура.
- Значение cmdFile указывает, что применяется команда получения набора данных, сохраненных в файле
- Значение cmdTableDirect указывает, что применяется команда получения набора данных из таблицы напрямую.

Свойство CommandText используется для определения операторов выполняемой команды. Команды, заданные в свойстве CommandText, выполняются при активизации набора данных. В листинге 3.8 приведен пример использования обоих свойств.

Листинг 3.8. Использование объекта Command

```
with ADODataSet1 do begin
  CommandType := cmdText;
  CommandText := 'SELECT * FROM CustomerTable';
  Open;
```

Для настройки параметров процесса выполнения команд следует обратиться к свойству ExecuteOptions. Данное свойство может принимать четыре значения:

- Значение eoAsyncExecute указывает, что команды выполняются асинхронно.
- Значение eoAsyncFetch указывает, что используется асинхронное выполнение команд обновления набора данных.
- Значение eoAsyncFetchNonBlocking указывает, что будет применяться асинхронное обновление набора данных без наложения блокировок.
- Значение eoExecuteNoRecords указывает, что в данном режиме выполняемые команды и хранимые процедуры не возвращают результатов.

Во время разработки свойство Parameters используется для доступа к параметрам SQL-запросов, используемых компонентами ADO, инкапсулирующими наборы данных. Свойства Parameters можно использовать для установки и получения значений параметров или для установки и проверки их атрибутов. Свойство TParameters содержит коллекцию объектов Parameters.

К любому параметру, хранящемуся в данном экземпляре класса TParameters, можно обратиться по его индексу через свойство Items. А значения параметра можно получить при помощи свойства ParamValues, возвращающего вариантный массив, содержащий значения параметров. Имя параметра передается через константу ParamName. Если необходимо получить доступ более чем к одному параметру, то их названия разделяются точкой с запятой.

К параметру можно также обратиться по его имени, используя метод ParamByName. Имя параметра задается параметром Value.

Для добавления параметра в коллекцию необходимо использовать метод CreateParameter. Имя параметра содержится в свойстве Name. А тип данных параметра задается свойством DataType. Тип данных параметра связан с полем, с которым он взаимодействует. В целочисленном свойстве Size указывается размер значения параметра строкового типа.

Вид параметра определяется свойством TParameterDirection. Значения этого списка указаны далее:

- Значение pdUnknown применяется, когда тип параметра не был задан.
- Значение pdInput указывает, что передан был входной параметр, который используется в хранимых процедурах и запросах.
- Значение pdOutput применяется для обозначения выходного параметра, который используется в хранимых процедурах и запросах.
- Значение pdInputOutput определяет, что используется входной и выходной параметр, который используется в хранимых процедурах и запросах.
- Значение pdReturnValue указывает, что данный параметр возвращает значение.

Значение параметра содержится в свойстве Value. Свойство Attributes содержит атрибуты параметра, характеризующие тип его значения:

- Значение psSigned указывает, что параметру может быть присвоено символьное значение.
- Значение psNullable указывает, что значение параметра может быть пустым.
- Значение psLong указывает, что параметру могут быть присвоены данные типа BLOB.

Также параметру можно присвоить значения, загрузив их из файла или из потока методами LoadFromFile и LoadFromStream соответственно.

Для получения прямого доступа к объекту ADO Parameter можно обратиться к свойству ParameterObject. Конечно, обращаться напрямую к этому объекту не рекомендуется, но в некоторых случаях класс TParameter не предоставляет нужные свойства или методы. В листинге 3.9 приведен пример создания параметра и обращения к нему.

Листинг 3.9. Работа с параметрами

```
var Parametr : TParameter;
begin
  Parametr:=TParameter.Create(ADOQuery1.Parameters);
  Parametr.Name:='TestParam';
  Parametr.DataType:= ftString;
```

```
  Parametr.Direction:= pdInput;
  Parametr.Size:=10;
  Parametr.Value:='TestString';
  Memo1.Lines.Clear;
  for I:= 0 to ADOQuery1.Parameters.Count -1 do begin
    Memo1.Lines.Add(ADOQuery1.Parameters[I].Name);
  end;
end;
```

Класс TCustomADODataSet поддерживает помимо обычной фильтрации, наследованной от класса TDataSet, фильтрацию записей по их состоянию. Свойство FilterGroup задает условие фильтрации записей, основываясь на их состоянии. Возможные значения свойства указаны в списке:

- Значение fgUnassigned показывает, что условия фильтрации не заданы.
- Значение fgNone указывает, что условия фильтрации снимаются. Все записи становятся видимыми. Использование этого значения равноценно присваиванию свойству Filtered значения False.
- Значение fgPendingRecords показывает, что отображаются те записи, которые были изменены, но не были сохранены в хранилище данных (UpdateBatch), или те, изменения которых были отменены (CancelBatch).
- Значение fgAffectedRecords показывает, что фильтр отображает те записи, которые были затронуты при последнем сохранении.
- Значение fgFetchedRecords показывает, что отображаются записи, полученные при последнем обновлении из источника данных.
- Значение fgPredicate показывает, что отображаются удаленные записи.
- Значение fgConflictingRecords показывает, что отображаются записи, которые не удалось сохранить в хранилище данных вследствие возникших ошибок.

Для того чтобы фильтрация могла быть выполнена, необходимо свойству LockType присвоить значение ltBatchOptimistic и включить ее, присвоив свойству Filter значение True. В листинге 3.10 приведен пример использования фильтрации.

Листинг 3.10. Групповая фильтрация

```
  with ADOTable1 do begin
    close;
    LockType:=ltBatchOptimistic;
    Filtered:=True;
    open;
    FilterGroup:= fgFetchedRecords;
end;
```

Для поиска в наборе данных по полям текущего индекса можно использовать метод Seek. В параметре этого метода KeyValues перечисляются значения индекса, по которым будет вестись поиск. Если индекс состоит из одного поля, то поиск будет вестись по нему. Если индекс составляют несколько полей, то значения полей указываются в порядке возрастания важности. Параметр SeekOption определяет порядок возврата результата поиска:

- Значение soFirstEQ указывает, что курсор устанавливается на первую найденную запись. Если запись не найдена, то курсор переводится на последнюю запись набора.
- Значение soLastEQ указывает, что курсор устанавливается на последнюю найденную запись. Если запись не найдена, то курсор переводится на последнюю запись набора.
- Значение soAfterEQ указывает, что курсор устанавливается на найденную запись или, если она не найдена, после того места, где она могла бы находиться.
- Значение soAfter указывает, что курсор устанавливается за найденной записью.
- Значение soBeforeEQ указывает, что курсор устанавливается на найденную запись или, если она не найдена, перед тем местом, где она могла бы находиться.
- Значение soBefore указывает, что курсор устанавливается перед найденной записью.

Для того чтобы использовать данный метод, необходимо выполнить несколько условий:

- Индекс, по которому будет производиться поиск, должен быть выбран в свойстве IndexName.
- Свойство CommandType должно иметь значение cmdTableDirect (для TADOTable свойству TableDirect необходимо присвоить значение True).
- Свойство CursorLocation должно иметь значение clUseServer.
- Свойство CursorType должно иметь значение ctKeySet.

В листинге 3.11 приведены примеры использования данного метода.

Листинг 3.11. Использование метода Seek

```
ADODataSet1.Seek(VarArrayOf([90030, 90020]), soFirstEQ);
SuccessVar := ADODataSet1.Seek('Jones', soFirstEQ);
```

Данные, хранящиеся в кэше набора данных, можно сохранить в файл и загрузить их из файла, используя методы SaveToFile и LoadFromFile соответственно. Параметр FileName содержит имя файла, с которым будут оперировать методы. В параметре Format указывается формат, в котором будет сохранен набор данных. По умолчанию используется значение pfADTG.

Компонент TADODataSet

Данный компонент представляет набор данных, получаемых из хранилища ADO. Компонент наследует свойства и методы класса TCustomADODataSet и добавляет несколько своих. Компонент имеет возможность получать результирующие наборы данных от одной или нескольких таблиц.

Используя свойство CommandText, можно указать текст команды, при помощи которой будут получены данные. Это может быть SQL-запрос, название таблицы или название хранимой процедуры. А в свойстве CommandType указывается тип исполняемой команды. Данный компонент не может выполнять SQL-операторы, не возвращающие результирующие наборы (операторы DELETE, INSERT и UPDATE). Соединение с базой данных задается свойствами Connection или ConnectionString. Компонент связывается с компонентами отображения данных стандартным способом.

Компонент TADOTable

Компонент TADOTable используется для доступа к хранилищам данных ADO и представления информации из них в табличном виде. Компонент предоставляет прямой доступ к каждой записи и ее полям, наследуя свойства и методы класса TCustomADODataSet. Компонент связывается с базой данных через свойства Connection или ConnectionString.

Имя таблицы указывается в свойстве TableName. Свойство TableDirect указывает, каким образом набор данных связывается с хранилищем данных. Так как не все провайдеры поддерживают прямое соединение с набором данных, то в некоторых случаях для связи с хранилищем данных приходится использовать SQL-операторы. При установке свойству значения True компонент использует фоновые SQL-запросы для доступа к данным.

Используя свойство ReadOnly, можно установить ограничение «только для чтения» на данную таблицу, запретив, таким образом, возможность изменять данные. В свойстве MasterSource указывается компонент TDataSource, используемый для создания отношения ссылочной целостности Master-Detail.

Метод GetIndexNames возвращает список индексов, доступных компоненту в качестве списка.

Компонент TADOQuery

Компонент TADOQuery позволяет выполнять SQL-запросы при работе с данными через ADO. Соединение с хранилищем данных осуществляется стандартным методом. Текст запроса содержится в свойстве SQL. В листинге 3.12 приведен пример типичного SQL-запроса, содержащегося в упомянутом свойстве.

Листинг 3.12. Выполнение SQL-запроса

```
with ADOQuery1 do begin
  Close;
  with SQL do begin
```

```
    Clear;
    Add('SELECT EmpNo, LastName, FirstName, HireDate');
    Add('FROM Employee');
  end;
  Open;
end;
```

Параметры запроса содержатся в свойстве Parameters. В случае, если компонент возвращает набор данных, его следует открывать методом Open или присвоить свойству Active значение True. Если запрос не должен возвращать набор данных (операторы INSERT, UPDATE, DELETE и CREATE TABLE), то запрос следует выполнять вызовом метода ExecSQL. Метод возвращает число обработанных запросом записей.

Свойство RowsAffected содержит число записей, которые затронул последний выполнявшийся запрос.

Компонент TADOStoredProc

Компонент TADOStoredProc позволяет обращаться к хранимым процедурам, содержащимся в базах данных. Связь компонента с хранилищем данных осуществляется стандартно, через названные ранее свойства.

Имя хранимой процедуры определяется свойством ProcedureName. Во время разработки хранимую процедуру можно выбрать из списка уже существующих процедур.

Для определения входных и выходных параметров процедуры используется свойство Parameters. Через параметры хранимая процедура получает аргументы и возвращает результаты своей работы.

В случае, если хранимая процедура будет вызываться множество раз с одними и теми же аргументами, рекомендуется заранее подготовить ее к исполнению. СУБД разместит ее в оперативной памяти. Для этого свойству Prepared следует присвоить значение True.

Пример связи с Access через ADO

Ранее уже рассматривалась процедура соединения с базой Access. В этом разделе будет приведен пример приложения, оперирующего базами данных в этом формате.

В новом проекте необходимо создать отдельный модуль данных. В нем надо расположить компонент TADOConnection и настроить соединение с базой данных. Свойству LoginPrompt следует присвоить значение True и активировать соединение. Также потребуются связанные друг с другом компоненты TADOTable и TDataSource.

Компонент TADOTable должен быть связан с компонентом TADOConnection при помощи свойства Connection. В свойстве TableName следует выбрать таблицу Customer, а затем активировать компонент.

На главной форме надо разместить компонент TDBGrid и связать его с компонентом TDataSource. Также потребуется один компонент TMainMenu. В этом меню будут реализованы пункты для создания новой записи, принятии изменений, сохранении изменений, удаления записи, записи данных в файл и загрузки их из файла, а также для установки соединения с базой данных и отключения от нее. Код примера приведен в листинге 3.13, а внешний вид приложения показан на рис. 3.30. Методы SaveToFile и LoadFromFile позволяют сохранить набор данных в файл, а позже считать его оттуда. Используя данные методы, можно загрузить набор данных с помощью компонента TADOConnection, сохранить его и потом работать с локальной базой данных. Эта технология довольно проста:

1. Загрузить данные с помощью TADOConnection в TADOTable.
2. Методом SaveToFile компонента TADOTable сохранить набор данных в выбранном файле.
3. Удалить компонент TADOConnection.
4. Методами LoadFromFile и SaveToFile можно загружать набор данных из файла и сохранять его в файле.

Листинг 3.13. Работа с Access

```
procedure TForm1.SaveClick(Sender: TObject);
begin
  DataMod.ADOTable1.UpdateBatch();
end;

procedure TForm1.SaveToFileClick(Sender: TObject);
begin
  DataMod.ADOTable1.SaveToFile('Base.bas');
end;

procedure TForm1.NewClick(Sender: TObject);
begin
  DataMod.ADOTable1.Insert;
end;

procedure TForm1.DeleteClick(Sender: TObject);
begin
  DataMod.ADOTable1.Delete;
end;

procedure TForm1.LoadFromFileClick(Sender: TObject);
```

продолжение

Листинг 3.13 *(продолжение)*

```
begin
  DataMod.ADOTable1.LoadFromFile('Base.bas');
end;

procedure TForm1.ExitClick(Sender: TObject);
begin
  Close;
end;

procedure TForm1.DisconnectClick(Sender: TObject);
begin
  DataMod.ADOTable1.SaveToFile('Base.bas');
  DataMod.ADOConnection.Connected:=False;
end;

procedure TForm1.ConnectClick(Sender: TObject);
begin
  DataMod.ADOConnection.Connected:=True;
  DataMod.ADOTable1.Active:=True;

end;

procedure TForm1.PostClick(Sender: TObject);
begin
  DataMod.ADOTable1.Post;
end;
```

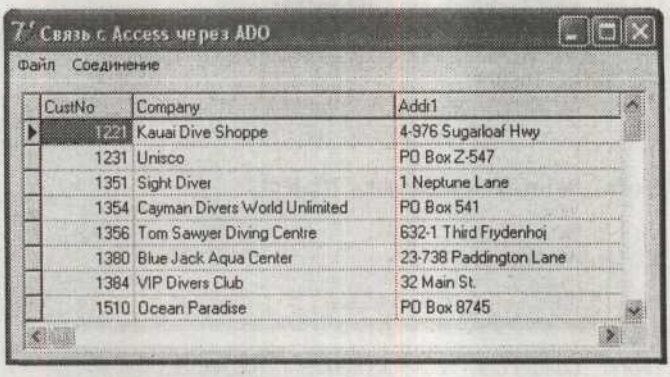


Рис. 3.30. Связь с таблицей Access через ADO

Пример связи с SQL Server 2000 через ADO

В этом разделе будет рассмотрен пример взаимодействия с SQL Server 2000. В новом приложении необходимо на форме расположить компонент TADOConnection. После этого необходимо установить соединение с сервером. В окне настройки параметров соединения требуется выбрать провайдер Microsoft OLE DB Provider SQL Server. На вкладке Connection в поле Select or enter a server name нужно выбрать сервер из списка. Во втором поле следует указать тип авторизации. Логин и пароль могут передаваться на сервер от текущей учетной записи, зарегистрированного в системе пользователя либо их можно явно указывать при соединении. В третьем пункте из списка надо выбрать базу данных Northwind. Функциональность примера останется той же, что и в предыдущем методе, поэтому в компоненте TADOTable можно просто выбрать Customers. На стадии визуального проектирования это будет единственным изменением предыдущего примера.

В главном меню TMainMenu метод Сохранить надо заменить методом Обновить. Также стоит установить максимальный уровень изоляции транзакции и понаблюдать за тем, как изменяются данные. Форма приложения показана на рис. 3.31. В процессе экспериментов могут возникнуть ошибки, которые нужно обработать с помощью свойства OnPostError. Параметр этого метода E содержит в себе объект ошибки и ее текст, а параметр Action определяет тип реакции на ошибку:

- Значение daFail указывает, что операция прерывается и возвращается сообщение об ошибке.
- Значение daAbort указывает, что операция прерывается без сообщения об ошибке.
- Значение daRetry указывает, что операция, в процессе выполнения которой возникла ошибка, будет повторена.

Разработать простое диалоговое окно не составляет труда. Достаточно лишь выбрать тип действия и передать его параметру Action.

Рис. 3.31. Связь с SQL Server 2000 через ADO

dbExpress

Технология *dbExpress* была разработана Borland для того, чтобы заменить несколько устаревшую технологию BDE. Эта технология доступа к данным обеспечивает взаимодействие приложений с СУБД. Данная технология является кросс-платформенной, не зависит от конкретной СУБД и имеет открытые интерфейсы, предоставляющие методы для выполнения динамически формируемых запросов. dbExpress возвращает только однонаправленные курсоры, не поддерживает кэширование и, следовательно, не позволяет напрямую редактировать наборы данных.

Технология dbExpress представляет собой совокупность драйверов, компонентов, инкапсулирующих соединения, транзакции, запросы и наборы данных, а также интерфейсов, обеспечивающих универсальный доступ к функциям dbExpress. Драйверы доступа к СУБД реализованы в виде динамических библиотек. Данная особенность позволяет распространять приложения, использующие эту технологию доступа к данным, без лишних хлопот. На машине конечного пользователя обязательно должна быть установлена клиентская часть СУБД.

Драйверы для доступа к СУБД поставляются Borland и сторонними разработчиками.

Интерфейсы dbExpress

Базовые классы `SQLDriver`, `SQLConnection`, `SQLCommand` и `SQLCursor` составляют ядро рассматриваемой технологии.

Интерфейс `ISQLDriver` предоставляет методы, необходимые для обслуживания соединения с драйвером, и производит определенные действия, например, загрузку поставщика клиента, инициализацию окружения, определение необходимых указателей. После отработки возвращается объект `SQLConnection`. Интерфейс поддерживает три метода. Методы `setOption` и `getOption` позволяют работать с параметрами драйвера. Метод `getSQLConnection` возвращает указатель на интерфейс `ISQLConnection`.

Интерфейс `ISQLConnection` устанавливает соединение с базой данных, возвращает экземпляры объекта `SQLCommand` для выполнения запросов и хранимых процедур, возвращает экземпляры объекта `SQLMetaData` для получения метаданных и управляет транзакциями.

Метод `Connect` устанавливает соединение с базой данных. Его параметры `pszServerName`, `pszUserName` и `pszPassword` содержат название базы данных, имя пользователя и пароль.

Завершить текущее соединение с базой данных можно при помощи метода `Disconnect`. Метод `getSQLCommand` предоставляет интерфейс `ISQLCommand`, который может быть использован при обработке SQL-запроса или хранимой процедуры. Данный метод следует вызывать только после того, как объект `SQLConnection` будет связан с базой данных.

Для получения информации о таблицах, хранимых процедурах, индексах, схемах и других объектах базы данных следует вызвать метод getSQLMetaData. Он возвращает интерфейс ISQLMetaData. Используя методы этого интерфейса, можно получить доступ к метаданным, перечисленным выше. Данный метод следует вызывать только после того, как объект SQLConnection будет связан с базой данных.

Используя методы setOption и getOption, можно управлять параметрами соединения и получать информацию о них. Для управления транзакциями предназначены методы beginTransaction, commit и rollback. Их названия достаточно прозрачно отражают функциональность методов. Параметр ulTransDesc содержит указатель на структуру TransactionDesc. Свойства структуры перечислены ниже:

- Свойство TransactionID содержит уникальный идентификатор транзакции.
- Свойство GlobalID содержит глобальный идентификатор транзакции, используется в СУБД Oracle.
- Свойство IsolationLevel указывает уровень изоляции транзакции.
- Свойство CustomIsolation содержит уровень изоляции транзакции, установленный данной СУБД. СУБД присваивает значение, если уровень изоляции не был задан явно.

Метод getErrorMessage используется для получения информации о последней ошибке, произошедшей при выполнении какой-либо операции. Для определения длины строки, содержащей сообщение о последней произошедшей ошибке, следует вызвать метод getErrorMessageLen.

Интерфейс ISQLCommand предоставляет методы для выполнения запросов и хранимых процедур. Он может быть использован для получения экземпляра интерфейса ISQLCursor после выполнения команды, возвращающей курсор. Он поддерживает выполнение «подготовленных» запросов.

Для установки свойств запроса можно использовать метод setOption. В его параметре eSOption указывается свойство команды, значение которого будет изменено. Параметр lValue содержит устанавливаемое значение.

Для получения информации о свойствах запроса следует использовать метод getOption. В его параметре eSOption указывается имя параметра, значение которого будет получено. Параметр plValue является указателем на переменную, в которую будет помещено полученное значение. Параметр iMaxLength определяет максимальный размер в байтах, который может быть возвращен в параметре plValue. Соответственно, в параметре Length возвращается длина значения, содержащегося в plValue.

Метод setParameter устанавливает значения параметров запросов и хранимых процедур до их выполнения. Передаваемое значение uParameterNumber содержит порядковый номер параметра запроса или хранимой процедуры. Если параметр является дочерним для сложных типов данных, параметр задает его порядковый номер. Значение ePType указывает тип параметра, а значение

uLogType содержит тип его данных. Значение uSubType определяет соответствующий подтип параметра uLogType, а lMaxPrecision задает максимальный класс точности возвращаемого значения. Свойство lMaxScale содержит максимальный размер значения в байтах, ulLength — размер буфера, pBuffer — указатель на буфер, содержащий значение параметра, а bIsNull является флагом, определяющим, может ли значение параметра иметь нулевое значение.

Для получения значений параметров можно использовать метод getParameter. Значение pData содержит указатель на буфер, в который передается значение параметра ulLength.

Метод prepare указывает СУБД на то, чтобы она подготовила SQL-запрос или хранимую процедуру к выполнению.

Для выполнения запроса или хранимой процедуры используется метод execute, возвращающий экземпляр интерфейса ISQLCursor в параметре Cursor, если запрос или хранимая процедура возвращают курсор. Пример приведен в листинге 3.14.

Листинг 3.14. Выполнение запроса

```
var
  Status : SQLResult;
  CustID : Integer;
  Cursor: ISQLCursor;
begin
  Status := Command.Prepare('Select * from Customer where CustID = ?', 1);
  if Status = SQL_SUCCESS then
  begin
    CustID := 201;
    Status := Command.SetParameter(1, 0, paramIN, fldINT32, 0, 4, 0, 4,
Addr(CustID), 0);
  end;
  if Status = SQL_SUCCESS then
    Status := Command.Execute(Cursor);
```

Метод executeImmediate выполняет запросы и хранимые процедуры, не требующие подготовки. Приложения не должны вызывать метод Prepare перед вызовом данного метода. При выполнении с помощью данного метода хранимые процедуры и запросы не могут содержать параметров. Текст запроса или хранимая процедура указываются в свойстве pszSQL. Если команда возвращает курсор, то он содержится в параметре Cursor.

Метод getNextCursor возвращает в параметре Cursor курсор следующего набора данных, если выполнялась хранимая процедура, возвращающая множество курсоров.

Для получения количества строк, затронутых в ходе выполнения данной хранимой процедуры или запроса, используется метод getRowsAffected. В его параметре Rows возвращается количество измененных строк.

Интерфейс ISQLCursor владеет данными и метаданными, полученными в ходе выполнения запроса или хранимой процедуры. Он имеет методы для получения информации об отдельных полях и для получения их значений.

Методы getColumnCount, getColumnNameLength, getColumnName, getColumnType, getColumnLength, getColumnPrecision и getColumnScale позволяют получить различную информацию о полях. Параметр этих методов uColumnNumber содержит номер поля, о котором необходимо получить информацию.

Метод isNullable позволяет определить, может ли данное поле содержать нулевое значение. Режим доступа к полю можно узнать, используя метод isReadOnly. Этот метод возвращает значение True, если поле имеет разрешение только на чтение и не может быть модифицировано.

Метод next обновляет курсор, занося в него значение из следующей строки набора данных. Обратиться к значению поля можно при помощи метода getString.

На базе четырех основных интерфейсов построен интерфейс ISQLMetaData. Данный интерфейс позволяет получить различные метаданные, принадлежащие открытому соединению с базой данных.

Методами setOption и getOption можно получить и настроить значения различных параметров курсора.

Методы getObjectList, getTables, getProcedures, getColumns, getProcedureParams и getIndices позволяют получить различную информацию о структуре базы данных.

В основе библиотеки VCL-компонентов, реализующих технологию dbExpress, лежат несколько описанных выше интерфейсов. Изложенная информация позволит лучше разобраться в механизме их работы.

Компонент TSQLConnection

Данный компонент используется для установления соединения с СУБД. С одним компонентом TSQLConnection может быть связано несколько компонентов набора данных через их свойство Connection. Компонент взаимодействует с драйвером и двумя файлами. Файл dbxdrivers.ini содержит список установленных драйверов и набор настроек для каждого драйвера. Файл dbxconnections.ini содержит базовый набор настроек.

Для установки соединения с базой данных следует для свойства Connected установить значение True. Также открыть или закрыть соединение можно при помощи методов Open и Close. При открытии и закрытии соединения с базой данных возникают события AfterConnect, AfterDisconnect, BeforeConnect и BeforeDisconnect. При помощи соответствующих методов можно реализовать различные проверки данных.

Свойство ConnectionName содержит имя настроенного ранее соединения. Его можно выбрать из списка. А свойство DriverName определяет драйвер dbExpress, используемый для взаимодействия с СУБД. Если необходимо получить имя

динамической библиотеки драйвера dbExpress, следует обратить внимание на значение свойства LibraryName.

В свойстве Params содержится список параметров соединения. В параметрах можно указать имя пользователя и пароль, которые будут подставляться автоматически при установке соответствующего свойства.

Эти свойства получают свои значения автоматически при выборе конкретного соединения с базой данных в свойстве ConnectionName.

Свойство ConnectionState указывает текущее состояние соединения.

Если для каких-либо целей потребуется создать копию объекта SQLConnection, следует воспользоваться методом CloneConnection.

Свойство ActiveStatements позволяет узнать общее число запросов, выполняющихся на сервере в данный момент. А в свойстве MaxStmtsPerConn указывается максимальное число одновременно выполняющихся запросов, установленное сервером. Если свойство имеет нулевое значение, то ограничение просто не установлено.

Обратиться к конкретному набору данных можно через его индекс, используя свойство DataSets. А метод GetTableNames позволяет получить список таблиц базы данных. Если его параметру SystemTables присвоить значение True, то метод также вернет имена системных таблиц.

Свойство TableScope позволяет задать типы таблиц, которые попадут в список, полученный при вызове метода GetTableNames. Список значений этого свойства не так уж велик:

- Значение tsSynonym указывает, что будут возвращаться синонимы таблиц.
- Значение tsSysTable позволит получить имена системных таблиц.
- Значение tsTable включает в список таблицы базы данных.
- Значение tsView добавляет в список представления.

Свойство TableScope будет работать только в том случае, когда параметр SystemTables имеет значение False.

Для получения списка полей конкретной таблицы следует использовать метод GetFieldNames. В его параметре TableName указывается имя таблицы, имена полей которой необходимо получить.

Метод GetProcedureNames возвращает список хранимых процедур. Для получения списка индексов используется метод GetIndexNames. Как и ранее, в параметре TableName указывается таблица, для которой возвращаются индексы.

Метод GetProcedureParams возвращает параметры процедуры, указанной в параметре ProcedureName. Если установлено соединение с СУБД Oracle, дополнительно необходимо задать значение параметра PackageName.

Выполнение команды на сервере инициируется методом Execute. Параметр SQL содержит выполняемую команду, а в параметре Params указывается коллекция объектов типа TParams, содержащая параметры команды. Пример использования данного метода приведен в листинге 3.15.

Листинг 3.15. Пример использования метода Execute

```
procedure TForm1.InsertWithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    SQLConnection1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    SQLConnection1.Execute(SQLstmt, stmtParams);
  finally
    stmtParams.Free;
  end;
end;
```

Начало, откат и фиксацию транзакции выполняют методы **StartTransaction**, **Commit** и **Rollback** соответственно. Параметры транзакции содержатся в записи **TTransactionDesc**.

В листинге 3.16 приведен пример работы с транзакцией. Тело транзакции помещено в блок **try-except**, что позволяет обработать возможную ошибку и произвести откат изменений.

Листинг 3.16. Пример работы с транзакциями

```
procedure TForm1.TransferButtonClick(Sender: TObject);
var
  Amt: Integer;
  TD: TTransactionDesc;
begin
  if not SQLConnection1.InTransaction then
  begin
    TD.TransactionID := 1;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
    try
      Amt := StrToInt(AmtEdit.Text);
      Debit.Params.ParamValues['Amount'] := Amt;
```

продолжение

Листинг 3.16 *(продолжение)*

```
      Credit.Params.ParamValues['Amount'] := Amt;
      SQLConnection1.Commit(TD); {on success, commit the changes};
    except
      SQLConnection1.Rollback(TD); {on failure, undo the changes};
    end;
  end;
end;
```

Свойство MultipleTransactionsSupported указывает, поддерживает ли данный сервер одновременное выполнение нескольких транзакций. Если свойство имеет значение True, то сервер поддерживает множественные транзакции. В этом случае каждой транзакции, запущенной методом StartTransaction, необходимо присваивать уникальный идентификатор.

Свойство InTransaction позволяет выяснить, выполняется ли в данный момент транзакция. Если транзакция запущена, свойство имеет значение True.

Соединение с сервером баз данных

Для создания соединения с сервером используется компонент TSQLConnection. С этим компонентом связываются все остальные компоненты, которым требуется доступ к данным в рамках технологии dbExpress.

Как было отмечено ранее, свойство ConnectionName позволяет выбрать из полученного списка настроенное соединение с СУБД. Для того чтобы настроить соединение, необходимо вызвать редактор соединений. Для этого достаточно дважды щелкнуть по компоненту TSQLConnection. Появится окно редактора соединений, показанное на рис. 3.32. В списке Connection Name содержатся существующие соединения. В правой части окна, в редакторе свойств Connection Settings указываются параметры выбранного соединения. В верхней части окна расположена панелька с кнопками, позволяющими добавить соединение в список, удалить его из списка, переименовать и проверить его.

Теперь нужно создать новое соединение. Для этого следует нажать на кнопку +. В результате будет отображено диалоговое окно, показанное на рис. 3.33, в котором нужно выбрать тип драйвера из списка Driver Name и указать название соединения в поле Connection Name.

Необходимо выбрать тип драйвера Interbase, а соединение можно назвать TestConnection. После ввода данных нужно нажать кнопку OK.

В диалоговом окне редактора соединений только что созданное соединение нужно выбрать в списке Connection Name. Для него в поле DataBase нужно указать путь к базе данных EMPLOYEE.GDB. Данная база данных находится обычно в каталоге Program Files\Borland\InterBase\examples\Database. Помимо указания местоположения базы данных, можно задать имя пользователя и пароль в полях User_Name и Password. Также можно настроить другие параметры соединения.

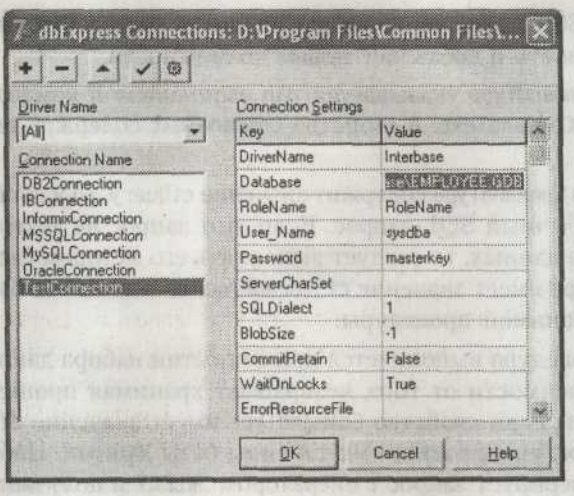


Рис. 3.32. Окно редактора соединений

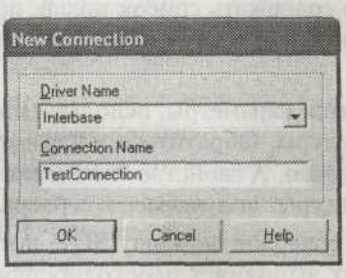


Рис. 3.33. Создание нового соединения

Класс TCustomSQLDataSet

Класс `TCustomSQLDataSet` является базовым классом для всех компонентов dbExpress, инкапсулирующих наборы данных. Он имеет свойства, сообщения и методы для работы с данными, полученными от dbExpress. Приложения не могут непосредственно обращаться к методам данного класса. Для этого можно использовать лишь унаследованные классы `TSQLDataSet`, `TSQLTable`, `TSQLQuery` и `TSQLStoredProc`. Класс `TCustomSQLDataSet` и его потомки являются однонаправленными наборами данных. Подобные наборы данных не могут хранить множество записей в буфере, поэтому они имеют некоторые ограничения:

- Невозможны поиск или свободное перемещение по набору данных. Поддерживается лишь последовательное перемещение между записями.
- Прямое редактирование набора данных невозможно. Данные могут быть модифицированы только при явном выполнении SQL-запроса при помощи операторов `UPDATE`, `INSERT` и `DELETE`.
- Не поддерживаются фильтрация и поля синхронного просмотра.

Класс TCustomSQLDataSet является наследником класса TDataSet, наследует его функциональность и добавляет новые возможности.

В свойстве CommandType указывается тип выполняемой команды, содержащейся в свойстве CommandText. А свойство CommandText содержит выполняемую команду.

Если свойство CommandType содержит значение ctQuery, то в свойстве CommandText содержится обычный SQL-запрос. Если этот запрос не возвращает результирующий набор данных, то следует выполнять его методом ExecSQL. Если свойство CommandType имеет значение ctStoredProc, то в свойстве CommandText указывается имя хранимой процедуры.

Хранимая процедура выполняется при открытии набора данных или методом ExecSQL, в зависимости от того, возвращает хранимая процедура записи или нет. И наконец, если свойство CommandType имеет значение ctTable, то в свойстве CommandText указывается имя таблицы базы данных. Набор данных автоматически генерирует запрос с оператором SELECT и получает все записи.

Для определения порядка сортировки записей используется свойство SortFieldNames. Оно должно содержать список полей, перечисленных через точку с запятой. Имена полей используются при составлении запроса в выражении ORDER BY.

В свойстве Params хранятся параметры, используемые при выполнении запроса или хранимой процедуры. Обратиться к параметру по его имени можно, используя метод ParamByName. А свойство ProcParams содержит описание параметров хранимой процедуры, получаемое с сервера.

Свойство NativeCommand содержит команду SQL, посылаемую на сервер. В некоторых случаях объект выполняет перекодировку команды, содержащейся в свойстве CommandText, и помещает ее в данное свойство.

По умолчанию свойство Params автоматически обновляется при изменении параметров запроса, содержащегося в свойстве CommandText. Свойство ParamCheck позволяет указать, проводить автоматическое обновление списка параметров или нет.

Свойство NoMetadata позволяет запретить передачу метаданных на сервер. Установка свойству значения False в некоторых случаях несколько увеличит производительность работы, но не позволит совершать некоторые операции, зависящие от индексов.

Для обработки ошибок часто используется свойство LastError, которое содержит сообщение о последней ошибке dbExpress.

В некоторых случаях наборы данных SQL используют метаданные чаще, чем результаты запросов и хранимых процедур. Свойство SchemaInfo определяет, передаются ли метаданные, и в случае их передачи устанавливает их тип. Разработчик может определить типы метаданных, которые будут переданы приложению, в структуре SchemaInfo, обратиться к которой можно через одноименное свойство. Поля структуры перечислены в списке:

- Поле FType определяет тип метаданных, которые будут запрошены.

- Поле ObjectName содержит название таблицы или хранимой процедуры, для которых получается информация о столбцах, индексах и параметрах. Этот параметр будет содержать или не содержать значение, в зависимости от типа метаданных, указанного в параметре FType.
- Поле Pattern определяет ограничения, накладываемые на метаданные.
- В поле PackageName указывается имя пакета, содержащего хранимую процедуру, о которой запрашивается информация. Данное свойство используется только для СУБД Oracle.

Значение поля FType принадлежит к перечислимому типу:

- Значение stNoSchema свидетельствует, что схема метаданных не определена.
- Значение stTables возвращает информацию обо всех таблицах с данными. Данный параметр сравним с критерием, задаваемым в свойстве TableScope компонента TSQLConnection.
- Значение stSysTables возвращает информацию об используемых системных таблицах. Следует учитывать, что не все СУБД используют системные таблицы для хранения метаданных.
- Значение stProcedures возвращает информацию обо всех хранимых процедурах базы данных.
- Значение stColumns возвращает информацию обо всех полях таблицы.
- Значение stProcedureParams возвращает информацию о параметрах хранимой процедуры.
- Значение stIndexes возвращает информацию обо всех индексах, определенных для данной таблицы.
- Значение stPackages возвращает информацию обо всех пакетах данной базы данных. Используется только для сервера Oracle.

Метод SetSchemaInfo используется для определения параметров структуры SchemaType, так как прямой доступ к ее полям невозможен.

Компонент TSQLDataSet

Компонент TSQLDataSet представляет собой однонаправленный набор данных, полученный при помощи технологии dbExpress. Он позволяет выполнять запросы и хранимые процедуры, а также получать данные из таблиц.

В свойстве CommandType указывается тип выполняемой команды, а в свойстве CommandText — ее текст либо имя таблицы или хранимой процедуры.

Для открытия или закрытия набора данных используются методы Open и Close. Впрочем, можно оперировать и свойством Active. Если запрос или хранимая процедура не возвращают набор данных, следует использовать метод ExecSQL. Его параметр ExecDirect указывает на необходимость подготовки команды перед выполнением, если она содержит параметры. Если команда не содержит параметров, параметру следует присвоить значение True.

Через свойство TransactionLevel можно определить уровень изоляции транзакции.

Соединение с компонентом TSQLConnection устанавливается через свойство SQLConnection. А свойство IndexDefs содержит описание всех индексов, используемых в результирующем наборе данных.

Компонент TSQLTable

Компонент TSQLTable инкапсулирует однонаправленный набор данных и используется для представления информации, получаемой из базы данных, в табличном виде. Компонент генерирует запросы на получение всех записей со всеми полями для выбранной таблицы.

Свойство IndexName содержит имя текущего индекса, по которому производится сортировка записей и по которому устанавливаются реляционные отношения между таблицами. Через свойство IndexFields можно обратиться к полям, входящим в данный индекс. Номер соответствующего поля указывается в параметре Index.

В свойстве IndexFieldNames перечислены поля, входящие в индекс, установленный текущим. Поля перечисляются через точку с запятой. Если необходимо узнать количество полей, входящих в текущий индекс, стоит обратиться к свойству IndexFieldCount. Метод GetIndexNames возвращает список индексов для рассматриваемой таблицы.

Для настройки отношений ссылочной целостности между таблицами используются свойства MasterFields и MasterSource.

Метод PrepareStatement генерирует SQL-запрос для получения данных от СУБД. А метод DeleteRecords применяется для удаления из таблицы всех записей.

Компонент TSQLQuery

Данный компонент используется для выполнения SQL-запросов на сервере. Он может возвращать результаты запроса в виде однонаправленного набора данных.

В свойстве SQL указывается текст SQL-запроса, который будет выполнен на сервере. А свойство Text представляет SQL-запрос в виде текстовой строки.

Метод PrepareStatement вызывается для подготовки запроса к выполнению на сервере. После того как метод будет подготовлен, его можно выполнить при помощи метода ExecSQL. Этот метод обычно используется для тех SQL-запросов, которые не возвращают результатов.

Компонент TSimpleDataSet

Компонент используется для получения данных и обеспечивает их кэширование. Объект TSimpleDataSet является клиентским набором данных, исполь-

зующим TSQLDataSet и TDataSetProvider для получения данных и передачи произведенных изменений на сервер. Он сочетает быстрый доступ к данным и легкость развертывания однонаправленных наборов с возможностями редактирования и навигации, предоставляемыми клиентскими наборами данных. Таким образом, компонент предоставляет в распоряжение разработчика двунаправленный курсор, позволяет отображать данные в таблицах и редактировать их в режиме кэширования.

Соединение с источником данных осуществляется при помощи свойства Connection. Но можно использовать и дублирующее свойство ConnectionName, в котором указывается только наименование соединения.

Тип выполняемой команды указывается в свойстве CommandType, а ее содержание задается в свойстве CommandText.

В свойстве FetchOnDemand можно указать возможность получения пакетов данных по запросу. По умолчанию свойство имеет значение True, то есть клиентский набор данных получает дополнительные пакеты при перемещении курсора по набору данных. Если свойство имеет значение False, разработчик обязан явно производить получение новых данных при помощи метода GetNextPacket, который возвращает число переданных в пакете записей.

Для дополнительной организации процедуры подкачки данных можно использовать методы, обрабатывающие события BeforeGetRecords и AfterGetRecords.

Для ограничения количества записей, передаваемых в одном пакете, используется целочисленное свойство PacketRecords, в котором можно указать количество записей, передаваемых в одном пакете. Если свойство имеет значение –1, то в набор данных передаются все записи. Если свойство имеет нулевое значение, то в набор данных передаются только метаданные. Положительные значения явно задают количество записей, помещаемых в пакет.

Метод Post помещает измененные или добавленные данные в кэш. Для сохранения их в базе данных следует использовать метод ApplyUpdates, пересылающий модифицированные, добавленные или удаленные данные на сервер для записи их в базу. Параметр MaxErrors определяет максимально допустимое число ошибок, при превышении которого обновление данных будет остановлено и транзакция будет прервана. Если параметру присвоить значение –1, сервер не обратит внимания на ошибки.

Метод CancelUpdates позволяет отменить все изменения, содержащиеся в локальном кэше. При этом данные, измененные пользователем или приложением, не будут сохранены в базе данных.

Метод Reconcile удаляет из кэша те записи, которые были сохранены на сервере. В параметре Results передается вариантный массив, возвращаемый методом ApplyUpdates. Событие OnReconcileError возникает в том случае, когда методом ApplyUpdates были сохранены не все записи и требуется их согласование с конфликтными значениями. Параметр E содержит информацию об ошибке. Параметр UpdateKind содержит тип действия, которое вызвало ошибку. Это может быть обновление, удаление или добавление записи. В параметре Action указывается тип действия, которое можно применить, чтобы исправить конфликт-

ное значение и вновь попытаться сохранить данные. В листинге 3.17 приведен пример обработки ошибки с помощью вызова метода HandleReconcileError.

Листинг 3.17. Пример использования метода-обработчика OnReconcileError

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Функция HandleReconcileError генерирует диалоговое окно, в котором указывается конфликтное значение, исходное значение поля и действия, которые можно применить.

Компонент TSQLStoredProc

Компонент TSQLStoredProc используется для выполнения хранимых процедур на сервере и представления результатов их работы. Как и для других компонентов, соединение с базой данных указывается в свойстве SQLConnection.

После соединения с базой данных можно выбрать выполняемую процедуру в свойстве StoredProcName. При разработке в инспекторе объектов для этого свойства отображается список хранимых процедур, расположенных на сервере. После выбора хранимой процедуры компонент TSQLStoredProc запрашивает список ее параметров и использует полученную информацию для инициализации свойства Params. Некоторые СУБД не позволяют вернуть информацию о параметрах хранимой процедуры. В этом случае параметры необходимо определять самостоятельно.

В свойстве Params указываются параметры хранимой процедуры. С его помощью хранимой процедуре передаются аргументы и обрабатываются результаты выполнения процедуры.

Если хранимая процедура возвращает несколько наборов данных, то для получения доступа к ним используется метод NextRecordSet. В листинге 3.18 приведен пример обращения к наборам данных.

Листинг 3.18. Обращение к нескольким наборам данных, возвращенных хранимой процедурой

```
var
  TempDataSet: TCustomSQLDataSet;
  nRows: Integer;
begin
  TempDataSet := SQLStoredProc1; // start with 1st record set
  while TempDataSet <> nil do
  begin
    TempDataSet.First;
    while not TempDataSet.Eof do
```

```
    begin
      // process each record
      TempDataSet.Next;
    end;
    if TempDataSet <> SQLStoredProc1 then // don't free the original!
      TempDataSet.Free;
    TempDataSet := SQLStoredProc1.NextRecordSet; // get next set
  end;
end;
```

Если хранимая процедура не возвращает набор данных, следует использовать метод ExecProc. Для выполнения процедуры, возвращающей значения, используется метод Open.

Компонент TSQLMonitor

Компонент TSQLMonitor перехватывает сообщения, пересылаемые между компонентом TSQLConnection и СУБД, и сохраняет их в списке. Его очень удобно использовать для отслеживания взаимодействий приложения с сервером.

Соединение с компонентом TSQLConnection устанавливается при помощи свойства SQLConnection. В свойстве FileName указывается имя файла, в который сохраняется журнал сообщений.

Свойство TraceList содержит список сообщений, передаваемых от приложения к серверу и обратно.

В свойстве AutoSave можно указать режим автоматического сохранения сообщений в файл из списка TraceList, иначе данную возможность придется реализовывать вручную.

В свойстве TraceCount указывается число сообщений, зарегистрированных на данный момент. А целочисленное свойство MaxTraceCount позволяет указать максимальное число сообщений, которое будет зарегистрировано. Если свойство имеет значение –1, то ограничения на число сообщений нет. Если указано нулевое значение, то TSQLMonitor не ведет журнал сообщений.

Метод SaveToFile принудительно сохраняет содержимое свойства TraceList в файл. А метод LoadFromFile позволяет загрузить в свойство TraceList информацию из файла.

Событие OnTrace вызывается в момент, когда сообщение зафиксировано, но еще не сохранено в списке. Сразу после того как сообщение было добавлено в список, инициируется событие OnLogTrace. Метод-обработчик этого события можно использовать для того, чтобы периодически сохранять содержимое списка в файл.

Пример работы с Interbase

В этом разделе будет приведен пример работы с сервером InterBase. Можно использовать ранее настроенное соединение с СУБД InterBase, которое было названо TestConnection.

Необходимо создать новое приложение и добавить модуль данных с именем DataMod. В модуле нужно разместить компоненты TSQLConnection, TSQLQuery, TSQLStoredProc, TSQLMonitor и TDataSource. Для установки соединения в свойстве ConnectionName компонента TSQLConnection нужно выбрать значение TestConnection. Свойство LoginPrompt должно получить значение False. Компоненты TSQLQuery, TSQLStoredProc и TSQLMonitor нужно связать с компонентом TSQLConnection при помощи свойства Connection.

В свойстве StoredProcName компонента TSQLStoredProc следует указать имя хранимой процедуры MAIL_LABEL. Эта процедура возвращает несколько значений из таблицы Customer, принимая в качестве аргумента номер покупателя. В свойство Params компонента будет автоматически загружен список параметров.

Потребуется также указать текст SQL-запроса. Для этого в свойство SQL компонента TSQLQuery нужно ввести строку SELECT * FROM Customer и активировать компонент. После этого можно связать компонент TDataSource с компонентом TQuery.

Теперь можно перейти к компоненту TSQLMonitor. В свойстве FileName нужно ввести имя файла Log.txt. В нем будет храниться журнал соединений с сервером. Свойство AutoSave должно получить значение True. После этого можно активировать компонент.

Настало время создания главной формы. На ней надо разместить компоненты TEdit, TMemo, TStringGrid и две кнопки. В компонент TStringGrid путем последовательного перемещения по набору данных, полученного от компонента TSQLQuery, будут загружены записи таблицы Customer. Так как TSQLQuery может возвращать только однонаправленные наборы, нельзя использовать компонент TDBGrid.

В компоненте TEdit указывается номер покупателя, который передается в качестве входного параметра хранимой процедуры компоненту TSQLStoredProc. При нажатии на кнопку Выполнить хранимой процедуре передается параметр, содержащийся в компоненте TEdit, а результаты, возвращаемые процедурой, отображаются в компоненте TMemo. Окно приложения показано на рис. 3.34.

Рис. 3.34. Работа с хранимой процедурой

В листинге 3.19 приведен код этого приложения.

Листинг 3.19. Работа с InterBase

```
procedure TForm1.ExcuteBtnClick(Sender: TObject);
var I : Integer;
begin
  with Memo1, ConnectionModule.SQLStoredProc1 do begin
    Lines.Clear;
    ParamByName('CUST_NO').AsInteger:=StrToInt(Edit1.Text);
    ExecProc;
    for I:=1 to Params.Count -1 do begin
      Lines.Add(Params[i].AsString)
    end;
  end;
end;

procedure TForm1.FillBtnClick(Sender: TObject);
var I, J : Integer;
begin
  {Заполняем StringGrid значениями, полученными компонентом Query}
  {Создаем шапку таблицы}
  TableGrid.ColCount:=ConnectionModule.SQLQuery1.FieldCount;
  for I:=0 to TableGrid.ColCount -1 do begin
    TableGrid.Cells[I,0]:=ConnectionModule.SQLQuery1.Fields[I].DisplayName;
  end;
  {Заполняем таблицу значениями}
  while not ConnectionModule.SQLQuery1.Eof do
  begin
    J:=J+1;
    TableGrid.RowCount:=TableGrid.RowCount+1;
    for I:=0 to TableGrid.ColCount -1 do begin
      TableGrid.Cells[I,J]:=ConnectionModule.SQLQuery1.Fields[I].AsString;
    end;
    ConnectionModule.SQLQuery1.Next;
  end;
end;

procedure TForm1.TableGridSelectCell(Sender: TObject; ACol,
  ARow: Integer; var CanSelect: Boolean);
```

продолжение

Листинг 3.19 *(продолжение)*

```
begin
  Edit1.Text:=TableGrid.Cells[0,ARow];
end;
end.
```

Следует обратить внимание на записываемые журнал, в которых фиксируется работа пользователя.

Пример работы с SQL Server 2000

В этом разделе будет приведен пример приложения, взаимодействующего с SQL Server 2000. Как обычно, следует создать новый проект и добавить в него модуль данных. В модуле данных потребуется установить компонент TSQLConnection. Двойной щелчок мышью на нем активирует окно редактора соединений.

Нужно создать новое соединение, указать драйвер MSSQL и дать ему имя ConnectToNorthwind. Теперь необходимо настроить соединение. В свойстве HostName потребуется указать сетевое имя компьютера, которое можно получить после выполнения команды System Properties ▸ Computer Name ▸ Full Computer Name. Диалоговое окно System Properties можно также вызвать через контекстное меню My Computer ▸ Properties.

В поле DataBase нужно выбрать базу данных Northwind. В поле User_Name необходимо ввести dbo, а в поле Password —набор символов, создающий пароль. В поле MSSQL TransIsolation нужно выбрать из списка значение DirtyRead. В поле OS Authentication можно выбрать значение True. В этом случае SQL Server 2000 будет использовать средства аутентификации операционной системы, не используя собственных методов. Именно поэтому в качестве пароля для учетной записи dbo был использован ничего не значащий набор символов.

Также потребуется один компонент TSimpleDataSet, который нужно связать с TSQLConnection. Свойство DataSet компонента TSimpleDataSet предоставляет доступ к внутреннему набору данных, получающему данные от сервера. В свойстве CommandType нужно выбрать значение ctTable. А в свойстве CommandText указать таблицу Customers. Осталось лишь активировать набор данных, присвоив свойству Active значение True. После этого можно активировать компонент TSimpleDataSet.

В модуле данных нужно разместить и настроить компонент TSQLMonitor. Также потребуется компонент TDataSource, который необходимо связать с компонентом TSimpleDataSet при помощи свойства DataSet.

На главной форме следует расположить компонент TDBGrid и пять кнопок. После связывания модуля данных с главной формой приложения можно связать TDBGrid с компонентом TDataSource. Кнопки Принять (PostBtn), Удалить (DeleteBtn), Отменить (CancelBtn), Сохранить (SaveBtn) и Обновить (RefreshBtn) обрабатываются кодом, приведенным в листинге 3.20. При нажатии на кнопку Принять изменения вносятся в кэш и могут быть отменены нажатием на

кнопку Удалить. На рис. 3.35 показан внешний вид основного окна программы.

Рис. 3.35. Пример работы с компонентом TSimpleDataSet

Листинг 3.20. Код главной формы примера работы с компонентом TSimpleDataSet

```
procedure TMainForm.PostBtnClick(Sender: TObject);
begin
  with ConnectionModule.SimpleDataSet1 do begin
    if State in [dsInsert, dsEdit] then
      Post;
  end;
end;

procedure TMainForm.DeleteBtnClick(Sender: TObject);
begin
  with ConnectionModule.SimpleDataSet1 do begin
    if State = dsBrowse then
      if MessageDlg('Вы уверены в том, что хотите удалить запись?',
mtConfirmation, [mbYes,mbNo], 0) = mrYes then
        Delete;
  end;
end;

procedure TMainForm.SaveBtnClick(Sender: TObject);
```

продолжение

Листинг 3.20 *(продолжение)*

```
begin
  with ConnectionModule.SimpleDataSet1 do begin
    if (ChangeCount > 0) then
      ApplyUpdates(-1);
  end;
end;

procedure TMainForm.CancelBtnClick(Sender: TObject);
begin
  ConnectionModule.SimpleDataSet1.CancelUpdates;
end;

procedure TMainForm.RefreshBtnClick(Sender: TObject);
begin
  ConnectionModule.SimpleDataSet1.Refresh;
end;
```

Модуль данных содержит код обработчика ошибок, возникающих при сохранении данных на сервере вызовом метода `ApplyUpdates`. Код приведен в листинге 3.21.

Листинг 3.21. Код обработчика ошибок

```
procedure TConnectionModule.SimpleDataSet1ReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action:= HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Можно попробовать запустить несколько копий приложения и посмотреть, как реализовано изменение одной записи несколькими пользователями.

УРОК 4 Основы технологии COM

Технология COM (Component Object Model – модель многокомпонентных объектов) представляет собой платформенно-независимую распределенную объектно-ориентированную систему, обеспечивающую взаимодействие между своими компонентами. Технология COM является основой для технологий OLE, ActiveX и многих других.

COM не определяет структуру приложения. Язык программирования, структура и реализация деталей оставлены на усмотрение программиста. Скорее, COM предоставляет разработчику объектную модель и код, необходимые для получения доступа к данному объекту и взаимодействия с другими объектами. Объекты могут быть расположены «внутри» данного процесса, в другом процессе или на удаленной машине. Для того чтобы иметь возможность работать с COM-объектами, язык программирования должен поддерживать структуру указателей и также, явно или неявно, вызов функций через указатели.

В общем случае объект приложения представляет собой совокупность данных и функций, манипулирующих ими. Объект COM взаимодействует с данными при помощи зависимых функций. Эти функции называются интерфейсами, а функции, вызываемые интерфейсами, называют методами. Кроме того, COM определяет, что получить доступ к методам интерфейса можно только через указатель на интерфейс. С другой стороны, устанавливая стандарт простых двоичных объектов, COM определяет универсальные интерфейсы, предоставляющие функции для работы со всеми COM-технологиями.

Базовые понятия

В технологии COM приложение предоставляет для использования свои службы, применяя для этого COM-объекты. Приложение может использовать один или несколько объектов. Любой объект COM может иметь один или несколько интерфейсов. Интерфейс предоставляет приложению методы, позволяющие получить доступ к данным и работать с ними. Как правило, в интерфейсы

объединяются методы, предназначенные для выполнения операций одного типа.

Клиентское приложение может обращаться к функциям объекта только через интерфейсы и их методы. Приложению достаточно знать несколько базовых методов для того, чтобы получить информацию о совокупности свойств, интерфейсов и методов данного объекта.

Объект работает в составе *сервера COM*. Сервер COM может быть исполняемым файлом либо динамической библиотекой. В любом случае, он обязательно предоставляет приложению описание объектов, входящих в его состав. Для доступа к методам объекта клиент должен получить указатели на соответствующие интерфейсы. Используя методы объекта, клиент может вызывать его службы и обращаться к его свойствам. Взаимодействие между объектом и клиентом обеспечивается средствами COM и скрыто от клиента. На рис. 4.1 представлена схема, изображающая сказанное. Объект COM может находиться в другом процессе, библиотеке или даже на другой машине. Клиентское приложение будет работать с ним точно так же, как если бы он находился в адресном пространстве приложения. Механизм, позволяющий клиенту вызывать функции «удаленных» объектов, называется маршалингом.

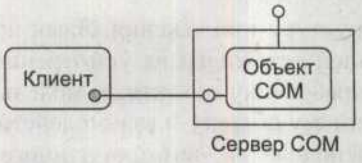


Рис. 4.1. Схема взаимодействия клиента и объекта

Любой объект COM является обычным экземпляром некоего класса, документирующего свои свойства и методы. Информация обо всех зарегистрированных и доступных в данной операционной системе классах COM собрана в специальной библиотеке COM, которая используется для инициализации экземпляра класса. Сначала клиент обращается к библиотеке COM, передавая ей имена требуемого класса и необходимого интерфейса. Библиотека находит нужный класс и сначала запускает сервер, который затем создает объект — экземпляр класса. После этого библиотека возвращает клиенту указатели на объект и интерфейс. В последующей работе клиент может обращаться непосредственно к объекту.

За создание объекта отвечает специальный объект COM, называемый *фабрикой классов*. После создания объекта наступает фаза его инициализации, в ходе которой считываются настройки из системного реестра и загружаются необходимые стартовые данные. За эти процессы отвечают специальные объекты COM, называемые моникерами. С любым объектом COM поставляется *библиотека типов*, в которой описываются его свойства, методы и интерфейсы. Данная библиотека создается с помощью специального языка описания интерфейса Interface Definition Language (IDL).

Объект

Объект COM является обычным объектом со специфичной реализацией ряда свойств. Физически он реализован в виде исполняемого кода, который выполняет какую-либо функцию и имеет один и более интерфейс. Каждый объект COM реализует экземпляр соответствующего класса. Объект может иметь любое число интерфейсов, и каждый интерфейс имеет свой собственный указатель. Основное отличие COM-объекта от объекта ООП заключается в том, что он наследует не реализацию интерфейсов, а только их объявления. Если какому-либо объекту нужен наследуемый метод другого объекта, то он вызывает его — данный механизм называется включением.

Каждый объект COM имеет свой уникальный идентификационный номер. К примеру, какой-либо объект «знает» о том, что другой объект имеет интерфейс с необходимым методом. Если сторонний разработчик переопределил данный метод, то информация, которую имеет первый объект, будет неверной. Следовательно, возникнет ошибка со всеми вытекающими последствиями. Механизм агрегирования позволяет использовать интерфейсы других объектов, передавая вызывающему объекту указатели на них.

Интерфейс

Интерфейс является связующим звеном между объектом COM и клиентским приложением. Через интерфейс приложение может корректно обратиться к объекту и получить от него данные. Интерфейс является группой логически и семантически связанных методов, обеспечивающих взаимодействие между сервисом сервера COM и его клиентом. Любой интерфейс однозначно идентифицируется глобальным уникальным идентификатором *GUID* (Globally Unique Identifier). Этот идентификатор является 128-битным номером, вероятность повторения которого стремится к нулю. Данный номер для интерфейсов называется *IID* (Interface Identifiers). Также интерфейс имеет имя, перед которым ставится символ «I». Идентификатор интерфейса позволяет избежать конфликта имен между разными версиями данного интерфейса или интерфейсами других объектов.

Все объекты COM поддерживают базовый интерфейс `IUnknown`. Данный интерфейс имеет метод `QueryInterface`, используя который, клиентское приложение может получить информацию обо всех интерфейсах объекта. Вместе с объектом COM поставляется библиотека типов, в которой на языке IDL описываются методы интерфейса. Так как объекты COM хранятся в двоичном формате (скомпилированном виде), то это обеспечивает их независимость от конкретного языка программирования, что и делает эту технологию универсальной.

Интерфейс имеет специальную *виртуальную таблицу* (VTable), которая содержит массив указателей на все его методы. Клиентское приложение обращается к определенному интерфейсу через его указатель, затем данный интерфейс получает указатель на необходимый метод, используя виртуальную таблицу. На рис. 4.2 приведена соответствующая схема.

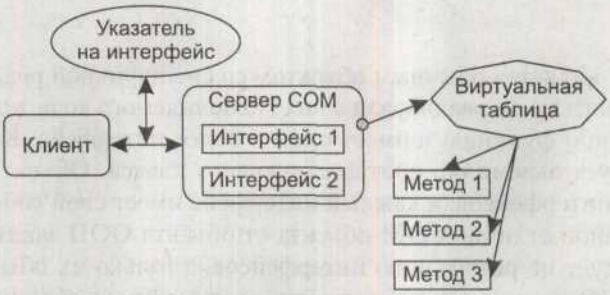


Рис. 4.2. Структура интерфейса

Когда указатель на нужный метод получен, можно выполнять соответствующий метод. Таким образом, обращение происходит по цепочке. Виртуальная таблица представляет собой механизм, сходный с таблицами виртуальных функций в Delphi.

Интерфейс IUnknown

Интерфейс `IUnknown` является базовым интерфейсом для всех объектов. Он управляет механизмом учета ссылок и позволяет клиентам получать указатели на другие интерфейсы данного объекта, используя метод `QueryInterface`. Все интерфейсы, прямо или косвенно, берут свое начало от этого интерфейса. В виртуальной таблице VTable три метода интерфейса `IUnknown` расположены первыми, а за ними следуют собственные методы интерфейса.

Метод `QueryInterface` возвращает указатель на интерфейс объекта, идентификатор IID которого указывает на запрашиваемый интерфейс. Если объект не имеет интерфейса с данным IID, то метод возвращает значение `NULL`.

Интерфейс `IUnknown` реализует механизм учета ссылок с помощью методов `AddRef` и `Release`. Объект должен существовать до тех пор, пока с ним работает хотя бы один клиент. Объект имеет специальный счетчик ссылок, который увеличивается на единицу при подключении нового клиента и уменьшается на единицу при завершении его работы. Клиент не может уничтожить объект, так как с ним могут работать другие клиенты.

Увеличение счетчика ссылок выполняется методом `AddRef`. Метод возвращает увеличенное на единицу количество ссылок на объект. Данный метод должен вызываться, когда указатель на интерфейс передается какому-либо объекту. Метод `Release` уменьшает счетчик ссылок на объект на единицу. Когда значение счетчика достигает нуля, объект завершает свою работу и выгружается из памяти. Данный метод должен обязательно вызываться клиентом, использовавшим данный интерфейс.

Сервер COM

Сервер COM является библиотекой или приложением, предоставляющим сервисы клиентскому приложению или библиотеке. Сервер COM может со-

стоять из одного и более объектов COM. Объекты COM выступают в роли наборов свойств и методов. Клиенту не обязательно знать, где расположен объект COM, так как технология предоставляет прозрачный доступ независимо от расположения объекта.

Различают три типа серверов:

- Внутренний сервер (In-process server) является библиотекой DLL, подключаемой к клиентскому приложению и работающей с ним в одном адресном пространстве. В качестве примера можно привести элемент управления ActiveX, отображаемый на веб-странице. Элемент управления запускается внутри некоторого процесса браузера. Схема показана на рис. 4.3.

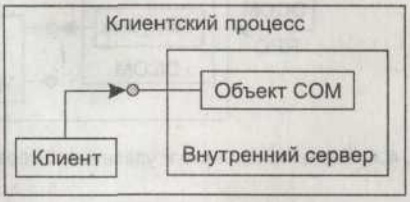


Рис. 4.3. Внутренний сервер

- Локальный сервер (Local server) создается отдельным процессом, работающим на той же машине, что и клиентское приложение.
- Удаленный сервер (Remote server) является динамической библиотекой или приложением, которое выполняется на другой машине. Удаленный сервер использует технологию Distributed COM (DCOM) для предоставления доступа к интерфейсам.

Как показано рис. 4.4, если объект COM расположен вне текущего процесса на той же машине, что и клиент, или на удаленной, то получаемый клиентом указатель интерфейса ссылается на специальный *proxy-объект* COM, который функционирует внутри клиентского процесса. Прокси предоставляет клиенту те же интерфейсы, что и вызываемый объект COM на локальном или удаленном сервере. Получив вызов от клиента, прокси упаковывает его параметры и с помощью специальных служб операционной системы передает его серверу. На стороне сервера расположен специальный объект COM — *заглушка* (*Stub*). Он распаковывает вызов и передает его требуемому объекту COM. Результаты возвращаются тем же путем, но в обратном порядке.

Удаленный сервер функционирует так же, как и локальный, за исключением того, что передача вызовов между клиентом и сервером осуществляется средствами DCOM при помощи механизма *вызова удаленных процедур* (Remote Procedure Calls).

Механизм, позволяющий клиенту получать доступ к объектам, расположенным в другом адресном пространстве или на другой машине, называется *маршалингом* (Marshaling). Маршалинг должен выполнять две основные функции:

- принимать указатель на интерфейс из процесса сервера и делать доступным указатель на прокси в процессе клиента;
- передавать аргументы вызовов интерфейса таким образом, как будто они пришли от клиента, и размещать аргументы в процесс удаленного объекта.

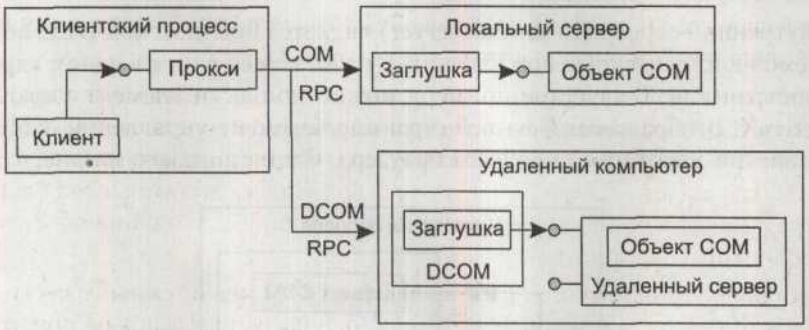


Рис. 4.4. Схема локального и удаленного серверов

Маршалинг обеспечивает процесс упаковки параметров запроса, а демаршалинг, соответственно, их распаковку. Может использоваться стандартный тип маршалинга, реализуемый сервером COM, либо альтернативный, в случае использования которого для каждого объекта динамически настраиваются параметры прокси и заглушки.

Фабрика класса

Объекты COM представляют собой экземпляры класса `CoClass`. Объект `CoClass` всегда содержит один или несколько интерфейсов. Для создания экземпляра класса используется специальный объект — *фабрика класса* (Class factory). С помощью фабрики класса можно создать один объект или несколько его экземпляров. Фабрика класса является специальным COM-объектом, который поддерживает интерфейс `IClassFactory` и отвечает за создание экземпляров того класса, с которым связана данная фабрика.

В листинге 4.1. приведено объявление интерфейса `IClassFactory` в модуле `ActiveX.pas`.

Листинг 4.1. Объявление интерфейса IClassFactory

```
IClassFactory = interface(IUnknown)
    ['{00000001-0000-0000-C000-000000000046}']
    function CreateInstance(const unkOuter: IUnknown; const iid: TIID;
      out obj): HResult; stdcall;
    function LockServer(fLock: BOOL): HResult; stdcall;
  end;
```

Как видно из кода, приведенного в листинге 4.1, интерфейс имеет два метода. Метод `CreateInstance` создает новый экземпляр объекта. Часто передается толь-

ко идентификатор объекта, на основе которого фабрика класса заполняет остальные параметры. Метод `LockServer` вызывается клиентским приложением для того, чтобы указать, что данный сервер COM должен храниться в памяти. При каждом вызове сервера параметру `fLock` присваивается значение `True`, тем самым увеличивая счетчик ссылок. После завершения использования объекта свойству `fLock` следует присвоить значение `False`. Когда счетчик достигнет нуля, сервер будет выгружен из памяти.

Для явного вызова фабрики класса можно использовать функцию `CoGetClassObject`. Данной функции в качестве параметров передаются CLSID нужного класса и IID интерфейса `IClassFactory`. Функция ищет требуемую фабрику и возвращает указатель на интерфейс. Далее, вызвав метод `CoCreateInstance`, клиентское приложение инициирует процесс создания объекта данной фабрикой класса.

При установке приложения, работающего с COM, в системный реестр записывается информация обо всех объектах, используемых данным приложением и содержащихся в нем. На рис. 4.5 приведено изображение редактора реестра, в окне которого выбран объект COM, и указаны его свойства.

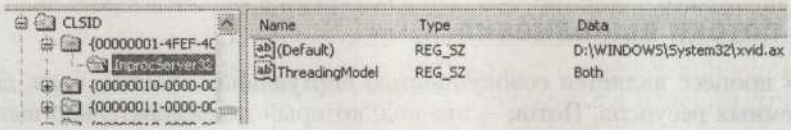


Рис. 4.5. Информация об объекте COM

В системном реестре хранится информация об объекте:

- идентификатор класса, который однозначно определяет класс объекта;
- тип сервера — внутренний, локальный или удаленный;
- имя динамической библиотеки или исполняемого файла, а также путь к ним для локальных и внутренних серверов;
- сетевой адрес и путь для удаленных серверов;
- тип потока выполнения (Threading Model).

Используя сведения, содержащиеся в реестре, приложения могут обращаться к объектам COM через CSLID и создавать экземпляры объектов, используя информацию о параметрах, содержащуюся в реестре.

Библиотека типов

В процессе разработки может возникнуть необходимость получить информацию о том, какие интерфейсы имеет объект, какие методы имеют его интерфейсы, информацию об их входных и выходных параметрах. Эта информация содержится в библиотеке типов, которая выполнена в виде специального файла, содержащего информацию об объекте COM. Вся информация описы-

вается в строго определенном формате на языке IDL (Interface Definition Language). Библиотека может содержать информацию о свойствах и методах, а также сведения об используемых заглушках и заместителях.

Для создания библиотеки типов, описываемой с помощью языка IDL, используются специальные компиляторы. Доступ к библиотеке осуществляется по идентификатору CLSID класса. Кроме того, библиотека имеет свой собственный GUID, который хранится в системном реестре при регистрации объекта, для которого создана библиотека.

Каждая библиотека типов имеет интерфейс `ITypeLib`, который дает возможность работать с ней как с единым объектом. Для доступа к информации об отдельном интерфейсе используется интерфейс `ITypeInfo`. Если необходимо получить доступ к библиотеке по GUID, то используется функция `LoadRegTypeLib`. А если клиентскому приложению известно имя библиотеки, то можно воспользоваться функцией `LoadTypeLib`.

При создании серверов COM с помощью мастеров Delphi библиотеки типов и соответствующий им код генерируются автоматически, избавляя от необходимости писать его вручную.

COM и потоки выполнения

Любой процесс является совокупностью виртуальной памяти, кода, данных и системных ресурсов. Поток — это код, который последовательно выполняется внутри процесса. Любое приложение Windows имеет как минимум один поток, который выполняется процессором. Этот поток называется главным. Процесс может иметь дополнительные потоки, которые могут выполняться на параллельных процессорах и реализовывать какие-либо действия.

Взаимодействие между потоками осуществляется через службу RPC при помощи передачи сообщений. Потоки могут взаимодействовать друг с другом, выполняясь в разных процессах на одной машине либо на удаленных машинах. В любом случае обмен информацией будет осуществляться через RPC.

Поток выполняется до тех пор, пока не будет завершен либо пока не будет прерван потоком с более высоким приоритетом (действие пользователя или планировщик выполнения потоков ОС). Поток может выполняться в отдельном адресном пространстве либо разделять область памяти с другими потоками. Потоки, выполняемые в некотором адресном пространстве, обслуживаются разделяемыми стеками.

Каждый поток в процессе может работать с глобальными переменными и ресурсами. Планировщик потоков определяет частоту использования конкретного потока, опираясь на комбинацию приоритетов процесса и данного потока. Разработчик может вручную настроить приоритет потока, вызвав функцию `SetPriorityClass` для установки приоритета процесса и функцию `SetThreadPriority` для установки приоритета потока.

Клиент и сервер COM могут выполняться в разных процессах или потоках, и к серверу одновременно может обращаться несколько клиентов. В COM

изящно решен данный вопрос. Все объекты COM процесса помещаются в специальные группы, называемые апартаментами (Apartments). Объект COM существует только внутри одного апартамента, и его методы могут быть вызваны только тем потоком, которые принадлежат этому апартаменту. Другие потоки могут вызывать методы данного объекта COM только через прокси.

Апартаменты бывают однопоточные (Single Threaded Apartments), многопоточные (Multiple Threaded Apartments) и нейтральные (Neutral Apartment). Их стоит рассмотреть подробнее:

- Однопоточный апартамент (Single Threaded Apartments) состоит из одного потока. Организуется очередь вызовов методов, и каждый из них обрабатывается только после того, как будут обработаны предшествующие вызовы. Внутри апартамента все объекты взаимодействуют напрямую. Использование апартамента данного типа является предпочтительным для реализации однопоточных серверов COM.
- В случае использования многопоточного апартамента (Multiple Threaded Apartments) может быть создано множество потоков и объектов, причем каждый поток не привязывается к какому-либо потоку и любой метод объекта может быть вызван в любом из потоков. Сервер COM через службу RPC создает пул свободных потоков, к которым могут подключаться клиентские приложения. При вызове со стороны клиента многопоточный апартамент выбирает свободный поток из пула и предоставляет его в пользование клиенту. Когда клиент завершает свою работу, апартамент вновь помещает поток в пул. В этом случае не требуется затрачивать ресурсы на то, чтобы заново создать поток. Указатели на интерфейсы внутри апартамента передаются напрямую. Сервер COM, использующий данный апартамент, способен обеспечить более высокое быстродействие по сравнению с однопоточным (особенно на многопроцессорных системах), но его реализация является гораздо более трудоемкой.
- Нейтральные апартаменты (Neutral Apartments) используются при создании серверов COM+. Методы объектов COM вызываются в потоке, из которого к ним обратился клиент. Использование данного типа апартамента позволяет снизить количество переключений потоков.

Реализация COM в Delphi

Объект COM описывается обычным классом `TCOMObject`, который порожден непосредственно от класса `TObject`. Все свойства и методы объекта описываются в объявлении класса. При создании объекта COM с ним связывается вспомогательный класс `CoClass`, который описывает все его интерфейсы. При создании объекта к имени его класса добавляется приставка `Co` и создается `CoClass` этого объекта.

Описание `CoClass` содержится в библиотеке типов. Стандартное объявление класса обеспечивает создание кода объекта, который будет скомпилирован

в двоичный код. CoClass обеспечивает представление экземпляра класса в соответствии со спецификой COM и гарантирует корректное обращение клиента к объекту.

Класс TComObject

Класс TComObject является базовым классом, на основе которого создаются простые классы COM, такие как, например, расширения оболочек. Класс TComObject является COM-объектом, который поддерживает интерфейсы IUnknown и ISupportErrorInfo. Класс реализует простые объекты, обладающие базовым списком возможностей:

- Объект обладает идентификатором класса CLSID, который используется для создания экземпляров класса с использованием фабрики класса.
- Объект поддерживает агрегирование при помощи методов интерфейса IUnknown.
- Объект поддерживает безопасные вызовы и обработку исключительных ситуаций OLE, используя интерфейс IProvideErrorInfo.
- Объект использует в работе интерфейс IErrorInfo.

Класс TComObject может использоваться как базовый класс для создания классов объектов COM, которые должны иметь идентификатор класса (CLSID). Идентификатор CLSID, как было сказано ранее, используется для регистрации класса в реестре и для создания его экземпляра извне при помощи вызова фабрики класса.

Объект COM, как и любой другой объект, создается конструктором Create. Конструктор создает объект как самостоятельный экземпляр класса, не входящий в агрегат. Под *агрегатом* следует понимать совокупность объектов COM, предоставляющих свои интерфейсы и имеющих один общий — IUnknown.

Для создания объекта COM и включения его в агрегат используется метод CreateAggregated, который создает новый объект как часть агрегата. В параметре Controller указывается общий управляющий интерфейс IUnknown и передается свойству Controller.

Метод CreateFromFactory используется для создания объекта и его инициализации. В свойстве Factory указывается фабрика класса объекта. После создания объекта метод Initialize позволяет произвести его инициализацию. В ходе инициализации счетчик ссылок на объект увеличивается на единицу.

В свойстве RefCount содержится число ссылок на объект. Свойство RefCount определяет, когда созданный объект может быть уничтожен. Когда свойству присваивается нулевое значение, объект уничтожается, так как с ним не работает ни один клиент. Значение свойства увеличивается методом AddRef интерфейса IUnknown и реализуется методом ObjAddRef. Уменьшение значения свойства производится при помощи метода ObjRelease, реализуя метод Release интерфейса IUnknown.

Метод ObjQueryInterface позволяет выяснить, имеет ли данный объект COM интерфейс с идентификатором, заданным IID. Данный метод является реализацией метода QueryInterface интерфейса IUnknown.

Класс TTypedComObject

Класс TTypedComObject является прямым наследником класса TComObject. Он предназначен для создания объектов COM с использованием библиотеки типов. Класс TTypedComObject имеет интерфейс IProvideClassInfo, в состав которого входит единственный метод, предназначенный для получения указателя на CoClass объекта.

Метод GetClassInfo является реализацией метода GetClassInfo интерфейса IProvideClassInfo.

Интерфейс IUnknown

Интерфейс IUnknown является базовым для всех интерфейсов. Он является прямым потомком класса Interface, на основе которого в Delphi строятся все интерфейсы.

В состав интерфейса входят методы AddRef, Release и QueryInterface, которые рассматривались ранее.

В коде Delphi интерфейс IUnknown подменяет собой класс Interface.

Класс TComObjectFactory

Класс TComObjectFactory является фабрикой класса для объектов COM, порожденных от класса TComObject. Класс TComObjectFactory обеспечивает функционирование интерфейсов IUnknown, IClassFactory и IClassFactory2. Интерфейс IClassFactory создает объект, принимая в качестве параметра его идентификатор CLSID. Интерфейс IClassFactory2 используется для обеспечения лицензирования объекта COM. Создание объекта фабрики класса может быть инициировано извне при помощи функции API CoCreateClassObject. Для создания объекта также могут быть использованы функции CreateComObject и CreateOleObject. Если создается несколько объектов одного класса, эффективнее вызывать фабрику класса, получая указатель на его интерфейс IClassFactory и используя его собственные методы. Для управления фабриками классов на сервере COM используется специальный класс TComClassManager, доступ к которому можно получить, используя функцию ComClassManager.

Метод CreateComObject инициирует вызов конструктора класса TComObjectFactory. Конструктор Create создает фабрику класса во время запуска сервера. Конструктор фабрики класса описывается в секции инициализации модуля, включающего сервер COM. В параметре ComServer указывается сервер COM, в составе которого будет функционировать объект. В параметре ComClass указывается тип класса, который используется методом GetFactoryFromClass менед-

жера классов для идентификации фабрики. Параметр ClassID задает идентификатор класса, создаваемого этой фабрикой, а параметр Instancing задает способ создания объекта. Этот способ регламентируется одним из перечисленных ниже значений:

- Значение ciInternal указывает, что объект создается в процессе как сервер COM. Другие приложения не могут создать экземпляр объекта напрямую, но могут использовать методы приложения для создания экземпляра документа.
- Значение ciSingleInstance указывает, что для каждого клиента, обратившегося к серверу COM, создается собственный экземпляр объекта. Все объекты создаются в единственном экземпляре сервера.
- Значение ciMultiInstance указывает, что для каждого клиента создается собственный экземпляр сервера, в котором содержится экземпляр объекта.

В параметре ThreadingModel задается способ взаимодействия сервера и клиента. Значения параметра перечислены ниже:

- Значение tmSingle указывает, что сервер последовательно выполняет запросы клиентов.
- Значение tmApartment свидетельствует, что вызов объекта осуществляется только в том потоке, в котором он создан. Различные объекты некоторого сервера могут быть вызваны в различных потоках, при этом один объект может обслуживать одновременно только одного клиента.
- Значение tmFree указывает, что объект может одновременно использоваться произвольным числом клиентов.
- Значение tmBoth указывает, что клиент может использовать модели tmApartment или tmFree.
- Значение tmNeutral указывает, что к данному объекту одновременно могут обращаться несколько клиентов в различных потоках. Но защиту от возможных конфликтов обязан обеспечить программист, используя критические секции, моникеры и иные соответствующие средства. Данная модель доступна только для технологии COM+. При использовании COM применяется модель tmApartment.

Метод RegisterClassObject регистрирует класс создаваемого объекта. Приложения, содержащие сервер COM, вызывают этот метод при запуске соответствующего сервиса. Метод UpdateRegistry вызывается для регистрирования объекта COM или удаления его регистрации. Регистрация объекта производится при первом запуске приложения. В свойстве ClassID указывается идентификатор класса, а свойство ClassName определяет имя класса объекта.

В свойстве ErrorIID содержится GUID интерфейса, в котором произошла ошибка. Свойство ShowErrors включает или отключает показ сообщений об ошибках при создании объекта. Если свойство имеет значение True, то сообщение о возникающих ошибках выводится в информационном окне.

Класс TTypedComObjectFactory

Класс TTypedComObjectFactory является прямым наследником класса TComObjectFactory. Он используется для создания объектов класса TypedComObject.

Свойство ClassInfo позволяет получить информацию о типе без необходимости применения загрузки библиотеки типов. Для получения информации об объекте используется интерфейс ITypeInfo.

Метод GetInterfaceTypeInfo возвращает информацию об объекте COM, созданном данной фабрикой.

Класс TComClassManager

Класс TComClassManager используется для управления фабриками классов. Экземпляр класса TComClassManager возвращается функцией ComClassManager, которая содержится в модуле ComObj.pas. Этот экземпляр управляет фабриками классов объектов, владельцем которых является данный сервер COM. Экземпляр класса получает от сервера COM список фабрик и обновляет его при создании нового объекта или уничтожении старого. Разработчику предоставляются методы, позволяющие управлять фабриками классов.

Функция ComClassManager возвращает ссылку на экземпляр класса. А метод ForEachFactory последовательно выполняет определенные действия над всеми фабриками классов данного сервера COM. В параметре ComServer указывается сервер COM, а в параметре FactoryProc — ссылка на метод, выполняющий необходимые действия.

Метод GetFactoryFromClass возвращает фабрику класса для класса, указанного в параметре ComClass. А метод GetFactoryFromClassID возвращает фабрику класса, принимая в качестве входного параметра идентификатор класса ClassID.

Класс TComServer

Класс TComServer предназначен для создания экземпляров серверов COM. При создании объекта COM в его модуль автоматически добавляется модуль ComServ, содержащий реализацию класса TComServer. Класс TComServer содержит всю необходимую информацию о сервере. В нем упоминаются библиотека типов, имя, файл справки и многие другие параметры. Также сервер содержит информацию о том, как он должен быть загружен и как он должен быть выгружен из адресного пространства памяти. Класс сервера применяется для создания экземпляров фабрик классов объектов, используя их идентификаторы CLSID.

Создается объект сервера COM методом Create. Метод Initialize вызывается при создании сервера. При первом запуске он производит регистрацию всех связанных объектов COM в системном реестре.

Режим запуска сервера определяется в свойстве StartMode. Его возможные значения перечислены далее:

- Значение smAutomation указывает, что сервер запускается как сервер автоматизации в ответ на запрос от контроллера автоматизации.
- Значение smRegServer указывает, что сервер запускается с целью регистрации в системном реестре (первый запуск) либо с использованием ключа /regserver.
- Значение smStandalone указывает, что сервер запускается пользователем как отдельное приложение.
- Значение smUnregServer указывает, что сервер запускается с целью удаления регистрации из системного реестра (используется ключ /unregserver).

Свойство IsInprocServer указывает, является ли данный сервер COM внутренним сервером или нет. Для внутреннего сервера используется значение True.

В свойстве ServerKey указывается, является ли данный сервер COM внутренним или локальным. Свойство содержит строку InprocServer32, если сервер является внутренним и оформлен в виде библиотеки DLL. Если сервер является локальным и существует в форме EXE-файла, то используется значение LocalServer32.

Для загрузки библиотеки типов следует вызвать метод LoadTypeLib. В случае возникновения ошибки будет возвращен объект класса исключения EOleSysError с информацией об ошибке.

Получить доступ к библиотеке типов, связанной с данным сервером, можно через свойство TypeLib, возвращающее интерфейс ITypeLib. Выяснить количество объектов, запущенных в данном сервере COM, можно при помощи свойства ObjectCount. Значение свойства увеличивается на единицу, когда новый объект создается фабрикой класса, и уменьшается, когда он уничтожается.

В свойстве ServerName содержится имя сервера. Задать его можно методом SetServerName. Если сервер использует библиотеку типов, то метод не будет выполнен, так как свойство получит значение автоматически из библиотеки типов.

В свойстве ServerFileName содержится полный путь к файлу сервера и его имя. А путь к соответствующему файлу справки содержится в свойстве HelpFileName.

Создание внутреннего сервера COM и работа с ним

Для создания серверов COM Delphi предоставляет широкий выбор мастеров, автоматизирующих выполнение трудоемких задач. Мастера доступны на вкладке ActiveX репозитория объектов. Перед созданием внутреннего сервера COM в виде DLL необходимо создать специальную библиотеку, которая имеет несколько функций и описывается по правилам COM.

Репозиторий объектов можно активировать при помощи команды меню File ▸ New ▸ Other. Для дальнейшей работы потребуется вкладка ActiveX. Нужно создать новый проект, выбрав значок ActiveX Library. Окно репозитория показано на рис. 4.6.

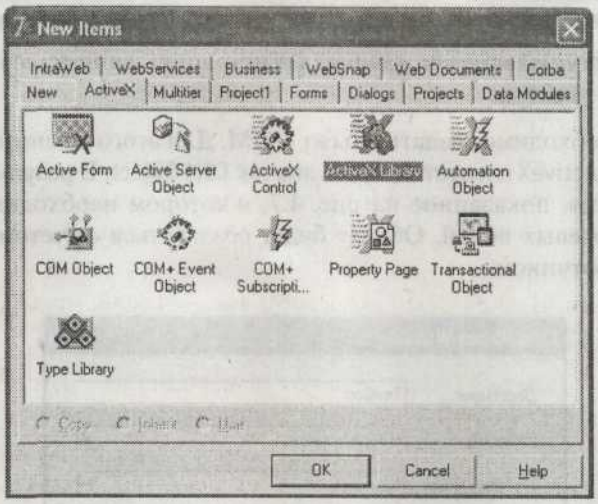


Рис. 4.6. Репозиторий объектов

Созданный проект нужно сохранить с именем TestServ. В листинге 4.2 приведен код созданной библиотеки.

Листинг 4.2. Код библиотеки TestServ

```
library TestServ;
uses
  ComServ;
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

Созданная библиотека экспортирует функции `DllGetClassObject`, `DllCanUnloadNow`, `DllRegisterServer` и `DllUnregisterServer`, необходимые для работы с COM-технологией. Эти функции объявляются в модуле `ComServ`:

- Метод `DllGetClassObject` возвращает фабрику класса для объекта COM.
- Метод `DllCanUnloadNow` производит проверку на предмет того, возможно ли выгрузить из памяти DLL-библиотеку данного сервера COM. Функция возвращает значение `S_OK`, если выгрузка возможна. В противном случае возвращается значение `S_FALSE`.

- Метод DllRegisterServer регистрирует сервер COM в системном реестре.
- Метод DllUnregisterServer удаляет регистрацию данного сервера COM из системного реестра.

После этого необходимо создать объект COM. Для этого в репозитории объектов на вкладке ActiveX следует выбрать значок COM Object. В результате появится диалоговое окно, показанное на рис. 4.7, в котором необходимо заполнить несколько ключевых полей. Объект будет создаваться с учетом данных, введенных разработчиком.

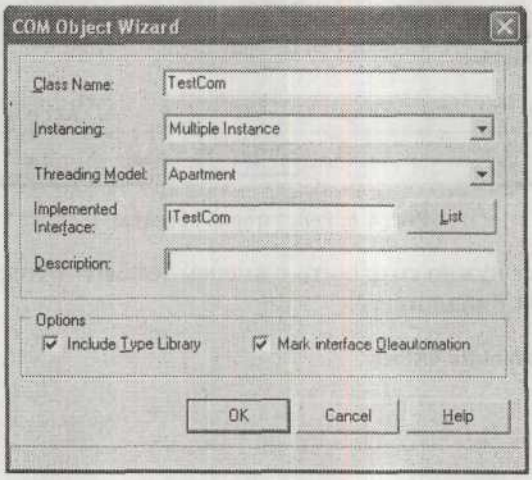


Рис. 4.7. Мастер создания COM-объекта

В поле ClassName указывается имя создаваемого класса. Под этим именем он будет зарегистрирован в реестре, и оно же будет использовано для названия класса CoClass. На рисунке видно, что было выбрано имя TestCom. В списке Instancing определяют способ создания объекта. Следует установить значение Multiple Instance. В списке Threading Model указывается способ взаимодействия сервера и клиента. В данном случае требуется использовать значение Apartament. В поле Implemented Interface указываются интерфейсы, входящие в состав создаваемого объекта COM. По умолчанию для объекта создается собственный интерфейс. Также интерфейс можно выбрать из списка, получаемого при нажатии кнопки List. Активируемое диалоговое окно показано на рис. 4.8. Установка значка Include Type Library приводит к включению в сервер библиотеки типов. Если флажок установлен, то объект наследуется от класса TTypedComObject, если снят — от класса TCOMObject. Установка флажка Mark interface Oleautomation указывает, что сервер COM создается как совместимый с технологией OLE.

Созданный модуль нужно сохранить с именем ComUnit. После того как к серверу будет добавлен объект, в секции Uses библиотеки появится указатель на него и на библиотеку типов сервера, как показано в листинге 4.3.

Рис. 4.8. Список интерфейсов, зарегистрированных на компьютере

Листинг 4.3. Секция Uses библиотеки

```
uses
  ComServ,
  TestServ_TLB in 'TestServ_TLB.pas',
  ComUnit in 'ComUnit.pas' {TestCom: CoClass};
```

Код модуля объекта COM, включенный в проект после его создания, приведен в листинге 4.4.

Листинг 4.4. Код объекта TestCom

```
unit ComUnit;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, TestServ_TLB, StdVcl;

type
  TTestCom = class(TTypedComObject, ITestCom)
  protected
    {Declare ITestCom methods here}
  end;
```

продолжение

Листинг 4.4 *(продолжение)*

```
implementation

uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TTestCom, Class_TestCom,
    ciMultiInstance, tmApartment);
end.
```

Как видно из строки `TTestCom = class(TTypedComObject, ITestCom)`, предком класса `TTestCom` является класс `TTypedComObject`. В секции инициализации располагается фабрика класса объекта. В листинге 4.5 приведен код автоматически созданной библиотеки типов, которая была сохранена в модуле **TestServ_TLB.pas**.

Листинг 4.5. Код модуля TestServ_TLB.pas

```
unit TestServ_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from
// a Type Library. If this type library is explicitly or indirectly
// (via another type library referring to this type library) re-imported,
// or the 'Refresh' command of the Type Library Editor activated while editing
// the Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : 1.2
// File generated on 05.10.2004 0:45:41 from Type Library described below.

// ************************************************************************ //
/
// Type Lib: D:\Documents and Settings\Андрей\Desktop\Книга\Тексты
программ\4.1 Создания внутреннего сервера COM\Сервер\TestServ.tlb (1)
// LIBID: {2BF7428F-C768-43C9-94A2-B7AAA4B74AA5}
// LCID: 0
// Helpfile:
// HelpString: TestServ Library
// DepndLst:
//   (1) v2.0 stdole, (D:\WINDOWS\System32\STDOLE2.TLB)
// ************************************************************************ //
```

```
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}
{$VARPROPSETTER ON}
interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;


// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  TestServMajorVersion = 1;
  TestServMinorVersion = 0;

  LIBID_TestServ: TGUID = '{2BF7428F-C768-43C9-94A2-B7AAA4B74AA5}';

  IID_ITestCom: TGUID = '{C03B7EBC-8BB8-4A25-8BCD-542D91D078F4}';
  CLASS_TestCom: TGUID = '{73CB7E38-B15A-40CC-A2F2-ECB81DACBB0E}';
type

// ************************************************************************//
// Forward declaration of types defined in TypeLibrary
// ************************************************************************//
  ITestCom = interface;

// ************************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// ************************************************************************//
  TestCom = ITestCom;

// ************************************************************************//
```

продолжение

Листинг 4.5 *(продолжение)*

```
// Interface: ITestCom
// Flags:     (256) OleAutomation
// GUID:      {C03B7EBC-8BB8-4A25-8BCD-542D91D078F4}
// ************************************************************************//
  ITestCom = interface(IUnknown)
    ['{C03B7EBC-8BB8-4A25-8BCD-542D91D078F4}']
  end;

// ************************************************************************//
// The Class CoTestCom provides a Create and CreateRemote method to
// create instances of the default interface ITestCom exposed by
// the CoClass TestCom. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
// ************************************************************************//
  CoTestCom = class
    class function Create: ITestCom;
    class function CreateRemote(const MachineName: string): ITestCom;
  end;

implementation

uses ComObj;

class function CoTestCom.Create: ITestCom;
begin
  Result := CreateComObject(CLASS_TestCom) as ITestCom;
end;

class function CoTestCom.CreateRemote(const MachineName: string): ITestCom;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_TestCom) as ITestCom;
end;

end.
```

В библиотеке типов содержится описание интерфейса `ITestCom` и указывается его GUID. Класс `CoTestCom` содержит два метода. Метод `Create` применяется для создания объекта COM, используемого для работы с сервером, расположенным на данной машине, а метод `CreateRemote` — для создания объекта, предназначенного для работы с удаленным сервером.

Теперь необходимо добавить созданному интерфейсу функциональности, то есть создать новый метод. Для этого нужно запустить редактор библиотеки типов, выполнив команду меню View ▸ Type Library. Окно редактора показано на рис. 4.9. В списке указаны интерфейс ITestCom и класс TestCom. В данном редакторе можно добавлять новые методы и интерфейсы к выбранному объекту.

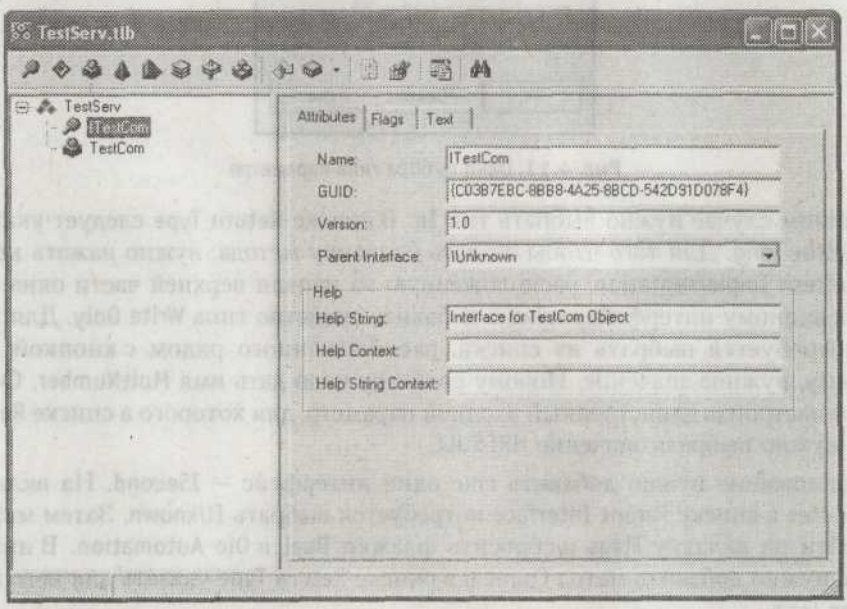


Рис. 4.9. Редактор библиотеки типов

В списке надо выбрать интерфейс ITestCom, а затем выполнить команду контекстного меню New ▸ Method. К интерфейсу будет добавлен новый метод, который в рассматриваемом примере получил имя SimpleMethod. В левой части окна нужно выбрать вкладку Parameters, как показано на рис. 4.10.

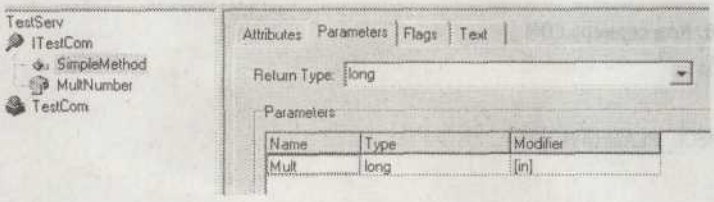


Рис. 4.10. Установка параметров метода

В списке Parametrs можно определять входные и выходные параметры метода, их типы и имена. Добавить новый параметр можно кнопкой Add. В поле Name нужно указать имя Mult, в поле Type потребуется выбрать значение Long. После этого нужно дважды щелкнуть мышью в поле Modifier. Появится окно, показанное на рис. 4.11, в котором будет предложено отметить тип параметра.

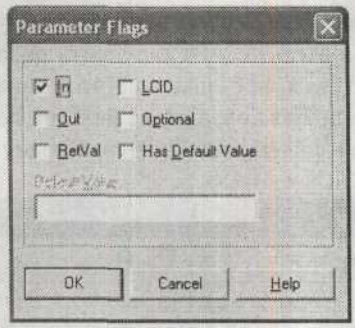


Рис. 4.11. Окно выбора типа параметра

В данном случае нужно выбрать тип In. В списке Return Type следует указать значение long. Для того чтобы создать болванку метода, нужно нажать кнопку Refresh Implementation, расположенную на панели верхней части окна. Затем к данному интерфейсу нужно добавить свойство типа Write Only. Для этого потребуется выбрать из списка, расположенного рядом с кнопкой New Property, нужное значение. Новому свойству надо дать имя MultNumber. Останется настроить единственный входной параметр, для которого в списке Return Type нужно выбрать значение HRESULT.

В приложение нужно добавить еще один интерфейс — ISecond. На вкладке Attributes в списке Parent Interface потребуется выбрать IUknown. Затем можно перейти на вкладку Flags и сбросить флажки Dual и Ole Automation. В интерфейс нужно добавить метод Count и в списке Return Type указать для него значение long.

Теперь нужно выбрать значок класса TestCom, перейти на вкладку Implements и выполнить команду контекстного меню Insert Interface. Из появившегося списка нужно выбрать интерфейс ISecond. Это позволит связать второй интерфейс с объектом. Останется лишь нажать кнопку Refresh. В листинге 4.6. показан код модуля ComUnit, содержащего реализацию методов и свойств сервера.

Листинг 4.6. Код сервера COM

```
unit ComUnit;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, TestServ_TLB, StdVcl;

type
```

```
  TTestCom = class(TTypedComObject, ITestCom, ISecond)
  protected
    function SimpleMethod(Mult: Integer): Integer; stdcall;
    function Set_MultNumber(Mult: Integer): HResult; stdcall;
    function Count: Integer; stdcall;
    {Declare ITestCom methods here}
  end;

implementation

uses ComServ;

var Number : Integer;
    MultNumber : Integer;

function TTestCom.SimpleMethod(Mult: Integer): Integer;
begin
  Result:=Mult*Number;
end;

function TTestCom.Set_MultNumber(Mult: Integer): HResult;
begin
  Number:=Mult;
end;

function TTestCom.Count: Integer;
begin
  Result:=Number+MultNumber;
end;

initialization
  TTypedComObjectFactory.Create(ComServer, TTestCom, Class_TestCom,
    ciMultiInstance, tmApartment);
end.
```

В соответствии с внесенными добавлениями будет модифицирована библиотека типов. Стоит обратить внимание на то, что в нее были добавлены описания интерфейсов и их методов. Теперь проект можно откомпилировать. После этого необходимо зарегистрировать библиотеку в системном реестре. Для этого достаточно выполнить команду меню Run ▸ Register ActiveX Server. В результате будет выведено сообщение о том, что библиотека зарегистрирована в системном реестре.

Теперь нужно создать простое клиентское приложение. Если посмотреть на листинг внимательнее, станет понятно, что первый интерфейс перемножает два числа, а второй складывает их. Одно из чисел передается в перемножающий метод SimpleMethod через свойство MultNumber.

После создания нового проекта надо разместить на форме три компонента TEdit и две кнопки. Для того чтобы обратиться к серверу, необходимо добавить в проект файл библиотеки типов TestServ_TLB.pas. В листинге 4.7 приведен код клиентской части.

Листинг 4.7. Код клиентского приложения

```
implementation
uses TestServ_TLB;
var
TestCom : ITestCom;
Second : ISecond;
{$R *.dfm}
procedure TForm1.MultBtnClick(Sender: TObject);
begin
TestCom:=CoTestCom.Create;
TestCom.QueryInterface(IID_ISecond, Second);
TestCom.Set_MultNumber(StrToInt(Edit1.Text));
Edit3.Text:=IntToStr(TestCom.SimpleMethod(StrToInt(Edit2.Text)));
end;
procedure TForm1.SumBtnClick(Sender: TObject);
begin
TestCom:=CoTestCom.Create;
Edit3.Text:=IntToStr(Second.Count);
end;
end.
```

Следует обратить внимание на следующий фрагмент кода:

```
TestCom:=CoTestCom.Create;
TestCom.QueryInterface(IID_ISecond, Second);
```

Рис. 4.12. Клиентская часть

В этом фрагменте создается CoClass класса TTestCom. Его конструктор возвращает указатель на базовый интерфейс ITestCom в переменную TestCom. Для

получения указателя на второй интерфейс была использована функция QueryInterface первого интерфейса, в качестве параметров которой были переданы идентификатор интерфейса IID_ISecond и переменная, в которую был возвращен указатель. На рис. 4.12 изображено окно клиентского приложения.

Создание локального сервера COM и работа с ним

Для разработки локального сервера потребуется создать новый проект и сохранить его под именем Serv. На главной форме нужно расположить компонент TEdit. Сам модуль надо сохранить с именем LocalServProj.

Теперь к проекту нужно добавить объект COM. Для этого на вкладке ActiveX репозитория объектов следует выбрать значок COM Object и добавить соответствующий объект в проект. В поле ClassName окна мастера создания объекта COM нужно ввести значение Editor. В списке Instancing потребуется выбрать значение Multiple Instance, а в списке Threading Model — значение Apartament. Модуль объекта COM нужно сохранить с именем ComUnit.

Выполнив пункт меню View ▸ Type Library, нужно вызвать библиотеку типов объекта. К интерфейсу IEditor следует добавить метод ReplaceStr. На вкладке Parametrs нужно установить значения параметров метода. В поле Name должно оказаться значение Str, в поле Type нужно выбрать значение BSTR. В поле Modifier нужно установить значение In, а в списке Return Type выбрать значение HRESULT. После этого нужно нажать кнопку Refresh Implementation для формирования шаблона метода и его описания.

Данный COM-объект будет взаимодействовать с формой приложения. С помощью метода ReplaceStr будет изменяться содержимое редактора TEdit. Код объекта приведен в листинге 4.8. Для того чтобы зарегистрировать сервер в системе, достаточно просто запустить его.

Листинг 4.8. Код локального сервера COM

```
unit ComUnit;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, Serv_TLB, StdVcl;

type
  TEditor = class(TTypedComObject, IEditor)
  protected
    function ReplaceStr(const Str: WideString): HResult; stdcall;
```

продолжение ⇩

Листинг 4.8 *(продолжение)*

```
    {Declare IEditor methods here}
  end;

implementation

uses ComServ, LocalServProj;

function TEditor.ReplaceStr(const Str: WideString): HResult;
begin
  Main.Edit1.Text:=Str;
end;

initialization
  TTypedComObjectFactory.Create(ComServer, TEditor, Class_Editor,
    ciMultiInstance, tmApartment);
end.
```

Теперь нужно разработать клиентскую часть приложения. В новом проекте на главной форме нужно расположить компонент TEdit и одну кнопку. Чтобы иметь возможность обращаться к серверу, необходимо подключить библиотеку типов. Для этого в секции uses надо подключить библиотеку Serv_TLB. Если сервер *располагается в ином каталоге, нежели клиентская* часть, то необходимо скопировать библиотеку типа в каталог с клиентской частью. Код клиентского приложения приведен в листинге 4.9.

Листинг 4.9. Код клиентской части

```
var
  Form1: TForm1;

implementation

uses Serv_TLB;
var
   LocalInterface : IEditor;

{$R *.dfm}

procedure TForm1.SendBtnClick(Sender: TObject);
begin
  LocalInterface:=CoEditor.Create;
```

```
  LocalInterface.ReplaceStr(SendEdit.Text);
end;
end.
```

В методе `SendBtn` создается `CoClass` объекта и в переменную `LocalInterface` передается указатель на интерфейс `IEditor` — базовый интерфейс класса. После этого вызывается метод, в качестве параметра которому передается строка. На рис. 4.13 показана основная форма приложения и окно сервера.

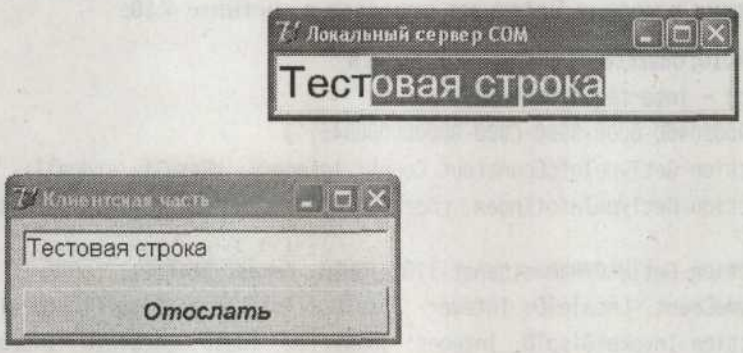


Рис. 4.13. Взаимодействие клиентского приложения и сервера

Можно запустить несколько экземпляров клиентского приложения. Следует обратить внимание на то, что сервер будет выгружен из памяти только тогда, когда отключится последний клиент.

Автоматизация

Автоматизация (*Automation*, ранее называвшаяся *OLE Automation*) является технологией, позволяющей пакетам приложений предоставлять свои методы другим приложениям. Автоматизация основана на технологии COM и широко применяется в системе Windows. Взаимодействие между приложениями осуществляется с помощью вызовов методов интерфейсов, основанных на интерфейсе `IDispatch`.

Основным отличием автоматизации от родительской технологии COM является способ вызова методов интерфейсов. При использовании COM вызов интерфейса возможен только после получения указателя на метод в виртуальной таблице. Этот способ называет ранним связыванием. При использовании автоматизации тоже можно использовать раннее связывание, но возможно использование и другого способа, когда обращение к нужному методу производится при помощи вызова специального метода `Invoke`, возвращающего указатель на нужный метод по его идентификатору. Этот метод называют поздним связыванием.

Объекты приложений, или программные средства, построенные с использованием автоматизации, называются объектами ActiveX. Приложения, обраща-

ющиеся к этим объектам, называются ActiveX-клиентами или контроллерами автоматизации. В общем случае реализация технологии автоматизации имеет ту же структуру, что и реализация COM. Ярким примером пакета, построенного на использовании данной технологии, является пакет MS Office.

Интерфейс IDispatch

Интерфейс IDispatch является базовым интерфейсом автоматизации. Код его объявления в модуле System.pas приведен в листинге 4.10.

Листинг 4.10. Объявление интерфейса IDispatch

```
IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HResult;
stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params: VarResult, ExcepInfo, ArgErr: Pointer):
HResult; stdcall;
  end;
```

Как видно из объявления, интерфейса является наследником IUnknown и, в дополнение к его трем методам, имеет четыре собственных.

Метод Invoke используется для обращения к методам интерфейса. В параметре DispID указывается уникальный идентификатор вызываемого метода. В параметре IID указывается GUID-идентификатор интерфейса, метод которого вызывается. Если приложение поддерживает многоязычность, в параметре LocaleID указывается идентификатор системной локали. На практике этот параметр обычно игнорируется. В параметре Flags указывается способ вызова метод. Он может быть вызван как обычный метод, как метод чтения или как метод записи. В параметре Params хранится указатель на структуру pDispParams, содержащую параметры вызываемого метода. Указатель на результат, возвращенный вызванным методом, содержится в параметре VarResult. Если метод ничего не возвратил, указатель передает значение Null.

Если при вызове метода возникла ошибка, в параметр ExcepInfo помещается указатель на структуру pExcepInfo, которая содержит информацию об ошибке. В параметр ArgErr помещается индекс неверного заданного параметра, указанного разработчиком в структуре pDispParams.

Метод GetIDsOfNames возвращает в параметре DispIDs указатель на массив rgDispId, содержащий идентификаторы методов, имена которых были переданы методу. Эти идентификаторы перечисляются в массиве rgszNames, указатель на который содержится в параметре Names. В параметре Name указывается

количество обработанных методом имен. Часто этот метод используется для определения размерности массива rgszNames.

Метод GetTypeInfo возвращает указатель на интерфейс ITypeInfo библиотеки типов. Этот указатель хранится в параметре TypeInfo. В случае успешного выполнения метода параметр Index принимает нулевое значение.

Метод GetTypeInfoCount позволяет выяснить, может ли возвращать данный объект информацию о типе во время выполнения. В параметре Count может быть возвращено одно из двух значений. Нулевое значение свидетельствует о том, что объект не возвращает информацию о типе во время выполнения. В этом случае для работы с объектом можно использовать виртуальную таблицу или интерфейс IDispatch. Единичное значение указывает, что объект возвращает информацию о типе во время выполнения.

Интерфейсы диспетчеризации и дуальные интерфейсы

Интерфейс диспетчеризации представляет собой интерфейс, в котором содержится список всех доступных свойств и методов, к которым можно обратиться с помощью метода Invoke. Когда контроллеру необходимо использовать тот или иной метод интерфейса, он вызывает метод Invoke и передает ему в качестве параметра идентификатор интерфейса DispID. В листинге 4.11 приведено типовое описание интерфейса.

Листинг 4.11. Код описания диспетчерского интерфейса в библиотеке типов

```
IExampleDisp = dispinterface
  ['{1499FEFE-9AAC-44AB-86D8-6399D75EF4F1}']
  procedure Method1; dispid 201;
  procedure Method2; dispid 202;
  property Property1: Integer readonly dispid 203;
  property Property2: Integer writeonly dispid 204;
end;
```

Диспетчерский интерфейс является наследником интерфейса IDispatch. При его объявлении используется ключевое слово Dispinterface. При передаче параметров и результатов от клиента к серверу (и обратно) производится приведение данных к вариантному типу и обратное преобразование. Поэтому параметры свойств и методов могут иметь только те типы, которые могут быть свободно конвертированы в Variant и обратно.

Дуальный интерфейс сочетает в себе свойства обычного и диспетчерского интерфейса. К его методам можно обратиться как через вызов метода Invoke, так и через виртуальную таблицу. Для контроллеров, которые могут взаимодействовать только с теми объектами, которые предоставляют информацию о типах только во время выполнения, будет использован диспетчерский интерфейс. В другом случае для тех объектов, что также поддерживают раннее связывание, наиболее эффективным будет использование виртуальной таблицы.

На рис. 4.14 приведена схема, на которой показано, как производится обращение к методам объекта автоматизации.

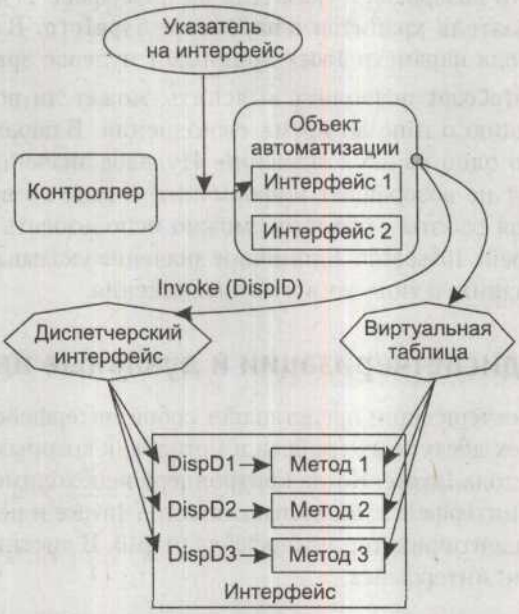


Рис. 4.14. Механизм вызова методов объекта автоматизации

Класс TAutoObject

Класс `TAutoObject` реализует объект автоматизации в Delphi. Он является базовым классом для всех объектов автоматизации. Этот класс инкапсулирует интерфейс `IDispatch`, а значит, реализует его методы `GetIDsOfNames`, `GetTypeInfo`, `GetTypeInfoCount` и `Invoke`.

Свойство `AutoFactory` позволяет обратиться к фабрике класса, с помощью которой создан данный объект.

Метод `Initialize` вызывается при создании объекта. В данном классе он не используется, но может быть переопределен в потомках для инициализации нужных свойств или выполнения нужных действий.

Класс TAutoObjectFactory

Класс `TAutoObjectFactory` реализует собой фабрику класса объекта автоматизации `TAutoObject`. Он является наследником класса `TTypedComObjectFactory`.

Для создания фабрики класса объекта автоматизации используется метод `Create`. Свойство `DispIntfEntry` содержит указатель на структуру `TInterfaceEntry`, предоставляющую информацию, необходимую для вызова интерфейсов диспетчеризации.

Метод GetIntfEntry позволяет получить указатель на структуру PInterfaceEntry для диспетчерского интерфейса, определенного параметром Guid. А через свойство DispTypeInfo можно получить указатель на интерфейс ITypeInfo, который содержит описание диспетчерского интерфейса. Данное свойство используется для получения информации о типах во время создания объекта автоматизации.

TAutoIntfObject

Класс TAutoIntfObject используется для создания объектов автоматизации, поддерживающих диспетчерские интерфейсы. Класс TAutoIntfObject включает в себя интерфейсы IDispatch и ISupportErrorInfo. Если объект автоматизации будет использоваться только внутри приложения, то для его создания можно использовать класс TAutoIntfObject, не требующий использования библиотеки типов. Класс TAutoIntfObject не имеет фабрики класса и поэтому для создания объекта вызывает конструктор Create. Конструктор создает объект с глобальным идентификатором DispIntf на основе информации о типе, содержащейся в параметре TypeLib.

После создания объекта автоматизации его глобальный идентификатор помещается в свойство DispIID. Для получения информации о диспетчерском интерфейсе следует обратиться к свойству DispIntfEntry, которое содержит указатель на структуру PInterfaceEntry. Если же необходимо получить тип диспетчерского интерфейса, то надо обратить внимание на свойство DispTypeInfo, содержащее указатель на интерфейс ITypeInfo.

Класс TAutoIntfObject похож на класс TAutoObject, так как оба класса реализуют интерфейсы IDispatch и ISupportErrorInfo. А также оба они требуют наличия библиотеки типов для создания дуальных интерфейсов. Класс TAutoIntfObject отличается от класса TAutoObject тем, что не имеет фабрики класса.

Сервер автоматизации и пример его реализации

Под сервером автоматизации следует понимать любое приложение, в состав которого входит объект автоматизации. Также это приложение должно быть зарегистрировано в операционной системе. Сервер автоматизации может быть внутренним (In-process), локальным (Local) или удаленным (Remote). В этом разделе будет приведен пример создания подобного приложения.

Для начала, как всегда, потребуется создать проект. В состав приложения необходимо включить объект автоматизации. Для этого при помощи пункта меню File ▸ New ▸ Other надо открыть репозитарий объектов и перейти на его вкладку ActiveX. На этой вкладке надо активировать элемент Automation Object. После его активации будет отображено диалоговое окно, показанное на рис. 4.15.

В поле CoClass необходимо определить имя нового объекта. В списке Instancing надо выбрать значение Multiple Instance, которое задает способ создания объекта. Элементы списка Threading Model указывают, как будут взаимодействовать

сервер и контроллеры. В данном случае надо выбрать значение Apartment. После подтверждения правильности введенных данных при помощи кнопки OK надо сохранить добавленный к проекту модуль объекта с именем AutoObject.

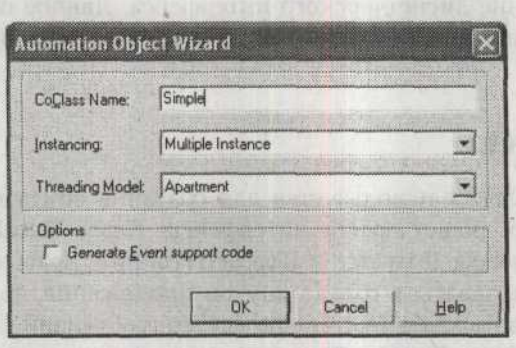


Рис. 4.15. Окно мастера создания объекта автоматизации

Теперь нужно перейти в редактор библиотеки типов, выполнив команду меню View ▸ Type Library. В редакторе надо добавить три метода без параметров с именами Stop, Start и Reset. Также потребуется одно свойство CurentTimeValue типа Read Only, которое будет возвращать значение типа Integer. В окне параметров свойства CurentTimeValue и в поле Return Type нужно выбрать значение Long. В свойстве Modifier указываются значения out, retval. На рис. 4.16 приведено изображение окна библиотеки типов.

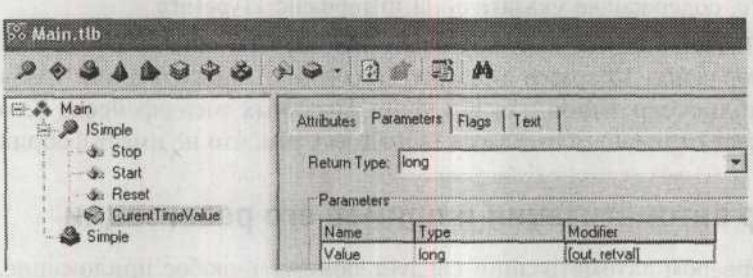


Рис. 4.16. Библиотека типов сервера автоматизации

После нажатия на кнопку Refresh будет сгенерирован код объекта и библиотеки типов. На главной форме приложения потребуется разместить три кнопки, два компонента TLabel и один компонент TTimer. В качестве примера будет разработан секундомер, управление которым будет осуществляться при помощи кнопок Старт, Стоп и Сброс. Кнопками будет управляться компонент TTimer. В листинге 4.12 приведен код модуля главной формы сервера автоматизации.

Листинг 4.12. Код модуля главной формы сервера автоматизации

```
var
  Time : Integer;
```

```
procedure TMainForm.Timer1Timer(Sender: TObject);
begin
  Time:=Time+1;
  TimeLabel.Caption:=IntToStr(Time);
end;

procedure TMainForm.StartBtnClick(Sender: TObject);
begin
  Timer1.Enabled:=True;
end;

procedure TMainForm.StopBtnClick(Sender: TObject);
begin
  Timer1.Enabled:=False;
end;

procedure TMainForm.ResetBtnClick(Sender: TObject);
begin
  Time:=0;
  TimeLabel.Caption:='0';
end;

end.
end.
```

На рис. 4.17 приведено изображение окна сервера автоматизации.

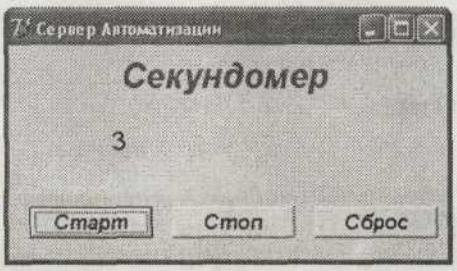


Рис. 4.17. Сервер автоматизации

В листинге 4.13 приведен код объекта автоматизации.

Листинг 4.13. Код модуля

```
unit AutoObject;
{$WARN SYMBOL_PLATFORM OFF}
```

продолжение

Листинг 4.13 *(продолжение)*

```
interface

uses
  ComObj, ActiveX, Main_TLB, StdVcl;

type
  TSimple = class(TAutoObject, ISimple)
  protected
    procedure Start; safecall;
    procedure Stop; safecall;
    procedure Reset; safecall;
    function Get_CurentTimeValue: Integer; safecall;

  end;

implementation

uses ComServ, AutoServProj, SysUtils;

procedure TSimple.Start;
begin
  MainForm.StartBtn.Click;
end;

procedure TSimple.Stop;
begin
  MainForm.StopBtn.Click;
end;

procedure TSimple.Reset;
begin
  MainForm.ResetBtn.Click;
end;

function TSimple.Get_CurentTimeValue: Integer;
begin
  Result:=StrToInt(MainForm.TimeLabel.Caption);
end;
```

```
initialization
  TAutoObjectFactory.Create(ComServer, TSimple, Class_Simple,
    ciMultiInstance, tmApartment);
end.
```

Легко заметить, что код, показанный в листинге 4.13, похож на код обычного сервера COM. В листинге 4.14 приведен код библиотеки типов

Листинг 4.14. Код библиотеки типов

```
unit Main_TLB;

{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}
{$VARPROPSETTER ON}
interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;

// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  MainMajorVersion = 1;
  MainMinorVersion = 0;

  LIBID_Main: TGUID = '{E703122F-1A3C-45F3-B150-73CB2A106A74}';

  IID_ISimple: TGUID = '{EE729D51-8409-4AAB-9017-BCA15EE86A7B}';
  CLASS_Simple: TGUID = '{C554CB1E-A957-4DAE-85AF-AD0BA52F32F2}';
type

// ************************************************************************//
// Forward declaration of types defined in TypeLibrary
// ************************************************************************//
  ISimple = interface;
```

продолжение ⇩

Листинг 4.14 *(продолжение)*

```
  ISimpleDisp = dispinterface;

// ********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// ********************************************************************//
  Simple = ISimple;


// ********************************************************************//
// Interface: ISimple
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {EE729D51-8409-4AAB-9017-BCA15EE86A7B}
// ********************************************************************//
  ISimple = interface(IDispatch)
    ['{EE729D51-8409-4AAB-9017-BCA15EE86A7B}']
    procedure Stop; safecall;
    procedure Start; safecall;
    procedure Reset; safecall;
    function Get_CurrentTimeValue: Integer; safecall;
    property CurrentTimeValue: Integer read Get_CurrentTimeValue;
  end;

// ********************************************************************//
// DispIntf:  ISimpleDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {EE729D51-8409-4AAB-9017-BCA15EE86A7B}
// ********************************************************************//
  ISimpleDisp = dispinterface
    ['{EE729D51-8409-4AAB-9017-BCA15EE86A7B}']
    procedure Stop; dispid 201;
    procedure Start; dispid 202;
    procedure Reset; dispid 203;
    property CurrentTimeValue: Integer readonly dispid 204;
  end;

// ********************************************************************//
// The Class CoSimple provides a Create and CreateRemote method to
// create instances of the default interface ISimple exposed by
```

```
// the CoClass Simple. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this Type Library.
// ************************************************************************//
  CoSimple = class
    class function Create: ISimple;
    class function CreateRemote(const MachineName: string): ISimple;
  end;

implementation

uses ComObj;

class function CoSimple.Create: ISimple;
begin
  Result := CreateComObject(CLASS_Simple) as ISimple;
end;

class function CoSimple.CreateRemote(const MachineName: string): ISimple;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_Simple) as ISimple;
end;

end.
```

Нужно обратить внимание на тот факт, что для интерфейса `ISimple` был создан диспетчерский интерфейс `ISimpleDisp`, содержащий список методов с их идентификационными номерами.

Контроллер автоматизации и пример его реализации

Контроллером автоматизации называют любое приложение, которое использует методы сервера автоматизации. Используя библиотеку типов, можно получить информацию об интерфейсах данного сервера. Если библиотека типов создана в Delphi, то можно просто подключить ее модуль. В ином случае придется применять импортирование. Для импортирования библиотеки следует использовать команду меню **Project ▸ Import Type Library**. Появится окно, показанное на рис. 4.18.

В диалоговом окне отображаются зарегистрированные в данной системе библиотеки типов. Используя кнопки **Add** и **Remove**, можно добавлять или удалять регистрацию о них. Нужно нажать кнопку **Add** и в диалоговом окне выбрать файл сервера автоматизации `main.exe`, который был создан в предыдущем

разделе. В списке Class names перечисляются объекты данного сервера автоматизации. В поле Palette Page указывается, на какую страницу компонентов будет помещен объект, если будет перемещен в компонентную оболочку. Соответственно, в поле Unit dir name указывается директория, в которую будут помещаться модули библиотек типов.

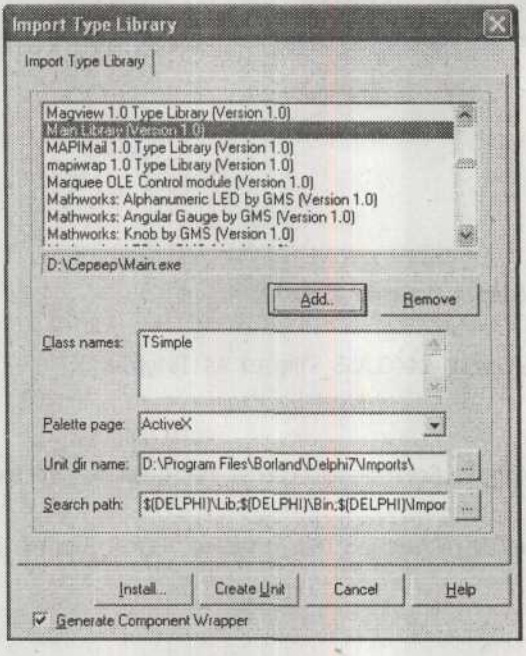


Рис. 4.18. Импорт библиотеки типов

Щелчок на кнопке Install создает модуль с описанием интерфейсов и автоматически регистрирует в системе все необходимые компоненты. Нажатие на кнопку Create Unit создает модуль библиотек типов с описанием интерфейсов, но не устанавливает его в среду.

Теперь можно перейти к разработке приложения. Как всегда нужно создать новый проект, который можно сохранить с именем ClientProj. Затем потребуется создать библиотеку типов и зарегистрировать ее в Delphi. Когда будет выбрана необходимая библиотека, нужно нажать кнопку Install. Появится диалоговое окно, предлагающее зарегистрировать компонент в среде. Так и стоит сделать.

На странице компонентов ActiveX среды разработки появился новый компонент — TSimple. Его нужно перенести на форму. Его свойству AutoConnect нужно присвоить значение True, а свойству ConnectKind — ckNewInstance. На форме надо разместить четыре кнопки и один компонент TEdit. На рис. 4.19 показаны основные окна контроллера и сервера.

В листинге 4.15 приведен код приложения контроллера.

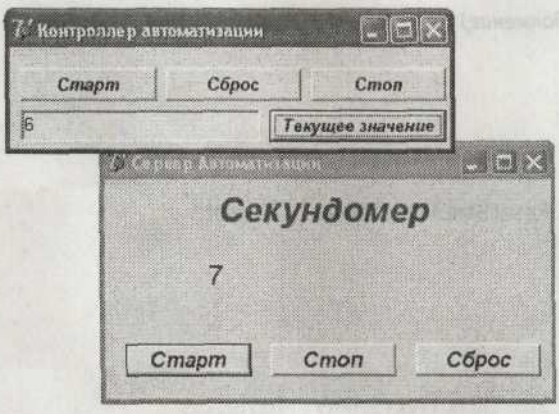


Рис. 4.19. Контроллер автоматизации

Листинг 4.15. Код модуля контроллера

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, OleServer, StdCtrls, Main_TLB;

type
  TForm1 = class(TForm)
    StartBtn: TButton;
    StopBtn: TButton;
    Button3: TButton;
    TimeValueEdit: TEdit;
    ResetBtn: TButton;
    Simple1: TSimple;
    procedure ResetBtnClick(Sender: TObject);
    procedure StartBtnClick(Sender: TObject);
    procedure StopBtnClick(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

продолжение

Листинг 4.15 *(продолжение)*

```
implementation

{$R *.dfm}

procedure TForm1.ResetBtnClick(Sender: TObject);
begin
  Simple1.Reset
end;

procedure TForm1.StartBtnClick(Sender: TObject);
begin
  Simple1.Start;
end;

procedure TForm1.StopBtnClick(Sender: TObject);
begin
  Simple1.Stop;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  TimeValueEdit.Text:=IntToStr(Simple1.CurentTimeValue);
end;
```

Теперь можно реализовать функциональность сервера автоматизации, подключив модуль библиотеки типов напрямую. Для начала следует подготовить проект. Прежде всего нужно удалить с основной формы компонент TSimple. Затем потребуется объявить библиотеку типов Main_TLB в секции Uses. Также нужно добавить переменную Simple, в которой будет размещен указатель на интерфейс ISimple. Код модуля приведен в листинге 4.16.

Листинг 4.16. Код модуля контроллера

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, OleServer, StdCtrls, Main_TLB;

type
  TForm1 = class(TForm)
    StartBtn: TButton;
```

```
    StopBtn: TButton;
    Button3: TButton;
    TimeValueEdit: TEdit;
    ResetBtn: TButton;
    procedure FormCreate(Sender: TObject);
    procedure StartBtnClick(Sender: TObject);
    procedure ResetBtnClick(Sender: TObject);
    procedure StopBtnClick(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    Simple : ISimple;
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation


{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Simple:=CoSimple.Create();
end;

procedure TForm1.StartBtnClick(Sender: TObject);
begin
  Simple.Start;
end;

procedure TForm1.ResetBtnClick(Sender: TObject);
begin
  Simple.Reset
end;
```

продолжение ⇩

Листинг 4.16 *(продолжение)*

```
procedure TForm1.StopBtnClick(Sender: TObject);
begin
  Simple.Stop;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  TimeValueEdit.Text:=IntToStr(Simple.CurentTimeValue);
end;
```

На этом можно завершить обзор технологии COM, так как были рассмотрены все основные варианты ее реализации.

5 УРОК Технология DataSnap

Технология DataSnap применяется для создания распределенных систем, состоящих в общем случае из сервера баз данных, сервера приложения и клиентского приложения, часто называемого «тонким клиентом». Как правило, тонкий клиент представляет собой «облегченное» приложение, предоставляющее пользователю возможность получать и редактировать данные. Клиентское приложение взаимодействует с сервером приложения (Application Server), который напрямую взаимодействует с сервером баз данных, передает ему запросы и возвращает полученные данные тонкому клиенту. В рамках технологии DataSnap сервер приложения является сервером COM, предоставляющим свои интерфейсы конечным пользователям. На рис. 5.1 представлена схема взаимодействия участников соединения.

Рис. 5.1. Структура распределенного приложения

В общем случае сервер приложения располагается на той же машине, что и сервер СУБД, но может располагаться и на удаленном сервере.

Технология DataSnap

Как было отмечено ранее, эта технология обеспечивает взаимодействие удаленных клиентов и серверов, используя для этой цели промежуточный слой. В качестве промежуточного слоя используется сервер приложения. Этот сервер взаимодействует с сервером БД, используя одну из технологий доступа к данным, предоставляемых Delphi. Как известно, Delphi позволяет для доступа к данным использовать ADO, BDE, InterBase eXpress и dbExpress.

Удаленные клиентские приложения строятся на основе компонентов палитры DataSnap. Как правило, клиентские компьютеры являются довольно слабыми системами, поэтому использование сервера приложений позволяет существенно снизить нагрузку на них и поднять производительность системы в целом.

Передача данных между сервером приложений и клиентом осуществляется с помощью интерфейса `IAppServer`, предоставляемого сервером приложения. Этот интерфейс использует компоненты, основанные на классе `TDataSetProvider` на стороне клиента и компоненты `TClientDataSet` на стороне сервера.

Сервер приложения

Основой сервера приложений является *удаленный модуль данных*. Он инкапсулирует все необходимые объекты и интерфейсы для разработки многозвенных баз данных. Удаленный модуль данных может содержать невизуальные компоненты доступа к данным. Также этот модуль данных может включать в себя готовый удаленный сервер. Удаленный модуль данных осуществляет взаимодействие с клиентскими приложениями при помощи интерфейса `IAppServer`. Интерфейс предоставляет методы, используя которые, клиентское приложение может получать и передавать данные. В модуле данных также размещаются компоненты-провайдеры `TDataSetProvider`, связываемые со всеми наборами данных. Они получают данные и передают их компонентам `TClientDataSet`, расположенным на клиентской стороне.

В состав Delphi входят несколько удаленных модулей данных. Они располагаются на страницах палитры компонентов `Multitier`, `WebServices` и `WebSnap`:

- Компонент `CORBA Data Module` позволяет реализовать сервер CORBA.
- Компонент `Remote Data Module` создает сервер автоматизации.
- Компонент `Transactional Data Module` является расширением `Remote Data Module` и реализует сервер MTS.
- Компонент `WebSnap Data Module` позволяет реализовать веб-сервер.
- Компонент `Soap Server Data Module` используется для создания сервера SOAP (Simple Object Access Protocol).

Клиентское приложение

Клиентское приложение строится на базе компонента-соединения, предоставляющего доступ к серверу приложения через интерфейс `IAppServer`. В Delphi

реализовано несколько компонентов соединений, взаимодействующих по различным протоколам:

- Компонент TDCOMConnection использует для взаимодействия технологию DCOM.
- Компонент TSocketConnection для той же цели использует сокеты.
- Компонент TWebConnection использует протокол HTTP.
- Компонент TSOAPConnection использует протокол SOAP.

Компоненты TClientDataSet получают данные и работают с ними в режиме кэширования. На рис. 5.2 приведена схема, иллюстрирующая порядок работы.

Рис. 5.2. Схема трехзвенного распределенного приложения

Компонент TDCOMConnection

Технология *Microsoft Distributed Component Object Model* (DCOM) является расширением COM, позволяющим взаимодействовать объектам COM, расположенным на разных машинах. Используя эту технологию, взаимодействующие объекты COM могут функционировать в локальной сети, в Интернете и иных подключенных системах.

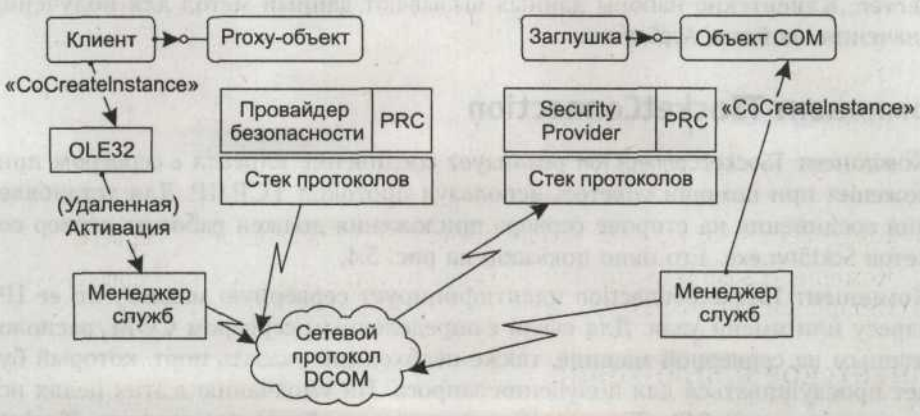


Рис. 5.3. Структура DCOM

На рис. 5.3 представлена схема работы технологии DCOM. Одной из центральных частей технологии является механизм установления соединения и создание нового экземпляра объекта. Клиент вызывает метод CoCreateInstance, в качестве параметра которому передает идентификатор CLSID требуемого объекта. Если CLSID и имя сервера известны, библиотека COM вызывает *менеджер служб* (Service Control Manager) клиентской машины, который передает запрос менеджеру служб сервера и инициирует создание объекта COM. Далее взаимодействие происходит через службу RPC.

Компонент TDCOMConnection используется для создания транспортного канала между клиентским приложением и сервером COM на основе технологии DCOM. В свойстве ComputerName указывается имя компьютера, на котором будет загружен сервер автоматизации. Если поле не содержит значения, компонент TDCOMConnection полагает, что сервер приложения функционирует на локальной машине. Если же это не так, то значение свойства можно получить из системного реестра.

В свойстве ServerName указывается имя сервера приложения, а в свойстве ServerGUID — его идентификатор GUID. Для установления или разрыва соединения следует использовать свойство Connected. Присвоение значения True приводит к установке соединения, а False — к разрыву. Впрочем, для тех же целей можно использовать методы Open и Close.

До и после создания и разрыва соединения инициируются события BeforeConnect, AfterConnect, BeforeDisconnect и AfterDisconnect соответственно.

Метод, вызываемый при возникновении события OnLogin, выполняется после открытия соединения, если свойство LoginPrompt имеет значение True. Параметры Username и Password должны содержать имя пользователя и пароль, указанные в диалоге авторизации.

При помощи свойства AppServer компонент предоставляет интерфейс IAppServer. Этот интерфейс обеспечивает взаимодействие между клиентским приложением и сервером. Для получения интерфейса можно использовать метод GetServer. Клиентские наборы данных вызывают данный метод для получения значения свойства AppServer.

Компонент TSocketConnection

Компонент TSocketConnection реализует соединение клиента с сервером приложения при помощи сокетов, используя протокол TCP/IP. Для установления соединения на стороне сервера приложения должен работать сервер сокетов ScktSrvr.exe. Его окно показано на рис. 5.4.

Компонент TSocketConnection идентифицирует серверную машину по ее IP-адресу или имени узла. Для связи с определенным сервером COM, расположенным на серверной машине, также необходимо указать порт, который будет прослушиваться для получения запроса. По умолчанию в этих целях используется порт 211. Три соответствующих свойства компонента TSocketConnection позволяют определить эти параметры.

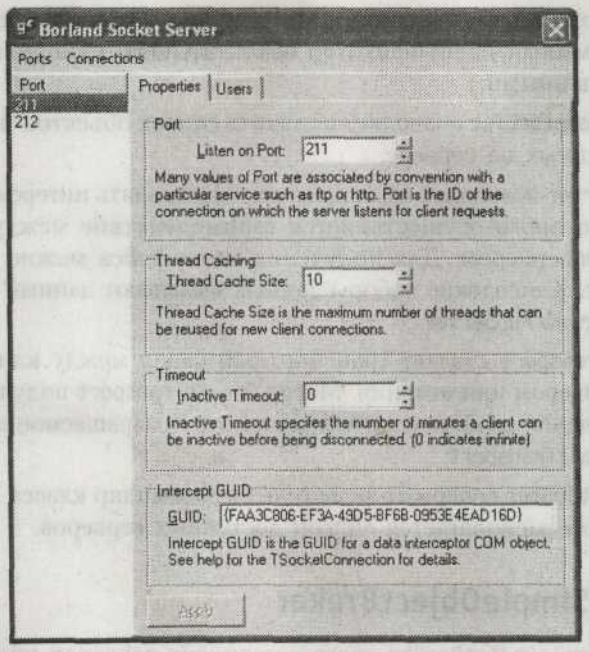


Рис. 5.4. Сервер сокетов

В свойстве `Address` указывается IP-адрес сервера, которому будет послан запрос на установление соединения с клиентским приложением. В свойстве `Host` указывается имя сервера. Значением свойства может служить либо сетевое имя сервера, либо его IP-адрес. В свойстве `Port` необходимо указать номер порта сервера сокетов, который будет прослушиваться диспетчером.

В свойстве `ServerGUID` указывается идентификатор GUID сервера приложения, с которым будет взаимодействовать клиентское приложение. Если на серверной машине существует сервер COM с таким GUID, то свойству `ServerName` автоматически присваивается имя сервера автоматизации.

Для установления или разрыва соединения следует использовать свойство `Connected`. Присвоение значения `True` приводит к установке соединения, а `False` — к разрыву. Впрочем, для тех же целей можно использовать методы `Open` и `Close`.

Метод `GetServerList` позволяет получить список зарегистрированных серверов. Список возвращается в виде вариантного массива.

В свойстве `InterceptGUID` определяется идентификатор GUID объекта COM, который может манипулировать данными до того, как они будут отосланы клиентскому приложению. Объект служит перехватчиком сообщений между компонентом соединения и сервером приложения. Данный объект COM должен реализовывать интерфейс `IDataIntercept`. Используя методы данного интерфейса можно реализовать шифрование пакетов с данными. На стороне сервера тогда необходимо зарегистрировать объект COM, выполняющий обратную операцию. Для этого тоже используется сервер сокетов, идентификатор

GUID которого указывается в поле InterceptGUID. Свойство InterceptName содержит программный идентификатор объекта COM, манипулирующего пересылаемыми данными.

Метод GetInterceptorList позволяет получить список объектов-перехватчиков, зарегистрированных на сервере.

Свойство AppServer позволяет компоненту предоставлять интерфейс IAppServer, при помощи которого осуществляется взаимодействие между клиентским приложением и сервером. Для получения интерфейса можно использовать метод GetServer. Клиентские наборы данных вызывают данный метод для заполнения свойства AppServer.

Метод CreateTransport создает транспортный канал между клиентским приложением и сервером приложения. Метод CreateTransport получает в качестве параметров значения свойств Host, Address и Port. Возвращаемое значение имеет тип интерфейса ITransport.

Свойство ObjectBroker содержит указатель на экземпляр класса TSimpleObjectBroker, в котором располагается список доступных серверов.

Компонент TSimpleObjectBroker

Компонент TSimpleObjectBroker содержит список серверов приложений, доступных данному клиентскому приложению. Когда компонент соединения запрашивает сервер, TSimpleObjectBroker возвращает его имя из списка доступных серверов. Такая возможность позволяет клиентским приложениям обращаться к любому серверу из списка динамически, во время выполнения. Если имеются резервные серверы приложений, то в случае падения сервера этот компонент автоматически выберет другой сервер, что позволит продолжить работу приложения. Компонент может случайным образом выбирать серверы приложений из списка, тем самым реализуя простую систему распределения нагрузки.

В свойстве Servers содержится список серверов, которые могут быть предоставлены компоненту соединения. Этот список заполняется на этапе разработки с помощью специализированно редактора. Значение свойства является коллекцией объектов класса TServerItem, каждый из которых описывает соединение с конкретным сервером. На рис. 5.5 показано окно редактора, в котором показаны имена двух соответствующих серверов.

У объектов класса TServerItem в свойстве ComputerName указывается имя компьютера, на котором расположен сервер приложения. Значение, содержащееся в свойстве, зависит от типа компонента соединения, запрашивающего данные от брокера. Если используется компонент соединения TDCOMConnection, то значение свойства содержит сетевое имя компьютера. Если применяется TSocketConnection, то свойство содержит IP-адрес или сетевое имя хоста.

Задать имя сервера можно при помощи свойства DisplayName. А в свойстве Port указывается порт, используемый для соединения с сервером приложения по

протоколу TCP/IP. Значение этого свойства возвращается методом GetPort-ForComputer компонента TSimpleObjectBroker.

Рис. 5.5. Редактор списка серверов

Когда компонент соединения не может установить связь с сервером приложения, он информирует об этом компонент TSimpleObjectBroker, который присваивает свойству HasFailed значение True. А если свойство имеет значение False, то клиентское приложение все же может использовать данный сервер. При помощи свойства Enabled можно включать в список данный сервер или убирать его из списка.

Свойство LoadBalanced указывает, как компонент TSimpleObjectBroker выбирает сервер из своего списка. Если свойство имеет значение True, то компонент выбирает сервер случайным образом. Если установить значение False, то будет использоваться самый первый доступный сервер.

Метод GetComputerForGUID возвращает строку, в которой содержится сетевое имя компьютера, на котором зарегистрирован сервер с данным GUID. Метод GetComputerForProgID возвращает строку, в которой содержится имя компьютера, на котором зарегистрирован сервер с данным программным идентификатором.

Компонент TConnectionBroker

Компонент используется для централизованного соединения клиентских наборов данных с сервером приложения. Связав клиентские наборы данных через свойство ConnectionBroker, можно изменять значение свойства Connection, в котором содержится ссылка на соответствующий компонент соединения. Таким образом, вместо того чтобы изменять значения свойств RemoteServer всех компонентов клиентских наборов данных, в случае изменения протокола передачи данных достаточно изменить значение в компоненте TConnectionBroker. Этот компонент обеспечивает централизованное использование интерфейса IAppServer и позволяет определять реакцию на сообщения, возникающие при открытии и закрытии соединения с сервером приложения.

В свойстве Connection указывается компонент соединения. При помощи значения свойства AppServer можно обратиться к интерфейсу IAppServer. Также для этой цели можно воспользоваться методом GetServer.

Компонент TLocalConnection

Компонент TlocalConnection позволяет объявлять соединение между клиентским набором данных (TClientDataSet) и провайдером (TDataSetProvider), расположенными в одном модуле данных. Используя свойство AppServer, можно получить доступ к интерфейсу IAppServer. При помощи этого интерфейса можно вызывать методы провайдера данных, до которых в обычной ситуации добраться нельзя из-за ограниченной области видимости. Для тех же целей можно использовать метод GetServer.

Свойство Providers содержит список компонентов-провайдеров, расположенных в том же модуле данных, что и компонент TLocalConnection. Обратиться к провайдеру можно по его имени, которое указывается в параметре ProviderName. В список помещаются только те компоненты-провайдеры, свойство Exported которых имеет значение True.

В свойстве ProviderCount хранится количество провайдеров, на которые содержатся ссылки в свойстве Providers.

Компонент TSharedConnection

Компонент TSharedConnection используется для соединения клиентского приложения с дочерним модулем данных. Он управляет соединением с дочерним модулем данных. Конечно, клиентские приложения могут напрямую обратиться к дочернему модулю данных. Но чаще используется единственное соединение с главным удаленным модулем данных. Это позволяет клиентскому приложению использовать единое соединение для всех модулей данных. Естественно, этот механизм более предпочтителен, чем использование отдельного соединения для каждого модуля данных. На серверной стороне указатель на интерфейс дочернего модуля данных содержится в свойстве интерфейса главного модуля данных. Компонент TSharedConnection использует имя в свойстве для указания того, какой дочерний модуль будет использован.

В свойстве ParentConnection указывается компонент соединения, который связан с главным модулем данных сервера приложения. Имя дочернего модуля данных задается в свойстве ChildName. Для того чтобы обратиться к интерфейсу IAppServer дочернего модуля данных, можно использовать свойство AppServer. Также этот интерфейс можно получить от метода GetServer.

Сервер приложения

Многозвенные приложения баз данных позволяют организовать эффективный доступ удаленных клиентов к данным, используя для этой цели различные протоколы. Промежуточным звеном, служащим для взаимодействия между сервером баз данных и клиентским приложением, является *сервер приложения*. Основой сервера приложения является модуль данных, осуществляющий взаимодействие с клиентскими приложениями через интерфейс IApp-

Server. За обмен данными с сервером отвечают компоненты соединений и провайдеры TDataSetProvider, связываемые с наборами данных. На рис. 5.6 приведена схема работы этого сервера.

Рис. 5.6. Структура сервера приложения

Как видно из схемы, в модуле данных размещаются компоненты наборов данных, связываемые через компонент соединения с базой данных. Также в модуль данных помещаются провайдеры TDataSetProvider, которые получают данные от компонентов наборов данных и пересылают их клиентскому приложению. Как правило, сервер приложения обеспечивает авторизацию пользователей, реализует часть бизнес-логики распределенного приложения и обрабатывает запросы пользователей, возвращая им результат.

Интерфейс IAppServer

Клиентские наборы данных реализуют многие свои свойства и методы при помощи интерфейса IAppServer. Этот интерфейс осуществляет базовое взаимодействие между клиентским набором данных и компонентом-провайдером, через который они передают данные и сохраняют изменения в базе данных. Когда клиентский набор данных использует интерфейс для соединения с провайдером, в качестве одного из параметров он передает имя провайдера при обращении ко всем его методам.

По умолчанию интерфейс не сохраняет значений своих свойств, то есть последующие вызовы функций не могут получить ранее полученный результат. Им приходится получать его самостоятельно. Поэтому интерфейс не имеет свойств, в которых бы хранилась информация о состоянии. Однако в некоторых случаях клиентские наборы данных получают информацию о положении курсора базы данных для получения следующего пакета записей.

Метод AS_Execute выполняет запрос или хранимую процедуру. В параметре CommandText указывается имя хранимой процедуры или текст SQL-запроса.

Метод AS_GetRecords возвращает пакет данных с запрошенными записями. В параметре Count указывается количество записей, которые должен будет возвратить метод. Если параметр имеет значение -1, то возвращаются все записи набора данных. Если параметр имеет нулевое значение, то метод вернет метаданные. Любое положительное значение параметра определяет реально возвращаемое количество записей. В параметре CommandText указывается SQL-запрос, который необходимо выполнить на сервере. Данный параметр будет использован только в том случае, если свойство Options включает значение poAllowCommandText. Параметр TParams позволяет передавать любые параметры, в том числе значения параметров хранимых процедур. В параметре RecsOut возвращается количество переданных записей.

Для получения текущих значений параметров набора данных, ассоциированных с провайдером, следует использовать метод AS_GetParams. А для получения списка провайдеров удаленного модуля данных следует обратиться к методу AS_GetProviderNames. Полученные значения отображаются в свойстве ProviderName клиентского набора данных.

Метод AS_RowRequest возвращает текущую запись набора данных, определенную в параметре Row.

Для того чтобы сохранить измененные данные в базе, следует вызвать метод AS_ApplyUpdates, который передаст их от клиентского набора данных соответствующему провайдеру. В параметре Delta содержатся измененные данные, которые будут переданы СУБД для внесения соответствующих записей в базу данных. В параметре MaxErrors указывается максимально допустимое количество ошибок. При превышении этого параметра обновление будет остановлено и транзакция будет отменена. В параметре ErrorCount возвращается число ошибок, возникших в процессе внесения изменений.

Удаленный модуль данных

Как было сказано ранее, основой сервера приложения является удаленный модуль данных. Он инкапсулирует сервер приложения. В этом разделе будет рассмотрен удаленный модуль данных TRemoteDataModule, реализующий сервер автоматизации. Для создания удаленного модуля данных нужно выполнить команду меню File ▸ New ▸ Other. В репозитории объектов нужно перейти на вкладку Multitier и дважды щелкнуть мышью на значке Remote Data Module. Это активирует диалоговое окно создания удаленного модуля данных, внешний вид которого показан на рис. 5.7.

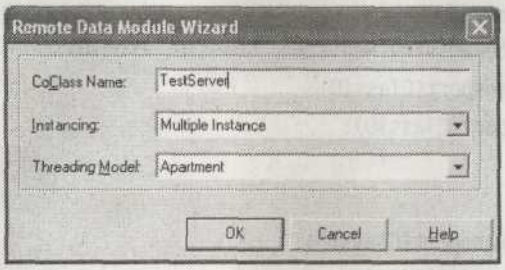


Рис. 5.7. Создание удаленного модуля данных

В поле CoClass Name указывается имя базового интерфейса автоматизации удаленного модуля данных. В листинге 5.1 приведен код удаленного модуля данных.

Листинг 5.1. Код удаленного модуля данных

```
interface

uses
  Windows, Messages, SysUtils, Classes, ComServ, ComObj, VCLCom, DataBkr,
  DBClient, ServProj_TLB, StdVcl;

type
  TTestServer = class(TRemoteDataModule, ITestServer)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean; const ClassID, ProgID:
string); override;
  public
    { Public declarations }
  end;

implementation

{$R *.DFM}

class procedure TTestServer.UpdateRegistry(Register: Boolean; const ClassID,
ProgID: string);
begin
  if Register then
  begin
```

продолжение

Листинг 5.1 *(продолжение)*

```
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;

initialization
  TComponentFactory.Create(ComServer, TTestServer,
    Class_TestServer, ciMultiInstance, tmApartment);
end.
```

При запуске удаленный модуль данных регистрируется в системном реестре как сервер автоматизации, для чего используется процедура `UpdateRegistry`. Затем вызываются процедуры `EnableSocketTransport` и `EnableWebTransport`, которые производят запись в системном реестре о том, что к данному удаленному модулю данных можно получить доступ через сокеты или через Интернет. Соответственно, процедуры `DisableSocketTransport` и `DisableWebTransport` осуществляют обратное действие. Экземпляр сервера приложения создается фабрикой класса, расположенной в секции инициализации. В листинге 5.2 приведен код библиотеки типов.

Листинг 5.2. Код библиотеки типов

```
unit ServProj_TLB;

{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
{$WARN SYMBOL_PLATFORM OFF}
{$WRITEABLECONST ON}
{$VARPROPSETTER ON}
interface

uses Windows, ActiveX, Classes, Graphics, Midas, StdVCL, Variants;

const
  // TypeLibrary Major and minor versions
  ServProjMajorVersion = 1;
```

```
  ServProjMinorVersion = 0;

  LIBID_ServProj: TGUID = '{A51C3468-631B-4C48-A6D9-8371A4F91668}';

  IID_ITestServer: TGUID = '{CCF362F1-132A-4DFB-AFA3-017D47AE25DC}';
  CLASS_TestServer: TGUID = '{7239CDA0-FEC3-4DFB-85A7-1987817F7B51}';
type

// *********************************************************************//
// Forward declaration of types defined in Type Library
// *********************************************************************//
  ITestServer = interface;
  ITestServerDisp = dispinterface;

// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
  TestServer = ITestServer;

// *********************************************************************//
// Interface: ITestServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CCF362F1-132A-4DFB-AFA3-017D47AE25DC}
// *********************************************************************//
  ITestServer = interface(IAppServer)
    ['{CCF362F1-132A-4DFB-AFA3-017D47AE25DC}']
  end;

// *********************************************************************//
// DispIntf:  ITestServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {CCF362F1-132A-4DFB-AFA3-017D47AE25DC}
// *********************************************************************//
  ITestServerDisp = dispinterface
    ['{CCF362F1-132A-4DFB-AFA3-017D47AE25DC}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta:
OleVariant; MaxErrors: Integer;
```

продолжение ⇨

Листинг 5.2 *(продолжение)*

```
                                out ErrorCount: Integer; var OwnerData:
OleVariant): OleVariant; dispid 20000000;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer; out
RecsOut: Integer;
                           Options: Integer; const CommandText: WideString;
var Params: OleVariant;
                           var OwnerData: OleVariant): OleVariant; dispid
20000001;
    function AS_DataRequest(const ProviderName: WideString; Data: OleVariant):
OleVariant; dispid 20000002;
    function AS_GetProviderNames: OleVariant; dispid 20000003;
    function AS_GetParams(const ProviderName: WideString; var OwnerData:
OleVariant): OleVariant; dispid 20000004;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
RequestType: Integer;
                           var OwnerData: OleVariant): OleVariant; dispid
20000005;
    procedure AS_Execute(const ProviderName: WideString; const CommandText:
WideString;
                         var Params: OleVariant; var OwnerData: OleVariant);
dispid 20000006;
  end;

  CoTestServer = class
    class function Create: ITestServer;
    class function CreateRemote(const MachineName: string): ITestServer;
  end;

implementation

uses ComObj;

class function CoTestServer.Create: ITestServer;
begin
  Result := CreateComObject(CLASS_TestServer) as ITestServer;
end;

class function CoTestServer.CreateRemote(const MachineName: string):
ITestServer;
begin
```

```
  Result := CreateRemoteComObject(MachineName, CLASS_TestServer) as
ITestServer;
end;

end.
```

Как было отмечено ранее, при создании сервера автоматизации для него создается интерфейс ITestServer. Данный интерфейс является наследником интерфейса IAppServer и наследует ряд его методов. Так как удаленный модуль данных реализует сервер автоматизации, дополнительно к дуальному интерфейсу ITestServer был создан диспетчерский интерфейс ITestServerDisp. Создание экземпляров класса осуществляется методами Create и CreateRemote. Оба метода возвращают ссылку на основной интерфейс класса. В данном случае возвращается ссылка на ITestServer.

Как правило, дочерние модули данных создаются в связи с требованиями бизнес-логики. Для того чтобы создать дочерний удаленный модуль данных, необходимо всего лишь добавить его в приложение. В библиотеке типов появится соответствующее описание, и его можно будет выбрать через компонент соединения. Для того чтобы использовать единственный компонент соединения, необходимо применять компоненты TSharedConnection.

Как было сказано ранее, для того чтобы получить доступ к дочернему модулю данных через компонент TSharedConnection, необходимо возвращать указатель на него. Это реализуется довольно просто. К базовому интерфейсу примера к интерфейсу ITestServer, добавляется свойство SecModule типа ReadOnly. На вкладке Attributes в поле Type необходимо выбрать базовый интерфейс дочернего модуля данных, на который будет получен указатель. На рис. 5.8 приведен пример выбора правильного указателя.

Рис. 5.8. Получение указателя на дочерний модуль данных

Далее на вкладке Parameters появится новый параметр. В поле Name указывается произвольное имя параметра, в поле Type будет содержаться ссылка на интерфейс ISecondModule **, а в поле Modifier будут указаны флаги Out и Retval. В листинге 5.3 приведен код реализации свойства SecModule.

Листинг 5.3. Код свойства SecModule

```
function TTestServer.Get_SecModule: ISecondModule;
begin
```

продолжение

Листинг 5.3 *(продолжение)*

```
  Result:=CoSecondModule.Create;
end;
```

Как видно из приведенного кода, свойство возвращает указатель на интерфейс ISecondModule. Для получения указателя необходимо вызвать метод Create соответствующего кокласса. Теперь в свойстве ChildName компонента TSharedConnection можно выбрать значение SecModule и получить доступ к дочернему модулю данных SecondModule.

Провайдеры данных

Провайдер TDataSetProvider предназначен для получения данных от сервера БД и передачи их клиентскому приложению. Как правило, данный компонент располагается в удаленном модуле данных сервера приложения и является частью трехзвенной архитектуры. Компонент TDataSetProvider упаковывает получаемые данные и пересылает их в виде пакетов клиентскому набору данных TClientDataSet. Когда клиентский набор данных получает пакет с данными от сервера, он помещает их в оперативную память. Если пользователь модифицировал данные и отсылает их на сервер, компонент TDataSetProvider распаковывает их и при помощи транзакций вносит в базу данных необходимые изменения. Клиентские наборы данных взаимодействуют с компонентом-провайдером через интерфейс IAppServer.

В свойстве DataSet указывается набор данных, из которого компонент TDataSetProvider будет получать данные и вносить в них определенные изменения. Свойство ResolveToDataSet позволяет определить режим внесения изменений в базу данных. Когда свойство имеет значение True, изменения вносятся в связанный с компонентом TDataSetProvider набор данных и только потом в саму базу данных. Если свойство имеет значение False, изменения, переданные компоненту-провайдеру, вносятся напрямую в базу данных, минуя связанный набор данных. В этом режиме несколько увеличивается скорость работы приложения.

Для того чтобы разрешить или запретить клиентским приложениям использовать интерфейс IAppServer, следует использовать свойство Exported. Когда используется значение True, клиентские приложения могут напрямую обращаться к провайдеру, а в ином случае это запрещается.

В свойстве Constraints указывается, будут ли переданы ограничения от серверного набора данных в клиентское приложение. Для передачи ограничений необходимо присвоить свойству значение True.

Свойство Options позволяет определить параметры взаимодействия между клиентским набором данных и компонентом-провайдером. Возможные значения свойства перечислены в списке:

- Значение poFetchBlobsOnDemand указывает, что содержимое BLOB-полей не включается в пакет с данными. Вместо этого при необходимости клиентс-

кое приложение должно запрашивать их. Если свойство FetchOnDemand компонента TClientDataSet имеет значение True, то запрос производится автоматически. В ином случае клиентское приложение должно вызывать метод FetchBlobs.

- Значение poFetchDetailsOnDemand указывает, что когда провайдер представляет в отношениях ссылочной целостности родительскую таблицу, то вложенные таблицы не включаются в пакет данных. В случае необходимости клиентские приложения запрашивают записи связанных таблиц. Если свойство FetchOnDemand компонента TClientDataSet имеет значение True, то клиентское приложение запрашивает эти данные автоматически, иначе для получения данных вложенных таблиц необходимо вызывать метод FetchDetails.
- Значение poIncFieldProps указывает, что в пакет данных включается информация о свойствах Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask и DisplayValues, присущих каждому полю.
- Значение poCascadeDeletes извещает сервер о том, что при удалении записи из главной таблицы в случае наличия отношений ссылочной целостности следует удалять записи подчиненных связанных таблиц. Для использования данной возможности соответствующие настройки должна иметь СУБД.
- Значение poCascadeUpdates указывает, что если таблицы сервера находятся в отношении ссылочной целостности, то при изменении значения ключевого поля в главной таблице автоматически изменяются его значения в подчиненных.
- Значение poReadOnly указывает, что набор данных сервера переводится в режим «только для чтения».
- Значение poAllowMultiRecordUpdates включает режим одновременного изменения нескольких записей. Если свойство имеет значение False, то пакеты обновлений с несколькими записями будут автоматически забракованы.
- Значение poDisableInserts указывает, что клиенты не могут добавлять новые записи.
- Значение poDisableEdits не позволяет клиентам вносить в записи изменения.
- Значение poDisableDeletes указывает, что клиенты не могут удалять записи.
- Значение poNoReset сбрасывает флаг Reset, требующий, чтобы записи в пакете данных начинались с первой. Данный флаг устанавливается в параметре Options метода AS_GetRecords интерфейса IAppServer.
- Значение poAutoRefresh включает автоматическое обновление записей клиентского набора данных.
- Значение poPropogateChanges указывает, что изменения, сделанные в методах, обрабатывающих события BeforeUpdateRecord и AfterUpdateRecord, отсылаются обратно клиентскому набору данных и сливаются с ним.

- Значение poAllowCommandText позволяет выполнять SQL-запросы, определяемые клиентским приложением, то есть выполнять хранимые процедуры и запросы с параметрами.
- Значение poRetainServerOrder запрещает клиентскому приложению изменять порядок сортировки записей.

До и после того как провайдер сохранил изменения, полученные от клиентского набора данных, в базе инициируются события BeforeApplyUpdates и AfterApplyUpdates.

Тип TRemoteEvent обеспечивает работу механизма обмена сообщениями между клиентскими наборами данных и компонентами-провайдерами. Взаимодействие осуществляется поверх методов интерфейса IAppServer. Перед тем как команда будет передана компоненту-провайдеру, будет отослано соответствующее сообщение.

Перед тем и после того как будет выполнен SQL-запрос или хранимая процедура, возникают события AfterExecute и BeforeExecute. А события AfterGetRecords, AfterGetParams, BeforeGetRecords и BeforeGetParams инициируются до и после того, как будут отправлены пакеты с записями и их параметры.

Событие BeforeRowRequest вызывается до того, как провайдер создаст пакет с данными о текущей записи (или блоке записей), а событие AfterRowRequest инициируется после того, как был отослан пакет с запрашиваемыми записями.

В момент поступления запроса на получение данных вызывается метод-обработчик события OnDataRequest. А перед моментом обновления единичной записи на сервере и после него инициируются события BeforeUpdateRecord и AfterUpdateRecord.

Метод-обработчик события OnGetData вызывается уже после того, как компонент-провайдер получил данные от сервера, но еще перед отсылкой их клиентскому приложению. А при сохранении изменений в наборе данных сервера вызывается метод-обработчик OnUpdateData. Если во время сохранения изменений на сервере произошла ошибка, то инициируется событие OnUpdateError.

Метод-обработчик события OnGetTableName вызывается, когда компонент-провайдер получает имя таблицы, в которую будут внесены изменения. А событие OnGetDataSetProperties возникает, когда провайдер добавляет дополнительную информацию в создаваемый пакет данных.

Пример разработки сервера приложения

Для создания сервера приложения нужно открыть новый проект и добавить в него удаленный модуль данных. Создаваемый класс должен получить имя TestServer, под которым его и нужно сохранить.

В модуле данных надо разместить компоненты TADOConnection, TADOTable и TDataSetProvider с именем Customer. В компоненте TADOConnection надо настроить соединение с базой данных dbdemos. Свойству IsolationLevel требуется присвоить значение iChaos. Компоненты TADOTable и TADOConnection надо связать

между собой и свойству TableName последнего присвоить значение Customer. Также потребуется связать компоненты TADOTable и TDataSetProvider.

В проект нужно добавить еще один дочерний удаленный модуль данных с именем SecondModule. В нем нужно создать тот же набор компонентов, что и в родительском модуле. Через компонент TADOConnection следует установить соединение с СУБД SQL Server 2000 и открыть базу данных Northwind. Свойство IsolationLevel должно получить значение iChaos. Также потребуется установить соединение компонента TADOTable с базой данных через его свойство Connection и связь с компонентом-провайдером TDataSetProvider, получившим имя DSPEmployee.

На форме надо расположить компонент TDataSource, который потребуется связать с компонентом ADOTable1. Также нужно добавить второй компонент TADOTable. Он должен быть связан с компонентами TADOConnection и TDataSetProvider. В свойстве TableName надо выбрать таблицу Customers и установить соединение с базой, присвоив свойству Active значение True.

Теперь требуется настроить отношения ссылочной целостности. В свойстве MasterSource компонента ADOTable2 нужно выбрать компонент TDataSource, а в свойстве MasterFields установить связь по полям CustomerID и EmployeeID. После этого можно перейти в редактор библиотеки типов, выполнив команду меню View ▸ Type Library.

К интерфейсу ITestServer нужно добавить два метода и свойство типа Read only. Через это свойство интерфейсу ITestServer будет передан указатель на интерфейс ISecond. Таким образом, к нему можно будет обратиться через компонент TSharedConnection.

В методах будет реализован подсчет количества клиентов, работающих в этот момент с базой данных. Первый метод AddUser будет увеличивать счетчик, а метод DeleteUser, соответственно, уменьшать значение счетчика. Для этого нужно разместить на форме компонент TTimer и несколько компонентов TLabel.

В листинге 5.4 приведен код главного удаленного модуля данных. Для регистрации сервера приложения достаточно просто запустить его. При распространении приложений на стороне сервера будет необходимо зарегистрировать библиотеку midas.dll командной regsvr32 midas.dll.

Листинг 5.4. Код главного модуля данных

```
class procedure TTestServer.UpdateRegistry(Register: Boolean; const ClassID,
ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
```

продолжение ⇩

Листинг 5.4 *(продолжение)*

```
    EnableWebTransport(ClassID);
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;

function TTestServer.Get_SecModule: ISecondModule;
begin
  Result:=CoSecondModule.Create;
end;

procedure TTestServer.AddUser;
var Count : Integer;
begin
  Count:=StrToInt(ServerForm.Label4.Caption);
  ServerForm.Label4.Caption:=IntToStr(Count+1);
end;

procedure TTestServer.DeleteUser;
var Count : Integer;
begin
  Count:=StrToInt(ServerForm.Label4.Caption);
  ServerForm.Label4.Caption:=IntToStr(Count-1);
end;

function TTestServer.Get_CountsUser: Integer;
begin
  Result:=StrToInt(ServerForm.Label4.Caption);
end;

initialization
  TComponentFactory.Create(ComServer, TTestServer,
    Class_TestServer, ciMultiInstance  , tmApartment);
end.
```

На рис. 5.9 показано окно сервера приложения.

Рис. 5.9. Окно сервера приложения

Клиентское приложение

Как правило, в многозвенных базах данных клиентское приложение не использует много ресурсов и называется тонким клиентом. На клиентское приложение возлагаются функции представления данных в удобном виде и организации хорошего интерфейса, предназначенного для работы с ними. Клиентское приложение через интерфейс `IappServer` взаимодействует с сервером приложения, получает от него запрашиваемые данные и возвращает измененные. Основой клиентского приложения является компонент `TClientDataSet`, который взаимодействует с компонентом-провайдером `TDataSetProvider`. Соединение с удаленным модулем данных обеспечивают компоненты `TSocketConnection` и `TDCOMConnection`. Данные, получаемые компонентом `TClientDataSet`, хранятся в кэше. Таким образом, пользователь работает именно с кэшем, используя механизм отложенного обновления. Как и обычное приложение баз данных, клиентское приложение содержит компоненты отображения данных и компоненты пользовательского интерфейса. На 5.10 показана соответствующая схема.

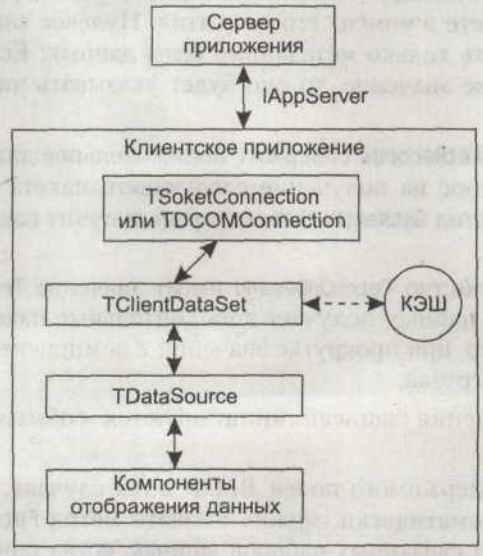


Рис. 5.10. Структура клиентского приложения

Компонент TClientDataSet

Компонент TClientDataSet представляет собой набор данных, размещаемый в памяти. Клиентский набор данных, который реализует компонент TClientDataSet, может быть использован как полнофункциональный автономный набор данных, хранилищем данных для которого является файл. Подобная функциональность обычно применяется в однозвенных приложениях баз данных. Иногда подобный набор данных используется локально в оперативной памяти, получая данные от другого набора данных. Эта функциональность применяется в многозвенных приложениях баз данных.

Класс TClientDataSet является потомком класса TDataSet, поэтому наследует свойства базового набора данных. Соединение с удаленным модулем данных осуществляется при помощи свойства RemoteServer, в котором указывается соответствующий компонент соединения.

В свойстве ProviderName выбирается компонент-провайдер, который передает в компонент TClientDataSet данные от сервера и возвращает ему внесенные изменения.

После того как были произведены все необходимые настройки, можно установить соединение с базой данных. Для этого следует присвоить свойству Active значение True или вызвать метод Open.

После установления соединения с удаленным модулем данных через свойство AppServer можно получить доступ к интерфейсу IAppServer, при помощи которого вызываются методы удаленного модуля данных.

Используя свойство PacketRecords, можно указать компоненту-провайдеру, какое количество записей включать в один пакет с данными. Если свойство имеет значение -1 по умолчанию, то все записи будет переданы в клиентский набор в одном пакете в момент его открытия. Нулевое значение свойства заставляет передавать только метаданные базы данных. Если свойство содержит положительное значение, то оно будет указывать число записей, помещаемых в пакет.

Если свойство PacketRecords содержит положительное значение, необходимо инициировать запрос на получение следующего пакета данных. Для этого следует вызвать метод GetNextPacket, который получит следующий блок записей от провайдера.

По умолчанию свойство FetchOnDemand имеет значение True. Это значит, что клиентский набор данных получает дополнительные пакеты данных по требованию. Например, при прокрутке значений в компоненте TDBGrid будет осуществляться их загрузка.

До и после получения записей инициируются события BeforeGetRecords и AfterGetRecords.

Для получения содержимого полей BLOB в тех случаях, когда информация не передается автоматически, можно вызвать метод FetchBlobs. Также для получения записей связанных наборов данных, когда они не передаются автоматически, используется метод FetchDetails.

Используя свойство CommandText можно указать SQL-запрос, имя таблицы или хранимой процедуры, которые будут переданы на сервер. Для выполнения запроса следует вызвать метод Execute, который инициирует выполнение соответствующей команды. Метод может быть использован в тех случаях, когда запрос или хранимая процедура не возвращают набор данных. Для того чтобы компонент-провайдер мог использовать данную возможность, необходимо присвоить значение True параметру poAllowCommandText свойства Options компонента TDataSetProvider.

До того как запрос будет выполнен на сервере, он пересылается провайдеру. В результате инициируется событие BeforeExecute. После того как запрос будет выполнен, вызывается метод-обработчик события AfterExecute. А при помощи свойства Params запросы и хранимые процедуры передают свои параметры серверу.

Получить текущие значения параметров компонента-провайдера можно при помощи метода FetchParams. На этапе разработки параметры можно просмотреть, выполнив команду Fetch Params контекстного меню компонента TClientDataSet.

Для записи внесенных изменений в базу данных используется метод ApplyUpdates, пересылающий все измененные, добавленные и удаленные записи на сервер. В параметре MaxErrors указывается максимальное число ошибок, после возникновения которого процесс сохранения прерывается. Если параметр имеет значение -1, то лимит количества ошибок не установлен, и сохранение будет прервано при первой же ошибке.

До и после сохранения изменений в базе данных инициируются события BeforeApplyUpdates и AfterApplyUpdates. Все внесенные в кэшируемый набор данных изменения сохраняются в специальном буфере, доступ к которому можно получить через свойство Delta. Количество записей, содержащихся в буфере Delta, указано в свойстве ChangeCount. Метод ApplyUpdates передает информацию из этого буфера компоненту-провайдеру. Для того чтобы очистить свойство Delta и локально отменить изменения, можно воспользоваться методом CancelUpdates. Метод RevertRecord по своему действию аналогичен методу CancelUpdates, но используется только для отката изменений текущей записи.

Для того чтобы получить копию данных с сервера, необходимо вызвать метод RefreshRecord, который произведет обновление всех записей, содержащихся в клиентском наборе данных. Все внесенные пользователем изменения будут потеряны. Перед обновлением записей инициируется событие BeforeRowRequest, а после обновления — AfterRowRequest.

Каждая запись клиентского набора данных имеет определенный статус, который возвращается методом UpdateStatus. Возвращаемое значение принадлежит к перечислимому типу. В списке указаны возможные значения:

- Значение usUnmodified указывает, что запись не была изменена.
- Значение usModified указывает, что запись была изменена.
- Значение usInserted указывает, что запись была добавлена, но на данный момент не сохранена в базе данных.

- Значение usDeleted указывает, что данная запись была удалена, но изменения не были сохранены в базе данных, а содержаться в локальном кэше.

Каждое поле в клиентском наборе данных может иметь три значения. Первым является базовое значение, то есть то, которое хранилось в локальной копии до начала редактирования. Второе возможное значение — текущее. Это то значение, которое указывает пользователь. Третьим значением является измененное и подтвержденное пользователем значение. В свойстве OldValue содержится базовое значение поля. В свойстве CurValue содержится текущее значение поля, включающее в себя изменения, сделанные другими пользователями базы данных. Значение данного поля используется в тех случаях, когда возникают ошибки сохранения данных. Например, значение поля могло быть изменено другой транзакцией и не может быть изменено текущей в силу налагаемых ограничений. А значение поля, которое будет сохранено, содержится в свойстве NewValue.

Компонент TClientDataSet позволяет сохранять данные в файле или в потоке, а затем читать их оттуда. Используя данное свойство компонента, можно организовывать простые файловые СУБД и даже обмен данными на сменных носителях, копируя часть записей во временный набор. Имя файла, в котором будет сохранен набор данных, указывается в свойстве FileName. Для загрузки набора данных из файла и записи данных в него применяются методы LoadFromFile и SaveToFile. В параметре FileName указывается имя файла, в котором будет храниться набор данных. Если параметру не присвоено значение, то оно будет взято из свойства FileName. В параметре Format определяется формат, в котором будет сохранен набор данных. Существуют три возможных значения формата:

- Значение dfBinary указывает, что будет использоваться двоичный формат данных.
- Значение dfXML указывает, что будет использоваться формат XML.
- Значение dfXMLUTF8 указывает, что будет использоваться формат данных XML в кодировке UTF8.

Для сохранения набора данных в потоке и чтения из него предназначены методы SaveToStream и LoadFromStream соответственно. В параметре Stream указывается имя потока.

Как и обычный компонент набора данных, компонент TClientDataSet имеет возможность работать с индексами. Для создания нового индекса следует воспользоваться методом AddIndex. В параметре Name указывается имя создаваемого индекса, а в параметре Fields перечисляются входящие в него поля. Используя параметр Options, можно задать необходимые атрибуты индекса. Возможные значения этого параметра перечислены ниже:

- Значение ixPrimary указывает, что индекс является первичным.
- Значение ixUnique указывает, что каждое значение индекса уникально.
- Значение ixDescending указывает, что индекс будет сортировать записи в порядке убывания.

- Значение ixExpression указывает, что индекс основывается на использовании ключевого выражения. Эта возможность применяется только для таблиц Dbase.
- Значение ixCaseInsensitive указывает, что индекс сортирует записи, не учитывая регистр символов.
- Значение ixNonMaintained указывает, что индекс не будет автоматически перестраиваться при изменении данных.

В параметре DescFields указывается список полей, по которым будет производиться сортировка в порядке убывания. Фактически, этот параметр дублирует значение ixDescending параметра TIndexOptions. В параметре CaseInsFields указывается список полей, по которым будет производиться сортировка. Но при этом регистр символов учитываться не будет. В параметре GroupingLevel указывается уровень группировки записей.

Для того чтобы подключить созданный индекс или сделать его активным, следует указать его имя в свойстве IndexName. В свойстве IndexFieldNames через точку с запятой указывается список полей, которые будут использоваться в качестве индекса. Свойства IndexName и IndexFieldNames не могут использоваться одновременно.

Для получения информации об индексах клиентского набора данных можно использовать свойство IndexDefs. Используя свойство IndexFields, можно получить список полей, входящих в текущий индекс. Для получения их количества следует обратиться к свойству IndexFieldCount. Для получения списка индексов набора данных можно использовать метод GetIndexNames.

Стоит упомянуть и механизм обеспечения ссылочной целостности. Данный механизм реализован, как в стандартных компонентах, но с одним исключением. Ограничения ссылочной целостности должны быть определены не только на клиентской стороне, в компоненте TClientDataSet, но и на стороне сервера приложения в компоненте, реализующем табличную функциональность. Для обеспечения связи родительской и дочерней таблиц в свойстве MasterSource необходимо выбрать компонент TDataSource, указывающий на набор данных, представляющий главную таблицу. После этого можно определить связи через свойство MasterFields.

Обработка ошибок сохранения данных

В случае возникновения ошибок сохранения записей компонент-провайдер возвращает их компоненту TClientDataSet в пакете данных Delta и инициирует для каждой записи событие OnReconcileError, в котором можно определить реакцию на возникающие ошибки.

Метод, выполняющийся при возникновении этого события, имеет несколько параметров. В параметре DataSet содержится ссылка на клиентский набор данных, в котором произошла ошибка. Если ошибка произошла в подчиненном наборе, то метод главного набора данных вернет в параметре DataSet ука-

затель на подчиненный набор. Получить доступ к значениям данной записи можно при помощи свойств CurValue, OldValue и NewValue. В параметре E содержится указатель на объект класса исключения EReconcileError, предоставляющий информацию об ошибке. В параметре UpdateKind содержится описание причины ошибки. Параметр может принимать три значения:

- Значение ukModify указывает, что ошибка возникла при изменении данных.
- Значение ukInsert свидетельствует, что добавление новой записи привело к ошибке.
- Значение ukDelete указывает, что ошибка произошла при удалении записи.

Параметр Action позволяет определить реакцию клиентского набора данных на возникшую ошибку. Параметр может принимать несколько заранее определенных значений:

- Значение raSkip отменяет операцию сохранения для текущей записи и возвращает ее в буфер записей.
- Значение raAbort полностью отменяет операцию сохранения для записи.
- Значение raMerge объединяет значение вызвавшей ошибку записи со значением записи на сервере.
- Значение raCorrect повторно сохраняет изменения.
- Значение raCancel отменяет сделанные изменения и устанавливает полям их текущие значения.
- Значение raRefresh отменяет все сделанные изменения, присваивая полям их базовые значения, то есть значения полей, хранящиеся на сервере.

Разработчик, используя данную информацию, может самостоятельно реализовать механизм обработки ошибок. Можно также воспользоваться методом HandleReconcileError, который отображает диалоговое окно, сообщающее пользователю о произошедшей ошибке и предоставляющее ему возможность выбрать определенные действия для ее устранения.

В листинге 5.5 приведен пример использования данного метода.

Листинг 5.5. Пример использования метода HandleReconcileError

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind; var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

На рис 5.11 показано диалоговое окно, вызываемое этим методом.

Так же как и обычные компоненты наборов данных, компонент TClientDataSet наследует методы-обработчики, вызывающиеся при модификации данных, содержащихся в кэше. При возникновении ошибки удаления записи инициируется событие OnDeleteError. В случае возникновения ошибки ввода новой

записи или модификации существующей записи вызывается метод-обработчик OnPostError. Событие OnEditError инициируется, если приложение пытается модифицировать данные.

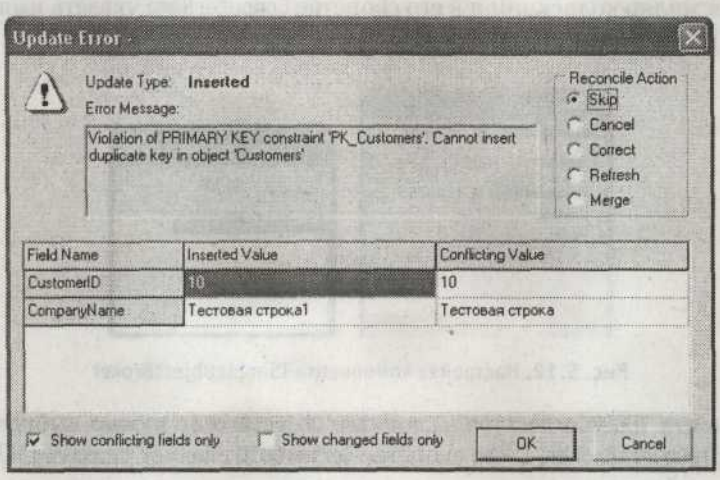


Рис. 5.11. Окно обработки ошибок сохранения данных в БД

Соответствующие методы содержат указатель на тип TDataSetErrorEvent. Тип TDataSetErrorEvent похож на тип TReconcileErrorEvent. Параметр DataSet указывает на набор данных, в котором произошла ошибка. В параметре E содержится указатель на объект класса исключения EDatabaseError, с помощью которого можно получить информацию об ошибке. В параметре Action можно определить реакцию на возникшую ошибку. Параметр может принимать несколько строго определенных значений:

- Значение daFail приводит к отмене операции и отображению сообщения об ошибке.
- Значение daAbort отменяет операцию без отображения сообщения об ошибке.
- Значение daRetry приводит к повтору операции. В этом случае следует написать обработчик, который будет позволять корректировать введенные данные.

Пример разработки клиентского приложения с использованием DCOM

К новому проекту DCOMPrj следует добавить модуль данных. В нем надо разместить три компонента TClientDataSet, три компонента TDataSource и компоненты TDCOMConnection, TSimpleObjectBroker, TSharedConnection и ConnectionBroker. В свойстве Connection компонента TConnectionBroker нужно выбрать соответствующий компонент TDCOMConnection. Это поможет связать с ним все компоненты TClientDataSet. В случае изменения настроек соединения не придется

перенастраивать все компоненты по одному. Достаточно будет изменить только компонент TConnectionBroker.

Для компонента TSimpleObjectBroker нужно выбрать свойство Servers, добавить новый экземпляр коллекции и в его свойстве ComputerName указать имя компьютера, как это показано на рис. 5.12.

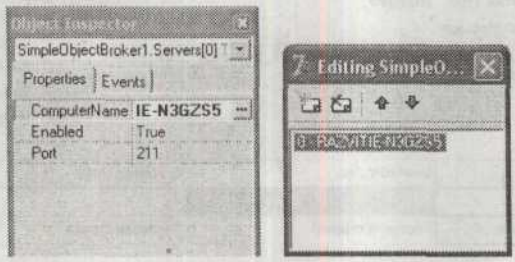


Рис. 5.12. Настройка компонента TSimpleObjectBroker

У компонента TDCOMConnection в свойстве ObjectBroker нужно выбрать компонент TSimpleObjectBroker. В его свойстве ServerGUID следует указать GUID-идентификатор объекта.

В списке ServerName надо выбрать удаленный модуль данных ServProj.TestServer. В списке доступен также другой модуль данных — ServProj.SecondModule. Но в данном случае будет использоваться только один компонент соединения, доступ к которому будет осуществляться при помощи компонента TSharedConnection. Его свойству LoginPrompt нужно присвоить значение False, а свойству Connected — значение True.

Теперь необходимо выбрать компонент TSharedConnection. В его свойстве ParentConnection нужно указать компонент TDCOMConnection, а в свойстве ChildName выбрать из списка значение SecModule. Это значение является именем свойства, возвращающего указатель на дочерний модуль данных.

В свойстве RemoteServer компонента ClientDataSet1 нужно выбрать компонент TConnectionBroker, а в свойстве ProviderName — значение Customer. Свойство Active должно получить значение True. После этого нужно связать компонент ClientDataSet1 с компонентом DataSource1.

В свойстве RemoteServer компонента ClientDataSet2 нужно выбрать компонент TSharedConnection, а в свойстве ProviderName указать таблицу DSPEmploeey. Этот компонент нужно связать с DataSource2. Точно так же следует настроить компонент ClientDataSet3, указав в его свойстве ProviderName значение DSPCustomers.

Теперь нужно определить отношения ссылочной целостности. В свойстве MasterSource компонента ClientDataSet3 нужно выбрать компонент DataSource2, а потом установить связь между таблицами в свойстве MasterFields по полям EmployeeID и CustomerID. Оба компонента следует активировать.

На главной форме надо разместить три компонента TDBGrid и три кнопки. Модуль данных следует объявить в модуле главной формы приложения. Ком-

поненты TDBGrid нужно связать с компонентами TDataSource. В примере показана связь таблиц и работа с ними. На рис. 5.13 изображено работающее приложение. Код главной формы приложения приведен в листинге 5.6.

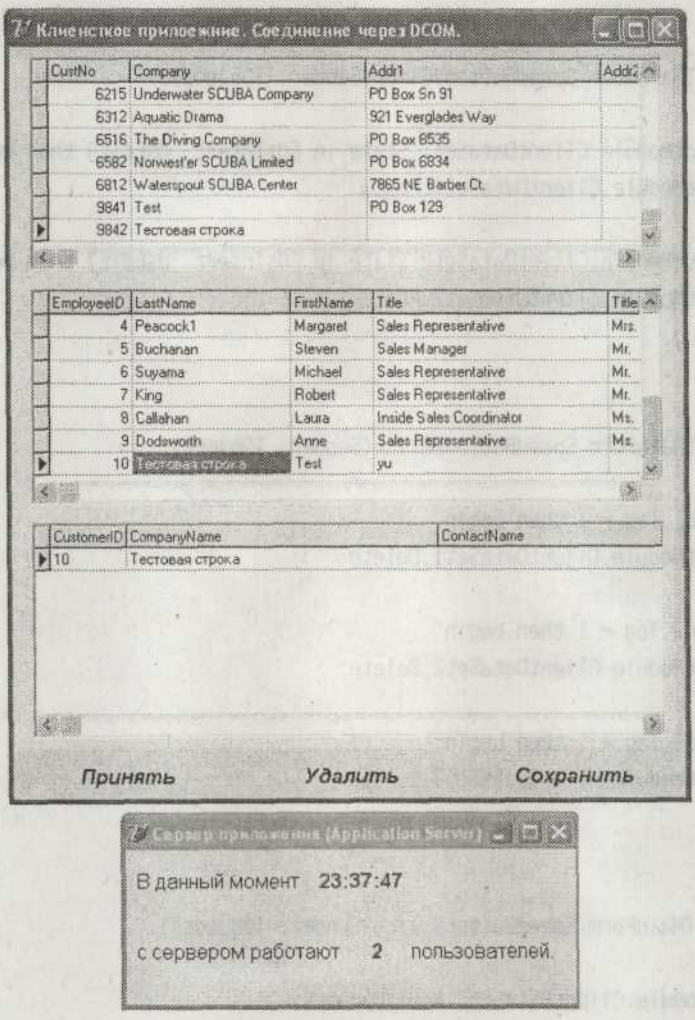


Рис. 5.13. Трехзвенное приложение с двумя клиентами

Листинг 5.6. Код клиентского приложения

```
uses ConnectionModule, Math;

{$R *.dfm}
procedure TMainForm.FormShow(Sender: TObject);
```

продолжение

Листинг 5.6 *(продолжение)*

```
begin
  ConnectModule.DCOMConnection.AppServer.AddUser;
end;

procedure TMainForm.SpeedButton1Click(Sender: TObject);
begin
  if ConnectModule.ClientDataSet1.State in [dsInsert, dsEdit] then begin
    ConnectModule.ClientDataSet1.Post;
  end;
  if ConnectModule.ClientDataSet2.State in [dsInsert, dsEdit] then begin
    ConnectModule.ClientDataSet2.Post;
  end;
end;

procedure TMainForm.SpeedButton2Click(Sender: TObject);
begin
  if DBGrid1.Tag = 1 then begin
    ConnectModule.ClientDataSet1.Delete;
  end;
  if DBGrid2.Tag = 1 then begin
    ConnectModule.ClientDataSet2.Delete;
  end;
  if DBGrid3.Tag = 1 then begin
    ConnectModule.ClientDataSet3.Delete;
  end;
end;

procedure TMainForm.SpeedButton3Click(Sender: TObject);
begin
  ConnectModule.ClientDataSet1.ApplyUpdates(-1);
  ConnectModule.ClientDataSet2.ApplyUpdates(-1);
  ConnectModule.ClientDataSet3.ApplyUpdates(-1);
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ConnectModule.DCOMConnection.AppServer.DeleteUser;
end;
```

```
procedure TMainForm.DBGrid1CellClick(Column: TColumn);
begin
  DBGrid1.Tag:=1;
  DBGrid2.Tag:=0;
  DBGrid3.Tag:=0;
end;

procedure TMainForm.DBGrid2CellClick(Column: TColumn);
begin
  DBGrid1.Tag:=0;
  DBGrid2.Tag:=1;
  DBGrid3.Tag:=0;
end;

procedure TMainForm.DBGrid3CellClick(Column: TColumn);
begin
  DBGrid1.Tag:=0;
  DBGrid2.Tag:=0;
  DBGrid3.Tag:=1;
end;
```

Чтобы определить, из какой таблицы следует удалить запись, был реализован простейший код, в котором значение тега устанавливается в единицу при выборе ячейки. А после остается лишь произвести проверку соответствующих тегов и выполнить удаление записей. Впрочем, в реальных проектах подобная технология может оказаться неэффективной.

Пример разработки клиентского приложения с использованием сокетов

Теперь можно разработать то же самое приложение, но связь с сервером приложения устанавливать при помощи сокетов. Для начала следует запустить сервер сокетов `scktsrvr.exe` и выбрать для работы какой-либо порт. По умолчанию используется порт 211. В поле `GUID` необходимо указать GUID-идентификатор сервера.

В модуле данных клиентского приложения нужно заменить компонент `TDCOMConnection` компонентом `TSocketConnection`. После этого потребуется настроить компонент `TSimpleObjectBroker`. В качестве значения свойства `ComputerName` нужно использовать IP-адрес компьютера, на котором производится разработка. Обычно для этого можно использовать стандартный адрес `127.0.0.1`. Также потребуется указать выбранный номер порта, используя свойство `Port`. В свойстве `ObjectBroker` компонента `TSocketConnection` нужно выбрать ком-

понент TSimpleObjectBroker, а в свойстве ServerGUID указать GUID-идентификатор объекта.

После этого в списке ServerName необходимо выбрать удаленный модуль данных ServProj.TestServer. Свойство Connected должно получить значение True. В свойстве Connection компонента TConnectionBroker нужно выбрать компонент TSocketConnection. Его же надо указать в свойстве ParentConnection компонента TSharedConnection.

Эти небольшие изменения приведут к тому, что приложение будет работать с другим механизмом установки соединения с сервером приложений.

6 УРОК 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.

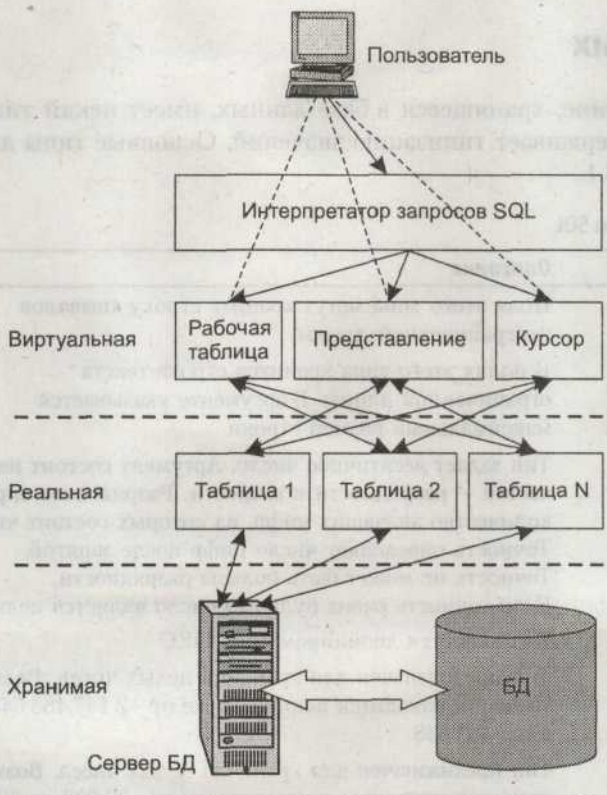


Рис. 6.1. Взаимодействие с базой данных при помощи SQL

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.

Этот язык предоставил пользователю простой и универсальный аппарат для доступа к данным и осуществления с ними различных операций. На рис. 6.1 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.

Типы данных

Каждое значение, хранящееся в базе данных, имеет некий тип. Язык SQL, конечно, поддерживает типизацию значений. Основные типы данных приведены в табл. 6.1.

Таблица 6.1. Типы SQL

Название	Описание
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

Название	Описание
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
DATA	Тип предназначен для хранения даты
TIME	Тип предназначен для хранения времени

Запросы к отдельным таблицам

Достаточно распространенной является задача получения данных из одной или нескольких таблиц и формирования на их основе каких-либо отчетов. В данном разделе будут изложены базовые понятия SQL и способы создания соответствующих запросов.

Использование выражения SELECT

Команда SELECT представляет собой выражение, инициирующее выполнение запроса. В данном случае запрос является командой на получение данных. Выражение SELECT имеет строго определенный формат:

```
SELECT[[ALL] | DISTINCT]{ * | элемент_SELECT [,элемент_SELECT] ...}
FROM{базовая_таблица | представление} [псевдоним]
[,{базовая_таблица | представление} [псевдоним]] ...
[WHEREусловие]
[GROUP BY название поля(полей) [HAVING фраза]];
```

Можно рассмотреть пример использования выражения SELECT при помощи соответствующего тестового приложения. На основной форме нужно разместить два компонента TADOConnection, компоненты TADOQuery, TADOCommand, TDataSource, TMemo, две кнопки TRadioButton, компонент TCheckBox и одну обычную кнопку. Для компонента TADOConnection1 надо настроить соединение с базой DBDEMOS (dbdemos.mdb), а для TADOConnection2 — с базой Northwind сервера SQL Server 2000. Компоненты TADOQuery и TADOCommand надо связать с компонентом TDataSource, а его, в свою очередь, с компонентом TDBGrid.

Листинг 6.1. Обработчик события кнопки

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
try
with ADOQuery1 do begin
```

продолжение ⇩

Листинг 6.1 *(продолжение)*

```
if RadioButton1.Checked then begin
Connection:=ADOConnection1;
end;
if RadioButton2.Checked then begin
Connection:=ADOConnection2;
end;
Close;
SQL.Clear;
SQL.Text:=Memo1.Lines.Text;
Open;
end;
except
ShowMessage('SQL-запрос введен неверно!');
end;
end;
```

При выборе того или иного сервера будет производиться соединение при помощи соответствующего компонента TADOConnection. На рис. 6.2 показано окно программы.

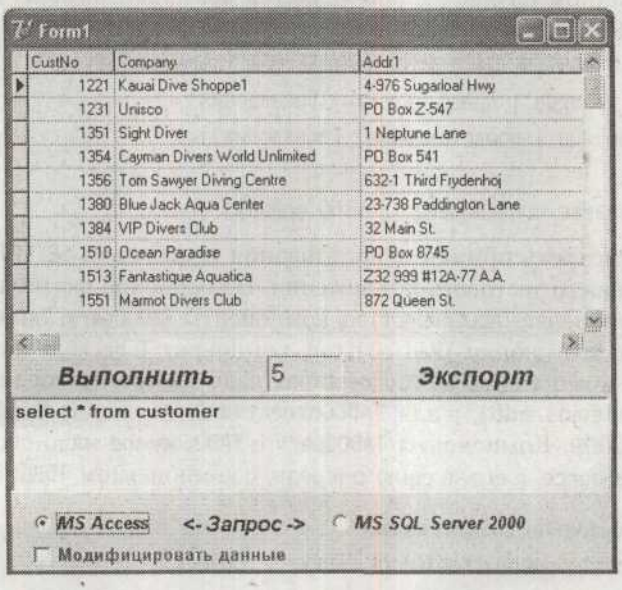


Рис. 6.2. Окно программы, выполняющей SQL-запросы

В поле ввода нужно ввести текст запроса:

```
SELECT * FROM Customer
```

При выполнении этой команды из таблицы Customer будут возвращены все записи. Выражение SELECT является ключевым словом, сообщающим базе данных о том, что эта команда является запросом. Далее следует список полей, которые будут возвращены в запросе, либо символ *, указывающий на то, что должны быть возвращены все поля. Ключевое слово FROM должно присутствовать в каждом запросе. После него указывается имя таблицы, к которой будет осуществлен запрос.

Можно использовать и другой запрос:

```
SELECT Country, City, Company FROM Customer
```

Результат выполнения запроса показан в табл. 6.2. Поля таблицы указаны не по порядку их реального следования. То есть значения полей будут выводиться в том порядке, который указан при составлении запроса.

Таблица 6.2. *Пример выборки значений нескольких полей*

Country	City	Company
US	Kapaa Kauai	Kauai Dive Shoppe1
Bahamas	Freeport	Unisco
Cyprus	Kato Paphos	Sight Diver
British West Indies	Grand Cayman	Cayman Divers World Unlimited
US Virgin Islands	Christiansted	Tom Sawyer Diving Centre

Выборка по условию

Выборку по условию реализует оператор WHERE. Оператор является частью выражения SELECT и служит для задания условий отбора записей в результирующий набор. В ходе выполнения запроса происходит проверка всех записей на соответствие условию отбора. В качестве примера можно привести довольно простой запрос:

```
SELECT State, City, Company FROM Customer
WHERE State = 'CA'
```

При обработке запроса был производен отбор всех записей, поле State которых имеет значение CA. В табл. 6.3 представлен результат выполнения запроса.

Таблица 6.3. Выборка по условию, налагаемому на поле State

State	City	Company
CA	Santa Maria	San Pablo Dive Center
CA	San Jose	Underwater Sports Co
CA	Lomita	American SCUBA Supply
CA	Catalina Island	Catamaran Dive Club
CA	Downey	Diver's Grotto

Можно произвести выборку по совпадающим значениям полей. Например, необходимо найти компании, в которых телефон и факс имеют

один и тот же номер. Условие запроса в этом случае будет довольно простым:

```
SELECT Company, Phone, Fax FROM Customer
WHERE Phone = Fax
```

Результаты выполнения запроса приведены в табл. 6.4.

Таблица 6.4. Отбор записей по соответствию значений полей

Company	Phone	Fax
Jamaica SCUBA Centre	011-3-697043	011-3-697043
SCUBA Heaven	011-32-09485	011-32-09485

Как видно из текста запроса, отбор значений был произведен для записей, значения полей Phone и Fax в которых одинаковы.

Исключение повторяющихся значений

Для получения результатов без повторяющихся значений используется оператор DISTINCT. Проще всего увидеть результаты применения этого оператора на соответствующих примерах. Сначала можно выполнить запрос без оператора DISTINCT:

```
SELECT Country FROM Customer
```

Результат выполнения запроса приведен в табл. 6.5.

Таблица 6.5. Результат выполнения запроса без оператора DISTINCT

Country
US
Bahamas
Cyprus
British West Indies
US Virgin Islands
US
US Virgin Islands
US

Второй вариант запроса уже включает в себя этот оператор:

```
SELECT DISTINCT Country FROM Customer
```

Результат выполнения запроса приведен в табл. 6.6.

Легко заметить, что в табл. 6.6 отсутствуют повторяющиеся значения поля Country, присутствовавшие в табл. 6.5. Но в некоторых случаях исключение дубликатов может привести к потере ценной информации. Предположим, необходимо получить список покупателей, используя их полные имена. Но полное совпадение фамилии, имени и отчества не такое уж маловероятное

событие. Поэтому при использовании в запросе оператора DISTINCT часть информации может быть потеряна.

Таблица 6.6. Результат выполнения запроса с оператором DISTINCT

Country
Columbia
Cyprus
Fiji
Greece
Republic So. Africa
US
US Virgin Islands
Venezuela

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. 6.7. 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 6.7. 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. Этот запрос можно выполнить в тестовом приложении и убедиться в получении результата.

Операторы сравнения и логические операторы

Реляционным оператором называют математический символ, который указывает на определенный тип сравнения между двумя значениями. Реляционные операторы, или операторы сравнения, могут использоваться для определения того, отвечает ли данное значение какому-либо диапазону.

Логические операторы позволяют задать в запросе логические условия. Оператор AND реализует логическое «И». Оператор OR реализует логическое «ИЛИ». Выражение с его использованием, будет считаться истинным, если хотя бы одно из условий тоже истинно. Оператор NOT означает логическое отрицание. Его действие сводится к тому, что он инвертирует логическое условие, перед которым его располагают.

В качестве примера можно привести запрос, позволяющий выбрать сотрудников, получающих заработную плату в определенном численном промежутке. Данные будут извлекаться из таблицы Employee:

```
SELECT LastName, FirstName, Salary FROM Employee
WHERE Salary >= 25000 AND Salary <= 30000
```

В результате выполнения запроса возвращаются имена сотрудников с заработной платой от 25 до 30 тысяч включительно. В данном случае оператор AND используется для задания диапазона выбираемых значений. Результат выполнения запроса приведен в табл. 6.8.

Таблица 6.8. Запрос по диапазону значений

LastName	FirstName	Salary
Lambert	Kim	25 000
Johnson	Leslie	25 050
Forest	Phil	25 050
Papadopoulos	Chris	25 050
De Souza	Roger	25 500

Теперь можно изменить данный запрос. Можно отыскать всех сотрудников, поле PhoneExt которых имеет значение 22:

```
SELECT LastName, FirstName, Salary, PhoneExt FROM Employee
WHERE Salary >= 25000 AND Salary <= 30000 and PhoneExt = '22'
```

Результат выполнения данного запроса приведен в табл. 6.9.

Таблица 6.9. Запрос по диапазону с дополнительным условием

LastName	FirstName	Salary	PhoneExt
Lambert	Kim	25 000	22
Ichida	Yuki	25 689	22

Если же потребуется отыскать сотрудников, поле PhoneExt которых имеет значение, не равное 22, запрос будет незначительно изменен:

```
SELECT LastName, FirstName, Salary, PhoneExt FROM Employee
WHERE Salary >= 25000 AND Salary <= 30000 and not PhoneExt = '22'
```

Результат выполнения запроса представлен в табл. 6.10.

Таблица 6.10. Запрос по диапазону с условием

LastName	FirstName	Salary	PhoneExt
Johnson	Leslie	25 050	410
Forest	Phil	25 050	229
Papadopoulos	Chris	25 050	887
De Souza	Roger	25 500	288
Williams	Randy	28 900	892

Как видно из текста запроса, логическое условие NOT позволило исключить ненужные номера.

Теперь можно рассмотреть пример запроса с использованием оператора OR. Предположим, менеджеру понадобилось получить списки всех сотрудников по фамилии Johnson. Либо тех сотрудников, которые получают заработную плату более 45 000. Составить запрос будет нетрудно:

```
SELECT LastName, FirstName, Salary FROM Employee
WHERE LastName = 'Johnson' or Salary > 45000
```

Результаты обработки запроса приведены в табл. 6.11.

Таблица 6.11. Результаты выполнения запроса со смешанным условием

LastName	FirstName	Salary
Young	Bruce	55 500
Johnson	Leslie	25 050
Lee	Terri	45 332
Burbank	Jennifer M	45 332
Page	Mary	48 000

Стоит обратить внимание на действие оператора OR. В набор данных были включены записи, значение поля Salary которых превышало 40 000, и те записи, в которых поле LastName имело значение Johnson.

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. Результат выполнения запроса представлен в табл. 6.12.

Таблица 6.12. Результат использования оператора IN для числовых значений

LastName	FirstName	Salary
Nelson	Roberto	40 000
Young	Bruce	55 500
Lambert	Kim	25 000

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. Результат выполнения запроса представлен в табл. 6.13.

Таблица 6.13. Результат использования оператора IN для строковых значений

Company	City
Cayman Divers World Unlimited	Grand Cayman
Tom Sawyer Diving Centre	Christiansted
VIP Divers Club	Christiansted
Fisherman's Eye	Grand Cayman
Action Diver Supply	St. Thomas

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 для SQL Server 2000:

```
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. В табл. 6.14 приведен результирующий набор данных.

Таблица 6.14. Использование оператора BETWEEN

CustomerID	EmployeeID	ShipName
VINET	5	Vins et alcools Chevalier
HANAR	4	Hanari Carnes
VICTE	3	Victuailles en stock
SUPRD	4	Supremes delices
HANAR	3	Hanari Carnes

Следующий пример показывает, как можно выбрать номера заказов, сделанных за определенный промежуток времени от 04.07.1996 до 08.07.1996:

```
SELECT OrderID, OrderDate, ShipName FROM Orders
WHERE OrderDate BETWEEN '07.04.1996' AND '07.08.1996'
```

Дата записана в американском формате. В табл. 6.15 представлены найденные записи.

Таблица 6.15. Использование оператора BETWEEN для выбора значений по диапазону дат

OrderID	OrderDate	ShipName
10248	04.07.1996	Vins et alcools Chevalier
10249	05.07.1996	Toms Spezialitaten
10250	08.07.1996	Hanari Carnes
10251	08.07.1996	Victuailles en stock

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 '07.04.1996' AND '07.08.1996' AND Freight > 100
```

Результат выполнения запроса представлен в табл. 6.16.

Таблица 6.16. Использование оператора BETWEEN в композитном запросе

OrderID	OrderDate	ShipName	Freight
10255	12.07.1996	Richter Supermarket	148,33
10258	17.07.1996	Ernst Handel	140,51
10263	23.07.1996	Ernst Handel	146,06
10267	29.07.1996	Frankenversand	208,58
10270	01.08.1996	Wartian Herkku	136,54

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'
```

В табл. 6.17 приведен результат выполнения запроса. Как видно, была найдена только одна запись, удовлетворяющая условию.

Таблица 6.17. Использование оператора LIKE с поиском пропущенного символа

CompanyName	ContactName
Simons bistro	Jytte Petersen

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%'
```

Результат приведен в табл. 6.18.

Таблица 6.18. Использование оператора LIKE с поиском подстроки

CompanyName	ContactName
FISSA Fabrica Inter. Salchichas S.A	Diego Roel
Pericles Comidas clasicas	Guillermo Fernandez
Ricardo Adocicados	Janete Limeira
Richter Supermarket	Michael Holz

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. Результаты выполнения соответствующего запроса приведены в табл. 6.19:

```
SELECT CompanyName, ContactName FROM Customers
WHERE CompanyName LIKE '%R_c%'
```

Таблица 6.19. Использование оператора LIKE в поиске подстроки с пропущенным символом

CompanyName	ContactName
Drachenblut Delikatessen	Sven Ottlieb
FISSA Fabrica Inter. Salchichas S.A	Diego Roel
Hungry Owl All-Night Grocers	Patricia McKenna
Pericles Comidas clasicas	Guillermo Fernandez
Rattlesnake Canyon Grocery	Paula Wilson

Теперь можно ввести дополнительное условие отбора данных. К уже существующему критерию нужно добавить условие, находящее в поле ContactName последовательность символов «?en»:

```
SELECT CompanyName, ContactName FROM Customers
WHERE CompanyName LIKE '%R_c%' AND ContactName LIKE '%_en%'
```

В результате были отобраны две записи, показанные в табл. 6.20.

Таблица 6.20. Оператор LIKE с двойным условием отбора

CompanyName	ContactName
Drachenblut Delikatessen	Sven Ottlieb
Hungry Owl All-Night Grocers	Patricia McKenna

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 поля.

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. Результат выполнения запроса приведен в табл. 6.21:

```
SELECT AVG (Freight) AS Среднее, MIN (Freight) AS Мин, MAX (Freight) AS Макс,
SUM (Freight) AS Суммарное, COUNT (Freight) AS Количество FROM Orders
WHERE Freight > 300
```

Таблица 6.21. Использование агрегатных функций

Среднее	Мин	Макс	Суммарное	Количество
5 146 465	306,07	1007,64	18 012,63	35

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 Число_городов FROM Customers
```

В ходе выполнения запроса с оператором DISTINCT было зафиксировано 69 записей. Без использования оператора — 91. 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. 6.22.

Таблица 6.22. 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 ко-

нечный 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. 6.23.

Таблица 6.23. 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. Результат выполнения запроса приведен в табл. 6.24. Стоит обратить внимание на то, как были упорядочены записи.

Таблица 6.24. Использование оператора ORDER BY

CompanyName	ContactName	City
Furia Bacalhau e Frutos do Mar	Lino Rodriguez	Lisboa
Princesa Isabel Vinhos	Isabel de Castro	Lisboa
Around the Horn	Thomas Hardy	London
B's Beverages	Victoria Ashworth	London
Consolidated Holdings	Elizabeth Brown	London
Eastern Connection	Ann Devon	London

продолжение

Таблица 6.24 *(продолжение)*

CompanyName	ContactName	City
North/South	Simon Crowther	London
Seven Seas Imports	Hari Kumar	London
Berglunds snabbkop	Christina Berglund	Lulea
Victuailles en stock	Mary Saveley	Lyon
Bolido Comidas preparadas	Martin Sommer	Madrid

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, City ASC
```

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. 6.25 приведен результат выполнения запроса.

Таблица 6.25. 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 *объеди-*

нением 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. 6.26.

Таблица 6.26. 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 17.07.1998:

```
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 '03.17.1998' AND '07.17.1998'
```

Результат выполнения запроса представлен в табл. 6.27.

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 необ-

ходимо использовать временные имена, присваиваемые таблице. Эти имена называют псевдонимами:

```
SELECT FirstTable.CompanyName, SecondTable.CompanyName FROM Customers
FirstTable, Customers SecondTable
```

Таблица 6.27. 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

Следует обратить внимание на то, что было создано два псевдонима для одной таблицы. В запросе использовались псевдонимы `FirstTable` и `SecondTable`. Результат выполнения запроса приведен в табл. 6.28.

Таблица 6.28. Объединение одной таблицы

CompanyName	CompanyName_1
Antonio Moreno Taqueria	Antonio Moreno Taqueria
Around the Horn	Antonio Moreno Taqueria
Berglunds snabbkop	Antonio Moreno Taqueria
Blauer See Delikatessen12	Antonio Moreno Taqueria
Blondesddsl pere et fils	Antonio Moreno Taqueria

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` возвращает массив значений, а скалярный оператор равенства — только одно.

Выполнение запросов с вложенными подзапросами всегда начинается с подзапроса, располагающегося на самом нижнем уровне. В рассматриваемом под-

запросе отыскивается индивидуальный номер сотрудника EmployeeID по его имени и фамилии. Основной запрос принимает найденное значение в качестве параметра. В табл. 6.29 представлен результат выполнения запроса.

Таблица 6.29. 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) > 100)
ORDER BY FirstName, LastName
```

Во вложенном запросе производится отбор идентификаторов работников, встречающихся в таблице более 90 раз. В табл. 6.30 приведен результирующий набор данных.

Таблица 6.30. 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 EXIST и NOT EXIST 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. 6.31 приведен результат выполнения запроса.

Таблица 6.31. 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 табл. 6.32 представлен результат выполнения запроса.

В ходе выполнения запроса в результирующую таблицу (табл. 6.32) были включены записи из двух таблиц.

Таблица 6.32. Использование оператора UNION

CustomerID
ALFKI
ANATR
ANTON
AROUT
BERGS
BLAUS

Модификация данных

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 INSERT

Все записи в таблицы добавляются с использованием команды INSERT. В самой простой форме SQL-запрос INSERT использует следующий синтаксис:

```
INSERT INTO <table name> (<field>, < field2>, . . .)
VALUES ( <value1>, <value2> . . .);
```

Если требуется добавить записи в таблицу Employees, то понадобится выполнить несколько запросов:

```
INSERT INTO Employees (LastName, FirstName) VALUES ('Федорович', 'Сергей')
INSERT INTO Employees (LastName, FirstName) VALUES ('Сергеевич', 'Сергей')
INSERT INTO Employees (LastName, FirstName) VALUES ('Петрович', 'Владимир')
INSERT INTO Employees (LastName, FirstName) VALUES ('Кузьмич', 'Михаил')
INSERT INTO Employees (LastName, FirstName) VALUES ('Васильевич', 'Евгений')
```

Этот запрос можно выполнить в соответствующем приложении. В ходе выполнения запроса в поля LastName и FirstName будут добавлены новые данные. Эти поля указаны в тексте запроса в скобках. Значения, присваиваемые полям, должны следовать в запросе в том же порядке, что и список полей. Отобразить новые значения можно при помощи нового запроса, результат выполнения которого представлен в табл. 6.33:

```
SELECT FirstName AS Имя, LastName AS Отчество FROM Employees
ORDER BY FirstName DESC
```

Таблица 6.33. Результат вставки группы записей в таблицу Employees

Имя	Отчество
Сергей	Федорович
Сергей	Сергеевич
Владимир	Петрович
Михаил	Кузьмич
Евгений	Васильевич

Использование оператора UPDATE

Команда UPDATE используется для изменения данных в таблицах. Запрос на модификацию данных имеет следующую структуру:

```
UPDATE <table name>
SET field1 = <value1>, field2 = <value2> . . .
WHERE условие
```

Непосредственно после оператора UPDATE указывается имя таблицы, затем следует оператор SET, после которого перечисляются поля и присваиваемые им значения. Можно рассмотреть самый простой запрос модификации данных:

```
UPDATE Customers
SET CompanyName = Название компании
```

В ходе выполнения этого запроса поле CompanyName во всех записях получит значение «Название компании», так как не было указано условие для выбора изменяемых записей. Это упущение исправляется в следующем запросе:

```
UPDATE Customers
SET CompanyName = 'Название компании'
WHERE City = 'London'
```

Измененные данные можно просмотреть при помощи нового запроса, результат выполнения которого представлен в табл. 6.34:

```
SELECT CompanyName, ContactName, City FROM Customers
WHERE City = 'London'
```

Таблица 6.34. Изменение значения поля группы записей по условию

CompanyName	ContactName	City
Название компании	Thomas Hardy	London
Название компании	Victoria Ashworth	London
Название компании	Elizabeth Brown	London
Название компании	Ann Devon	London
Название компании	Simon Crowther	London
Название компании	Hari Kumar	London

Можно рассмотреть один из старых примеров, чтобы использовать более сложный запрос. Например, попробовать как-нибудь выделить тех работников, которые обслужили более ста кораблей, присвоив полю TitleOfCourtesy значение Лидер. В этом случае критерием для отбора записей будут служить значения, возвращаемые в подзапросе:

```
UPDATE Employees
SET TitleOfCourtesy = 'Лидер'
WHERE EmployeeID IN
(SELECT EmployeeID FROM Orders GROUP BY EmployeeID
HAVING COUNT(EmployeeID) > 100)
```

При помощи соответствующего запроса можно отыскать все измененные поля, список которых приведен в табл. 6.35:

```
SELECT TitleOfCourtesy, Firstname, LastName FROM Customers
WHERE TitleOfCourtesy = 'Лидер'
```

Таблица 6.35. Изменение значения поля группы

TitleOfCourtesy	FirstName	LastName
Лидер	Nancy	Davolio
Лидер	Janet	Leverling
Лидер	Margaret	Peacock
Лидер	Laura	Callahan

Использование DELETE

Оператор DELETE используется для удаления записей из таблицы. Структура запроса выглядит следующим образом:

```
DELETE FROM <table name>
WHERE условие отбора записей
```

Структура этого запроса очень проста. Ее иллюстрирует несложный запрос

```
DELETE FROM Employees
```

В ходе выполнения этого запроса будут удалены все записи, так как не задано условие их отбора. Лучше выполнить другой запрос:

```
DELETE FROM Employees
WHERE FirstName = 'Владимир'
```

Будут удалены те записи, у которых поле FirstName имеет значение Владимир. Ранее в таблицу Employees было добавлено несколько подобных записей. Теперь их можно удалить. Стоит исходить из того, что записи имеют это значение только в полях FirstName и LastName. Соответственно, нам необходимо удалить те записи, поля которых не имеют значений в других полях. Соответствующий запрос можно придумать довольно быстро:

```
DELETE FROM Employees
WHERE Title is NULL and City is NULL
```

Если поле какой-либо записи не имеет значения, то, как правило, ему присваивается значение NULL. Условие IS NULL используется для того, чтобы проверить, содержит ли поле значение NULL. Если содержит, то оператор возвращает значение True.

Другие операторы SQL

В этом разделе будут кратко рассмотрены остальные операторы языка SQL, которые позволяют выполнять более сложные операции с данными.

Работа с представлениями

Представления являются таблицами, содержание которых выбирается из других таблиц. Они работают в SQL-запросах так же, как и обычные таблицы. Структура запроса, создающего представление, довольно проста:

```
CREATE VIEW имя_представления
AS подзапрос
<с условием>
```

После предложения CREATE VIEW следует имя представления, а затем указывается подзапрос, в котором определяются возвращаемые поля и условия отбора записей. Следующий запрос создает простое представление:

```
CREATE VIEW SimpleView
AS SELECT * FROM Employees
WHERE City = 'London'
```

Условием отбора записей в представление явилось соответствие значений поля City значению London. Это представление потом можно использовать как обычную таблицу:

```
SELECT * FROM SimpleView
```

В ходе выполнения запроса были выбраны все записи представления. Если из преставления нужно извлечь только поля City, FirstName и LastName, следует применить другой запрос. Результаты выполнения этого запроса показаны в табл. 6.36:

```
SELECT City, FirstName, LastName FROM SimpleView
```

Таблица 6.36. Содержимое представления

City	FirstName	LastName
London	Steven	Buchanan
London	Michael	Suyama
London	Robert	King
London	Anne	Dodsworth

Можно создать другое представление, ограничив список возвращаемых полей при его описании. Предположим, что необходимо получить название то-

вара из поля ProductName и тип его упаковки из поля QuantityPerUnit, которые хранятся в таблице Products:

```
CREATE VIEW SecondView
AS SELECT ProductName, QuantityPerUnit FROM Products
```

Соответствующий запрос с условием поможет получить записи из этого представления:

```
SELECT * FROM SecondView
WHERE ProductName like '%Cha%'
```

В табл. 6.37 представлен результат, возвращенный запросом.

Таблица 6.37. Работа с представлением

ProductName	QuantityPerUnit
Chai	10 boxes x 20 bags
Chang	24 – 12 oz bottles
Chartreuse verte	750 cc per bottle

Используя представления, можно даже модифицировать данные. Эта возможность иллюстрируется следующим запросом:

```
UPDATE SecondView
SET QuantityPerUnit = '1'
WHERE ProductName like '%Cha%'
```

В ходе выполнения запроса полю QuantityPerUnit будет присвоено значение 1 для тех записей, в поле ProductName которых встретилась подстрока «Cha». После этого можно извлечь данные из представления:

```
SELECT * FROM SecondView
WHERE ProductName like '%Cha%'
```

В табл. 6.38 представлен результат выполнения модифицирующего представления.

Таблица 6.38. Модификация данных с использованием представления

ProductName	QuantityPerUnit
Chai	1
Chang	1
Chartreuse verte	1

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> [(<size>)],
<column name2 > <data type> [(<size>)] . . .)
```

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. 6.39.

Таблица 6.39. 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	Камшловский комбинат	55	21.04.2004
Плоскогубцы	Камшловский 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 индексирова-

ние. В качестве примера можно построить индекс по полю Firma для созданной таблицы:

```
CREATE INDEX SkladIndex ON Sklad (Firma)
```

Для того чтобы сделать индекс уникальным, необходимо перед ключевым словом INDEX использовать директиву UNIQUE. С учетом этого запрос несколько изменится:

```
CREATE UNIQUE INDEX SkladIndex ON Sklad (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 Sklad.SkladIndex
```

После команды DROP INDEX указываются имя таблицы и имя индекса.

На этом можно завершить обзор возможностей языка SQL.

7 УРОК СУБД MS Access

В этой главе будут изложены основные методы работы с СУБД MS Access. Несмотря на то что файлы этой базы данных очень быстро увеличивают свой объем во время работы, Access прекрасно подходит для написания небольших приложений баз данных.

Типы данных

Любая СУБД поддерживает собственный набор типов данных. Как правило, СУБД допускает преобразование данных из одного типа в другой. СУБД Access предоставляет пользователю довольно много типов данных:

- Текстовый тип создает символьное поле, которое может иметь размер до 255 символов.
- MEMO-поле предназначено для хранения больших объемов информации. Если поле заполняется двоичными данными, то оно может занимать до 65 535 знаков. Если поле заполняется символами и цифрами, то размер поля не ограничен.
- Байтовый тип предназначен для хранения целых чисел в диапазон от 0 до 255.
- Действительный тип предназначен для хранения чисел.
- Целочисленный тип реализует целые числа в диапазоне от –32 768 до 32 767.
- Длинное целое число может находиться в диапазоне от –2 147 483 648 до 2 147 483 647.
- Одинарное число с плавающей точкой с отрицательным значением находится в диапазоне от $-3{,}402823E^{38}$ до $-1{,}401298E^{-45}$, а с положительным значением — в диапазоне от $1{,}401298E^{-45}$ до $3{,}402823E^{38}$.
- Двойное число с плавающей точкой требует для своего хранения восемь байтов. Отрицательные значения этого типа располагаются в диапазоне от $1{,}79769313486231E^{308}$ до $-4{,}94065645841247E^{-324}$. Положительные значения располагаются в диапазоне от $4{,}94065645841247E^{-324}$ до $1{,}79769313486231E^{308}$.

- Тип дата/время позволяет создавать поля, содержащие данные о календарной дате или о времени.
- Денежный тип данных предназначен для хранения денежных сумм. Может иметь от одного до четырех знаков в дробной части и до пятнадцати — в целой.
- Счетчик представляет собой автоинкрементное поле.
- Логический тип может содержать только два значения — True или False.
- Поле объекта OLE предназначено для хранения OLE-объектов. Данные в этом поле могут иметь размер до одного гигабайта.

Создание базы данных

Для создания базы данных в Access достаточно выполнить команду меню **Файл ▸ Создать**. В правой части появившегося диалогового окна следует выбрать пункт **Новая база данных**. В результате будет отображено диалоговое окно выбора файла, в котором будет предложено указать существующий файл либо ввести название нового. Для создания тестового приложения необходимо создать новый файл с именем TeachDB. На рис. 7.1 показано основное окно определения структуры новой базы данных.

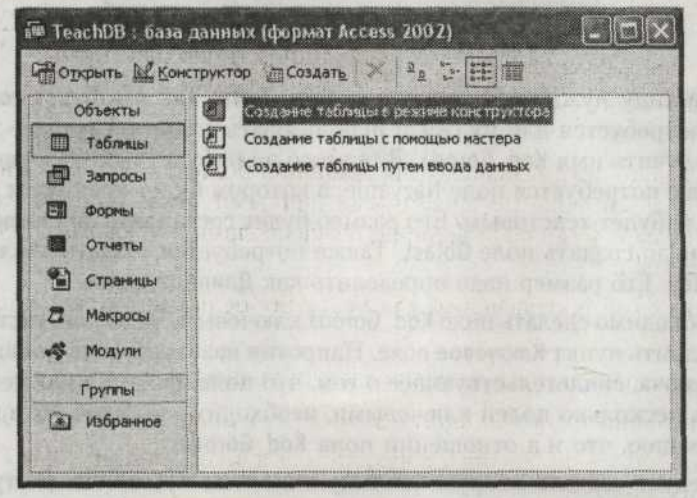


Рис. 7.1. Окно центра управления базой данных

Создание таблиц

Для создания таблиц следует нажать на кнопку **Создание таблицы в режиме конструктора**. Появится окно, показанное на рис. 7.2. В этом окне можно ввести название поля, указать его тип, а для символьных полей еще и задать размер поля.

В качестве примера нужно определить структуру двух таблиц. В первой таблице будут содержаться сведения о городах. Будут указаны область, количество жителей города, максимально низкая и самая высокая температура, зафиксированная метеорологами. Во второй таблице будут содержаться данные о состоянии погоды в городах, упорядоченные по дате наблюдения.

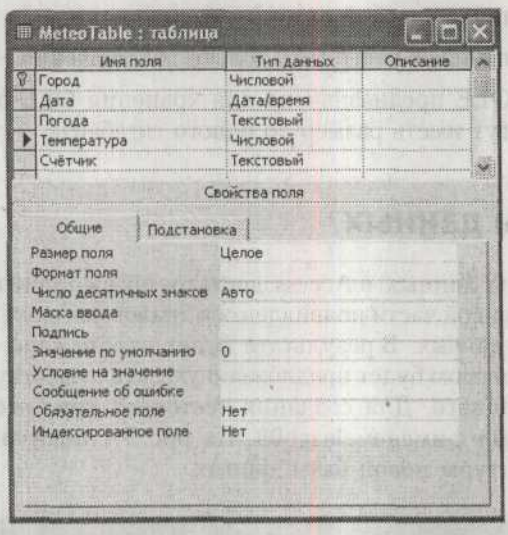


Рис. 7.2. Окно конструктора таблицы

Первую таблицу нужно сохранить с именем Town. Так как будет создано две таблицы, потребуется для их связи использовать соответствующее поле. Оно должно получить имя Kod_Goroda. Для этого поля будет использован тип Числовой. Также потребуется поле Nazvanie, в котором будет храниться имя города. Это поле будет текстовым. Его размер будет составлять 20 символов. Точно так же надо создать поле Oblast. Также потребуется создать числовое поле Chislo_Giteley. Его размер надо определить как Длинное целое.

Теперь необходимо сделать поле Kod_Goroda ключевым. В его контекстном меню надо выполнить пункт Ключевое поле. Напротив названия поля появится изображение ключа, свидетельствующее о том, что поле является ключевым. Чтобы сделать несколько полей ключевыми, необходимо выбрать их и проделать ту же операцию, что и в отношении поля Kod_Goroda.

Теперь следует создать вторую таблицу с именем MeteoTable. В эту таблицу необходимо добавить автоинкрементное поле Counter с типом Счетчик. Потребуется также поле Key строкового типа размером 10 символов. Его тоже необходимо сделать ключевым. Также потребуется поле Kod_Goroda, определение которого можно взять из первой таблицы. Оно будет использоваться в качестве связующего поля.

Осталось лишь добавить поле Data типа Дата/время и определить его формат как Краткий формат даты. Также потребуется поле Pogoda текстового типа размером 30 символов. А последнее поле в этой таблице будет иметь название

Temperature. Так как температура может быть как отрицательной, так и положительной, поле должно получить числовой тип, а размер поля надо объявить как Целое.

Определение ссылочной целостности

Для того чтобы определить ссылочную целостность между таблицами, необходимо выбрать пункт главного меню Сервис ▸ Схема данных. В результате будет активировано диалоговое окно Добавление таблицы. В нем нужно выбрать доступные таблицы, расположенные на одноименной вкладке. Затем нужно выбрать поле Kod_Goroda таблицы Town и, удерживая кнопку мыши, перетащить его на одноименное поле таблицы MeteoTable. Появится окно, показанное на рис. 7.3.

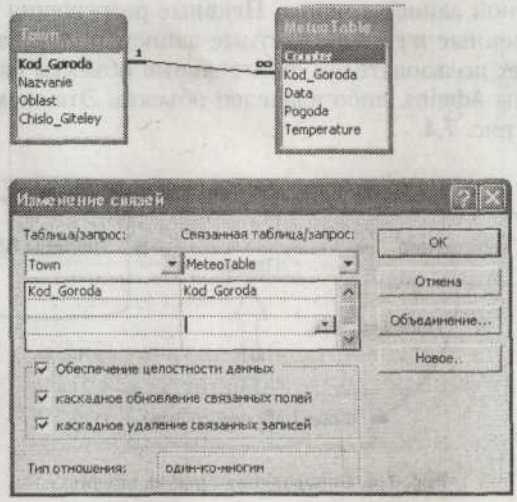


Рис. 7.3. Определение отношений ссылочной целостности

В левом списке Таблица/запрос окна Изменение связей указывается родительская таблица Town и выбирается ключевое поле связи. В правом списке, соответственно, выбирается подчиненная таблица и поле связи. Также потребуется установить флажки Обеспечение целостности данных, Каскадное обновление связанных полей и Каскадное удаление связанных записей.

Так как на данный момент таблица является пустой, операция пройдет без возникновения сообщений об ошибках. Созданную схему данных потребуется сохранить.

Администрирование базы данных

Администратор СУБД Access может сжимать таблицы, выполнять резервное копирование, репликацию баз данных, выдачу разрешений на работу с объек-

тами определенным пользователям и многие другие процедуры. Некоторые из этих операций могут быть выполнены при помощи объектов COM.

Управление рабочими группами и распределение прав

Под *рабочей группой* обычно понимают совокупность учетных записей пользователей, входящих в состав учетной записи группы. В учетной записи указывается набор прав для работы с объектами базы данных. Сведения о членах рабочих групп и их правах хранятся в файле рабочей группы. В момент открытия базы данных производится получение информации о правах данного пользователя, и в соответствии с ними на таблицы накладываются ограничения.

Права на доступ к объектам базы данных складываются из явных и неявных прав. Явными правами называются ограничения, непосредственно присвоенные учетной записи пользователя. Неявными считаются разрешения, присвоенные самой учетной записи группы. Неявные разрешения оказывают влияние на все включенные в группу учетные записи пользователей. Изменять разрешения других пользователей на отдельные объекты базы данных могут либо члены группы Admins, либо владелец объекта. Эта схема распределения прав показана на рис. 7.4.

Рис. 7.4. Определение прав на объект

Приложение посылает запрос на доступ к таблице, после этого производится проверка и определяются права доступа. При создании файла базы данных Access автоматически создает файл рабочих групп. Лучше всего при создании базы данных явно определить права доступа к ней. Для этого нужно выполнить команду главного меню Сервис ▸ Защита ▸ Администратор рабочих групп. В результате будет активировано диалоговое окно, показанное на рис. 7.5.

Кнопка Создать служит для вызова окна диалога создания файла рабочей группы, в котором следует указать имя пользователя, идентификационный номер и его расположение. Кнопка Связать позволяет установить связь с существующим файлом рабочей группы. Для создания учетных записей пользователей и рабочих групп достаточно выбрать пункт главного меню Сервис ▸ Защита ▸ Пользователи и группы. Диалоговое окно, активируемое этой командой, показано на рис. 7.6.

На вкладке Пользователи можно создать учетную запись для нового пользователя. Для этого достаточно нажать кнопку Создать. Появится диалоговое окно, показанное на рис. 7.7.

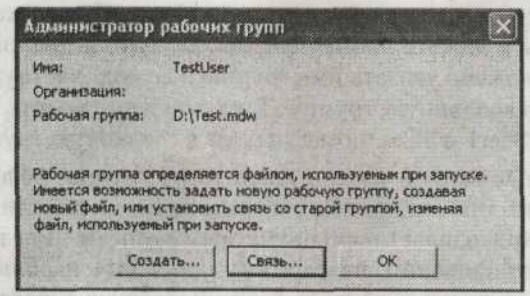


Рис. 7.5. Окно администратора рабочих групп

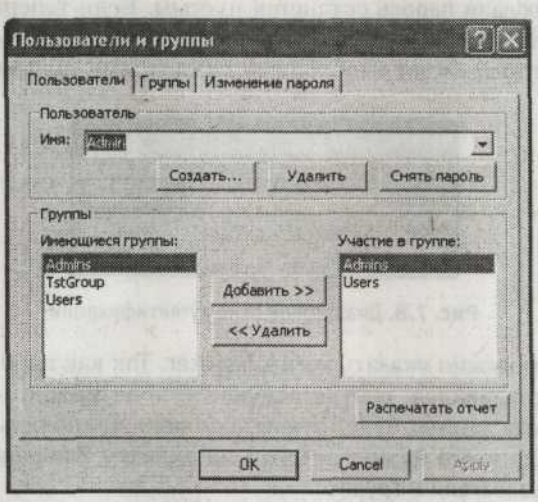


Рис. 7.6. Создание учетных записей пользователей и рабочих групп

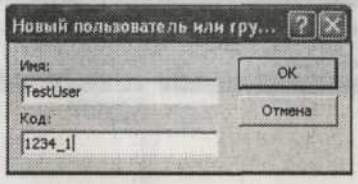


Рис. 7.7. Диалог создания учетной записи

На рисунке показано создание учетной записи TestUser. Код используется для идентификации учетной записи в группе. После создания новой учетной записи имя пользователя можно выбрать из списка Имя диалогового окна Пользователи и группы. В списке Имеющиеся группы показаны все существующие группы пользователей. В списке Участие в группах указывается, в какую группу входит данная учетная запись. Для тестового приложения необходимо выбрать Admins из списка Имеющиеся группы и нажать кнопку Добавить. В списке Участие в группах появится группа Admins. По умолчанию каждой учетной записи присваивается членство в группе Users.

Также потребуется создать новую группу TestGroup. Для этого нужно перейти на вкладку Группы и нажать кнопку Создать. Появится диалоговое окно, в котором будет предложено указать имя группы и ее код. Учетную запись TestUser надо поместить в созданную группу. Также нужно создать учетную запись пользователя TestUser1 и тоже поместить ее в созданную группу.

Обе новые учетные записи нужно удалить из группы Admins. После этого можно определить для созданных учетных записей пароли. По умолчанию СУБД Access не присваивает пароли учетным записям. Для того чтобы включить режим аутентификации, необходимо присвоить пароль учетной записи Admin. Для этого надо перейти на вкладку Изменение пароля и ввести в поля Новый пароль и Подтверждение пароль для учетной записи. Если пароль не был задан, то поле Текущий пароль останется пустым. Если теперь закрыть Access, а потом снова открыть базу данных, то появится диалоговое окно, показанное на рис. 7.8, в котором будет необходимо указать имя пользователя и пароль.

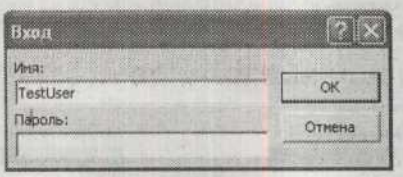


Рис. 7.8. Диалоговое окно аутентификации

В качестве имени нужно указать логин TestUser. Так как пароль для этой учетной записи не создавался, соответствующее поле можно оставить пустым. После того как база данных будет открыта, можно установить пароль для этого пользователя. Для этого нужно перейти на вкладку Изменение пароля диалогового окна Пользователи и группы.

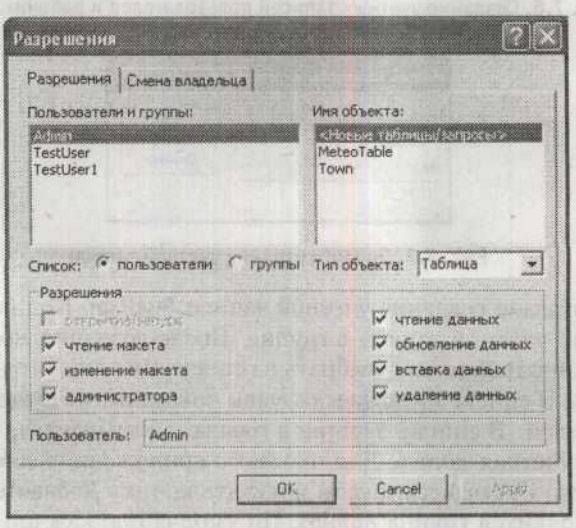


Рис. 7.9. Установка прав учетных записей

Эта учетная запись не входит в группу Admins, следовательно, она не имеет прав на то, чтобы производить какие-либо действия с учетными записями других пользователей.

Теперь нужно задать разрешения на работу с объектами конкретным учетным записям. Для этого нужно выполнить команду главного меню Сервис ▸ Защита ▸ Разрешения. На вкладке Разрешения располагаются два списка. В левом содержатся ученые записи, а в правом — объекты базы данных. Сначала нужно выбрать учетную запись Admin, а потом — таблицу MeteoTable, как это показано на рис. 7.9.

В группе флажков Разрешения нужно установить флажок Администратор, что приведет к установке всех разрешений на работу с объектом. Точно так же надо поступить с таблицей Town. Для пользователя TestUser в обеих таблицах надо отметить пункты чтение данных и вставка данных. При этом автоматически будет установлен флажок чтение макета. Эта возможность открывает доступ к структуре базы данных, но не позволяет изменять ее.

Сжатие и восстановление файлов Access

В процессе функционирования файлы Access имеют свойство быстро набирать объем. Это связано с тем, что во время работы побочный «мусор» не удаляется из базы физически. Например, если какая-либо запись была удалена, она исключается из таблицы на логическом уровне, но физически все еще занимает место в файле. Из-за этого происходит фрагментация страниц данных, что снижает производительность базы в целом.

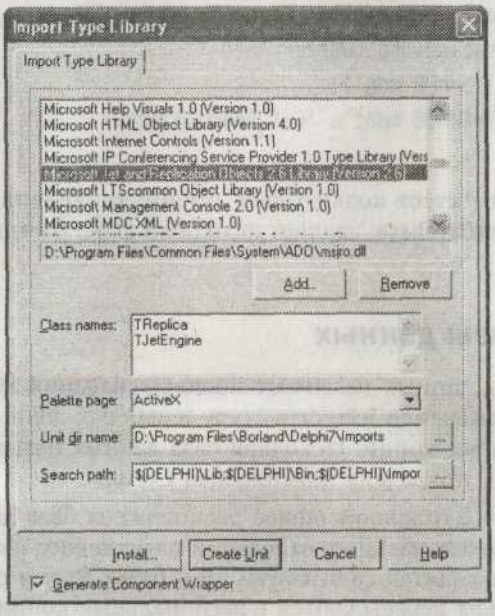


Рис. 7.10. Включение библиотеки типов JRO в проект Delphi

Сжать базу данных можно при помощи команды главного меню Сервис ▸ Служебные ▸ Сжать и восстановить базу данных. Также можно воспользоваться библиотекой расширения ADO – Microsoft Jet and Replication Objects (JRO). Чтобы получить доступ к этой библиотеке, необходимо включить ее библиотеку типов в проект Delphi. Для этого следует выбрать пункт главного меню Project ▸ Import Type Library. Появившееся диалоговое окно показано на рис. 7.10.

В основном списке необходимо выбрать элемент Microsoft Jet and Replication Objects 2.6 Library (Version 2.6), а затем нужно нажать кнопку Install для создания пакета. Объект JetEngine представляет механизм Jet Database Engine. При помощи одного из его методов CompactDatabase можно сжать базу данных. Метод принимает в качестве параметров строки соединения с исходной и конечной базами данных.

Проще всего рассмотреть пример использования этого метода. В Delphi нужно создать новый проект и поместить на форму кнопку и компонент TJetEngine. В листинге 7.1 приведен код, демонстрирующий работу с этим объектом.

Листинг 7.1. Сжатие базы данных

```
procedure TForm1.CompactBtnClick(Sender: TObject);
var JetEng : JetEngine;
    Provider, Src, Dst : String;
begin
  JetEng:=CoJetEngine.Create;
  Provider:='Provider=Microsoft.Jet.OLEDB.4.0;';
  Src:=Provider+'Data Source='+'C:\TeachDB.mdb;
  Dst:=Provider+'Data Source='+'C:\TeachDB1.mdb';
  JetEng.CompactDatabase(Src, Dst);
  DeleteFile('C:\ TeachDB.mdb');
  RenameFile('C:\ TeachDB.mdb1', 'C:\ TeachDB.mdb')
end;
```

Приведенный выше кусок кода создает экземпляр объекта JetEngine, вызывает его метод CompactDatabase, удаляет исходный файл базы и переименовывает созданный файл.

Репликация базы данных

Репликацией базы данных называют процесс создания копий базы данных, в ходе которого изменения в ее структуре и наборах данных асинхронно распространяются на все копии. Исходная база данных называется *главной репликой*. На рис. 7.11 приведена поясняющая схема.

Предположим, что в головном офисе расположена база данных. Обращение к базе данных из региональных отделений затруднено, поэтому в региональных отделениях находятся свои копии этой базы. Время от времени изменения из главной базы пересылаются в региональные копии. Этот процесс называется *синхронизацией*. Синхронизация может быть однонаправленной

и двунаправленной. При однонаправленной синхронизации изменения передаются от главной реплики к дочерним, при двунаправленной — передача производится в обоих направлениях.

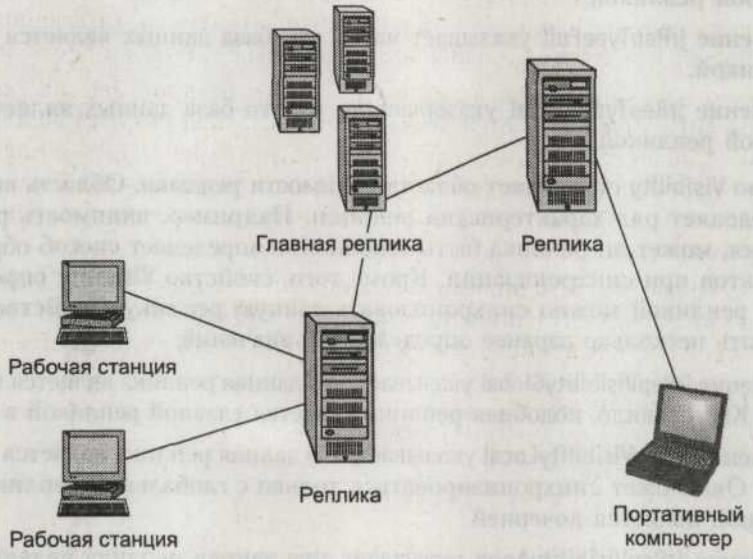


Рис. 7.11. Репликация базы данных

Реплика может быть полной и частичной. Полная реплика включает в себя все данные и объекты, помеченные как реплицируемые. В частичную реплику включаются не все объекты. Возможности MS Access не позволяют удобно работать с частичными репликами, поэтому стоит рассмотреть создание полных реплик.

Для того чтобы создать реплику, необходимо выполнить команду меню Сервис ▸ Репликация ▸ Создать реплику. Появится диалоговое окно выбора файла, в котором следует указать местоположение реплики и ее название. Для того чтобы синхронизировать реплики, нужно выполнить команду основного меню Сервис ▸ Репликация ▸ Синхронизация.

Чтобы выполнить репликацию базы данных Access из обычного приложения, можно воспользоваться объектом Replica библиотеки Microsoft Jet and Replication Objects. Свойство этого объекта ActiveConnection содержит ссылку на объект соединения ADO Connection. Чтобы присвоить свойству значение, необходимо использовать метод Set_ActiveConnection, который в качестве параметра принимает строку соединения. Так как репликация может быть двунаправленной, часто возникают ситуации, когда запись в разных репликах содержит разную информацию. В этом случае возникает конфликт. Свойство ConflictTables типа Recordset возвращает набор данных, содержащий поля TableName и ConflictTableName, в которых указываются наборы данных, между которыми возник конфликт в ходе процесса синхронизации.

Используя свойство ReplicaType, можно выяснить тип реплики. Свойство может принимать несколько заранее предопределенных значений:

- Значение jrRepTypeNotReplicable устанавливается по умолчанию. Оно указывает, что база данных не подвергалась реплицированию.
- Значение jrRepTypeDesignMaster указывает на то, что база данных является главной репликой.
- Значение jrRepTypeFull указывает на то, что база данных является полной репликой.
- Значение jrRepTypePartial указывает на то, что база данных является частичной репликой.

Свойство Visibility определяет область видимости реплики. Область видимости определяет ряд характеристик реплики. Например, видимость реплики указывает, может ли реплика быть основной, и определяет способ обработки конфликтов при синхронизации. Кроме того, свойство Visibility определяет, с какой репликой можно синхронизовать данную реплику. Свойство может принимать несколько заранее определенных значений:

- Значение jrRepVisibilityGlobal указывает, что данная реплика является глобальной. Как правило, подобная реплика является главной репликой в наборе.
- Значение jrRepVisibilityLocal указывает, что данная реплика является локальной. Она может синхронизироваться только с глобальной репликой, для которой является дочерней.
- Значение jrRepVisibilityAnon указывает, что данная реплика является анонимной. Может синхронизироваться только с родительской репликой. Реплика не несет в себе системной информации, из-за чего ее размер меньше, чем у локальной.

Локальные и анонимные реплики имеют нулевой приоритет. При возникновении конфликтов изменения, внесенные в записи в этих репликах, теряются. Если конфликта не происходит, то в процессе синхронизации изменения пересылаются главной реплике.

Каждой реплике назначается определенный приоритет в диапазоне от 0 до 100, значение которого можно получить из свойства Priority, доступного только для чтения. Последующим репликам в момент создания назначается приоритет, равный 90% от значения приоритета предыдущей созданной реплики. Приоритет реплики определяет учет изменений записей в случае конфликта при синхронизации. На рис. 7.12 приведен простой пример.

Главная реплика, расположенная на узле А, обладает максимальным приоритетом. Она имеет две дочерних реплики на узлах В и Б, имеющих приоритеты 95 и 90% соответственно. Если некая запись будет изменена в обеих дочерних репликах, то возникнет конфликт. Принято будет то значение, которое имеет больший приоритет, то есть реплика, расположенная на узле В.

В свойстве RetentionPeriod указывается число дней, в течение которого будет храниться реплика. Свойство может принимать в качестве параметра число от 5 до 32 000 и может быть установлено только для главной реплики базы данных. Метод CreateReplica используется для создания новой реплики текущей реплицируемой базы данных. Параметр ReplicaName содержит имя и путь

создаваемой реплики, а параметр Description содержит описание создаваемой реплики. Остальные параметры метода являются опциональными. Параметр ReplicaType определяет тип создаваемой реплики, параметр Visibility определяет область видимости реплики, а параметр Priority задает приоритет реплики. Параметр Updatability определяет возможность обновления реплики. По умолчанию устанавливается значение jrRepUpdFull, указывающее, что реплика может обновляться. Если свойству будет установлено значение jrRepUpdReadOnly, то реплика будет иметь доступ только на чтение.

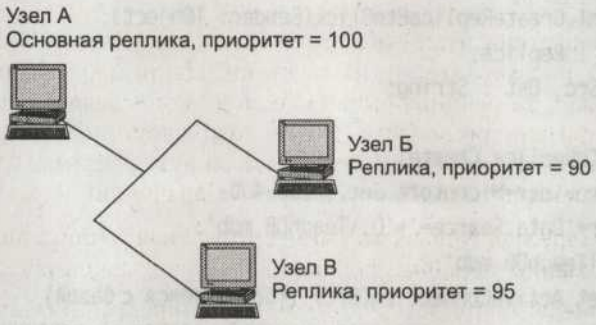


Рис. 7.12. Приоритет реплики

Метод GetObjectReplicability используется для определения того, реплицирован объект или нет. Если объект может быть реплицирован, то метод возвращает значение True.

Метод MakeReplicable переводит базу данных в реплицируемое состояние. В первом параметре метода указывается строка соединения с базой данных, а во втором — способ отслеживания изменений. Если второму параметру ColumnTracking присвоено значение True, то отслеживание изменений будет производиться по столбцам.

Для синхронизации изменений, произведенных в двух репликах, используется метод Synchronize. В параметре этого метода Target указываются путь и имя файла реплики, с которой будет производиться синхронизация. Параметр SyncType определяет направление синхронизации. Этот параметр может принимать несколько заранее определенных значений.

- Значение jrSyncTypeExport указывает, что изменения будут сохраняться из текущей реплики в целевую.
- Значение jrSyncTypeImport указывает, что изменения будут сохраняться из целевой реплики в главную.
- Значение jrSyncTypeImpExp устанавливается по умолчанию и обеспечивает двухсторонний обмен изменениями.

Параметр SyncMode устанавливает метод синхронизации:

- Значение jrSyncModeIndirect указывает, что будет использоваться косвенная синхронизация. Косвенная синхронизация представляет собой способ синхронизации без постоянного подключения к главной реплике.

- Значение jrSyncModeDirect указывает, что будет использоваться прямая синхронизация. В этом случае реплики подключены к локальной сети и располагаются в общих сетевых папках.
- Значение jrSyncModeInternet указывает, что для синхронизации будет использоваться Интернет.

В листинге 7.2 приведен пример создания реплики базы данных.

Листинг 7.2. Репликация базы данных

```
procedure TForm1.CreateReplicaBtnClick(Sender: TObject);
var ReplObject : Replica;
    Provider, Src, Dst : String;
begin
  ReplObject:=CoReplica.Create;
  Provider:='Provider=Microsoft.Jet.OLEDB.4.0;';
  Src:=Provider+'Data Source='+'D:\TeachDB.mdb';
  Dst:='D:\ReplTeachDB.mdb';
  ReplObject.Set_ActiveConnection(Src); {Соединяемся с базой}
  {Если база данных не является реплицируемой, делаем ее реплицируемой.}
  if ReplObject.ReplicaType = jrRepTypeNotReplicable then begin
    ReplObject._Set_ActiveConnection(nil);//Временно разрываем соединение.
    ReplObject.MakeReplicable(Src,True);
    ReplObject.Get_ActiveConnection;//А теперь восстанавливаем.
  end;
  {Создаем реплику базы данных}
  ReplObject.CreateReplica(Dst, 'Global replica',    jrRepTypeFull,
jrRepVisibilityGlobal, 90, jrRepUpdFull);
end;
```

В листинге 7.3 приведен пример синхронизации базы данных с созданной ранее репликой.

Листинг 7.3. Синхронизация базы данных

```
procedure TForm1.SynchronizeBtnClick(Sender: TObject);
var ReplObject : Replica;
    Provider, Src, Dst : String;
begin
  ReplObject:=CoReplica.Create;
  Provider:='Provider=Microsoft.Jet.OLEDB.4.0;';
  Src:=Provider+'Data Source='+'D:\TeachDB.mdb';
  Dst:='D:\ReplTeachDB.mdb';
  ReplObject.Set_ActiveConnection(Src); {Соединяемся с базой}
  ReplObject.Synchronize(Dst, jrSyncTypeImpExp, jrSyncModeDirect);
end;
```

Перед тем как опробовать на базе данных представленные примеры, стоит сохранить ее копию, так как эта операция необратима.

Работа с базой данных из Delphi

В этом разделе будет рассмотрен пример создания трехзвенного приложения базы данных, работающего с СУБД Access. Для работы с базами данных будет использоваться технология BDE.

Прежде всего нужно настроить соединение с базой TeachDB через ODBC. Для этого следует запустить утилиту Data Sources (ODBC) при помощи команды меню Start ▸ Control Panel ▸ Administrative Tools. В диалоговом окне ODBC Data Source Administrator, на вкладке User DSN нужно нажать кнопку Add и в появившемся окне, внешний вид которого показан на рис. 7.13, выбрать пункт Microsoft Access Driver (*.mdb). В поле Data Source Name нужно ввести имя, которое станет псевдонимом базы данных.

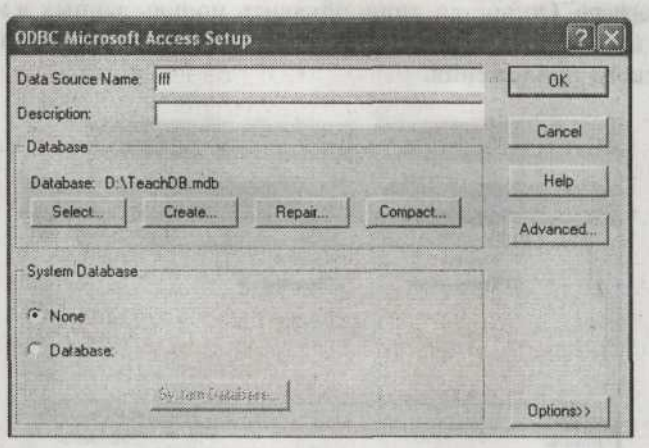


Рис. 7.13. Настройка связи с Access

Теперь нужно создать новый проект AccessRemProj. В него потребуется добавить модуль данных, а в модуле данных разместить два компонента TTable, два компонента TDataSetProvider, компоненты TDatabase, TDataSource и TSession. В поле AliasName компонента TDatabase нужно указать псевдоним используемой базы данных.

Нужно указывать имя той базы данных, которая будет видна в поле DatabaseName компонентов TTable. Свойству AutoSessionName компонента TSession необходимо присвоить значение True. Это свойство заставляет компонент присваивать каждому экземпляру сессии собственное имя.

Свойство DatabaseName компонентов TTable должно получить значение, указанное в свойстве DatabaseName компонента TDatabase. Первый компонент TTable надо связать с таблицей Town, а второй — с таблицей MeteoTable. Также нужно настроить связь таблиц по полю Kod_Goroda при помощи свойства MasterFields.

Свойства poAllowMultiRecordUpdates, poCascadeDeletes и poCascadeUpdates компонента TDataSetProvider должны получить значение True. Потом компоненты TDataSetProvider надо связать с компонентами TTable. После этого созданный проект нужно сохранить.

Также нужно будет создать еще один проект для клиентского приложения AsProj.dpr. В нем потребуется объявить модуль данных, в котором разместятся два компонента TClientDataSet, два компонента TDataSource и компонент TDCOMConnection. В свойстве ComputerName компонента TDCOMConnection нужно указать имя локальной машины, а в свойстве ServerGUID ввести GUID-идентификатор сервера приложения.

Компоненты TClientDataSet надо связать с компонентом TDCOMConnection при помощи свойства RemoteServer. В свойстве ProviderName первого компонента TClientDataSet нужно указать имя DataSetProvider1, а в том же свойстве второго — DataSetProvider2. Связь между таблицами нужно задать при помощи свойства MasterFields второго компонента TClientDataSet.

На форме приложения необходимо разместить две таблицы, две кнопки и компонент TPopupMenu. Останется лишь объявить модуль данных в модуле главной формы и настроить необходимые связи. На рис. 7.14 показан внешний вид клиентского приложения.

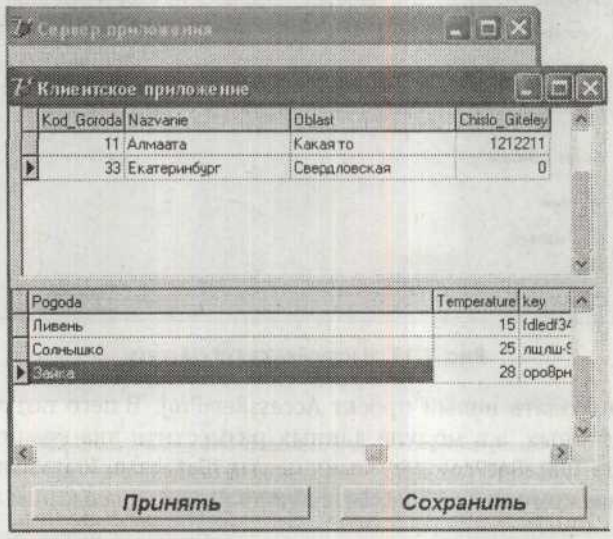


Рис. 7.14. Клиентское приложение

В листинге 7.4 приведен код приложения.

Листинг 7.4. Код клиентской части приложения

```
uses ConnectionModule, Math, DB;

{$R *.dfm}
```

```
procedure TMainForm.Button2Click(Sender: TObject);
begin
  DataMod.ClientDataSet1.ApplyUpdates(-1);
  DataMod.ClientDataSet2.ApplyUpdates(-1);
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  if DataMod.ClientDataSet1.Modified then begin
    DataMod.ClientDataSet1.Post;
  end;
  if DataMod.ClientDataSet2.Modified then begin
    DataMod.ClientDataSet2.Post;
  end;
end;

procedure TMainForm.NPostClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataMod.ClientDataSet1 do begin
      if modified then
        Post;
    end;
  end
  else
    with DataMod.ClientDataSet2 do begin
      if modified then
        Post;
    end;
end;

procedure TMainForm.NInsertClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataMod.ClientDataSet1 do begin
      Insert;
    end;
  end
  else
    with DataMod.ClientDataSet2 do begin
```

продолжение

Листинг 7.4 *(продолжение)*

```
      Insert;
    end;
end;

procedure TMainForm.NDeleteClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataMod.ClientDataSet1 do begin
      Delete;
    end;
  end
  else
    with DataMod.ClientDataSet2 do begin
      Delete;
    end;
end;

procedure TMainForm.NSaveClick(Sender: TObject);
begin
  if PopupMenu1.PopupComponent = DBGrid1 then begin
    with DataMod.ClientDataSet1 do begin
      ApplyUpdates(-1);
    end;
  end
  else
    with DataMod.ClientDataSet2 do begin
      ApplyUpdates(-1);
    end;
end;
```

Эта глава показывает, что даже при помощи относительно слабой СУБД Access можно реализовывать мощные приложения обработки баз данных.

8 УРОК Сервер InterBase

Промышленный сервер баз данных InterBase предназначен для решения широкого круга задач. Он сочетает в себе высокую надежность и простоту установки. Седьмая версия сервера обеспечивает поддержку параллельной работы на многопроцессорном оборудовании. С Delphi поставляется ряд компонентов InterBase eXpress (IBX), позволяющих без особого труда работать с этим сервером. С самим сервером, помимо мощных консольных утилит, поставляется некоторое количество вспомогательных инструментов. Одной из таких утилит является InterBase Guardian.

Данная утилита запускается как сервис и производит непрерывный мониторинг работы сервера. В случае отказа Guardian перезапускает сервер. На рис. 8.1 показана схема взаимодействия приложений с сервером InterBase.

Как видно из схемы, на основе InterBase можно разрабатывать двух и трехзвенные приложения баз данных. При разработке трехзвенного приложения необходимо в качестве промежуточного звена использовать сервер приложения. Взаимодействие с сервером может осуществляться напрямую с использованием API-сервера, при помощи компонентов InterBase eXpress либо через BDE или dbExpress. Следует отметить, что существует несколько параллельно развивающихся Open Source-проектов, таких как FireBird и Yaffil. До определенной версии они являются полностью совместимыми с InterBase.

Установка InterBase

Сервер InterBase в реализации для платформы Windows устанавливается очень просто. Достаточно запустить файл `setup.exe` и в диалоговом окне `InterBase Setup Launcher` выбрать пункт `InterBase`. После этого можно нажать кнопку `Next` и указать путь расположения сервера. В результате будет отображено окно выбора компонентов сервера, показанное на рис. 8.2.

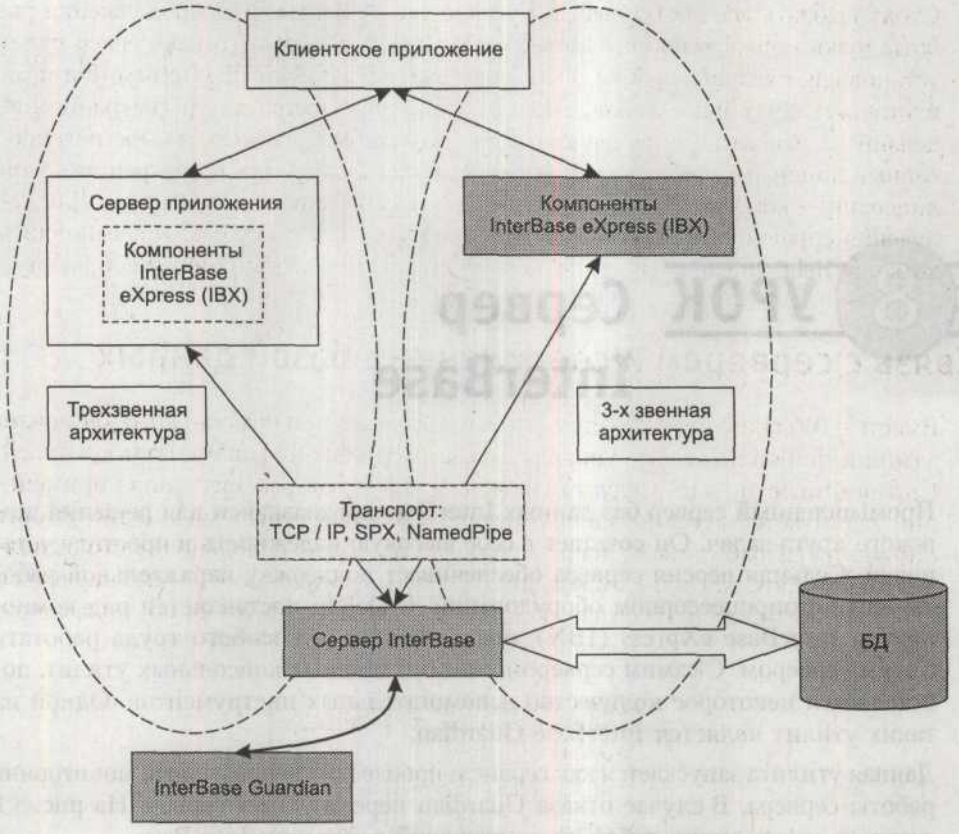


Рис. 8.1. Схема взаимодействия с InterBase

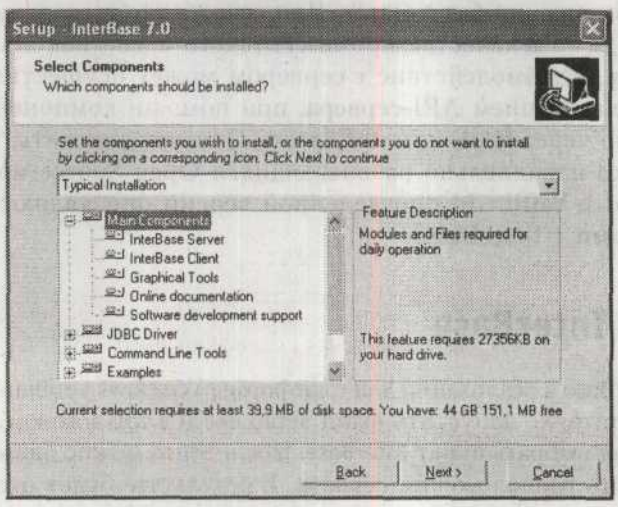


Рис. 8.2. Окно выбора устанавливаемых компонентов

Стоит выбрать все доступные для установки элементы. Для продолжения работы нужно нажать кнопку Next, а затем Install. После того как сервер будет установлен, следует перейти в каталог Program Files\Borland\InterBase\Bin и запустить утилиту iblicense.exe. Эта утилита используется для регистрации лицензий на использование сервера. Для того чтобы просмотреть зарегистрированные лицензии, следует выполнить команду Display, для регистрации новой лицензии — команду Add, для удаления ненужной лицензии — Remove. Для получения справки по доступным командам и их синтаксису следует выполнить команду Help. После регистрации своей лицензии можно продолжить работу.

Связь с сервером и соединение с базой данных

Вместе с InterBase поставляется утилита IBConsole, при помощи которой можно администрировать сервер, управлять базами данных и правами пользователей. Справедливости ради следует отметить, что в некоторых ситуациях применение утилиты является несколько неудобным. Существует множество утилит сторонних разработчиков, используя которые, можно довольно просто выполнять с сервером различные операции. Но сначала все же необходимо научиться работать с утилитой IBConsole. После ее запуска появится окно, показанное на рис. 8.3.

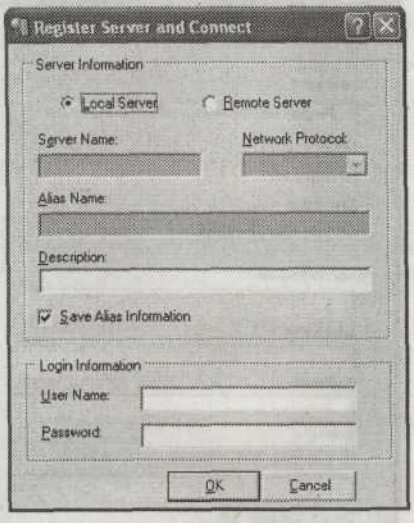


Рис. 8.3. Окно соединения с сервером утилиты IBConsole

Так как сервер расположен на том же компьютере, что и InterBase Client, следует устанавливать соединение с локальным сервером напрямую. Для этого потребуется выбрать пункт Local Server. В поле User Name следует указать логин SYSDBA, а в поле Password — пароль masterkey. Имя пользователя SYSDBA является системным, и пароль для него устанавливается по умолчанию. Соответствен-

но, в реальных разработках его необходимо будет сменить. После нажатия кнопки OK будет установлено соединение с сервером. На рис. 8.4 показана область окна, в которой отображается результат соединения с сервером.

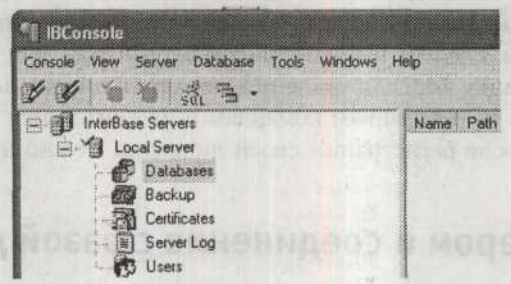


Рис. 8.4. Соединение с сервером установлено

Таким образом создается подключение к серверу InterBase. Теперь нужно установить соединение с конкретной базой данных. Для этого следует выбрать значок Databases из списка Local Server и вызвать пункт контекстного меню Register либо выполнить команду основного меню Database ▸ Register. Появится окно, показанное на рис. 8.5.

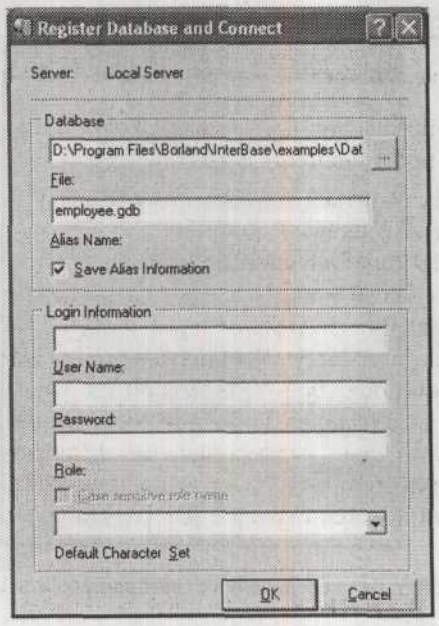


Рис. 8.5. *Соединение с базой данных*

В поле Database нужно указать путь к тестовой базе данных Employee, которая обычно располагается в каталоге examples\Database\employee.gdb внутри корневой папки InterBase. А в поле File надо указать название файла базы данных

Employee.gdb. После нажатия кнопки OK будет установлено соединение с базой данных. На рис. 8.6 показано окно IBConsole с подключенной базой данных.

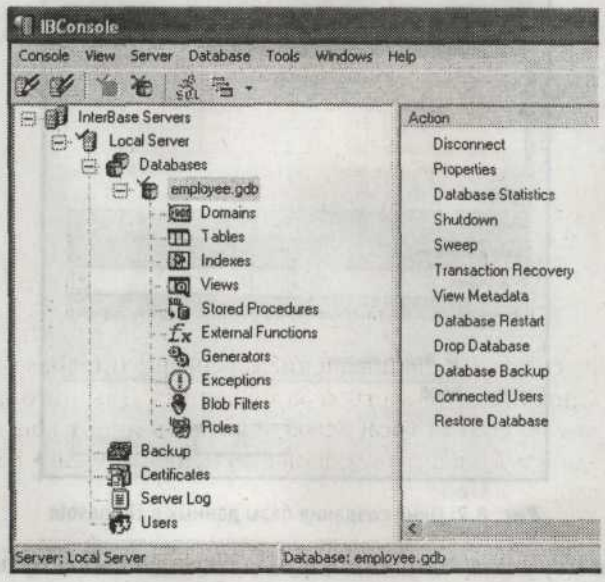


Рис. 8.6. Соединение с базой данных установлено

В списке отображены различные объекты базы данных, такие как домены, таблицы, индексы, представления и многое другое.

Для того чтобы подключиться к удаленному серверу базы данных, используя IBConsole, необходимо в окне соединения с сервером (рис. 8.3) выбрать пункт Remote. Из списка Network Protocol следует выбрать протокол соединения с базой данных, а в поле Server Name указать имя сервера и путь до базы данных. Например, для соединения с сервером через протокол TCP/IP следует выбрать данный протокол и в поле Server Name ввести IP-адрес сервера. Подключившись к серверу, можно подключить базу данных. Форматы строк соединения для разных протоколов различаются. Для соединения по протоколу TCP/IP используется формат <server_name>:<filename>. Для соединения по протоколу NetBEUI строка принимает вид \\<server_name>\<filename>. А для соединения по протоколу SPX требуется строка вида <server_name>@<filename>.

Создание базы данных

Создание базы данных является несложным процессом. Нужно запустить утилиту IBConsole и соединиться с сервером. Затем нужно выполнить команду меню Database ▸ Create Database. Будет отображено диалоговое окно, показанное на рис. 8.7.

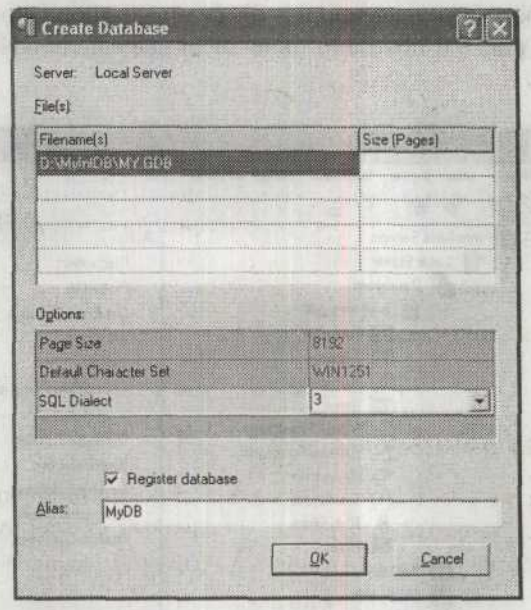


Рис. 8.7. Окно создания базы данных в IBConsole

В поле FileName(s) указываются имя файла базы данных и путь до него. В соседнем поле Size (Pages) указывается количество страниц, содержащихся в файле базе данных. Если в поле на указано значение, то количество страниц не ограничивается. В некоторых случаях необходимо разделить файл базы данных на несколько файлов определенного размера. В этом случае в поле FileName указывают имя первого файла и его размер в поле Size, во второй строчке — имя и размер вторичного файла. Размер страницы базы данных указывается в поле Page Size, он может принимать значения 1024, 2048, 4096 и 8192 байт. В поле Default Character Set можно выбрать кодировку, которая будет использоваться по умолчанию. Кодировка определяет набор символов национального алфавита, который будет использоваться в базе данных по умолчанию.

Если предстоит работать только с русским и английским языками, то имеет смысл выставить значение WIN1251. В поле SQL Dialect следует выбрать используемый диалект языка SQL. Также потребуется установить флажок Register database для того, чтобы при создании база данных была зарегистрирована. После этого потребуется определить псевдоним базы данных в поле Alias. После нажатия кнопки OK база данных будет создана.

С сервером InterBase поставляются мощные утилиты командной строки. Утилита *ISQL* предназначена для выполнения запросов SQL к серверу InterBase из командной строки. Запрос SQL, применяемый для создания базы данных, имеет довольно простую структуру.

```
CREATE {DATABASE | SHEMA} 'Имя файла БД'
[USER 'Имя пользователя' PASSWORD ['Пароль']]
```

```
[PAGE_SIZE [=] int]
[LENGTH [=] int [PAGE[S]]]
[DEFAULT CHARACTER SET charset]
{вторичный файл}
FILE 'имя файла' LENGTH [=] int [PAGE[S] | STARTING AT [PAGE]
```

Для создания базы данных MyDB необходимо выполнить соответствующий SQL-запрос в утилите isql. Она располагается в каталоге BIN-сервера:

```
CREATE DATABASE 'D:\MyIntDB\MY2.GDB'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 8192 LENGTH 500
DEFAULT CHARACTER SET WIN1251;
```

Страницы базы данных

База данных InterBase состоит из последовательно пронумерованных страниц. Нулевая страница является системной и содержит служебную информацию. На рис. 8.8 приведена схема страничного хранения информации.

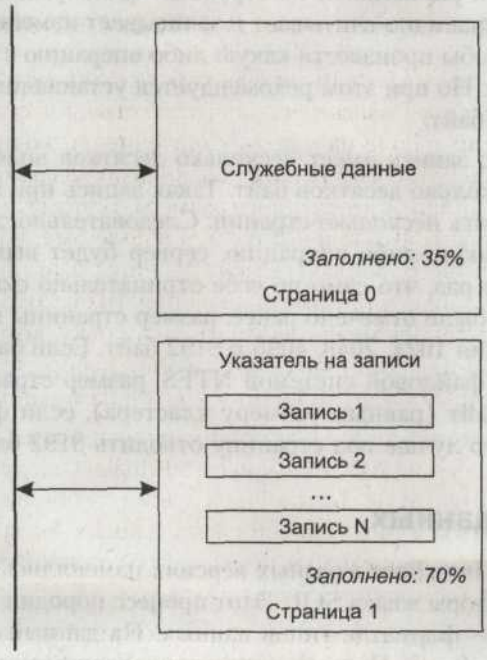


Рис. 8.8. Страничная организация хранения данных

На каждой странице базы данных последовательно располагаются записи. InterBase поддерживает многоверсионную структуру записей. При изменении записи какой-либо транзакцией создается копия записи и работа осуществ-

ляется именно с ней. Помимо данных исходной записи в копию заносятся номер транзакции и указатель на исходную запись. Исходная версия записи помечается как измененная. Каждая стартующая транзакция получает в свое распоряжение копию исходной записи и работает с ней, снимая, таким образом, вопросы доступа к записи, возникающие при ее блокировке.

Если с данной записью работает только одна транзакция, то исходная запись помечается как удаленная, а копия исходной записи становится текущей. Если исходную запись одновременно попытаются изменить несколько транзакций, возникает конфликт обновления записи.

После удаления записи на странице образуются пробелы. При добавлении новой записи производится анализ возможности размещения ее на данной странице. Если длина добавляемой записи меньше суммарной длины всех пробелов на странице, то запись вставляется вместо пробелов. Если длина записи больше суммарной длины пробелов, то она записывается на новой странице.

Размер страницы базы данных

Файл базы данных разбивается на страницы фиксированного размера. Сервер InterBase постранично считывает и записывает изменения в базу данных. Таким образом, чтобы произвести какую либо операцию с записью, он считывает всю страницу. Но при этом рекомендуется устанавливать размер страницы не менее 4096 байт.

Предположим, что запись имеет несколько десятков полей, каждое из которых занимает несколько десятков байт. Такая запись при малом объеме страницы будет занимать несколько страниц. Следовательно, для того чтобы осуществить с ней какую-либо операцию, сервер будет вынужден обратиться к диску несколько раз, что само по себе отрицательно скажется на производительности. Как было отмечено ранее, размер страницы в InterBase 7 может принимать значения 1024, 2048, 4096 и 8192 байт. Если база данных располагается на диске с файловой системой NTFS, размер страницы следует устанавливать 4096 байт (равным размеру кластера), если файловой системой является FAT32, то лучше под страницу отводить 8192 байтов.

Диалект базы данных

В ходе эволюции InterBase в разных версиях изменялись типы данных и используемые операторы языка SQL. Этот процесс породил необходимость создания диалектов — форматов типов данных. На данный момент существует три диалекта — 1, 2 и 3. Первый диалект поддерживают серверы InterBase версии 4 и 5, а третий диалект поддерживается серверами, начиная с шестой версии. В третьем диалекте выделены поля даты и времени. Также введены типы данных для работы с большими целыми числами. Третий диалект не поддерживает неявное приведение типов, в отличие от первого диалекта.

Второй диалект используется в качестве промежуточного для проверки возможности преобразования таблиц из первого в третий. При использовании третьего диалекта следует учитывать, что устаревшие технологии доступа, например BDE, не полностью поддерживают его. В любом случае, ориентироваться при создании баз данных следует именно на третий диалект, так как процесс перевода базы данных из одного формата в другой является довольно трудоемким.

Типы данных

InterBase поддерживает большинство типов данных SQL 92, поля типа BLOB и массивы. Любой тип данных имеет набор операций, которые можно выполнять со значениями этого типа, поэтому необходимо правильно выбрать тип на этапе разработки базы данных. В седьмой версии сервера определено 13 типов данных:

- BLOB — тип данных с динамически изменяемым размером, предназначенный для хранения данных большого размера. В этих полях хранятся графические данные, большие массивы текстов, музыка и многое другое. Данные хранятся в сегментах размером по 64 Кбайт.
- Boolean — логическое поле. Может принимать значения True, False и Unknown. Имеет размер два байта.
- CHAR(n) — символьное поле фиксированной длины. Размер поля определяется на этапе проектирования и указывается в качестве параметра n. Поле может занимать до 32 767 байт.
- DATE — поле даты. Может принимать значение от 1 января 100 года до 29 февраля 32768 года. Поле занимает четыре байта.
- DECIMAL — числовой тип данных с фиксированной точкой. Тип данных принимает в качестве аргументов разрядность и точность хранимых чисел. Разрядность определяет общее число цифр, а точность — число цифр после запятой. Разрядность может быть определена в пределах от 1 до 18 знаков, а точность — достигать определенной перед этим разрядности. Поле может занимать размер 16, 32 или 64 бита.
- DOUBLE PRECISION — вещественный тип данных повышенной точности. Число может находиться в диапазоне от $2{,}25\times10^{-308}$ до $1{,}797\times10^{308}$ и иметь размер до 15 знаков. Поле занимает восемь байтов.
- FLOAT — вещественный тип данных. Число может находиться в диапазоне от $1{,}175\times10^{-38}$ до$3{,}4002\times10^{38}$ и иметь размер до 15 знаков. Поле занимает четыре байта.
- INTEGER — знаковый целочисленный тип данных. Может принимать значение в диапазоне от −2 147 483 648 до 2 147 483 647. Поле занимает четыре байта.

- NUMERIC — эквивалентен типу DECIMAL.
- SMALLINT — знаковый целочисленный тип данных. Может принимать значение в диапазоне от –32 768 до 32 767. Поле занимает два байта.
- TIME —хранит данные о времени с точностью до десятитысячной доли секунды. Может принимать значение в диапазоне от 00:00 до 23:59.9999.
- TIMESTAMP — тип данных, хранящий информацию о дате и времени. Фактически, представляет собой комбинацию типов DATE и TIME.
- VARCHAR (n) — символьное поле переменной длины. Размер поля определяется на этапе проектирования и указывается в качестве параметра n. Поле может занимать до 32 767 байт.

Компоненты InterBase eXpress

Компоненты InterBase eXpress (IBX) предназначены для работы с сервером InterBase, используя InterBase API. Используя данные компоненты, можно получать данные, вносить в них изменения, управлять транзакциями, получать сведения о базе данных, отслеживать состояние процессов выполнения запросов и осуществлять другие действия. Основой кода IBX является Open Source-библиотека FreeIBComponents, разработанная Грегори Дилтцом в 1998 году. Компания Borland продолжила разработку этого набора компонентов.

Компонент TIBDataBase

Механизм доступа к данным InterBase eXpress использует для обращений к серверу возможности клиентской части InterBase, проинсталлированной на машине. Компонент TIBDataBase предназначен для установления соединения с базой данных, расположенной на сервере InterBase. Для установления связи с сервером необходимо указать имя сервера и полный путь к базе данных. Если сервер является локальным, то можно указать путь до файла базы данных. Для этого можно использовать свойство DatabaseName либо воспользоваться удобным редактором, окно которого представлено на рис. 8.9. Для вызова редактора нужно дважды щелкнуть на компоненте либо нажать на кнопку, расположенную в правом углу свойства DatabaseName.

Порядок заполнения полей для соединения с сервером не отличается от рассмотренного ранее. На панели Connection выбирается тип соединения, затем указываются адрес сервера и протокол связи. В поле Database прописывается путь до файла базы данных. В качестве примера можно попробовать установить соединение со своим сервером как с удаленным по протоколу TCP/IP. Для этого в поле Server надо ввесьти адрес 127.0.0.1 и выбрать из списка Protocol значение TCP. В поле Database необходимо указать путь до базы данных employee.gdb.

Активировать соединение можно при помощи свойства Connected. Свойство AllowStreamedConnected позволяет запретить соединение с базой данных во вре-

мя запуска приложения, даже если на этапе разработки свойству Connected было присвоено значение True. По умолчанию свойство принимает значение True, и соединение с базой данных во время запуска допускается. В ином случае необходимо присваивать свойству Connected значение True или вызывать метод Open.

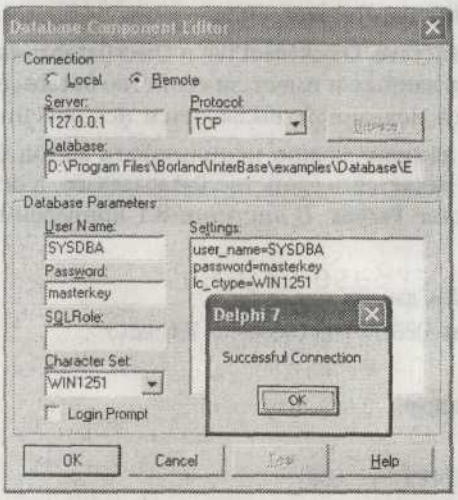


Рис. 8.9. Редактор соединения с базой данных компонента TIBDataBase

При помощи свойства Params можно определить параметры соединения, такие как имя пользователя, пароль и используемую кодировку символов. Свойство DBParamByDPB позволяет напрямую обратиться к отдельным параметрам соединения по их индексу. Например, для получения имени пользователя следует использовать конструкцию DBParamByDPB[isc_dpb_user_name].

Определить диалект базы данных можно при помощи свойства DBSQLDialect. Все действия с базой данных производятся в рамках транзакций. Свойство DefaultTransaction содержит ссылку на компонент транзакции TIBTransaction, используемый по умолчанию. Все компоненты, связанные с TIBDataBase, также будут использовать данный компонент транзакции. С одним компонентом TIBDataBase может быть связано несколько компонентов транзакций. Выяснить, сколько на самом деле их используется, можно при помощи свойства TransactionCount, а обратиться к компоненту транзакции можно по его индексу, используя свойство Transactions.

Поместить компонент транзакции в список используемых можно при помощи метода AddTransaction, а для удаления следует использовать метод RemoveTransaction. Метод RemoveTransactions применяется для удаления всех связанных транзакций.

Для получения индекса транзакции следует воспользоваться методом FindTransaction, получающим в качестве параметра указатель на компонент транзакции. С компонентом может быть связано несколько компонентов TIBEvents, автоматически отслеживающих события, возникающие в базе данных.

Для добавления объекта в список объектов, связанных с данной базой, используется метод AddEventNotifier. В свою очередь, для удаления объекта из списка используется метод RemoveEventNotifier. Целочисленное свойство IdleTimer позволяет определить временной интервал, по истечении которого неиспользуемое соединение будет разорвано. В момент истечения времени ожидания инициируется событие OnIdleTimer.

Группа методов CheckActive, CheckInactive и CheckDatabaseName позволяет выяснить, активна ли база данных и имеет ли свойство DatabaseName значение. Если проверка оказывается неудачной, вызывается исключение.

Для создания базы данных можно использовать метод CreateDatabase. Имя файла базы данных задается в свойстве DatabaseName, а остальные параметры указываются в свойстве Params. В листинге 8.1 приведен пример использования этого метода.

Листинг 8.1. Создание базы данных

```
procedure TForm1.CreateDBBtnClick(Sender: TObject);
begin
with IBDatabase2 do begin
Params.Clear;
DatabaseName:='D:\MyIntDB\My1.GDB';
SQLDialect:=3;
Params.Add('USER «SYSDBA»');
Params.Add('PASSWORD «masterkey»');
Params.Add ('PAGE_SIZE 8192');
CreateDatabase;
end;
end;
```

В момент соединения с базой данных сервер запрашивает логин и пароль. Это инициирует событие OnLogin.

Компонент TIBTransaction

Транзакцией называют логически связанный с базой данных блок операций, выполняющийся как единое целое или не выполняющийся вовсе. Компонент TIBTransaction предоставляет свойства и методы, предназначенные для управления транзакциями одной или нескольких баз данных. Список компонентов TIBDatabase, с которыми связан данный компонент транзакции TIBTransaction, содержится в свойстве Databases. Количество связанных с компонентом баз данных можно выяснить при помощи свойства DatabaseCount.

Свойство DefaultDatabase позволяет указать компонент базы данных TIBDatabase, который будет применяться в рамках данной транзакции. Для того чтобы добавить компонент базы, используется метод AddDatabase. А для разрыва связи используется метод RemoveDatabase. В качестве параметра данный метод при-

нимает индекс компонента. Для определения индекса компонента следует воспользоваться методом FindDatabase, который возвращает целочисленное значение.

Метод CheckDatabasesInList позволяет выяснить, существуют ли связанные с данным компонентом транзакции компоненты TIBDatabase. Если связанные компоненты существуют, метод возвращает значение True.

Для запуска новой транзакции на сервере базы данных используется метод StartTransaction. До вызова этого метода приложение обязано произвести проверку на выполнение другой транзакции в данный момент. Для осуществления проверки используется свойство InTransaction. Если свойство имеет значение True, которое автоматически устанавливается при старте транзакции, это свидетельствует, что транзакция уже выполняется. Попытка запуска транзакции методом StartTransaction без предыдущего ее завершения при помощи методов Commit или Rollback приведет к возникновению исключения.

Для завершения выполняемой транзакции и сохранения изменений, произведенных в ее рамках, используется метод Commit. Для завершения транзакции и отката всех изменений, произведенных в базе данных в рамках данной транзакции, используется метод Rollback. В листинге 8.2 приведен пример использования этих методов.

Листинг 8.2. Запуск и завершение транзакции

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
IBDatabase1.Open;
IBTransaction1.StartTransaction;
Table1.Insert;
Table1.FieldByName('QUANTITY').AsInteger := StrToInt(Edit1.Text);
Table1.Post;
IBTransaction1.Commit;
end;
```

Любой запрос к базе данных должен выполняться в контексте транзакции. Свойство AllowAutoStart указывает компоненту на необходимость автоматического запуска транзакции, когда это необходимо. А свойство AutoStopAction определяет, какое действие будет произведено в момент автоматического завершения транзакции. Это свойство может принимать одно из заранее предопределенных значений:

- Значение saNone указывает, что транзакция не может завершиться неявным образом.
- Значение saRollback указывает, что транзакция откатывается и закрывается при неявном завершении.
- Значение saCommit указывает, что транзакция подтверждается и закрывается при неявном завершении.

- Значение saRollbackRetaining указывает, что транзакция не завершается при закрытии последнего связанного набора данных, но все произведенные изменения откатываются.
- Значение saCommitRetaining указывает, что фиксация изменений будет производиться, но транзакция не будет завершена.

Целочисленное свойство IdleTimer определяет промежуток времени, по истечении которого транзакция будет автоматически зафиксирована или откачена. Для определения того, какое действие будет предпринято по истечении указанного времени, следует воспользоваться свойством DefaultAction. Оно может принимать одно из заранее определенных значений:

- Значение TARollback указывает, что будет произведен откат транзакции.
- Значение TACommit указывает, что транзакция будет зафиксирована.
- Значение TARollbackRetaining указывает, что будет произведен откат транзакции, но она не будет завершена.
- Значение TACommitRetaining указывает, что будет произведена фиксация транзакции, но она не будет завершена.

Метод CheckAutoStop выполняет действие, определенное в свойстве AutoStopAction. Данный метод вызывает наборы данных в моменты неявного завершения транзакции. Любой стартующей транзакции в обязательном порядке присваивается идентификатор. Он используется сервером базы данных для управления транзакцией, учета блокировок и других операций. Идентификатор транзакции хранится в соответствующем свойстве Handle.

Транзакция может иметь набор параметров. К этим параметрам относятся уровень изоляции транзакции, возможность просмотра или модификации таблиц, возможность одновременного доступа к таблице (странице данных) с другими транзакциями. Для получения доступа к буферу, хранящему эти параметры, следует использовать свойство TPB, предоставляющее доступ только для чтения. Для изменения параметров транзакции следует использовать свойство Params, позволяющее получить доступ к TPB. Также для более удобного определения параметров транзакций в компонент встроен редактор Transaction Editor, содержащий несколько заранее определенных комбинаций. Для получения доступа к редактору достаточно дважды щелкнуть мышью на компоненте TIBTransaction. В результате будет отображено диалоговое окно, показанное на рис. 8.10.

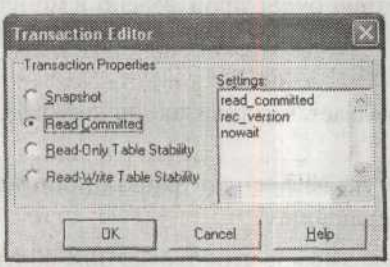


Рис. 8.10. Редактор параметров транзакций

Как было отмечено ранее, InterBase поддерживает многоверсионную архитектуру транзакций. Под версией следует понимать копию исходной записи, созданную при попытке ее изменения. Каждая транзакция имеет свой номер TID, который последовательно увеличивается. То есть поздняя версия имеет номер более высокий, чем предыдущие транзакции. В момент начала выполнения транзакции на *странице учета транзакций* (Transaction Inventory Pages) производится соответствующая отметка, включающая идентификатор транзакции и информацию о том, что транзакция находится в состоянии выполнения.

Необходимо понимать, чем отличаются друг от друга уровни изоляции транзакций, определяющие область видимости изменений, произведенных другими транзакциями. Этот уровень устанавливается при запуске транзакции. Всего существует три уровня:

- Уровень `READ COMMITED` позволяет транзакции видеть все подтвержденные изменения, произведенные в других, параллельно выполняющихся транзакциях. Если данные изменены параллельно выполняющейся транзакцией, но не подтверждены, доступ к ним не может быть получен. Данная транзакция может иметь два режима, определяемых в качестве необязательных параметров. Параметр `RECORD_VERSION` указывает, что в данном режиме разрешается чтение только подтвержденных данных. Транзакция читает лишь последнюю подтвержденную версию записи. Параметр `NO RECORD_VERSION` указывает, что если запрашивается неподтвержденная версия записи и определен режим блокировки `Wait`, то транзакция будет ожидать завершения или отмены транзакции, изменившей данные. В том случае, если установлен режим блокировки `No Wait` и данные, которые пытается прочесть транзакция, не являются зафиксированными, транзакция немедленно возвращает ошибку.
- Уровень `SNAPSHOT` указывает, что транзакция получает в свое распоряжение слепок текущего состояния активности транзакции и не может видеть любые изменения, произведенные другими транзакциями. То есть данная транзакция может видеть только те данные, которые были зафиксированы на момент ее запуска.
- Уровень `SNAPSHOT TABLE STABILITY` указывает, что параллельно выполняющиеся транзакции не могут модифицировать таблицу. Им разрешается только читать данные. Фактически, таблице запрещается записывать изменения во время работы транзакции.

Транзакция имеет определенный набор параметров, настраиваемых с помощью перечисления набора констант, определяющих поведение транзакции. Виды параметров транзакции представлены в табл. 8.1.

Таблица 8.1. Параметры транзакций

Группа параметров	Константа	Краткое описание константы
Режим доступа	Read	Разрешает только операции чтения
	Write	Разрешает операции записи

продолжение

Таблица 8.1 *(продолжение)*

Группы параметров	Константа	Краткое описание константы
Режим блокировки	Wait	Устанавливает режим отсроченного разрешения конфликтов
	NoWait	При возникновении конфликта немедленно возникает ошибка
Уровень изоляции	read_committed	Возможность читать подтвержденные данные других транзакций.
	rec_version	Дополнительный параметр rec_version позволяет читать записи, имеющие неподтвержденные версии
	read_committed	Возможность читать подтвержденные данные других транзакций.
	no_rec_version	Дополнительный параметр no_rec_version не позволяет читать данные, имеющие неподтвержденные версии
	concurrency	При запуске транзакции создается мгновенный «снимок» состояния базы данных других транзакций, не видных в этой транзакции
	consistency	Аналогичен уровню concurrency, но помимо прочего блокирует таблицу на запись

Транзакция может находиться в одном из четырех состояний, которое фиксируется в ее параметрах:

- Состояние `Active` указывает, что транзакция активна.
- Состояние `Commited` указывает, что изменения, произведенные транзакцией, зафиксированы.
- Состояние `Rolled back` указывает, что транзакция отменена.
- Состояние `In limbo` свидетельствует о неопределенном статусе транзакции. Это состояние возникает при использовании механизма двухфазного подтверждения записей.

При старте новой транзакции, претендующей на работу с блоком записей, производится создание версии данных и новый статус отмечается в буфере транзакции. После осуществления работы транзакция принимает или откатывает данные. В распоряжение следующей стартующей транзакции предоставляются исходная и новая версии блока записей. В этом случае читающая транзакция считывает все версии измененного блока записей, производит поиск транзакций, создавших эти записи, и выбирает транзакцию с максимальным номером. Далее производится проверка на предмет подтвержденности транзакции. Если транзакция не является подтвержденной, то берется предыдущая версия транзакции. Если транзакция является подтвержденной, соответствующий ей блок записи принимается в качестве актуального.

Этот процесс порождает «мусор» — неактуальные версии записей, что в конечном итоге увеличивает размер базы данных и замедляет работу сервера. Есть несколько механизмов, позволяющих проводить чистку базы данных. Один из них заключается в том, что когда выделен актуальной блок, неактуальные записи удаляются. Соответственно, удаляется запись из TIP-буфера. При этом версии записей активных на данный момент транзакций не удаляются.

Чистку можно проводить отдельной процедурой. По умолчанию интервал между чистками составляет 20 000 записей, то есть чистка произойдет тогда, когда TIP-буфер достигнет размера в 20 000 тысяч записей.

Класс TIBCustomDataSet

Класс `TIBCustomDataSet` является базовым классом для всех наборов данных, взаимодействующих с сервером, используя InterBase eXpress. Фактически, класс `TIBCustomDataSet`, являющийся потомком класса `TDataSet`, инкапсулирует механизм доступа к данным InterBase eXpress. Приложения не могут напрямую обращаться к свойствам и методам класса. Им позволяется лишь использовать свойства и методы потомков, таких как `TIBDataSet`, `TIBQuery`, `TIBStoredProc` и `TIBTable`, являющихся его наследниками.

Связь с базой данных осуществляется при помощи свойства `Database`, в котором указывается компонент базы данных. Доступ к базе данных осуществляется по ее идентификатору, представляющему собой указатель на файловую структуру. Для получения идентификатора базы данных следует использовать свойство `DBHandle`.

Доступ к связанному компоненту транзакции осуществляется при помощи свойства `Transaction`. Как было отмечено ранее, каждой выполняющейся транзакции присваивается идентификатор, используя который, можно управлять транзакцией, отслеживать ее состояние и выполнять иные действия при помощи InterBase API. Для получения доступа к идентификатору выполняющейся транзакции следует использовать свойство `TRHandle`.

Для определения параметров выполняющегося запроса следует использовать свойство `Params`, представляющее собой экземпляр класса `TIBXSQLDA`. Класс `TIBXSQLDA` предоставляет свойства и методы, предназначенные для использования с компонентом *IBSQL*. Данный класс инкапсулирует структуру InterBase API XSQLDA (extended SQL descriptor area), предназначенную для транспортировки параметров SQL-запроса на сервер и возврата результатов выполнения запроса. Структура XSQLDA формируется динамически для каждого запроса. Как правило, для приложения формируется две структуры XSQLDA. Одна из них предназначена для ввода, а другая для вывода данных. Свойство `Vars` содержит ссылку на массив структур XSQLVAR, инкапсулирующих массив значений каждого возвращаемого поля. Свойство `AsXSQLDA` содержит ссылку на структуру XSQLDA. Выяснить количество полей можно при помощи свойства `Count`. А используя свойство `Names`, можно выяснить имена полей, содержащихся в структуре.

Свойство Modified позволяет определить, возможно ли редактирование полей структуры XSQLDA. Свойство RecordSize возвращает размер записи структуры. А метод AddName позволяет добавить к структуре новое поле. Обратиться к экземпляру структуры XSQLVAR можно по его имени, используя метод ByName.

Стоит рассмотреть структуру XSQLVAR более подробно. Эта структура создается для каждого поля записи и содержит в себе массив возвращаемых значений. Класс TIBXSQLVAR, инкапсулирующий структуру, имеет ряд свойств и методов, позволяющих различным образом взаимодействовать с полем. Используя свойство AsXSQLVAR, можно получить значение поля в виде структуры XSQLVAR. Свойство Index возвращает индекс дескриптора структуры. Обратиться к значению поля можно при помощи свойства Value, а определить, содержит ли структура данные, можно при помощи свойства IsNull.

Метод Clear позволяет очистить структуру от содержащегося в ней значения.

Для обновления данных в наборах, запрещающих производить изменения, используется компонент TIBUpdateSQL, позволяющий выполнять автоматически формируемые SQL-запросы. Связать компонент TIBUpdateSQL с набором данных можно при помощи свойства UpdateObject.

Свойство LiveMode определяет тип операций, которые могут производиться над набором данных. Оно принимает одно из заранее определенных значений:

- Значение lmInsert указывает, что приложение может добавлять записи.
- Значение lmModify указывает, что приложение может изменять записи.
- Значение lmDelete указывает, что приложение может удалять записи.
- Значение lmRefresh указывает, что приложение может обновлять значения записей набора данных.

Для включения механизма кэширования изменений, производящихся в наборе данных, необходимо присвоить значение True свойству CachedUpdates. В том случае, если данный механизм включен, изменения, внесенные в набор данных, сохраняются во внешнем кэше. Когда изменения произведены, вызывается метод ApplyUpdates, отсылающий пакет с изменениями на сервер. Однако эти изменения не фиксируются. Для того чтобы их зафиксировать, необходимо явно вызвать метод Commit компонента TIBTransaction. Таким образом, изменения будут зафиксированы в рамках одной транзакции. Для отмены изменений, содержащихся в кэше, следует использовать метод CancelUpdates.

Свойство UpdatesPending указывает, могут ли измененные записи находиться в кэше. Если свойство имеет значение True, кэш может содержать измененные, добавленные или удаленные записи. При сохранении изменений в режиме кэширования будет инициировано событие OnUpdateRecord, позволяющее определить тип действия. Его обозначение указывается в параметре UpdateKind. Этот параметр может принимать три значения:

- Значение ukModify указывает, что запись обновляется.
- Значение ukInsert указывает, что запись добавляется в набор данных.
- Значение ukDelete указывает, что запись удаляется из набора данных.

Определить реакцию компонента на попытку обновления данных можно при помощи параметра UpdateAction. Его возможные значения перечислены в списке:

- Значение uaFail указывает, что операция обновления данных отменяется и будет выдано сообщение об ошибке.
- Значение uaAbort указывает, что операция обновления данных отменяется без отображения сообщения об ошибке.
- Значение uaSkip указывает, что записи, вызвавшие ошибку обновления, пропускаются и помещаются обратно в кэш.
- Значение uaRetry указывает, что производится повторная попытка обновления записи.
- Значение uaApplied указывает, что метод-обработчик фиксирует произведенные изменения и удаляет запись из кэша.
- Значение uaApply предназначено для внутреннего использования.

В процессе сохранения данных могут возникать ошибки, инициирующие событие OnUpdateError. Метод, обрабатывающий это событие, оперирует несколькими параметрами. В параметре DataSet возвращается указатель на набор данных, в котором произошла ошибка. В параметре E возвращается указатель на объект класса исключения EDatabaseError. Параметр UpdateKind позволяет определить тип операции, вызвавшей ошибку, а параметр UpdateAction — тип операции, применяемый для ее устранения.

При помощи свойства UpdateRecordTypes можно определить типы записей, видимых в режиме кэширования данных. Возможные значения свойства перечислены в списке:

- Значение cusModified указывает, что отображаются измененные записи.
- Значение cusInserted указывает, что отображаются добавленные записи.
- Значение cusDeleted указывает, что отображаются удаленные записи.
- Значение cusUnmodified указывает, что отображаются те записи, которые не были изменены.
- Значение cusUninserted указывает, что отображаются те записи, которые не были добавлены.

По умолчанию свойство UpdateRecordTypes в режиме кэширования имеет значения cusModified, cusInserted и cusUnmodified указывающие, что все существующие, измененные или добавленные записи являются видимыми. В листинге 8.3 показан пример использования свойства UpdateRecordTypes.

Листинг 8.3. Использование свойства UpdateRecordTypes

```
procedure UndeleteAll(DataSet: TIBCustomDataSet);
begin
with DataSet do
```

продолжение

Листинг 8.3 *(продолжение)*

```
begin
UpdateRecordTypes := [cusDeleted]; {make only deleted records visible}
try
First; {move to beginning of dataset}
while not EOF do
begin
Undelete; {undelete the current record}
Next; {move to the next record}
end;
UpdateRecordTypes := [cusUninserted];
try
First; {move to beginning of dataset}
while not EOF do
begin
RevertRecord; {undelete the current record}
Next; {move to the next record}
end;
finally
UpdateRecordTypes := [cusDeleted, cusModified, cusInserted, cusUninserted,
cusUnmodified];
end;
end;
end;
```

Для определения статуса записи в режиме кэширования изменений следует использовать метод `UpdateStatus`. Метод возвращает одно из четырех значений:

- Значение `usUnmodified` означает, что запись не была изменена.
- Значение `usModified` означает, что запись была обновлена.
- Значение `usInserted` означает, что запись была добавлена.
- Значение `usDeleted` означает, что запись была удалена.

Компоненты, инкапсулирующие набор данных, имеют специальный буфер, в который помещаются записи с целью ускорения их обработки. Задать размер буфера можно при помощи свойства `BufferChunks`. По умолчанию обновление записей набора данных осуществляется при вызове метода `Refresh` либо при расчете значений вычисляемых полей. Присвоив свойству `ForcedRefresh` начение `True`, можно заставить компонент набора данных производить обновление данных при каждом вызове метода `Post`. По умолчанию это свойство имеет значение `False`. Если присвоить свойству `InternalPrepared` значение `True`, то набор данных будет производить подготовку SQL-запросов перед их исполнением.

Метод GetCurrentRecord позволяет скопировать запись в буфер, указываемый в параметре Buffer. Для копирования значения одного поля в буфер необходимо использовать метод GetFieldData. В его параметрах FieldNo и Field указывается номер или идентификатор поля, из которого будет скопировано значение в буфер. Метод возвращает значение True, если данные были успешно скопированы в буфер.

Метод IsSequenced позволяет включить механизм упорядочивания записей по номерам их следования, что позволяет использовать метод RecNo для быстрого перемещения к нужной записи по ее номеру. Если метод принимает значение False, то навигация по набору данных до нужной записи будет осуществляться только прямым перемещением до нее от первой записи.

Для получения всех данных, начиная с текущей позиции курсора и до конца файла или хранилища, следует вызывать метод FetchAll. Данный метод рекомендуется использовать для уменьшения сетевого трафика между приложением и сервером InterBase при использовании кэширования данных. Метод перемещает на сервер измененные записи для записи и возвращает записи, измененные другими транзакциями.

Перед разрывом соединения с базой данных и после разрыва этого соединения вызываются методы, обрабатывающие события BeforeDatabaseDisconnect и AfterDatabaseDisconnect. Перед завершением связанной с набором транзакции и после завершения инициируются события BeforeTransactionEnd и AfterTransactionEnd.

При обнулении свойства Database набора данных вызывается метод DatabaseFree, а при обнулении свойства Transaction вызывается метод TransactionFree.

Компонент TIBDataSet

Компонент TIBDataSet предназначен для представления в приложениях наборов данных, полученных в ходе выполнения SQL-запросов. Так как класс TIBDataSet является предком класса TDataSet, он прекрасно работает с компонентами отображения данных. Компонент имеет ряд встроенных объектов, предназначенных для выполнения запросов на получение, удаление, модификацию и добавление записей. Основной запрос на выборку записей содержится в свойстве SelectSQL.

Для создания запроса можно использовать встроенный редактор, существенно упрощающий процесс. Для запуска редактора достаточно выбрать пункт контекстного меню Edit SQL. В результате будет отображено окно, показанное на рис. 8.11.

Вспомогательные запросы содержатся в свойствах InsertSQL, ModifySQL, DeleteSQL и RefreshSQL. Эти запросы предназначены, соответственно, для вставки, изменения, удаления и обновления записей. Данные свойства предоставляют прямой доступ к связанным с ними объектам SQL. Каждому запросу соответствует собственный объект IBSQL, предназначенный для выполнения запроса. Указатели на эти объекты SQL содержаться в свойствах QSelect, QInsert, QModify, QDelete и QRefresh.

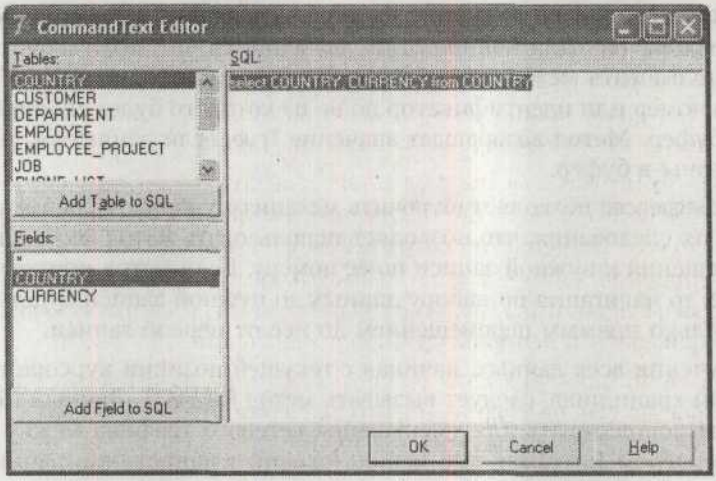


Рис. 8.11. Редактор запросов SQL

Компонент TIBDataSet инкапсулирует часть функциональности компонента TIBUpdateSQL. Для вызова редактора вспомогательных запросов к базе данных необходимо выбрать пункт контекстного меню компонента DataSet Editor. Внешний вид окна редактора запросов показан на рис. 8.12.

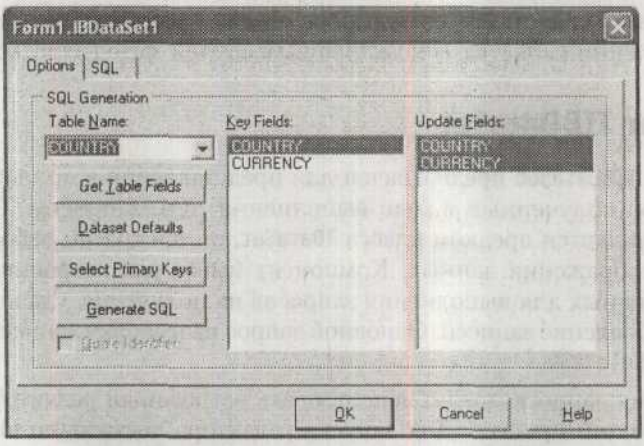


Рис. 8.12. Редактор вспомогательных запросов SQL

Формирование запросов производится очень просто. Достаточно всего лишь выбрать ключевые поля из списка Key Fields, по которым будет производиться поиск нужных записей, и выбрать изменяемые поля из списка Update Fields. После этого достаточно лишь нажать кнопку Generate Fields для составления запросов. Текст сформированных запросов можно посмотреть на вкладке SQL.

Сервер InterBase, как и любой другой сервер БД, имеет аналог автоинкрементного поля. Этот тип называется генератором. Как правило, значение генера-

тора увеличивается специально написанным триггером. Для того чтобы определить используемый с данной таблицей генератор, следует воспользоваться свойством GeneratorField, предоставляющим доступ к объекту TIBGeneratorField, используя который, можно указать поле, значение которого будет увеличиваться, и метод вычисления нового значения поля.

В свойстве Field указывается имя поля, для которого будет сгенерировано значение. А используя свойство IncrementBy, можно указать размер инкрементной части, то есть значение, на которое будет производиться увеличение счетчика. Как правило, данное свойство имеет единичное значение. В свойстве Generator указывается имя генератора, вызываемого объектом TIBGeneratorField при необходимости генерирования нового значения.

Используя свойство ApplyEvent, можно указать, когда серверу будет передаваться запрос на генерирование нового значения поля. Свойство может принимать одно из приведенных ниже значений:

- Значение gamOnNewRecord указывает, что набор данных вызывает метод генерирования нового значения поля немедленно после вставки или присоединения записи, но до инициирования события OnNewRecord.
- Значение gamOnPost указывает, что набор данных вызывает метод сервера для генерирования нового значения поля записи немедленно после вызова метода Post, но до вызова метода BeforePost.
- Значение gamOnServer указывает, что набор данных не производит вызов метода для генерации нового значения поля записи, а сервер генерирует значения автоматически при помощи соответствующего триггера.

Для вызова генератора на сервере и генерирования нового значения поля следует вызвать метод Apply, который помещает сгенерированное значение поля в инкрементируемое поле.

Удобнее всего определять параметры объекта TIBGeneratorField при помощи специального редактора, доступ к которому можно получить при помощи кнопки, расположенной в левой части свойства GeneratorField. Окно редактора показано на рис. 8.13.

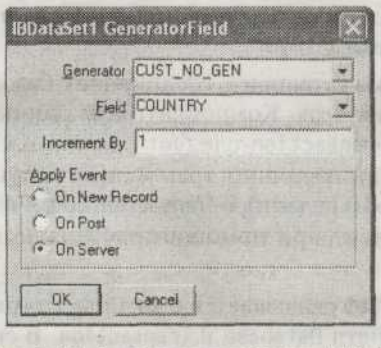


Рис. 8.13. Окно редактора GeneratorField

В поле Generator можно выбрать генератор из списка, в поле Field — поле, для которого будет генерироваться значение, а в поле Increment By можно указать значение, на которое будет производиться увеличение текущего значения генератора при создании новой записи. На панели Apply Event можно указать момент, когда новое значение будет генерироваться и отсылаться на сервер.

Теперь можно вернуться к рассмотрению компонента TIBDataSet. Определить, подготовлен ли SQL-запрос к выполнению, можно при помощи свойства Prepared. Если свойство имеет значение True, то запрос является подготовленным. Как правило, использование данного свойства рекомендуется в случае использования однотипных параметризованных запросов, в которых меняется не структура запроса, а только его параметры. Это связано с тем, что сервер разбирает и анализирует запрос перед выполнением только один раз. Все последующие вызовы данного запроса обрабатываются напрямую, используя результаты предыдущего анализа.

Для подготовки запроса к выполнению следует вызвать метод Prepare, а для его возврата к исходному состоянию — метод UnPrepare. Для выполнения SQL-запроса, не возвращающего набор данных, применяется метод ExecSQL.

Метод BatchInput используется для сохранения данных параметризованного SQL-запроса в объект типа TIBBatchInput. Чаще всего при помощи этого метода полученные данные записываются в файл или поток. Метод BatchOutput, в свою очередь, используется для получения данных из файла или потока в объект типа TIBBatchOutput.

Теперь, когда основные методы работы с соответствующими объектами были рассмотрены, следует создать тестовое приложение, иллюстрирующее методику работы. К новому проекту, как обычно, нужно добавить модуль данных и поместить в него компоненты TIBDatabase, TIBTransaction, TIBDataSet и TDataSource.

Для начала необходимо настроить компонент TDatabase. После двойного щелчка мышью на компоненте будет отображено окно Database Component Editor. Необходимо выбрать удаленный сервер и указать IP-адрес 127.0.0.1, который используется для адресации локальной машины. Также потребуется выбрать протокол TCP и в поле Database указать путь к тестовой базе данных employee.gdb. В полях User Name и Password нужно указать значения SYSDBA и masterkey соответственно. После нажатия кнопки OK в свойстве LoginPrompt потребуется выбрать значение False. В этом случае имя пользователя и пароль будут загружены из списка с параметрами.

После этого необходимо установить соединение с базой данных, задав для свойства Connected значение True. Компонент базы данных TIBDatabase при помощи свойства DefaultTransaction следует связать с компонентом транзакции TIBTransaction. Уровень изоляции транзакции устанавливается при помощи значения Read Commited в редакторе Transaction Editor компонента TIBTransaction. Активируется компонент при помощи присваивания свойству Active значения True.

Компонент TIBSQLDataSet связывается с компонентами базы данных и транзакции при помощи свойств Database и Transaction. В свойстве SQL надо указать запрос select * from COUNTRY, который получает все записи из таблицы COUNTRY.

Также следует определить вспомогательные запросы. Для этого нужно запустить редактор DataSet Editor, выбрать в списке Key Fields ключевые поля, из списка Update Fields — поля для изменений и сгенерировать запросы, нажав кнопку Generate SQL. Потребуется еще включить механизм кэширования изменений, присвоив свойству CachedUpdates значение True. Остается активировать компонент, присвоив свойству Active значение True.

Компонент TDataSource при помощи свойства DataSet нужно связать с компонентом TIBDataSet. Модуль данных нужно объявить в секции uses модуля главной формы приложения. На форме следует разместить компоненты TDBGrid, два компонента TDBEdit и четыре кнопки. Компоненты TDBGrid и TDBEdit нужно связать с компонентом TDataSource. На рис. 8.14 показано окно программы, а в листинге 8.4 приведен ее исходный код.

Рис. 8.14. Окно главной формы приложения

Листинг 8.4. Использование компонента TIBDataSet

```
procedure TForm1.PostBtnClick(Sender: TObject);
begin
with Connection.IBDataSet1 do begin
if Modified then
Post;
end;
end;
procedure TForm1.DeleteBtnClick(Sender: TObject);
begin
with Connection.IBDataSet1 do begin
Delete;
```

продолжение

Листинг 8.4 *(продолжение)*

```
end;
end;
procedure TForm1.InsertBtnClick(Sender: TObject);
begin
with Connection.IBDataSet1 do begin
Insert;
end;
end;
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
with Connection do begin
try
IBDataSet1.ApplyUpdates;
IBTransaction1.CommitRetaining;
except
IBTransaction1.Rollback;
end;
end;
end;
```

Следует обратить внимание на то, что в методе, сохраняющем изменения данных, после вызова метода `ApplyUpdates` вызывается метод `CommitRetaining` или `Commit`. Это необходимо делать из-за того, что метод `ApplyUpdates` не вызывает метод `CommitRetaining` автоматически.

Компонент TIBSQL

Компонент `TIBSQL` предназначен для выполнения SQL-запросов с минимальными затратами времени. Компонент не поддерживает работу с механизмами отображения данных и предоставляет разработчику только однонаправленный курсор. Непосредственным предком компонента является класс `TComponent`, поэтому он только передает запрос на сервер через компонент базы данных `TDatabase` и возвращает результат выполнения запроса.

Для связи с базой данных необходимо выбрать компонент базы данных в свойстве `Database`. Соответственно, для связи с компонентом транзакции необходимо использовать свойство `Transaction`.

Компонент в своей работе не использует объекты полей `TField`, но возвращает данные полей в объекте `TIBXSQLDA`, содержащем объекты `TIBXSQLVAR`. Каждый объект `TIBXSQLVAR` содержит в себе значение поля возвращаемого набора данных. Для обращения к полю по его индексу следует использовать свойство `Fields`. А для определения индекса поля по его имени следует использовать свойство `FieldIndex`. В свою очередь, для того чтобы получить указатель

на нужный объект TIBXSQLVAR по его имени, можно воспользоваться методом FieldByName.

Определить количество записей, содержащихся в наборе данных, можно при помощи целочисленного свойства RecordCount. Для перемещения на следующую запись можно использовать метод Next, возвращающий область дескрипторов следующей записи.

При достижении курсором начала набора данных свойство Bof принимает значение True. В свою очередь, при достижении конца набора данных свойство Eof принимает то же самое значение True. Для того чтобы заставить набор данных в момент открытия устанавливать курсор на первую запись, следует установить свойству GoToFirstRecordOnExecute значение True.

При использовании параметризованных запросов можно изменять их параметры во время выполнения приложения. Для того чтобы компонент мог автоматически обновлять список параметров, необходимо присвоить свойству ParamCheck значение True. Если использование параметризованных запросов не предвидится, лучше присвоить свойству значение False.

Для формирования списка имен параметров запроса используется свойство GenerateParamNames. Для формирования списка достаточно присвоить ему значение True.

В свойстве SQL указывается текст запроса. Запрос можно создавать в специальном редакторе, доступ к которому можно получить, нажав на кнопку, расположенную в правой части свойства. Произвести проверку корректности запроса поможет метод CheckValidStatement. Если запрос не является корректным, метод вернет исключение. Определить, подготовлен ли запрос к выполнению, можно при помощи свойства Prepared. А для подготовки запроса вызывается метод Prepare.

После того как запрос был подготовлен, можно посмотреть план его выполнения, который хранится в свойстве Plan. Для выполнения запроса следует вызвать метод ExecQuery. При помощи свойства SQLType можно выяснить тип выполненного запроса. Свойство может возвращать несколько значений:

- Значение SQLCommit указывает, что была произведена фиксация активной транзакции.
- Значение SQLDelete указывает, что была удалена запись.
- Значение SQLPutSegment указывает, что была произведена запись сегмента данных типа BLOB.
- Значение SQLRollback указывает, что был произведен откат транзакции.
- Значение SQLSetForUpdate указывает, что была выполнена хранимая процедура, изменившая набор данных.
- Значение SQLSetGenerator указывает, что было произведено увеличение значения генератора автоинкрементного поля.
- Значение SQLSelect указывает, что был выполнен запрос, результатом которого является возвращаемый набор данных.

- Значение SQLStartTransaction сигнализирует о запуске новой транзакции.
- Значение SQLUnknown применяется для обозначения неизвестного типа запроса.
- Значение SQLUpdate указывает, что был выполнен запрос для обновления данных.

В свойстве RowsAffected указывается количество записей, обработанных последним запросом. Каждому выполняемому запросу присваивается уникальный внутренний идентификатор, хранящийся в свойстве UniqueRelationName.

Метод Current возвращает ссылку на область дескриптора текущей записи. Для определения активности набора данных можно воспользоваться свойством Open, которое содержит значение логического типа. Метод CheckOpen создаст исключение, если набор данных неактивен. Чтобы закрыть набор данных, следует воспользоваться методом Close. Метод CheckClosed создаст исключение, если набор данных открыт. Метод Call возвращает сообщение о возникшей ошибке на основе ее кода.

Эту технологию необходимо рассмотреть на конкретном примере. Как было отмечено ранее, компонент TIBSQL не может отображать данные в стандартных компонентах отображения данных, поэтому полученные при запросе данные надо выводить в табличный компонент TStringGrid.

После создания нового проекта к нему надо добавить модуль данных. В модуле данных следует разместить компоненты TIBDataBase, TIBTransaction, TIBSQL и TDataSource. При помощи компонента TIBDataBase настраивается соединение с базой employee.gdb. После первичной настройки компонента транзакции TIBTransaction нужно указать уровень изоляции транзакции Snapshot.

Компонент TIBSQL связывается с базой данных при помощи свойства Database, а с компонентом транзакции — при помощи Transaction. В свойстве SQL нужно указать используемый запрос select * from EMPLOYEE, который позволяет получить все записи из таблицы EMPLOYEE.

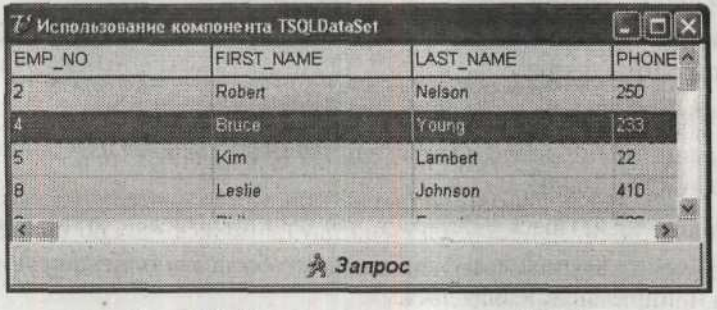


Рис. 8.15. Использование компонента TIBSQL

Компонент TDataSource при помощи свойства DataSet нужно связать с компонентом TIBSQL. После этого нужно объявить модуль данных в модуле главной формы приложения. На основной форме нужно расположить компонент TString-

Grid и одну кнопку. Метод, вызывающийся после нажатия кнопки, будет заполнять таблицу значениями и формировать заголовок. Окно программы показано на рис. 8.15, а основной код приложения приведен в листинге 8.5.

Листинг 8.5. Использование компонента TIBSQL

```
procedure TForm1.QueryBtnClick(Sender: TObject);
var I,J : Integer;
begin
with ConnectionModule.IBSQL1 do begin
ExecQuery;
StringGrid1.ColCount:=FieldCount;
for I:=0 to FieldCount-1 do begin
StringGrid1.Cells[I,0]:=Fields[I].Name;
end;
J:=0;
while not Eof do begin
J:=J+1;
StringGrid1.RowCount:=StringGrid1.RowCount+1;
for I:=0 to FieldCount-1 do begin
StringGrid1.Cells[I,J]:=Fields[I].AsString;
end;
Next;
end;
end;
end;
```

Компонент TIBTable

Компонент TIBTable, являющийся прямым наследником класса TIBCustomDataSet, инкапсулирует функциональность таблицы базы данных сервера InterBase. Компонент может использоваться для получения прямого доступа к данным таблицы или представления. Также компонент имеет возможность работать с множествами записей внутри таблицы базы данных, используя фильтры.

После соединения с базой данных в свойстве TableNames указывается список доступных таблиц. Используя свойство TableTypes, можно определить, будут ли доступны для просмотра системные таблицы и представления. Свойство может принимать два значения:

- Значение ttSystem указывает, что для просмотра доступны системные таблицы и представления.
- Значение ttView указывает, что для просмотра доступны пользовательские представления.

Свойство Filtered позволяет управлять режимом фильтрации записей. Если свойство имеет значение True, то фильтрация включена. Критерий, по которому производится фильтрация записей, указывается в текстовом свойстве Filter. В листинге 8.6. представлен пример использования этого свойства.

Листинг 8.6. Использование свойства Filter

```
IBTable1.Filtered:=False;
IBTable1.Filter:='Country like '+chr(39)+'%Aus%'+chr(39)+' or Country =
'+chr(39)+'Italy'+chr(39);
IBTable1.Filtered:=True;
```

Чтобы указать параметры фильтрации, следует использовать свойство FilterOptions.

В момент открытия набора данных производится попытка упорядочивания записей по первичному ключу. Упорядочивание будет производиться, если свойство DefaultIndex имеет значение True. Выбрать используемый индекс можно при помощи свойства IndexName. Поля индекса можно задать произвольно, перечислив их в свойстве IndexFieldNames.

Определить, существует ли в базе данных таблица, имя которой указано в свойстве TableNames, можно при помощи свойства Exists, которое возвратит значение True, если таблица существует. Как правило, данное свойство используется в тех случаях, когда необходимо создать таблицу базы данных. Сначала необходимо провести проверку на существование таблицы, и если таблица не существует, она создается методом CreateTable.

В листинге 8.7 представлен пример создания таблицы базы данных. Для того чтобы запустить его, следует настроить соединение с созданной ранее базой данных My.GDB. Также потребуется связать компоненты TIBDataBase, TIBTransaction и TIBTable. На форме надо расположить кнопку и для ее метода ввести код, приведенный в листинге 8.7.

Листинг 8.7. Создание таблицы базы данных

```
if not IBTable1.Exists then begin
with IBTable1 do begin
{ The Table component must not be active }
Active := False;
{ First, give the table a database }
Database  := IBDataBase1;
TableName := 'FirstTable';
{ Next, describe the fields in the table }
with FieldDefs do begin
Clear;
with AddFieldDef do begin
Name := 'Field1';
DataType := ftInteger;
```

```
Required := True;
end;
with AddFieldDef do begin
Name := 'Field2';
DataType := ftString;
Size := 30;
end;
end;
{ Next, describe any indexes }
with IndexDefs do begin
Clear;
with AddIndexDef do begin
Name := '';
Fields := 'Field1';
Options := [ixPrimary];
end;
with AddIndexDef do begin
Name := 'Fld2Indx';
Fields := 'Field2';
end;
end;
{ Call the CreateTable method to create the table }
CreateTable;
end;
end;
end;
```

В представленном примере создается таблица базы данных. В первой части кода производится описание структуры таблицы. Эти данные помещаются в структуру `FieldDefs`. Потом производится описание индексов и их добавление в структуру `IndexDefs`.

Для удаления таблицы можно использовать метод `DeleteTable`, а для ее очистки — метод `EmptyTable`.

В некоторых случаях возникает необходимость в синхронизации положения курсоров разных наборов данных. Для этого используется метод `GotoCurrent`, которому передается указатель на набор данных, чей курсор будет синхронизирован с положением текущего курсора.

Компонент TIBQuery

Компонент `TIBQuery` предназначен для выполнения SQL-запросов, возвращающих наборы данных. Непосредственным предком компонента является класс

TIBCustomDataSet. Компонент TIBQuery предоставляет разработчику нередактируемый набор данных. Для изменения данных необходимо использовать компонент TIBUpdateSQL.

Текст SQL-запроса содержится в свойстве SQL, с которым связан редактор запросов. Если есть необходимость рассмотреть код запроса, можно воспользоваться свойством Text. Параметры SQL-запроса содержатся в свойстве Params. В листинге 8.8 приведен пример работы со свойством Params.

Листинг 8.8. Пример работы со свойством Params

```
IBQuery2.Params[0].AsString := 'Liechtenstein';
IBQuery2.Params[1].AsString := 'Vaduz';
IBQuery2.Params[2].AsInteger := 420000;

IBQuery2.SQL.Clear;
IBQuery2.SQL.Add('INSERT INTO COUNTRY (NAME, CAPITAL, POPULATION)');
IBQuery2.SQL.Add('VALUES (:Name, :Capital, :Population)');
IBQuery2.ExecSQL;
```

Общее число параметров запроса хранится в свойстве ParamCount.

Чтобы выполнить запрос, содержащийся в свойстве SQL, необходимо присвоить свойству Active значение True либо вызвать метод Open. Для выполнения запроса, не возвращающего параметры, следует использовать метод ExecSQL.

Компонент TIBUpdateSQL

Компонент TIBUpdateSQL реализует объекты, предназначенные для редактирования наборов данных, доступных только для чтения. Компонент обычно используется совместно с компонентом TIBQuery, когда у последнего включен механизм кэширования данных. Вся работа механизма является прозрачной для конечного пользователя и практически не требует от программиста написания кода.

Связь компонента с компонентами, инкапсулирующими наборы данных, осуществляется при помощи их свойства UpdateObject. Запросы удобно создавать в специальном редакторе, доступ к которому можно получить при помощи команды контекстного меню UpdateSQL Editor.

Для разработки простого примера использования компонента необходимо создать новый проект и добавить к нему модуль данных. В модуле данных размещаются компоненты TIBQuery, TIBDatabase, TIBTransaction, TIBUpdateSQL и TDataSource. Также понадобится настроить соединение с базой данных My.GDB. В параметрах соединения надо прописать следующие строки:

```
user_name=sysdba
password=masterkey
lc_ctype=WIN1251
```

Компонент транзакции должен получить уровень изоляции транзакции Read Commited. После компиляции примера стоит опробовать различные режимы работы при разных уровнях изоляции транзакции.

Компонент TIBQuery нужно связать с компонентами базы данных и транзакции. Компоненты TIBQuery и TIBUpdateSQL нужно связать при помощи свойства UpdateObject. Также при помощи редактора надо сгенерировать запросы на модификацию данных таблицы FirstTable. В редакторе запросов перед генерацией необходимо установить флажок Quote Identifiers.

Необходимо также объявить модуль данных в модуле формы приложения. На основной форме нужно разместить компонент TDBGrid и пять кнопок. Окно приложения показано на рис. 8.16, а его код приведен в листинге 8.9.

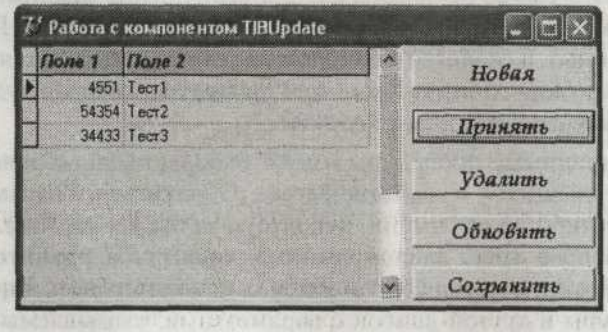


Рис. 8.16. Работа с компонентом TIBUpdateSQL

Листинг 8.9. Код работы с TIBUpdateSQL

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
try
DataModule1.IBQuery1.ApplyUpdates;
except
DataModule1.IBQuery1.Cancel;
end;
end;
procedure TForm1.Button4Click(Sender: TObject);
begin
DataModule1.IBQuery1.Refresh;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
DataModule1.IBQuery1.Delete;
end;
```

продолжение

Листинг 8.9 *(продолжение)*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with DataModule1 do begin
if IBQuery1.Modified then
IBQuery1.Post;
end;
end;
```

Компонент TIBStroredProc

Компонент TIBStoredProc предназначен для выполнения хранимых процедур, хранящихся на сервере, и возвращения результатов их работы. При соединении с базой данных в свойство StoredProcedureNames загружается список хранимых процедур базы данных. Хранимая процедура, которая будет выполнена, указывается в свойстве StoredProcName.

Как правило, хранимые процедуры имеют наборы входных и выходных параметров, содержащихся в свойстве Params. Обратиться к параметру можно по его индексу либо по его имени, используя метод ParamByName. Количество параметров, которое имеет данная хранимая процедура, хранится в свойстве ParamCount. Используя метод CopyParams, можно скопировать параметры хранимой процедуры в другой список с параметрами, указываемый в параметре Value.

Определить, готова ли хранимая процедура к выполнению, позволяет свойство Prepared. Если свойство имеет значение True, то хранимая процедура подготовлена к выполнению. Для подготовки хранимой процедуры к выполнению используется метод Prepare. Если при вызове приложением хранимой процедуры она еще не подготовлена, то вызывается метод Prepare. По завершении выполнения хранимой процедуры автоматически вызывается метод UnPrepare для возврата ее в исходное состояние.

Для выполнения хранимой процедуры на сервере следует вызвать метод ExecProc. В момент запуска хранимой процедуры на сервере ей присваивается уникальный идентификатор, доступ к которому можно получить при помощи свойства StmtHandle.

Рассмотрим простой пример использования компонента TIBStroredProc. К новому проекту надо добавить модуль данных. В модуле данных следует расположить компоненты TIBDataBase, TIBTransaction, TIBStoredProc, TIBTable и TDataSource. Затем нужно настроить соединение с базой данных employee.gdb.

Компонент TIBTable нужно связать с таблицей EMPLOYEE_PROJECT. Также потребуется настроить соединение компонента IBStoredProc с компонентами TIBStoredProc и TIBDataBase, а затем выбрать в свойстве StoredProcName хранимую процедуру GET_EMP_PROJ. В листинге 8.10 приведен код хранимой процедуры.

Листинг 8.10. Хранимая процедура GET_EMP_PROJ

```
BEGIN
FOR SELECT proj_id
FROM employee_project
WHERE emp_no = :emp_no
INTO :proj_id
DO
SUSPEND;
END
```

Следует обратить внимание на то, что хранимая процедура GET_EMP_PROJ имеет два параметра. Входной параметр имеет название :emp_no, а выходной — :proj_id. Эти же параметры доступны в свойстве Params компонента.

После активации всех компонентов нужно объявить модуль данных в модуле главной формы приложения. На форме приложения также нужно разместить компонент TDBGrid, кнопку и два компонента TEdit. На рис. 8.17 показано окно приложения, а в листинге 8.11 приведен код метода, вызываемого при нажатии кнопки.

Рис. 8.17. Работа с хранимой процедурой

Листинг 8.11. Вызов хранимой процедуры

```
procedure TForm1.ExecuteBtnClick(Sender: TObject);
begin
with DataMod do begin
IBStoredProc1.ParamByName('EMP_NO').AsString:=Edit1.Text;
IBStoredProc1.ExecProc;
Edit2.Text:=IBStoredProc1.ParamByName('PROJ_ID').AsString;
end;
end;
```

Компонент TIBDataBaseInfo

Компонент TIBDatabaseInfo предназначен для получения различной системной информации о базе данных. Свойство DBFileName возвращает имя файла базы данных, а свойство DBSiteName — имя сайта базы данных. Размер страницы базы данных возвращает свойство PageSize, а число выделенных базе данных страниц можно выяснить при помощи свойства Allocation. Диалект базы данных указывается в свойстве DBSQLDialect.

Версию базы данных возвращает свойство BaseLevel. В нем хранится число из двух разрядов. В первом разряде содержится номер базы данных, а во втором — ее версия. Также узнать версию базы данных можно, используя свойство Version.

Значение ODS позволяет получать информацию о структуре базы данных InterBase. В зависимости от версии структуры различается формат хранения данных и меняются возможности работы с базой данных. Каждая версия сервера имеет свою структуру. В процессе работы с базой данных значение версии ODS может изменяться. Свойство ODSMinorVersion возвращает начальное значение версии ODS. При изменении структуры ODS увеличивается значение версии. Свойство ODSMajorVersion возвращает последнюю версию структуры ODS. Свойство DBImplementationNo возвращает номер описания базы данных, а свойство DBImplementationClass возвращает номер класса описания.

При помощи свойства CurrentMemory можно выяснить объем оперативной памяти, занимаемой сервером на момент запроса. Используя свойство MaxMemory, можно выяснить максимальное значение загрузки оперативной памяти сервером с момента первого подключенного процесса. Свойство NumBuffers возвращает количество буферов памяти, выделенных серверу на данный момент.

Определить, доступна база данных для записи или она предоставляет возможность только читать данные, позволяет свойство логического типа ReadOnly.

Свойство DeleteCount возвращает количество записей, удаленных с момента последнего присоединения базы данных. А свойство PurgeCount возвращает количество полностью удаленных записей, то есть записей, работа с которыми в рамках транзакций завершена. Свойство ExpungeCount позволяет выяснить полное количество удаленных записей, включая те, которые являются родительскими. Узнать число удаленных версий записей можно при помощи свойства BackoutCount.

Количество записей, добавленных в базу данных с момента последнего рестарта сервера, возвращает свойство InsertCount. А свойство Writes позволяет определить количество операций записи страниц.

Выяснить режим, в котором производится запись данных, можно при помощи свойства ForcedWrites. Нулевое значение указывает, что записи фиксируются в асинхронном режиме, а единичное значение свидетельствует, что записи фиксируются в синхронном режиме.

Свойство SweepInterval возвращает количество транзакций, которые могут быть зафиксированы в базе данных в интервале между чистками. Определить, вы-

полняется ли резервирование места на страницах с данными под версии записей и удаленные записи, позволяет свойство NoReserve. Если свойство принимает единичное значение, то резервирование не производится. Если свойство имеет нулевое значение, то резервирование все же производится.

Свойство UserNames возвращает список пользователей, соединенных с базой данных в текущий момент.

Метод Call возвращает сообщение об ошибке, принимая в качестве параметра ее код.

Компонент TIBSQLMonitor

Компонент TIBSQLMonitor отслеживает динамические SQL-запросы и данные, пересылаемые между сервером и клиентским приложением. Тип операций, отслеживаемых компонентом, указывается в свойстве TraceFlags. Свойство может принимать несколько значений, перечисленных в списке:

- Значение tfQPrepare указывает, что будут отслеживаться сообщения о подготовке SQL-запроса или хранимой процедуры к выполнению.
- Значение tfQExecute указывает, что будут отслеживаться сообщения о выполнении SQL-запроса или хранимой процедуры.
- Значение tfError указывает, что будут отслеживаться сообщения об ошибках, возникающих во время выполнения запроса. В тексте сообщения будет возвращен код ошибки.
- Значение tfStmt указывает, что будут отслеживаться все операции с запросами.
- Значение tfConnect указывает, что будут отслеживаться моменты соединения и разрыва соединения с базой данных.
- Значение tfTransact указывает, что будут отслеживаться операции с транзакциями.
- Значение tfBlob указывает, что будут отслеживаться операции с полями типа BLOB.
- Значение tfService указывает, что будут отслеживаться различные сервисные операции.
- Значение tfMisc указывает, что будут отслеживаться все остальные операции.

В момент получения от сервера сообщения инициируется событие OnSQL. Метод, обрабатывающий его, содержит в параметре EventText текст сообщения, а в параметре EventTime — время регистрации сообщения. Используя данный метод, можно организовать работу журнала.

Компонент TIBEvents

InterBase имеет специальный оператор, позволяющий генерировать события при наступлении каких-либо условий. Сигнал о событии отсылается всем

клиентским приложениям, подключенным к серверу. Используя данный компонент, приложение может регистрировать интересующие его события, происходящие в базе данных, которая указывается в свойстве Database, реализовав отслеживание этих операций в реальном времени.

Для отслеживания возникновения событий в клиентском приложении предназначен компонент TIBEvents. В свойстве Events указывается список отслеживаемых событий. В момент регистрации отслеживаемого события вызывается метод, обрабатывающий событие OnEventAlert. В его параметре EventName указывается имя зарегистрированного события, а в параметре EventCount — количество событий, поступивших с момента начала регистрации.

Для того чтобы прекратить отслеживание событий, следует присвоить параметру CancelAlerts значение True. Если свойство имеет значение False, то регистрация событий будет продолжаться.

Для того чтобы сервер извещал приложение о наступлении события, это событие необходимо зарегистрировать на сервере. Для этого используется метод RegisterEvents. Метод RegisterEvents вызывается автоматически, если свойство AutoRegister имеет значение True. Метод SetAutoRegister позволяет задать значение свойства AutoRegister. А метод GetAutoRegister возвращает значение свойства AutoRegister. Определить, была ли произведена регистрация события, позволяет свойство Registered.

Для того чтобы отменить регистрацию события на сервере, используется метод UnRegisterEvents. Для определения регистрируемых событий удобно пользоваться редактором EventAlerter Events, запустить который можно при помощи кнопки, расположенной в правой части свойства Events. Окно редактора показано на рис. 8.18.

Рис. 8.18. Редактор событий компонента TIBEvents

Теперь следует рассмотреть пример использования этого свойства. Для начала необходимо создать в базе данных два триггера и хранимую процедуру. В утилите IBConsole нужно зарегистрировать базу данных My.GDB, созданную ранее. После этого потребуется запустить утилиту Interactive SQL. Для этого

нужно выполнить пункт меню Tools ▸ Interactive SQL утилиты IBConsole. SQL-запрос для создания хранимой процедуры выглядит довольно просто:

```
CREATE PROCEDURE GFGD
AS
BEGIN
POST_EVENT 'ProcEvent';
END
```

Триггер, вызываемый после удаления записи, создается новым запросом:

```
CREATE TRIGGER DELETE_NEW_ORDER FOR «FirstTable»
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
POST_EVENT 'DELETE';
END
```

Текст триггера, вызывающегося после вставки записи, описывается другим запросом:

```
CREATE TRIGGER POST_NEW_ORDER FOR «FirstTable»
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
POST_EVENT 'NEW_ORDER';
END
```

Как видно из текста запросов, оператор POST_EVENT будет вызван при наступлении некоего условия. В данном случае он будет активирован в моменты вызовов триггеров или хранимой процедуры.

После создания нового проекта к нему надо добавить модуль данных. В модуле данных следует разместить компоненты TIBDataBase, TIBTransaction, TIBTable, TIBStoredProc, TIBEvents и TDataSource. Как обычно, потребуется настроить соединение с базой данных My.GDB, присвоить свойствам Active и Connected компонентов TIBStoredProc и TIBDataBase соответственно значение True.

Компонент TIBDataBase нужно связать с базой данных и выбрать для работы таблицу FirstTable. Точно так же надо связать с базой данных компонент TIBStoredProc с базой данных и задать в свойстве StoredProcName хранимую процедуру GFGD.

Компонент TIBEvents при помощи свойства Database нужно связать с компонентом базы данных TIBDataBase, а потом надо установить режим автоматической регистрации событий на сервере, присвоив свойству AutoRegister значение True. Осталось связать компонент TDataSource с компонентом TIBTable и объявить модуль данных в модуле главной формы.

На основной форме следует расположить компонент TDBGrid и три кнопки. В метод, обрабатывающий событие OnEventAlert компонента TIBEvents, нужно

поместить код, который будет выводить в заголовок формы приложения имя события в момент его регистрации. В листинге 8.12 приведен код модуля данных.

Листинг 8.12. Код модуля данных

```
procedure TEventDataModule.IBEvents1EventAlert(Sender: TObject;
EventName: String; EventCount: Integer; var CancelAlerts: Boolean);
begin
Form1.Caption:=EventName;
end;
```

В листинге 8.13 приведен код методов, обрабатывающих нажатие кнопок на главной форме, внешний вид которой показан на рис. 8.19.

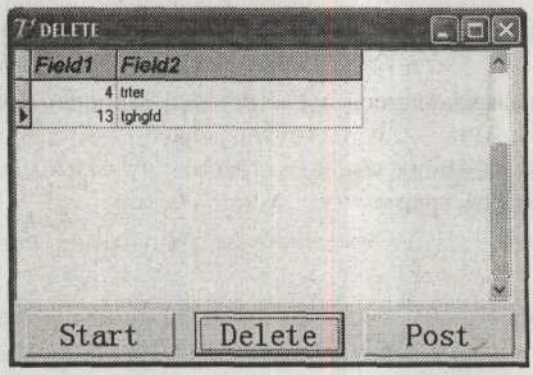


Рис. 8.19. Работа с компонентом TIBEvents

Листинг 8.13. Код обработки нажатия кнопок главной формы приложения

```
procedure TForm1.StartTransactionBtnClick(Sender: TObject);
begin
with EventDataModule do begin
IBStoredProc1.Prepare;
IBStoredProc1.ExecProc;
IBTransaction1.CommitRetaining;
end;
end;
procedure TForm1.DeleteBtnClick(Sender: TObject);
begin
with EventDataModule do begin
IBTable1.Delete;
IBTransaction1.CommitRetaining;
end;
end;
```

```
procedure TForm1.PostBtnClick(Sender: TObject);
begin
with EventDataModule do begin
if IBTable1.Modified then begin
IBTable1.Post;
IBTransaction1.CommitRetaining;
end;
end;
end;
```

При нажатии кнопок вызываются методы, инициирующие выполнение триггера или хранимой процедуры. Соответственно, приложение получает в ответ сгенерированное событие. Можно запустить несколько копий приложения и выполнить какую-либо операцию. Следует обратить внимание на то, что все экземпляры приложения получат сообщение.

Сервисные компоненты InterBase eXpress

На вкладке InterBase Admin расположены компоненты, предназначенные для управления сервером и получения различной информации о режимах его работы. Компоненты, фактически, представляют собой «обертку» для методов InterBase Services API. Перечень возможностей этих методов довольно широк:

- архивирование и восстановление баз данных, отключение и перезагрузка сервера, сборка «мусора», сканирование таблиц для поиска ошибок;
- создание, модификация и удаление учетных записей;
- отслеживание сертификатов, предоставляющих права на работу с сервером;
- получение информации о конфигурации сервера и баз данных.

За выполнение перечисленных выше задач отвечает объект Services Manager, получающий вызовы от клиентских приложений посредством Services API. На рис. 8.20 показана схема, отражающая смысл сказанного. Семейство функций Services API состоит из четырех функций:

- Функция `isc_service_attach` осуществляет установление соединения с объектом Services Manager.
- Функция `isc_service_start` осуществляет вызов нужного метода объекта Services Manager.
- Функция `isc_service_query` возвращает запрошенную информацию или результат выполнения вызванного ранее метода.
- Функция `isc_service_detach` разрывает соединение с объектом Services Manager.

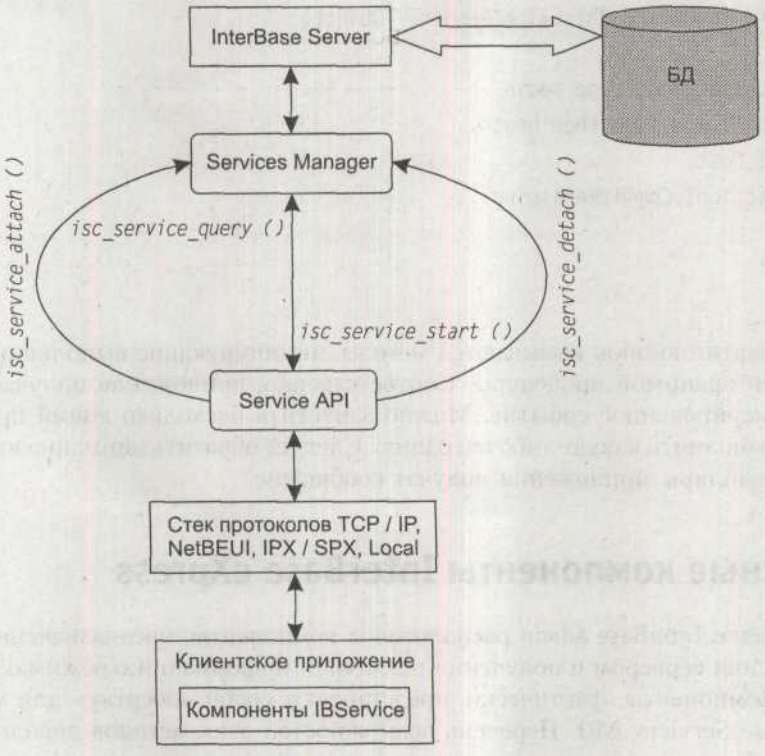


Рис. 8.20. Взаимодействие с InterBase Services

Для определения дополнительных параметров соединения, создаваемого методом `isc_service_attach`, следует воспользоваться специальным буфером Services Parameter Buffer (SPB), адрес которого передается методу в качестве одного из параметров. Буфер параметров представляет собой массив типа `Char` и может быть использован для передачи имени пользователя и пароля.

На рис. 8.21 представлена иерархия компонентов IBX Services. Основой всех компонентов является класс `TIBCustomService`.

Класс TIBCustomService

Класс `TIBCustomService` является общим предком для всех компонентов `IBService`. В его свойстве `ServerName` указывается имя сервера, на котором будут использоваться сервисы. Для того чтобы определить параметры соединения, следует использовать свойство `ServiceParamBySPB`, позволяющее обратиться к параметру буфера по его индексу. Например, выражение `ServiceParamBySPB[isc_SPB_user_name]` позволяет получить или установить имя пользователя.

Для доступа к параметрам базы данных следует использовать свойство `Params`, а для выбора сетевого протокола, по которому будет производиться соединение с сервером, используется свойство `Protocol`.

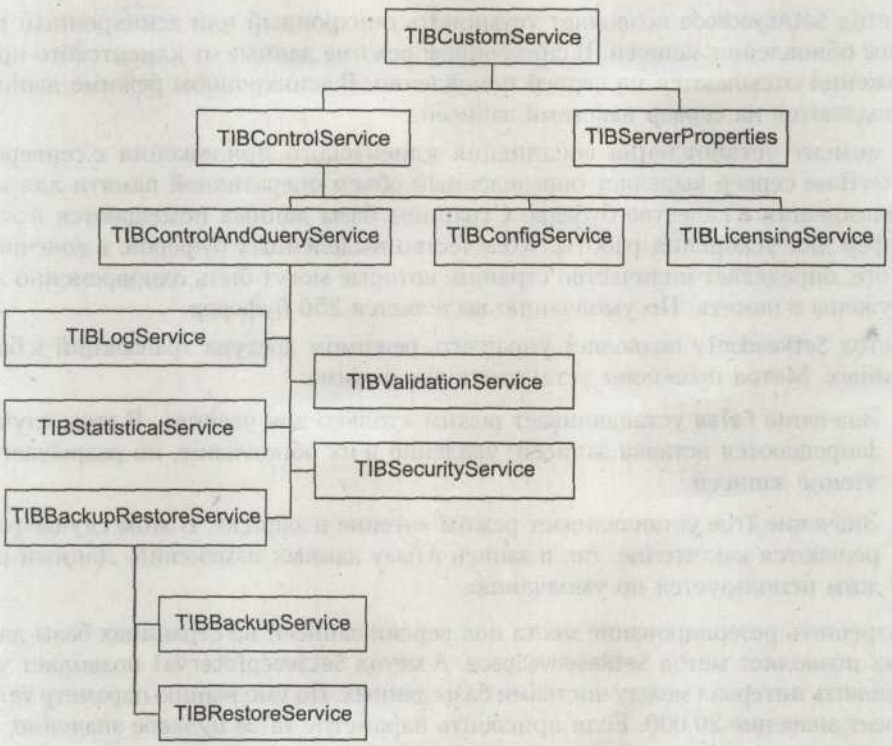


Рис. 8.21. Иерархия компонентов IBX Services

Метод `Attach` производит соединение с базой данных. В момент соединения с базой данных инициируется событие `OnAttach`.

Для активации сервиса следует присвоить свойству `Active` значение `True`, соответственно, для деактивации следует присвоить свойству значение `False`.

Компонент TIBConfigService

Этот компонент предназначен для управления параметрами базы данных. В свойстве `DatabaseName` указывается имя базы данных, параметры которой будут настраиваться. Определить, запущен сервис или нет, позволяет свойство `IsServiceRunning`.

Сервер InterBase позволяет восстанавливать базу данных в случае крушения жесткого диска, поломки сети или случайного удаления записей. Одним из способов резервирования данных является метод Shadowing. Суть метода заключается в создании полной копии базы данных и в синхронном или асинхронном поддержании ее соответствия оригиналу. В некоторых случаях возникает необходимость отключения основной базы данных. В этом случае достаточно просто подключить резервную копию. Для этого используется метод `ActivateShadow`.

Метод SetAsyncMode позволяет установить синхронный или асинхронный режим обновления записей. В синхронном режиме данные от клиентского приложения отсылаются на сервер немедленно. В асинхронном режиме данные отсылаются на сервер пакетами записей.

В момент установления соединения клиентского приложения с сервером InterBase сервер выделяет определенный объем оперативной памяти для использования в качестве буфера. Страницы базы данных помещаются в этот буфер для ускорения работы. Количество выделенных буферов, в конечном итоге, определяет количество страниц, которые могут быть одновременно загружены в память. По умолчанию выделяется 256 буферов.

Метод SetReadonly позволяет управлять режимом доступа транзакций к базе данных. Метод позволяет установить два режима:

- Значение False устанавливает режим «только для чтения». В этом случае запрещаются вставка записей, удаление и их обновление, но разрешается чтение записей.
- Значение True устанавливает режим «чтение и запись». В этом случае разрешаются как чтение, так и запись в базу данных изменений. Данный режим используется по умолчанию.

Разрешить резервирование места под версии записей на страницах базы данных позволяет метод SetReserveSpace. А метод SetSweepInterval позволяет установить интервал между чистками базы данных. По умолчанию параметр Value имеет значение 20 000. Если присвоить параметру Value нулевое значение, то периодические чистки базы данных производиться не будут.

Метод ShutdownDatabase позволяет выключить базу данных по истечении промежутка времени, заданного параметром Wait. В параметре Options можно определить условия завершения работы базы данных. Параметр может принимать одно из трех предопределенных значений:

- Значение Forced указывает, что база данных будет закрыта по истечении указанного времени. Для того чтобы закрыть базу данных немедленно, следует присвоить параметру Value нулевое значение.
- Значение DenyTransaction указывает, что база данных будет закрыта только в том случае, если в установленном промежутке времени все активные транзакции будут завершены. Если хотя бы одна транзакция останется активной, процесс завершения работы базы данных будет отменен.
- Значение DenyAttachment указывает, что база данных будет закрыта только в том случае, если не будет произведено новых подключений к ней. В случае наличия активных подключений к базе данных процесс будет отменен.

Компонент TIBBackupService

Компонент TIBBackupService предназначен для резервирования базы данных. Имя резервируемой базы данных указывается в свойстве DatabaseName, а имя файла резервной копии указывается в свойстве BackupFile.

Свойство Options позволяет определить параметры процесса резервирования данных. Возможные значения этого свойства приведены в списке:

- Значение IgnoreChecksums указывает, что проверка контрольной суммы в процессе резервирования производиться не будет.
- Значение IgnoreLimbo указывает, что транзакции, имеющие статус Limbo, в течение процесса резервирования будут игнорироваться.
- Значение MetadataOnly указывает, что выходной файл будет содержать только метаданные и пустые таблицы.
- Значение NoGarbageCollect указывает, что нормальный процесс сборки «мусора» в течение резервирования будет подавляться. В некоторых базах данных это увеличивает производительность.
- Значение OldMetadataDesc указывает, что метаданные будут сохраняться в устаревшем формате.
- Значение NonTransportable указывает, что данные будуь сохраняться не в XDR-формате. За счет этого увеличивается производительность и повышается экономия места в базах данных небольшого объема.
- Значение ConvertExtTables указывает, что будет производиться перемещение данных из внешних таблиц во внутренние.

Метод ServiceStart запускает соответствующий сервис. Для получения нового блока данных следует использовать метод GetNextChunk. В свою очередь, метод GetNextLine предназначен для получения новой строки данных.

В листинге 8.14 представлен пример использования компонента.

Листинг 8.14. Резервирование данных в несколько файлов

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  with IBBackupService1 do
  begin
    ServerName := 'Poulet';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Verbose := True;
      Options := [MetadataOnly, NoGarbageCollection];
      DatabaseName := 'c:\InterBase\examples\database\employee.gdb';
      BackupFile.Add('c:\temp\e1.gbk = 2048');

      BackupFile.Add('c:\temp\e2.gbk' = 4096);
```

продолжение

Листинг 8.14 *(продолжение)*

```
      BackupFile.Add('c:\temp\e3.gbk');
      ServiceStart;
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Компонент TIBRestoreService

Компонент TIBRestoreService предназначен для восстановления базы данных из резервной копии. В свойстве DatabaseName указывается имя восстанавливаемой базы данных. А в свойстве BackupFile указывается список файлов, из которых будет восстановлена база данных.

Свойство Options позволяет задать параметры восстановления данных. Возможные значения этого свойства приведены в списке:

- Значение DeactivateIndexes отключает индексы на время восстановления базы данных. В нормальном режиме работы InterBase перестраивает индексы во время восстановления базы данных. Если база данных содержит дубликаты записей, являющиеся уникальными, то процесс восстановления будет прерван. Дублирующиеся записи можно внести в базу данных, если временно отключить уникальный индекс.
- Значение NoShadow отключает перестройку резервных копий. В нормальном режиме InterBase перестраивает их во время процесса восстановления данных. Использование этого значения блокирует данную возможность.
- Значение NoValidityCheck отключает проверку ограничений целостности. Для восстановления базы данных, содержащей записи в неправильном формате, следует использовать это значение. При этом будет отключена проверка данных на соответствие существующим ограничениям.
- Значение OneRelationAtATime указывает на то, что данные и метаданные будут восстанавливаться для каждой таблицы отдельно.
- Значение Replace позволяет приложению перезаписать существующий файл базы данных. Если это значение не использовать, то при попытке восстановления базы данных будет произведена отмена операции.
- Значение CreateNew используется для того, чтобы произвести восстановление базы данных из резервной копии в новый файл.
- Значение UseAllSpace заставляет InterBase в процессе восстановления данных заполнять страницы данных на 100%. Если это значение не использовать, то страницы будут заполняться только на 80%.

Используя свойство PageBuffers, можно определить количество страниц буфера данных. Размер страницы задается при помощи целочисленного свойства PageSize.

Компонент TIBValidationService

Компонент TIBValidationService предназначен для проверки базы данных и завершения работы транзакций. Имя обрабатываемой базы данных указывается в свойстве DatabaseName.

При помощи свойства Options можно определить множество параметров процесса проверки и восстановления базы данных. Свойство может принимать одно или несколько значений из списка:

- Значение LimboTransactions указывает, что будет возвращен список транзакций со статусом «лимбо», то есть двухфазных транзакций, статус которых неизвестен.
- Значение CheckDB указывает, что будет производиться проверка таблиц базы данных на наличие ошибок. Но само восстановление базы производиться не будет. Это значение используется в связке с параметром ValidateDB.
- Значение IgnoreChecksum указывает, что ошибки несоответствия контрольной суммы эталонному значению будут игнорироваться.
- Значение KillShadows указывает, что все неиспользуемые Shadow-файлы будут уничтожаться.
- Значение MendDB подготавливает поврежденную базу данных к резервированию.
- Значение SweepDB производит чистку базы данных.
- Значение ValidateDB указывает, что будет производиться проверка структуры базы данных, а найденные ошибки будут исправлены.
- Значение ValidateFull производит проверку фрагментов записей базы данных на наличие ошибок в них и в случае обнаружения исправляет их.

Свойство GlobalAction позволяет определить, какие действия будут применены по отношению к транзакции. Свойство может принимать одно из заранее определенных значений:

- Значение CommitGlobal указывает, что транзакции будут зафиксированы.
- Значение RollbackGlobal указывает, что транзакции будут откачены назад.
- Значение RecoverTwoPhaseGlobal выполняет восстановление двухфазных транзакций.
- Значение NoGlobalAction указывает, что никаких действий не будет предпринято.

Свойство LimboTransactionInfo возвращает ссылку на структуру TLimboTransactionInfo, содержащую информацию о транзакциях со статусом «лимбо». Объявление этой структуры приведено ниже:

```
type
TLimboTransactionInfo = record
MultiDatabase: Boolean;
ID: Integer;
HostSite: String;
RemoteSite: String;
RemoteDatabasePath: String;
State: TTransactionState;
Advise: TTransactionAdvise;
Action: TTransactionAction;
end;
```

Также для получения информации о транзакциях со статусом «лимбо» можно использовать метод `FetchLimboTransactionInfo`. Свойство `LimboTransactionInfoCount` возвращает количество транзакций с этим статусом. Для исправления ошибок в подобных транзакциях следует воспользоваться методом `FixLimboTransactionErrors`.

Компонент TIBStatisticalService

Компонент `TIBStatisticalService` предназначен для получения разнообразной статистики о базе данных. Свойство `Options` позволяет определить параметры, которые возвратит компонент. Возможные значения свойства приведены в списке:

- Значение `DataPages` возвращает статистику о таблицах базы данных.
- Значение `DBLog` указывает, что будет включаться информация из журнала базы данных.
- Значение `HeaderPages` указывает, что будет включаться информация о системных страницах базы данных.
- Значение `IndexPages` возвращает статистическую информацию об индексах, созданных пользователем.
- Значение `SystemRelations` возвращает статистическую информацию о системных таблицах и индексах.

В листинге 8.15 приведен пример использования этого компонента.

Листинг 8.15. Получение статистической информации о базе данных

```
procedure TForm1.Button1Click(Sender: TObject);
begin
with IBStatisticalService1 do
  begin
  ServerName := '127.0.0.1';
  DatabaseName := 'D:\MyIntDB\My.gdb';
```

```
  LoginPrompt := False;
   Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    ServiceStart;
    try
      Options := [DataPages, DbLog];
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

Компонент TIBLogService

Компонент TIBLogService предназначен для получения содержания файла InterBase.log, расположенного на сервере. В листинге 8.16 представлен пример[1].

Листинг 8.16. Пример использования компонента TIBLogService

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with IBLogService1 do
  begin
    ServerName := '127.0.0.1';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    ServiceStart;
    try
      While not Eof do
        Memo1.Lines.Add(GetNextLine);
    finally
      Active := False;
    end;
  end;
end;
```

[1] Пример скопирован из справочной системы Delphi 7.

Компонент TIBSecurityService

Ядром системы безопасности сервера InterBase 7 является база данных admin.ib, содержащая учетные записи и пароли пользователей. Каждой учетной записи соответствует зашифрованный пароль. При соединении с сервером ему пересылаются имя пользователя и зашифрованный пароль, которые затем сравниваются с хранящимися в базе. На основании полученной информации пользователю предоставляются права на работу с объектами. В рамках этих прав пользователь может получать, изменять, удалять данные.

По умолчанию сервер имеет учетную запись SYSDBA, которой присвоен пароль masterkey. Учетная запись SYSDBA обладает абсолютными правами. Она позволяет создавать, изменять и удалять учетные записи пользователей, а также изменять права доступа. Компонент TIBSecurityService позволяет управлять правами пользователей, их группами, а также регулировать и ограничивать доступ к серверу из приложения.

Метод DisplayUsers возвращает информацию обо всех пользователях в формате структуры TUserInfo. Объявление этой структуры приведено ниже:

```
TUserInfo = record
UserName: string;
FirstName: string;
MiddleName: string;
LastName: string;
GroupID: Integer;
UserID: Integer;
end;
```

Фактически, структура TUserInfo имеет тот же перечень полей, что и таблица admin.ib. Как видно из формата структуры TUserInfo, каждая учетная запись и группа пользователей имеет свой уникальный номер.

Для получения информации об учетной записи следует вызвать метод DisplayUser, который вернет информацию в структуре TUserInfo. Доступ к структуре TUserInfo можно получить при помощи свойства UserInfo, принимающего в качестве параметра номер учетной записи. Для определения количества элементов в полученной структуре следует воспользоваться свойством UserInfoCount. В листинге 8.17 представлен пример использования этого свойства.

Листинг 8.17. Получение параметров учетной записи из структуры TUserInfo

```
procedure TForm1.Button3Click(Sender: TObject);
begin
 with IBSecurityService1 do
  begin
    ServerName := '127.0.0.1';
    LoginPrompt := False;
```

```
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      UserName := Edit1.Text;
      DisplayUser(UserName);
      Edit2.Text := UserInfo[0].FirstName;
      Edit3.Text := UserInfo[0].MiddleName;
      Edit4.Text := UserInfo[0].LastName;
      Edit5.Text := IntToStr(UserInfo[0].UserID);
      Edit6.Text := IntToStr(UserInfo[0].GroupID);
    finally
      Active := False;
    end;
  end;
end;
```

Свойство `UserName` содержит логин пользователя, а свойство `Password`, соответственно, — пароль.

Для того чтобы определить имя, отчество и фамилию пользователя, можно воспользоваться свойствами `FirstName`, `MiddleName` и `LastName`. Свойства `UserID`, `GroupID` и `SQLRole` позволяют задать идентификатор учетной записи, идентификатор группы и роль.

При помощи свойства `SecurityAction` можно определить тип операции, которую выполнит InterBase Security Service. Возможные значения свойства перечислены в списке:

- Значение `ActionAddUser` указывает, что будет произведено добавление новой учетной записи.
- Значение `ActionDeleteUser` указывает, что будет произведено удаление учетной записи.
- Значение `ActionDisplayUser` указывает, что будет получена информация об учетной записи.
- Значение `ActionModifyUser` указывает, что будет произведено изменение учетной записи.

Метод `AddUser` добавляет новую учетную запись пользователя. Для удаления учетной записи пользователя используется метод `DeleteUser`. Изменение учетной записи производится при помощи метода `ModifyUser`.

В листинге 8.18 приведен пример создания учетной записи. Следует обратить внимание на то, что учетная запись создается из-под учетной записи администратора.

Листинг 8.18. Изменение параметров учетной записи

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with IBSecurityService1 do
  begin
  ServerName := '127.0.0.1';
  LoginPrompt := False;
  Params.Add('user_name=sysdba');
  Params.Add('password=masterkey');
  Active := True;
    try
      UserName := Edit1.Text;
      FirstName := Edit2.Text;
      MiddleName := Edit3.Text;
      LastName := Edit4.Text;
      UserID := StrToInt(Edit5.Text);
      GroupID := StrToInt(Edit6.Text);
      Password := Edit7.Text;
      AddUser;
    finally
    Active := False;
    end;
  end;
end;
```

Компонент TIBServerProperties

Компонент `TIBServerProperties` предназначен для получения настроек сервера InterBase. Свойство `Options` возвращает указатель на структуру, содержащую интересующие параметры. Для прямого обращения к нужным структурам следует воспользоваться соответствующими свойствами. Свойство `DatabaseInfo` позволяет выяснить число прикрепленных файлов и количество баз данных, управляемых данным сервером. Метод `FetchDatabaseInfo` возвращает информацию о базах данных в структуре `TDatabaseInfo`.

В листинге 8.19 приведен пример использования этого свойства. Следует помнить, что InterBase не поддерживает постоянной связи с базами. Для того чтобы протестировать приведенный пример и получить наглядные результаты, следует запустить одно из созданных ранее приложений.

Листинг 8.19. Получение информации о базах данных

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
  I: Integer;
begin
 with IBServerProperties1 do
  begin
    ServerName := '127.0.0.1';
    LoginPrompt := False;
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Options := [Database];
      FetchDatabaseInfo;
      Label1.Caption := 'Number of Attachments = ' +
IntToStr(DatabaseInfo.NoOfAttachments);
      Label2.Caption := 'Number of Databases = ' +
IntToStr(DatabaseInfo.NoOfDatabases);
      for I:= 0 to High(DatabaseInfo.DbName) do
        Memo1.Lines.Add(DatabaseInfo.DbName[i]);
    finally
      Active := False;
    end;
  end;
end;
```

Свойство `LicenseInfo` позволяет получить информацию о лицензиях, поддерживаемых системой. Параметр `Key` содержит ключ лицензии, в параметре `LicensedUsers` указывается количество клиентов, имеющих право одновременно работать с сервером, параметр `Id` содержит идентификатор лицензии, а в параметре `Desc` указывается описание лицензии. Метод `FetchLicenseInfo` возвращает информацию о лицензиях в структуре `TLicenseInfo`.

В листинге 8.20 приведен соответствующий пример.

Листинг 8.20. Получение списка лицензий

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
with IBServerProperties1 do
  begin
    ServerName := '127.0.0.1';
    LoginPrompt := False;
```

продолжение ⇨

Листинг 8.20 *(продолжение)*

```
    Params.Add('user_name=sysdba');
    Params.Add('password=masterkey');
    Active := True;
    try
      Options := [License];
      FetchLicenseInfo;
      Label1.Caption := 'Licensed Users = ' +
IntToStr(LicenseInfo.LicensedUsers);
      for I:= 0 to High(LicenseInfo.Key) do
        Memo1.Lines.Add(LicenseInfo.Key[i] + ':' + LicenseInfo.ID[i] + ':' +
LicenseInfo.Desc[i]);
    finally;
      Active := False;
    end;
  end;
end;
```

Свойство `LicenseMaskInfo` позволяет получить информацию о маске лицензий и количество возможных вариантов маски. А метод `FetchLicenseMaskInfo` возвращает информацию о маске лицензий в виде структуры `TLicenseMaskInfo`. Список параметров сервера содержит свойство `ConfigParams`, в котором содержится *довольно много служебной информации о сервере.*

Метод `FetchConfigParams` возвращает список параметров сервера. При помощи свойства `VersionInfo` разработчик может получить версию используемого сервера, аппаратную платформу, на которой он работает, и номер версии соответствующих сервисов. Метод `FetchVersionInfo` заполняет данными это свойство типа `TVersionInfo`.

Компонент TIBLicensingService

Компонент `TIBLicensingService` *предназначен для управления лицензиями.* Свойство `Action` позволяет определить тип действия, производимого над лицензией. Значение `LicenseAdd` указывает, что лицензия будет добавлена, а значение `LicenseRemove` позволяет удалить лицензию.

В свойстве `Key` указывается ключ лицензии, а в свойстве `ID` — ее идентификатор. Для добавления лицензии можно воспользоваться методом `AddLicense`. Для удаления лицензии используется метод `RemoveLicense`.

В листинге 8.21 приведен пример использования этого компонента.

Листинг 8.21. Регистрация новой лицензии

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 with IBLicensingService1 do
```

```
  begin
    ServerName := '127.0.0.1';
    LoginPrompt := False;
    Protocol := Local;
    Params.Add('user_name=SYSDBA');
    Params.Add('password=masterkey');
    Active := True;
    try
      Key := Edit1.Text;
      ID := Edit2.Text;
      Action := LicenseAdd;
      ServiceStart;
    finally
    if Active then
      Active := False;
    end;
  end;
end;
```

Системные таблицы, временные таблицы и системные представления InterBase

Каждый объект базы данных имеет определенную структуру, которую и называют метаданными. Метаданные в InterBase хранятся вместе с пользовательскими таблицами в системных таблицах. Системные таблицы InterBase имеют префикс RDB$. Системные таблицы создаются автоматически при создании базы данных и изменяются всякий раз, когда выполняются команды, изменяющие структуру базы данных.

По умолчанию право просмотра системных таблиц имеют все пользователи, но право их изменения — только владельцы базы данных и учетной записи SYSDBA. Для просмотра содержания системных таблиц можно использовать системные представления, которые можно создать вручную. Помимо системных таблиц, в своей работе сервер использует временные таблицы, в имени которых обязательно присутствует префикс TMP$. Временные таблицы существуют только во время работы с базой данных.

Системные таблицы InterBase

Таблица RDB$CHARACTER_SETS (табл. 8.2) описывает кодировки символов, которые может использовать сервер InterBase. Кодировка набора данных создается совокупностью параметров.

Таблица 8.2. Структура RDB$CHARACTER_SETS

Наименование поля	Тип данных	Размер	Описание
RDB$CHARACTER_ SET_NAME	CHAR	67	Имя кодовой страницы, распознаваемой InterBase
RDB$FORM_OF_USE	CHAR	67	Предназначено для внутреннего использования
RDB$NUMBER_ OF_CHARACTERS	INTEGER		Число символов в кодовой таблице
RDB$DEFAULT_ COLLATE_NAME	CHAR	67	Порядок сравнения символов кодовой страницы
RDB$CHARACTER_ SET_ID	SMALLINT		Уникальный идентификатор кодовой страницы
RDB$SYSTEM_FLAG	SMALLINT		Кодовая страница определяется: — пользователем (имеет значение 0 или null); — системной установкой (*единичное значение*)
RDB$DESCRIPTION	BLOB		Содержит описание кодовой страницы
RDB$FUNCTION_ NAME	CHAR	67	Предназначено для внутреннего использования
RDB$BYTES_ PER_CHARACTER	SMALLINT		Размер символа в байтах

Таблица RDB$CHECK_CONSTRAINTS (табл. 8.3) содержит информацию об ограничениях ссылочной целостности. Также эта таблица хранит информацию об ограничениях типа NOT NULL.

Таблица 8.3. Структура RDB$CHECK_CONSTRAINTS

Наименование поля	Тип данных	Размер	Описание
RDB$CONSTRAINT_ NAME	CHAR	67	Имя ограничения
RDB$TRIGGER_ NAME	CHAR	67	Имя триггера, имеющего ограничения либо имя поля в RDB$RELATION_FIELDS, имеющее ограничение NOT NULL

Таблица RDB$COLLATIONS (табл. 8.4) содержит данные о порядке сравнения символьных данных, используемого для кодовой страницы, определенной в таблице RDB$CHARACTER_SETS.

Таблица 8.4. Структура RDB$COLLATIONS

Наименование поля	Тип данных	Размер	Описание
RDB$COLLATION_ NAME	CHAR	67	Наименование порядка сравнения символов
RDB$COLLATION_ID	SMALLINT		Уникальный идентификатор порядка сравнения

Наименование поля	Тип данных	Размер	Описание
RDB$CHARACTER_SET_ID	SMALLINT		Идентификатор кодовой страницы для данного порядка сравнения, фактически, определяет используемую кодовую страницу. Данное поле связано с полем RDB$CHARACTER_SET_ID таблицы RDB$CHARACTER_SETS
RDB$COLLATION_ATTRIBUTES	SMALLINT		Предназначено для внутреннего использования
RDB$SYSTEM_FLAG	SMALLINT		Указывает на то, кем определяется генератор. Нулевое значение указывает, что генератор задается пользователем, а ненулевое значение указывает, что генератор определяется системой
RDB$DESCRIPTION	BLOB		Содержит описание порядка сравнения символов
RDB$FUNCTION_NAME	CHAR	67	Предназначено для внутреннего использования

Таблица RDB$DATABASE (табл. 8.5) содержит описание базы данных.

Таблица 8.5. Структура RDB$DATABASE

Наименование поля	Тип данных	Размер	Описание
RDB$DESCRIPTION	BLOB		Содержит описание базы данных. Поле заполняется при создании базы данных либо при использовании инструкций CREATE или ALTER SCHEMA|DATABASE
RDB$RELATION_ID	SMALLINT		Предназначено для внутреннего использования
RDB$SECURITY_CLASS	CHAR	67	Класс безопасности определяется в таблице RDB$SECURITY_CLASSES. Ограничения, описанные в классе безопасности, применяются ко всей базе
RDB$CHARACTER_SET_NAME	CHAR	67	Имя кодовой страницы

Таблица RDB$DEPENDENCIES (табл. 8.6) содержит описание зависимостей между таблицами и полями, от которых зависят другие системные объекты. Эти объекты могут включать в себя представления, триггеры и вычисляемые значения полей. InterBase использует данную таблицу для гарантии того, что используемые другим объектом поле или таблица не будут удалены до тех пор, пока с ними производятся какие-либо действия.

Таблица 8.6. Структура RDB$DEPENDENCIES

Наименование поля	Тип данных	Размер	Описание
RDB$DEPENDENT_NAME	CHAR	67	Содержит имя зависимого объекта
RDB$DEPENDED_ON_NAME	CHAR	67	Имя таблицы, от которой зависит данный объект
RDB$FIELD_NAME	CHAR	67	Имя поля, от которого зависит данный объект
RDB$DEPENDENT_TYPE	SMALLINT		Описывает тип зависимого объекта: 0 — таблица; 1 — представление; 2 — триггер; 3 — вычисляемое поле; 4 — допустимое значение; 5 — процедура; 6 — индексное выражение; 7 — исключение; 8 — учетная запись пользователя; 9 — поле; 10 — индекс

Таблица RDB$EXCEPTIONS (табл. 8.7) содержит описание исключений базы данных, связанных с хранимыми процедурами, включая исключения, определенные пользователем.

Таблица 8.7. Структура RDB$EXCEPTIONS

Наименование поля	Тип данных	Размер	Описание
RDB$EXCEPTION_NAME	CHAR	67	Имя исключения
RDB$EXCEPTION_NUMBER	INTEGER		Номер исключения
RDB$MESSAGE	VARCHAR	78	Текст ошибки
RDB$DESCRIPTION	BLOB		Описание исключения
RDB$SYSTEM_FLAG	SMALLINT		Определяет тип исключения: — пользовательский (имеет значение 0); — системный (ненулевое значение)

В таблице RDB$FIELD_DIMENSIONS (табл. 8.8) содержится описание размерностей полей, являющихся массивами. Для каждого измерения массива используется отдельная запись. Соответственно, для каждого поля создается отдельный блок записей.

Таблица 8.8. Структура RDB$FIELD_DIMENSIONS

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_NAME	CHAR	67	Имя поля (массива), имеющего тип массив. Имя массива должно быть также описано в таблице RDB$FIELDS, в поле RDB$FIELD_NAME

Наименование поля	Тип данных	Размер	Описание
RDB$DIMENSION	SMALLINT		Указывает на номер измерения массива. Первое измерение указывается под номером 0
RDB$LOWER_BOUND	INTEGER		Содержит нижнюю границу измерения массива
RDB$UPPER_BOUND	INTEGER		Содержит верхнюю границу измерения массива

В таблице RDB$FIELDS (табл. 8.9) описываются характеристики полей и доменов. Каждому типу поля или домену соответствует одна строчка в таблице.

Таблица 8.9. Структура RDB$FIELDS

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_NAME	CHAR	67	Уникальное имя домена или системное имя поля (типизированное поле)
RDB$QUERY_NAME	CHAR	67	Не используется для объектов SQL
RDB$VALIDATION_BLR	BLOB		Не используется для объектов SQL
RDB$VALIDATION_SOURCE	BLOB		Не используется для объектов SQL
RDB$COMPUTED_BLR	BLOB		Содержит BLR (Binary Language Representation) — выражение, вычисляемое во время выполнения
RDB$COMPUTED_SOURCE	BLOB		Для вычисляемых полей, содержит исходное символьное выражение поля
RDB$DEFAULT_VALUE	BLOB		Содержит выражение типа BLR, используемое по умолчанию
RDB$DEFAULT_SOURCE	BLOB		Содержит выражение, используемое по умолчанию (в текстовом формате)
RDB$FIELD_LENGTH	SMALLINT		Содержит размер несимвольного поля
RDB$FIELD_PRECISION	SMALLINT		Содержит разрядность поля для числовых полей
RDB$FIELD_SCALE	SMALLINT		Содержит точность поля для числовых полей
RDB$FIELD_TYPE	SMALLINT		Содержит тип поля: BLOB — 261; CHAR — 14; CSTRING — 40; D_FLOAT — 11; DOUBLE — 27; FLOAT — 10; INT64 — 16; INTEGER — 8; QUAD — 9; SMALLINT — 7; DATE — 12; TIME — 13; VARCHAR — 37

продолжение

Таблица 8.9 *(продолжение)*

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_ SUB_TYPE	SMALLINT		Используется для различения подтипов BLOB, CHAR и INTEGER
RDB$MISSING_VALUE	BLOB		Не используется для объектов SQ
RDB$MISSING_ SOURCE	BLOB		Не используется для объектов SQL
RDB$DESCRIPTION	BLOB		Содержит описание типа поля или домена, присвоенное пользователем
RDB$SYSTEM_FLAG	SMALLINT		Используется для системных таблиц
RDB$QUERY_ HEADER	BLOB		Не используется для объектов SQL
RDB$SEGMENT_ LENGTH	SMALLINT		Содержит длину сегмента типа BLOB
RDB$EDIT_STRING	VARCHAR	125	Не используется для объектов SQL
RDB$EXTERNAL_ LENGTH	SMALLINT		Содержит размер поля внешней, подключенной таблицы. Для внутренних таблиц имеет нулевое значение
RDB$EXTERNAL_ SCALE	SMALLINT		Определяет наличие точной части числового поля внешней таблицы
RDB$EXTERNAL_ TYPE	SMALLINT		То же что и RDB$FIELD_TYPE
RDB$DIMENSIONS	SMALLINT		Для полей-массивов определяет число измерений. Для остальных типов полей имеет нулевое значение
RDB$NULL_FLAG	SMALLINT		Определяет, может ли поле принимать значение NULL. Значение EMPTY указывает, что это допускается, а единичное — что нет
RDB$CHARACTER_ LENGTH	SMALLINT		Длина символа в байтах
RDB$COLLATION_ID	SMALLINT		Уникальный идентификатор порядка сравнения
RDB$CHARACTER_ SET_ID	SMALLINT		Уникальный идентификатор кодовой страницы. Фактически, ссылается на поле CHARACTER_ SET_ID таблицы RDB$CHARACTER_SETS

Таблица RDB$FILES (табл. 8.10) содержит список вторичных файлов и Shadow-файлов базы данных.

Таблица 8.10. RDB$FILES

Наименование поля	Тип данных	Размер	Описание
RDB$FILE_NAME	VARCHAR	253	Содержит имя вторичного файла или файла Shadow
RDB$FILE_SEQUENCE	SMALLINT		Порядковый номер вторичного файла или файла Shadow
RDB$FILE_START	INTEGER		Указывает номер страницы, с которой начинается вторичный файл или файл Shadow
RDB$FILE_LENGTH	INTEGER		Длина файла в блоках
RDB$FILE_FLAGS	SMALLINT		Используется в системных целях
RDB$SHADOW_ NUMBER	SMALLINT		Указывает число, идентифицирующее набор файлов Shadow. Если число равно 0, то файл является вторичным

В таблице RDB$FILTERS (табл. 8.11) содержится информация о фильтрах полей типа BLOB.

Таблица 8.11. Структура RDB$FILTERS

Наименование поля	Тип данных	Размер	Описание
RDB$FUNCTION_NAME	CHAR	67	Уникальное имя фильтра
RDB$DESCRIPTION	BLOB	80	Описание фильтра, присвоенное пользователем
RDB$MODULE_ NAME	VARCHAR	253	Имя библиотеки, содержащей фильтр
RDB$ENTRYPOINT	CHAR	31	Точка входа в библиотеку, используемая для доступа к данному фильтру
RDB$INPUT_ SUB_TYPE	SMALLINT		Структура типа BLOB, используемая для ввода данных
RDB$OUTPUT_ SUB_TYPE	SMALLINT		Структура типа BLOB, используемая для вывода данных
RDB$SYSTEM_FLAG	SMALLINT		Указывает на то, кто создал фильтр. Нулевое значение говорит о том, что фильтр создал пользователь, а ненулевое — система

В таблице RDB$FORMATS (табл. 8.12) содержится история изменения структуры таблиц. Сервер InterBase назначает таблице новый номер формата при каждом изменении определения поля. Прямые операции с метаданными, такие как запросы ALTER TABLE, тоже увеличивают версию формата. То же самое относится к операциям создания и удаления триггеров.

Сервер InterBase допускает только 255 изменений структуры таблицы. При превышении данного предела дальнейшие попытки внести изменения в струк-

туру таблиц не увенчаются успехом. Для того чтобы вновь получить возможность изменять структуру таблиц, необходимо произвести резервирование базы данных и ее восстановление.

Таблица 8.12. Структура RDB$FORMATS

Наименование поля	Тип данных	Размер	Описание
RDB$RELATION_ID	SMALLINT		Содержит имя таблицы, в которую вносятся изменения
RDB$FORMAT	SMALLINT		Содержит текущий номер формата таблицы
RDB$DESCRIPTOR	BLOB		Содержит список полей таблицы, с описанием типов данных и размеров

В таблице RDB$FUNCTION_ARGUMENTS (табл. 8.13) содержится описание параметров пользовательских функций.

Таблица 8.13. Структура RDB$FUNCTION_ARFUMENTS

Наименование поля	Тип данных	Размер	Описание
RDB$FUNCTION_ NAME	CHAR	67	Имя функции. Должно быть уникальным. Связано с именем функции, содержащимся в таблице RDB$FUNCTIONS
RDB$ARGUMENT_ POSITION	SMALLINT		Номер параметра в списке параметров
RDB$MECHANISM	SMALLINT		Определяет способ передачи параметра. Нулевое значение говорит о том, что параметр передается по значению, а единичное значение — по ссылке
RDB$FIELD_TYPE	SMALLINT		Содержит тип параметра: BLOB — 261; CHAR — 14; CSTRING — 40; D_FLOAT — 11; DOUBLE — 27; FLOAT — 10; INT64 — 16; INTEGER — 8; QUAD — 9; SMALLINT — 7; DATE — 12; TIME — 13; VARCHAR — 37
RDB$FIELD_SCALE	SMALLINT		Определяет точность числового типа данных (дробная часть)
RDB$FIELD_LENGTH	SMALLINT		Размер параметра
RDB$FIELD_ SUB_TYPE	SMALLINT		Поле определяет для параметров, имеющих тип SMALLINT, INTEGER и INT64, как они будут трактоваться: 0 или NULL — значение RDB$FIELD_TYPE; 1 — числовое; 2 — десятичное

Наименование поля	Тип данных	Размер	Описание
RDB$CHARACTER_ SET_ID	SMALLINT		Уникальный идентификатор кодовой страницы.
RDB$FIELD_ PRECISION	SMALLINT		Определяет разрядность числового типа данных.

Таблица RDB$FUNCTIONS (табл. 8.14) содержит описание функций UDF. Для получения полного описания пользовательской функции необходимо также использовать только что рассмотренную таблицу RDB$FUNCTION_ARGUMENTS, связь с которой осуществляется по полю RDB$FUNCTION_NAME.

Таблица 8.14. Структура RDB$FUNCTIONS

Наименование поля	Тип данных	Размер	Описание
RDB$FUNCTION_ NAME	CHAR	67	Имя функции. Должно быть уникальным. Связано с именем функции, содержащимся в таблице RDB$FUNCTIONS
RDB$FUNCTION_ TYPE	SMALLINT		Зарезервировано
RDB$QUERY_NAME	CHAR	67	Альтернативное название функции. Может быть использовано в isql
RDB$DESCRIPTION	BLOB		Описание функции
RDB$MODULE_NAME	VARCHAR	253	Имя библиотеки, содержащей функцию UDF
RDB$ENTRYPOINT	CHAR	31	Точка входа в библиотеку, используемая для доступа обращения к функции UDF
RDB$RETURN_ ARGUMENT	SMALLINT		Номер параметра из списка параметров, возвращающего результат выполнения функции
RDB$SYSTEM_FLAG	SMALLINT		0 — пользовательская функция; 1 — системная

В таблице RDB$GENERATORS содержится информация о генераторах, предназначенных для создания уникальных значений полей.

Таблица 8.15. Структура RDB$GENERATORS

Наименование поля	Тип данных	Размер	Описание
RDB$GENERATOR_ NAME	CHAR	67	Имя генератора. Является уникальным
RDB$GENERATOR_ID	SMALLINT		Уникальный, назначаемый сервером номер генератора
DB$SYSTEM_FLAG			Нулевое значение говорит о том, что генератор создал пользователь, а единичное — сервер

Таблица RDB$INDEX_SEGMENTS (табл. 8.16) содержит информацию о столбцах, входящих в индекс. Модификация записей в таблице приводит к тому, что после следующей транзакции записи создаются заново. Для каждого поля, входящего в индекс, имеется соответствующая запись в таблице.

Таблица 8.16. Структура RDB$INDEX_SEGMENTS

Наименование поля	Тип данных	Размер	Описание
RDB$INDEX_NAME	CHAR	67	Имя индекса, в который входит данное поле
RDB$FIELD_NAME	CHAR	67	Имя поля, входящего в индекс
RDB$FIELD_POSITION	SMALLINT		Номер поля в индексе в соответствии с принятой сортировкой

В таблице RDB$INDICES (табл. 8.17) описывается структура индексов. Так как InterBase предоставляет возможность создания индексов как по одному полю, так и по нескольким, с данной таблицей связана таблица RDB$INDEX_SEGMENTS.

Таблица 8.17. Таблица RDB$INDICES

Наименование поля	Тип данных	Размер	Описание
RDB$INDEX_NAME	CHAR	31	Имя индекса
RDB$RELATION_NAME	CHAR	31	Имя индексированной таблицы
RDB$INDEX_ID	SMALLINT		Содержит внутренний идентификатор индекса
RDB$UNIQUE_FLAG	SMALLINT		Определяет, может ли индекс содержать дубликаты. Нулевое значение показывает, что дубликаты допустимы, а единичное — свидетельствует, что значения, входящие в индекс, уникальны
RDB$DESCRIPTION	BLOB		Описание индекса
RDB$SEGMENT_COUNT	SMALLINT		Количество полей, составляющих индекс. Если поле имеет значение 1, то индекс является простым
RDB$INDEX_INACTIVE	SMALLINT		Определяет состояние индекса. Нулевое значение говорит о том, что индекс активен. Для неактивного состояния используется единичное значение
RDB$INDEX_TYPE	SMALLINT		Порядок сортировки значений поля. Нулевое значение задает сортировку по возрастанию, а единичное — по убыванию
RDB$FOREIGN_KEY	CHAR	31	Имя внешнего ключа, для которого определен данный индекс

Наименование поля	Тип данных	Размер	Описание
RDB$SYSTEM_FLAG	SMALLINT		Нулевое значение говорит о том, что индекс создал пользователь, а единичное — сервер
RDB$EXPRESSION_BLR	BLOB		Содержит вычисляемое выражение типа BLR
RDB$EXPRESSION_SOURCE	BLOB		Содержит исходный текст вычисляемого выражения
RDB$STATISTICS	DOUBLE PRECISION		Коэффициент селективности. Используется оптимизатором сервера для выбора оптимальной стратегии выборки данных

Таблица RDB$PAGES (табл. 8.18) содержит историю выделения страниц в базе данных. Модификация данных в этой таблице вызовет повреждение базы данных.

Таблица 8.18. Структура RDB$PAGES

Наименование поля	Тип данных	Размер	Описание
RDB$PAGE_NUMBER	INTEGER		Номер физически выделенной страницы
RDB$RELATION_ID	SMALLINT		Идентификатор таблицы, для которой было произведено выделение
RDB$PAGE_SEQUENCE	INTEGER		Номер выделенной страницы
RDB$PAGE_TYPE	SMALLINT		Описывает тип страницы. Предназначено для системного использования

Таблица RDB$PROCEDURE_PARAMETERS (табл. 8.19) содержит описание параметров хранимых процедур.

Таблица 8.19. Структура RDB$PROCEDURE_PARAMETERS

Наименование поля	Тип данных	Размер	Описание
RDB$PARAMETER_NAME	CHAR	67	Имя параметра
RDB$PROCEDURE_NAME	CHAR	67	Имя процедуры, которой принадлежит данный параметр
RDB$PARAMETER_NUMBER	SMALLINT		Порядковый номер параметра
RDB$PARAMETER_TYPE	SMALLINT		Тип параметра. Нулевое значение применяется для входных параметров, а единичное — для выходных
RDB$FIELD_SOURCE	CHAR	31	Глобальное имя столбца

продолжение

Таблица 8.19 *(продолжение)*

Наименование поля	Тип данных	Размер	Описание
RDB$DESCRIPTION	BLOB		Описание хранимой процедуры
RDB$SYSTEM_FLAG	SMALLINT		Нулевое значение поля указывает, что параметр создан пользователем. Ненулевое значение используется для параметров, созданных сервером

В таблице RDB$PROCEDURES (табл. 8.20) хранится информация обо всех хранимых процедурах базы данных.

Таблица 8.20. Структура RDB$PROCEDURES

Наименование поля	Тип данных	Размер	Описание
RDB$PROCEDURE_NAME	CHAR	67	Имя хранимой процедуры
RDB$PROCEDURE_ID	SMALLINT		Идентификатор хранимой процедуры
RDB$PROCEDURE_INPUTS	SMALLINT		Количество входных параметров
PROCEDURE_OUTPUTS	SMALLINT		Количество выходных параметров
RDB$DESCRIPTION	BLOB		Описание хранимой процедуры
RDB$PROCEDURE_SOURCE	BLOB		Исходный код хранимой процедуры
RDB$PROCEDURE_BLR	BLOB		BLR-код хранимой процедуры
RDB$SECURITY_CLASS	CHAR	67	Класс безопасности процедуры
RDB$OWNER_NAME	CHAR	67	Учетная запись, создавшая процедуру
RDB$RUNTIME	BLOB		Описание метаданных процедуры. Используется для увеличения производительности
RDB$SYSTEM_FLAG	SMALLINT		Нулевое значение поля указывает, что процедура создана пользователем. Ненулевое значение используется для процедур, созданных сервером

В таблице RDB$REF_CONSTRAINTS (табл. 8.21) хранится информация об ограничениях ссылочной целостности.

Таблица 8.21. Структура RDB$REF_CONSTRAINTS

Наименование поля	Тип данных	Размер	Описание
RDB$CONSTRAINT_NAME	CHAR	67	Имя ограничения ссылочной целостности
RDB$CONST_NAME_UQ	CHAR	67	Имя ограничения первичного или уникального ключа

Наименование поля	Тип данных	Размер	Описание
RDB$MATCH_ OPTION	CHAR	7	Зарезервировано. По умолчанию принимает значение FULL
RDB$UPDATE_RULE	CHAR	11	Определяет тип действия по отношению к внешнему ключу (Foreign key), когда первичный ключ (Primary key) обновляется: NO ACTION; CASCADE; SET NULL; SET DEFAULT
RDB$DELETE_RULE	CHAR	11	Определяет тип действия по отношению к внешнему ключу (Foreign key), когда первичный ключ (Primary key) удаляется: – NO ACTION; – CASCADE; – SET NULL; – SET DEFAULT

Таблица RDB$RELATION_CONSTRAINTS (табл. 8.22) хранит информацию об ограничениях, наложенных на таблицы. Если ограничение связано с индексом, то в поле RDB$INDEX_NAME указывается имя индекса, таким образом создается ссылка на таблицу RDB$INDICES.

Таблица 8.22. Структура RDB$RELATION_CONSTRAINTS

Наименование поля	Тип данных	Размер	Описание
RDB$CONSTRAINT_ NAME	CHAR	67	Имя ограничения
RDB$CONSTRAINT_ TYPE	CHAR	11	Тип ограничения таблицы: PRIMARY KEY, UNIQUE, FOREIGN KEY, PCHECK, NOT NULL
RDB$RELATION_NAME	CHAR	67	Имя таблицы, для которой определено ограничение
RDB$DEFERRABLE	CHAR	3	Зарезервировано. По умолчанию принимает значение NO
RDB$INITIALLY_ DEFERRED	CHAR	3	Зарезервировано. По умолчанию принимает значение NO
RDB$INDEX_NAME	CHAR	67	Имя индекса, используемого ограничениями PRIMARY KEY, UNIQUE, FOREIGN KEY

Таблица RDB$RELATION_FIELDS (табл. 8.23) содержит список полей и их описания для доменов.

Таблица 8.23. Структура RDB$RELATION_FIELDS

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_NAME	CHAR	67	Имя поля
RDB$RELATION_NAME	CHAR	67	Имя таблицы, которой принадлежит данное поле

продолжение

Таблица 8.23 *(продолжение)*

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_SOURCE	CHAR	31	Имя описания поля в таблице RDB$FIELDS. Если поле построено на базе домена, то указывается имя домена
RDB$QUERY_NAME		31	Альтернативное имя поля. Может использоваться в утилите isql
RDB$BASE_FIELD	CHAR	31	Используется только для представлений. Содержит имя поля в таблице или представлении, которое является базовым для определенного поля
RDB$EDIT_STRING	VARCHAR	125	Не используется в SQL
RDB$FIELD_POSITION	SMALLINT		Номер поля в списке
RDB$QUERY_HEADER	BLOB		Не используется в SQL
RDB$UPDATE_FLAG	SMALLINT		Не используется в InterBase
RDB$FIELD_ID	SMALLINT		Идентификатор для использования в BLR для именования столбца
RDB$VIEW_CONTEXT	SMALLINT		Псевдоним, используемый для уточнения поля представления
RDB$DESCRIPTION	BLOB		Описание поля
RDB$DEFAULT_VALUE	BLOB		Выражение BLR, используемое по умолчанию
RDB$SYSTEM_FLAG	SMALLINT		Нулевое значение указывает, что поле создано пользователем, а ненулевое — сервером
RDB$SECURITY_CLASS	CHAR	67	Имя класса безопасности, определенного в таблице RDB$SECURITY_CLASSES. Накладывает ограничения на возможности работы с полем
RDB$COMPLEX_NAME	CHAR	67	Зарезервировано
RDB$NULL_FLAG	SMALLINT		Определяет, может ли поле содержать значение NULL
RDB$DEFAULT_SOURCE	BLOB		Код SQL столбца
RDB$COLLATION_ID	SMALLINT		Идентификатор порядка сравнения

Таблица RDB$RELATIONS (табл. 8.24) содержит описание таблиц и представлений базы данных.

Таблица 8.24. Структура RDB$RELATION_FIELDS

Наименование поля	Тип данных	Размер	Описание
RDB$VIEW_BLR	BLOB		Используется для представлений. Содержит запрос BLR

Наименование поля	Тип данных	Размер	Описание
RDB$VIEW_SOURCE	BLOB		Используется для представлений. Содержит текст запроса, определяющий структуру представления
RDB$_DESCRIPTION	BLOB		Описание таблицы
RDB$RELATION_ID			Содержит внутренний идентификатор, используемый в запросах BLR
RDB$SYSTEM_FLAG	SMALLINT		Определяет содержание таблицы. Нулевое значение задает пользовательское наполнение таблицы, а положительное значение — системное
RDB$DBKEY_LENGTH	SMALLINT		Длина ключа базы данных
RDB$FORMAT	SMALLINT		Для внутреннего использования
RDB$FIELD_ID	SMALLINT		Количество полей в таблице
RDB$RELATION_NAME	CHAR	67	Имя таблицы (уникальное)
RDB$SECURITY_CLASS	CHAR	67	Имя класса безопасности, определенного в таблице RDB$SECURITY_CLASSES. Накладывает ограничения на работу с таблицей
RDB$EXTERNAL_FILE	VARCHAR	253	Имя файла, в котором содержится внешняя таблица
RDB$RUNTIME	BLOB		Метаданные таблицы
RDB$EXTERNAL_ DESCRIPTION	BLOB		Описание внешнего файла
RDB$OWNER_NAME	CHAR	67	Учетная запись владельца таблицы
RDB$DEFAULT_CLASS	CHAR	31	Класс безопасности, применяемый к добавляемым в таблицу полям
RDB$FLAGS	SMALLINT		1 — триггер, созданный SQL-запросом; 2 — игнорирование ограничений

Таблица RDB$ROLES (табл. 8.25) содержит список ролей базы данных и их владельцев.

Таблица 8.25. Структура RDB$ROLES

Наименование поля	Тип данных	Размер	Описание
RDB$ROLE_NAME	CHAR	67	Имя роли
RDB$OWNER_NAME	CHAR	67	Имя владельца роли

Таблица RDB$SECURITY_CLASSES (табл. 8.26) содержит список доступа к базе данных, таблицам и представлениям. Для всех объектов базы данных информация этой таблицы дублирована в RDB$USER_PRIVILEGES.

Таблица 8.26. Структура RDB$SECURITY_CLACSSES

Наименование поля	Тип данных	Размер	Описание
RDB$SECURITY_CLASS	CHAR	67	Имя класса безопасности
RDB$ACL	BLOB		Список контроля доступа ACL (Access Control List), в котором указаны пользователи и присущие им права
RDB$DESCRIPTION	BLOB		Описание класса безопасности

В таблице RDB$TRANSACTIONS (табл. 8.27) хранится история транзакций, работающих с несколькими базами данных. Транзакции, работающие с одной базой данных, в таблицу не заносятся.

Таблица 8.27. Структура RDB$TRANSACTIONS

Наименование поля	Тип данных	Размер	Описание
RDB$TRANSACTION_ID	INTEGER		Идентификатор транзакции
RDB$TRANSACTION_ STATE	SMALLINT		Описывает состояние транзакции: 0 — Limbo; 1 — Commited; 2 — Rolled back
RDB$TIMESTAMP	DATE		Зарезервировано
RDB$TRANSACTION_ DESCRIPTION			Описание подготовленной транзакции, доступной при неудаче повторного соединения

Таблица RDB$TRIGGER_MESSAGES (табл. 8.28) описывает сообщения системных триггеров и ассоциирует их с конкретными триггерами.

Таблица 8.28. Структура RDB$TRIGGER_MESSAGES

Наименование поля	Тип данных	Размер	Описание
RDB$TRIGGER_NAME	CHAR	67	Имя триггера, с которым ассоциировано данное сообщение
RDB$MESSAGE_ NUMBER	SMALLINT		Номер сообщения
RDB$MESSAGE	VARCHAR	78	Текст сообщения

В таблице RDB$TRIGGERS (табл. 8.29) содержится описание всех триггеров базы данных.

Таблица 8.29. Структура RDB$TRIGGERS

Наименование поля	Тип данных	Размер	Описание
RDB$TRIGGER_NAME	CHAR	67	Имя триггера
RDB$RELATION_NAME	CHAR	67	Имя таблицы, с которой связан данный триггер
RDB$TRIGGER_ SEQUENCE	SMALLINT		Номер триггера, который определяет последовательность выполнения триггеров

Наименование поля	Тип данных	Размер	Описание
RDB$TRIGGER_TYPE	SMALLINT		Тип триггера: 1 – BEFORE INSERT; 2 – AFTER INSERT; 3 – BEFORE UPDATE; 4 – AFTER UPDATE; 5 – BEFORE DELETE; 6 – AFTER DELETE
RDB$TRIGGER_SOURCE	BLOB		Исходный текст триггера
RDB$TRIGGER_BLR	BLOB		Текст триггера (BLR)
RDB$DESCRIPTION	BLOB		Описание триггера
RDB$TRIGGER_INACTIVE	SMALLINT		Определяет состояние триггера: 0 – триггер активен; > 0 – триггер неактивен
RDB$SYSTEM_FLAG	SMALLINT		Триггер создан: 0 – пользователем; > 0 – сервером

В таблице RDB$TYPES (табл. 8.30) содержится описание типов данных, псевдонимов символьных наборов и порядков сравнения.

Таблица 8.30. Структура RDB$TYPES

Наименование поля	Тип данных	Размер	Описание
RDB$FIELD_NAME	CHAR	67	Имя поля, для которого определяется тип
RDB$TYPE	SMALLINT		Содержит внутренний код, идентифицирующий тип поля: 0 – таблица; 1 – представление; 2 – триггер; 3 – вычисляемое поле; 4 – ограничение; 5 – процедура
RDB$TYPE_NAME	CHAR	67	Текст, соответствующий внутреннему коду
RDB$DESCRIPTION	BLOB		Описание типа данных
RDB$SYSTEM_FLAG	SMALLINT		Тип данных определен: 0 – пользователем; > 0 – сервером

Таблица RDB$USER_PRIVILEGES (табл. 8.31) содержит сведения о выданных пользователям правах. Таблица формируется при вызове команды GRANT.

Таблица 8.31. Структура RDB$USER_PRIVELEGES

Наименование поля	Тип данных	Размер	Описание
RDB$USER	CHAR	31	Имя пользователя
RDB$GRANTOR	CHAR	31	Имя пользователя, присвоившего данные привилегии

продолжение

Таблица 8.31 *(продолжение)*

Наименование поля	Тип данных	Размер	Описание
RDB$PRIVILEGE	CHAR	6	Привилегии, доступные данному пользователю: ALL SELECT; DELETE; INSERT; UPDATE; REFERENCE; MEMBER OF (для ролей)
RDB$GRANT_OPTION	SMALLINT		Определяет, как была выдана привилегия: 1 — с опцией WITH GRANT OPTION; 0 — без данной опции
RDB$RELATION_NAME	CHAR	67	Определяет таблицу, на которую распространяется привилегия
RDB$FIELD_NAME	CHAR	67	Для привилегии UPDATE содержит имя поля, на которое она распространяется

В таблице RDB$VIEW_RELATIONS (табл. 8.32) содержится описание представлений базы данных.

Таблица 8.32. Структура RDB$VIEW_RELATIONS

Наименование поля	Тип данных	Размер	Описание
RDB$VIEW_NAME	CHAR	67	Имя представления
RDB$RELATION_NAME	CHAR	67	Имя таблицы, на основе которой строится представление
RDB$VIEW_CONTEXT	SMALLINT		Номер, используемый для уточнения состава полей представления. Используется в представлениях BLR
RDB$CONTEXT_NAME	CHAR	67	Текстовое представление поля RDB$VIEW_CONTEXT

Временные таблицы

Сервер InterBase хранит большой массив информации о базах данных, соединениях, транзакциях и выражениях. Эту информацию можно получить из соответствующих временных таблиц:

- Таблица TMP$ATTACHMENTS содержит описание соединения с базой данных.
- Таблица TMP$DATABASE содержит информацию о выделенных базе данных ресурсах.
- Таблица TMP$MEMORY содержит информацию о состоянии блоков памяти пула.
- Таблица TMP$POOLS содержит описание состояния всех пулов базы данных. Пул представляет собой выделенный под служебные цели массив памяти.

- Таблица TMP$PROCEDURES содержит информацию о выполняемых хранимых процедурах.
- Таблица TMP$RELATIONS содержит информацию об использовании таблиц баз данных.
- Таблица TMP$STATEMENTS содержит информацию о выполнении SQL-запросов.
- Таблица TMP$TRANSACTIONS содержит информацию о ходе выполнения транзакций.

Получить информацию из временных системных таблиц так же просто, как и из любых других таблиц баз данных. Чтобы просмотреть информацию из временных системных таблиц, можно воспользоваться утилитой isql.exe. Команда SHOW SYSTEM выведет список всех системных таблиц, а команда SHOW TABLE имя таблицы позволит получить список полей. После этого можно будет получить информацию из таблицы при помощи выражения SELECT.

Для соединения с базой данных MY.GDB необходимо запустить утилиту isql.exe. Затем потребуется последовательно ввести команды:

```
CONNECT D:\MyIntDB\my.gdb
USER SYSDBA
PASSWORD masterkey
;
```

Символ точки с запятой указывает на окончание ввода группы команд. В результате будет отображено сообщение о том, что было произведено соединение с базой данных. На рис. 8.22 приведен фрагмент окна, иллюстрирующий этот процесс.

Рис. 8.22. Соединение с базой данных при помощи утилиты isql

Системные представления InterBase

Системные представления позволяют легко получать различную информацию о базе данных. Сервер InterBase поддерживает четыре представления, которые определены в стандарте SQL-92. Имена системных представлений, как и названия возвращаемых полей, не имеют префикса RDB$.

Представление CHECK_CONSTRAINTS возвращает все ограничения CHECK, определенные в базе данных. Объявляется это представление довольно простым SQL-запросом:

```
CREATE VIEW CHECK_CONSTRAINTS (CONSTRAINT_NAME,CHECK_CLAUSE) AS
SELECT RDB$CONSTRAINT_NAME, RDB$TRIGGER_SOURCE
FROM RDB$CHECK_CONSTRAINTS RC, RDB$TRIGGERS RT
WHERE RT.RDB$TRIGGER_NAME = RC.RDB$TRIGGER_NAME;
```

В табл. 8.33 приведена структура этого представления.

Таблица 8.33. Структура CHECK_CONSTRAINTS

Наименование поля	Тип данных	Размер	Описание
CONSTRAINT_NAME	CHAR	67	Имя CHECK-ограничения
CHECK_CLAUSE	BLOB		Выражение, на соответствие которому производится проверка (триггером)

Представление CONSTRAINTS_COLUMN_USAGE содержит названия полей, используемые в ограничениях PRIMARY KEY и UNIQUE. Для ограничений типа FOREIGN KEY представление описывает столбцы, определяющие эти ограничения:

```
CREATE VIEW CONSTRAINTS_COLUMN_USAGE (TABLE_NAME,COLUMN_NAME,
CONSTRAINT_NAME
) AS
SELECT RDB$RELATION_NAME, RDB$FIELD_NAME, RDB$CONSTRAINT_NAME
FROM RDB$RELATION_CONSTRAINTS RC, RDB$INDEX_SEGMENTS RI
WHERE RI.RDB$INDEX_NAME = RC.RDB$INDEX_NAME;
```

В табл. 8.34 приведена структура этого представления.

Таблица 8.34. Структура CONSTRAINTS_COLUMN_USAGE

Наименование поля	Тип данных	Размер	Описание
TABLE_NAME	CHAR	67	Имя таблицы, для которой определено ограничение
COLUMN_NAME	CHAR	67	Имя поля, используемого в ограничении
CONSTRAINT_NAME	CHAR	67	Имя ограничения. Значение должно быть уникальным

Представление REFERENTIAL_CONSTRAINTS содержит описание всех ограничений ссылочной целостности, заданных в базе данных. Объявляется это представление соответствующим SQL-запросом:

```
CREATE VIEW REFERENTIAL_CONSTRAINTS (
CONSTRAINT_NAME,
UNIQUE_CONSTRAINT_NAME,
MATCH_OPTION,
UPDATE_RULE,
DELETE_RULE
) AS
```

```
SELECT RDB$CONSTRAINT_NAME, RDB$CONST_NAME_UQ, RDB$MATCH_OPTION,
RDB$UPDATE_RULE, RDB$DELETE_RULE
FROM RDB$REF_CONSTRAINTS;
```

В табл. 8.35 приведена структура этого представления.

Таблица 8.35. Структура REFERENTIAL_CONSTRAINTS

Наименование поля	Тип данных	Размер	Описание
CONSTRAINT_NAME	CHAR	67	Имя ограничения ссылочной целостности
UNIQUE_ CONSTRAINT_NAME	CHAR	67	Имя ограничения первичного или уникального ключа, связанное с определенным списком полей
MATCH_OPTION	CHAR	7	Зарезервировано. По умолчанию принимает значение FULL
UPDATE_RULE	CHAR	11	Зарезервировано. По умолчанию принимает значение RESTRICT
DELETE_RULE	CHAR	11	Зарезервировано. По умолчанию принимает значение RESTRICT

Представление TABLE_CONSTRAINTS описывает все ограничения, созданные в базе данных, для всех таблиц:

```
CREATE VIEW TABLE_CONSTRAINTS (
CONSTRAINT_NAME,
TABLE_NAME,
CONSTRAINT_TYPE,
IS_DEFERRABLE,
INITIALLY_DEFERRED
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$RELATION_NAME,
RDB$CONSTRAINT_TYPE, RDB$DEFERRABLE, RDB$INITIALLY_DEFERRED
FROM RDB$RELATION_CONSTRAINTS;
```

В табл. 8.36 приведена структура этого представления.

Таблица 8.36. Структура TABLE_CONSTRAINTS

Наименование поля	Описание
CONSTRAINT_NAME	Имя ограничения
TABLE_NAME	Имя таблицы, для которой определено ограничение
CONSTRAINT_TYPE	Возможные значения: UNIQUE; PRIMARY KEY; FOREIGN KEY; CHECK
IS_DEFERRABLE	Зарезервировано. По умолчанию принимает значение NO
INITIALLY_DEFERRED	Зарезервировано. По умолчанию принимает значение NO

Логика приложения

Сервер предоставляет в распоряжение пользователя мощный диалект языка SQL. В этом разделе будет рассмотрено создание различных объектов базы данных, работа с хранимыми процедурами и многие другие технологии, без которых нельзя полностью задействовать все возможности сервера InterBase.

Работа с доменами

Доменом называют глобальное описание типа поля, доступное всей базе данных. Как правило, домены используются в том случае, когда множество таблиц базы данных содержит идентичные описания полей. Поля, созданные на базе доменов, наследуют все их характеристики. Некоторые из этих характеристик могут быть переопределены в каждом конкретном случае. Синтаксис кода, определяющего домен, не так уж и сложен:

```
CREATE DOMAIN имя домена [AS] <datatype>
[DEFAULT { literal | NULL | USER}]
[NOT NULL] [CHECK ( <dom_search_condition>)]
[COLLATE collation];
```

При создании домена в базе данных следует определить уникальное имя и необходимые атрибуты. Обязательным атрибутом для домена является тип данных поля. В примере показано, как определяется домен для символьного поля длиной 20 символов:

```
CREATE DOMAIN TEST AS VARCHAR(20);
```

В определении домена можно указать значение, которое будет присваиваться полю по умолчанию. Это значение указывается после оператора DEFAULT. Значение по умолчанию может иметь строковый тип, числовой, дату, NULL либо USER. Константа USER содержит в себе имя пользователя, работающего с базой в данный момент, в данной сессии. Это иллюстрирует следующий пример:

```
CREATE DOMAIN TESTDOMAIN AS VARCHAR(20) DEFAULT USER;
CREATE TABLE TESTTABLE (Order_Date DATE, Entered_By TESTDOMAIN);
INSERT INTO TESTTABLE (ORDER_DATE)
VALUES ('01.01.2005');
```

Значением по умолчанию является константа USER. Данный код создает таблицу TESTTABLE с двумя полями. При вводе одного из значений в поле даты Order_Date во второе поле Entered_By, созданное на базе домена TESTDOMAIN, автоматически будет вводиться новое значение.

Также можно ввести требование обязательного ввода значения в объявляемое поле. Реализуется это при помощи оператора NOT NULL, как показано в примере:

```
CREATE DOMAIN TEST1 INTEGER NOT NULL;
```

Помимо требования обязательного ввода значения на домен можно наложить и другие ограничения. Ограничение CHECK производит проверку вводимых в по-

ле данных на соответствие определенным условиям. С оператором CHECK могут быть использованы другие операторы, которые указаны в синтаксисе:

```
<операторы соответствия> = {
VALUE <operator> <val>
| VALUE [NOT] BETWEEN <val> AND <val>
| VALUE [NOT] LIKE <val> [ESCAPE <val>]
| VALUE [NOT] IN ( <val> [ , <val> …])
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING <val>
| VALUE [NOT] STARTING [WITH] <val>
| (<dom_search_condition>)
| NOT <dom_search_condition>
| <dom_search_condition> OR <dom_search_condition>
| <dom_search_condition> AND <dom_search_condition>
}
<операторы отношения> = {= | < | > | <= | >= | !< | !> | <> | !=}
```

Можно привести небольшой пример использования этого ограничения:

```
CREATE DOMAIN CHECKDOMAIN AS INTEGER DEFAULT 162
CHECK (VALUE > 10);
```

Этот фрагмент кода указывает, что в целочисленном поле может быть сохранено значение, не превышающее десяти.

Оператор COLLATE позволяет определить порядок сравнения символьных данных:

```
CREATE DOMAIN COLDOM AS CHAR(30)
CHARACTER SET WIN1251 COLLATE PXW_CYRL;
```

Для изменения определения домена следует воспользоваться командой ALTER DOMAIN, синтаксис которой приведен ниже:

```
ALTER DOMAIN name {
[SET DEFAULT { literal | NULL | USER}]
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK ( <dom_search_condition>)]
| [DROP CONSTRAINT]
| new_col_name
| TYPE data_type
};
```

Например, для изменения имени домена TEST1 на NEWTEST должен использоваться следующий SQL-запрос:

```
ALTER DOMAIN TEST1 TO NEWTEST;
```

Чтобы удалить у домена CHECKDOMAIN соответствующее ограничение ссылочной целостности, потребуется воспользоваться другой конструкцией:

```
ALTER DOMAIN CHECKDOMAIN DROP CONSTRAINT;
```

Работа с таблицами

Для создания таблицы применяется команда CREATE TABLE, синтаксис которой определен ниже:

```
CREATE TABLE Table [EXTERNAL [FILE] ' filespec']
(<col_def> [, <col_def> | <tconstraint>...]);
```

В параметре Table указывается имя создаваемой таблицы. В параметре EXTERNAL [FILE] указывается файл, содержащий внешнюю таблицу с данными. Параметр <col_def> предназначен для описания поля таблицы. Синтаксис описания поля приведен ниже:

```
<col_def> = col {datatype | COMPUTED [BY] (< expr>) | domain}
[DEFAULT { literal | NULL | USER}]
[NOT NULL] [ <col_constraint>]
[COLLATE collation]
```

Легко заметить, что описание поля практически не отличается от описания домена. Поле может описываться самостоятельно на базе домена или являться вычисляемым. Простая таблица базы данных задается небольшим SQL-запросом:

```
CREATE TABLE MYTABLE (
MyField1 CHAR(6),
MyField2 CHAR(50) CHARACTER SET DOS437 COLLATE PDOX_INTL,
MyField3 INTEGER);
```

Для определения вычисляемого поля следует использовать другой синтаксис:

```
<col_name> COMPUTED [BY] (<expr>);
```

В параметре (<expr>) могут быть указаны любое арифметическое выражение или строковая операция. В следующем примере показано, как задается вычисляемое поле:

```
CREATE TABLE EMPLOYEE
(FIRST_NAME VARCHAR(10) NOT NULL,
LAST_NAME VARCHAR(15) NOT NULL,
FULL_NAME COMPUTED BY (LAST_NAME || ', ' || FIRST_NAME));
```

Поле FULL_NAME является вычисляемым. В нем будет храниться результат объединения двух строк из предыдущих полей. Если тип возвращаемого выражением значения не указан, InterBase самостоятельно определяет его тип в процессе вычислений.

Сервер InterBase позволяет задавать ограничения, накладываемые на поля. Эти ограничения могут создать отношения ссылочной целостности между таблицами и контроль целостность данных. Ограничения могут быть наложены как на всю таблицу целиком, так и на отдельное ее поле. Они производят отслеживание всех операций с данными. Синтаксис ограничения, накладываемого на отдельный столбец, приведен ниже:

```
< col_constraint> = [CONSTRAINT constraint] <constraint_def>
```

```
[ <col_constraint>...]
<constraint_def> = {UNIQUE | PRIMARY KEY
| CHECK ( <search_condition>)
| REFERENCES other_table [( other_col [, other_col …])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
}
```

Ограничения PRIMARY KEY и UNIQUE указывают на то, что значение поля должны быть уникальными. Следовательно, поле не может содержать пустого значения. Пример использования этого ограничения в определении таблицы приведен ниже:

```
CREATE TABLE COUNTRY
(COUNTRY VARCHAR(10) NOT NULL PRIMARY KEY,
CURRENCY VARCHAR(10) NOT NULL);
```

Есть и другой способ создать таблицу с первичным ключом:

```
CREATE TABLE COUNTRY (
COUNTRY VARCHAR(10) NOT NULL,
CURRENCY VARCHAR(10) NOT NULL,
PRIMARY KEY (COUNTRY));
```

Поле COUNTRYNAME используется в качестве первичного ключа таблицы COUNTRY.

Ограничения могут налагаться на таблицу в целом. Подобные ограничения указываются в параметре <tconstraint>. Их синтаксис показан ниже:

```
<tconstraint> = [CONSTRAINT <constraint>] <tconstraint_def>
[< tconstraint>...]
<tconstraint_def> = {{PRIMARY KEY | UNIQUE} ( col [, col …])
| FOREIGN KEY ( col [, col …]) REFERENCES other_table
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK ( <search_condition>)}
```

В параметре <constraint> можно указать имя ограничения. Если имя не указано, сервер сгенерирует его автоматически и поместит описание в таблицу RDB$RELATION_CONSTRAINTS. Параметр FOREIGN KEY позволяет определить, какое поле будет использоваться в качестве внешнего ключа. Следующий SQL-запрос показывает, как налагается ограничение UNIQUE на всю таблицу в целом:

```
CREATE TABLE STOCK
(MODEL SMALLINT NOT NULL,
MODELNAME CHAR(10) NOT NULL,
ITEMID INTEGER NOT NULL,
CONSTRAINT MOD_UNIQUE UNIQUE (MODEL, MODELNAME, ITEMID));
```

Если необходимо установить отношения ссылочной целостности между двумя таблицами, стоит обратить внимание на следующий фрагмент кода:

```
CREATE TABLE TABLE1 (
FieldNum1 VARCHAR(20) NOT NULL PRIMARY KEY,
FieldNum2 SMALLINT);
CREATE TABLE TABLE2 (
FieldNum1 VARCHAR(20) REFERENCES TABLE1 (FieldNum1),
FieldNum2 SMALLINT);
```

При помощи этого запроса создаются две таблицы. Таблица TABLE1 имеет первичный ключ FieldNum1, с которым связывается поле FieldNum1 таблицы TABLE2.

Оператор REFERENCES позволяет определять тип действия, производимого при изменении первичного ключа:

- Значение NO ACTION указывает, что значения внешнего ключа не изменяются, что может привести к нарушению целостности данных и к соответствующей ошибке.
- Значение CASCADE указывает, что внешний ключ изменяется или удаляется соответственно действиям, производимым над первичным ключом.
- Значение SET DEFAULT указывает, что при удалении или обновлении первичного ключа значениям внешнего ключа присваивается значение по умолчанию. Если это значение по умолчанию не будет соответствовать какому-либо внешнему ключу, возникнет ошибка.
- Значение SET NULL указывает, что всем полям внешнего ключа при изменении первичного ключа будет присвоено значение NULL.

Если описанные выше механизмы не будут использоваться, то необходимо будет написать соответствующие триггеры, выполняющие аналогичные функции. Иначе возможны потери данных и другие неприятные ошибки.

Перед удалением первичного ключа необходимо удалить все связанные с ним записи внешнего ключа подчиненной таблицы. При использовании встроенных механизмов InterBase сам выполняет данную функцию. Продемонстрировать создание таких возможностей лучше всего на отдельном примере. Для этого можно добавить механизм контроля в таблицу TABLE2. Перед этим прежнюю таблицу TABLE2 необходимо удалить командой DROP TABLE:

```
DROP TABLE TABLE2
```

После этого таблицу можно создать заново:

```
CREATE TABLE TABLE2 (
FieldNum1 VARCHAR(20) REFERENCES TABLE1 (FieldNum1)
ON DELETE CASCADE
ON UPDATE CASCADE,
FieldNum2 SMALLINT);
```

Этот запрос можно изменить. Но даже в измененном варианте он создаст такую же таблицу:

```
CREATE TABLE TABLE2 (
FieldNum1 VARCHAR(20),
```

```
FieldNum2 SMALLINT,
FOREIGN KEY (FieldNum1) REFERENCES TABLE1 (FieldNum1)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Для того чтобы создать именованное ограничение ссылочной целостности при создании таблицы, можно использовать дополнительный код:

```
CREATE TABLE TABLE2 (
FieldNum1 VARCHAR(20),
FieldNum2 SMALLINT,
CONSTRAINT TEST_CONSTR FOREIGN KEY (FieldNum1) REFERENCES TABLE1 (FieldNum1)
ON UPDATE CASCADE
ON DELETE CASCADE);
```

Командой ALTER TABLE можно изменять структуру существующих таблиц. Она позволяет добавлять новые поля в таблицу и удалять существующие поля, а также ограничения, наложенные на поля или таблицы, изменять имена полей, их положение и тип данных.

Синтаксис команды ALTER TABLE приведен ниже:

```
ALTER TABLE table <operation> [, <operation> …];
<operation> = {ADD <col_def>
| ADD <tconstraint>
| ALTER [COLUMN] column_name <alt_col_clause>
| DROP col
| DROP CONSTRAINT constraint}
```

Синтаксис оператора <alt_col_clause> необходимо описать отдельно:

```
<alt_col_clause> = {TO new_col_name
| TYPE new_col_datatype
| POSITION new_col_position}
```

Добавить новое поле в таблицу можно при помощи простого SQL-запроса:

```
ALTER TABLE TableName ADD <col_def>
```

Для добавления в таблицу TABLE1 нового столбца NewField типа SMALLINT используется простой SQL-запрос:

```
ALTER TABLE TABLE1 ADD NewField SMALLINT NOT NULL;
```

В таблицу можно добавить сразу несколько столбцов, используя оператор ADD:

```
ALTER TABLE EMPLOYEE
ADD EMP_NO EMPNO NOT NULL,
ADD FULL_NAME COMPUTED BY (LAST_NAME || ' ' || FIRST_NAME);
```

Для удаления поля из таблицы следует воспользоваться командой DROP:

```
ALTER TABLE name DROP colname [, colname...];
```

Для удаления из таблицы TABLE1 поля NewField нужно использовать дополнительный запрос:

```
ALTER TABLE TABLE1 DROP NewField
```

Чтобы посмотреть, какие ограничения на данный момент наложены на таблицу TABLE2, следует воспользоваться следующим кодом:

```
SELECT RDB$Constraint_Name FROM RDB$RELATION_CONSTRAINTS
WHERE RDB$Relation_Name = 'TABLE2'
```

Для добавления в таблицу нового ограничения следует воспользоваться командой, имеющей следующий синтаксис:

```
ALTER TABLE name ADD [CONSTRAINT ConstraintName] <tconstraint_opt>;
```

Для добавления в таблицу TABLE2 ограничения ссылочной целостности следует воспользоваться следующим кодом:

```
ALTER TABLE TABLE2 ADD CONSTRAINT TEST_CONSTR2 FOREIGN KEY (FieldNum2)
REFERENCES TABLE1
```

А для удаления ограничения используется команда с несколько иным синтаксисом:

```
ALTER TABLE name
DROP CONSTRAINT ConstraintName;
```

В качестве примера можно продемонстрировать запрос, удаляющий ограничение ссылочной целостности:

```
ALTER TABLE TABLE2
DROP CONSTRAINT TEST_CONSTR2;
```

Теперь следует рассмотреть более серьезный пример. Сначала потребуется удалить из базы данных все созданные представления и ограничения ссылочной целостности, а затем и все таблицы. После этого можно приступать к созданию новых таблиц.

Таблица Compact_Disc будет содержать поля Nazvanie, NomerCD, TipCD и Proizvoditel. Таблица Opisanie, в свою очередь, включает в себя поля NomerCD, KolvoCD и Primechanie. Связь между таблицами будет осуществляться по полю NomerCD.

Первая таблица будет создана при помощи следующего SQL-запроса:

```
CREATE TABLE COMPACT_DISC(
Nazvanie VARCHAR(50) NOT NULL PRIMARY KEY,
NomerCD SMALLINT NOT NULL UNIQUE,
TipCD VARCHAR(10) DEFAULT 'Data CD',
Proizvoditel VARCHAR(10) DEFAULT 'BASF');
```

Для объявления второй таблицы используется другой запрос:

```
CREATE TABLE OPISANIE(
NomerCD SMALLINT,
KolvoCD SMALLINT DEFAULT 1,
Primechanie BLOB,
CONSTRAINT CD_CONSTR FOREIGN KEY (NomerCD) REFERENCES COMPACT_DISC (NomerCD)
ON UPDATE CASCADE
ON DELETE CASCADE
);
```

Работать с базой данных будет отдельное приложение. Для его создания нужно открыть новый проект и добавить к нему модуль данных. В модуле данных потребуется разместить компоненты TIBDataBase, TIBTransaction, два компонента TIBTable и два компонента TDataSource.

После этого потребуется настроить соединение с базой данных и установить отношения ссылочной целостности между наборами данных TIBTable, инкапсулирующими таблицы COMPACT_DISC и OPISANIE при помощи свойств MasterSource и MasterFields. Модуль данных потребуется присоединить к главной форме приложения, на которой потом необходимо разместить две таблицы и кнопку. На рис. 8.23 показано окно приложения.

Рис. 8.23. Связанные таблицы

В листинге 8.22 представлен код, вызываемый после нажатия на кнопку.

Листинг 8.22. Код метода-обработчика кнопки

```
procedure TForm1.PostBtnClick(Sender: TObject);
begin
  with ConnectionModule do begin
    if IBTable1.Modified then IBTable1.Post;
    if IBTable2.Modified then IBTable2.Post;
  end;
end;
```

В заключение следует отметить, что создание таблиц ручным способом — не самый удобный способ работы. Существует множество утилит, упрощающих этот процесс, например, IB Expert.

Использование внешних наборов данных

Сервер InterBase позволяет получать данные из внешних источников данных, имеющих табличную структуру. Это могут быть, например, текстовые файлы

особого формата. Но перед тем как использовать набор данных, его необходимо подключить к базе данных. Для подключения внешнего файла необходимо использовать оператор EXTERNAL. Использование команды EXTERNAL FILE позволяет импортировать данные из внешнего файла, имеющего фиксированный формат. Также это позволяет осуществлять выборку из внешнего набора данных и экспорт данных в него.

На используемые внешние наборы данных налагается несколько ограничений:

- Каждая запись внешнего набора данных должна иметь фиксированную длину.
- При создании таблицы, используемой для импорта внешних наборов данных, в ней должно быть определено поле для хранения символов конца строки или перевода на новую строку. Для систем Unix, как правило, требуется один байт, а в Windows-системах требуется два байта.
- Несмотря на то что чтение числовых данных из файла напрямую допустимо, рекомендуется читать их как символьные и приводить к числовым типам при помощи функции CAST().

При работе с внешними таблицами разрешены только операции выборки SELECT и вставки INSERT. Внешние наборы данных не обслуживаются транзакциями, так как располагаются вне базы данных. Если есть необходимость использовать транзакции, следует импортировать данные из внешней таблицы во внутреннюю.

Пример работы с внешним набором данных будет не так уж сложен. Сначала потребуется создать текстовый файл, структура которого описана в табл. 8.37. Первое поле имеет длину 10 символов, последующие — по 4 символа. Вместо знака звездочки следует использовать знак пробела.

Таблица 8.37. Тестовая таблица, представляющая собой внешний набор данных

MarkaKonfet (10)	Ves (4)	CenaZaKilo (4)
Полет*****	***1	**70
Арлекино**	***3	**80
Мятные****	***1	*50
Шоколадные	***2	*100

Рис. 8.24. Генерирование значений полей внешнего набора данных

Созданный текстовый файл нужно сохранить в каталоге Program Files\Borland\InterBase\ext под именем MyTable.txt. Чтобы не записывать данные в файл вручную, можно воспользоваться простой программой, окно которой показано на рис. 8.24, а код приведен в листинге 8.23.

Листинг 8.23. Формирование файла со значениями фиксированной длины

```
procedure TForm1.InsertBtnClick(Sender: TObject);
var S,S1 : string;
    I,J,C : Integer;
begin
  I:=Length(Edit1.Text);
  S:=Edit1.Text;
  if I<10 then begin
    J:=10-I;
    for C:=0 to J-1 do begin
      S:=S+' ';
    end;
  end;
  S1:=S;
  I:=Length(Edit2.Text);
  S:=Edit2.Text;
  if I<4 then begin
    J:=4-I;
    for C:=0 to J-1 do begin
      S:=S+' ';
    end;
  end;
  S1:=S1+S;
  I:=Length(Edit3.Text);
  S:=Edit3.Text;
  if I<4 then begin
    J:=4-I;
    for C:=0 to J-1 do begin
      S:=S+' ';
    end;
  end;
  S1:=S1+S;
  Memo1.Lines.Add(S1);
end;

procedure TForm1.ClearBtnClick(Sender: TObject);
begin
  Memo1.Lines.Clear;
```

```
end;

procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  Memo1.Lines.SaveToFile(EdtPath.Text);
end;
```

После создания набора данных его необходимо подключить как внешнюю таблицу. Для этого нужно выполнить следующий SQL-запрос:

```
CREATE TABLE MYT EXTERNAL FILE 'MyTable.txt' (
MarkaKonfet CHAR(10) NOT NULL,
Ves CHAR(4),
CenaZaKilo CHAR(4),
NEWLINE CHAR(2));
```

Следует обратить внимание на то, что в таблицу включены три поля, значения которых необходимо получить, и четвертое поле, содержащее символ-разделитель. В качестве разделителя будут использоваться символ окончания строки и символ перевода каретки на новую строку. Поэтому в поле будут храниться два символа. В качестве небольшого примера можно привести запрос, извлекающий все значения таблицы MYT и цену каждого вида конфет.

```
SELECT MarkaKonfet, Ves, CenaZaKilo, Ves*CenaZaKilo AS Cena FROM MYT
```

Результат выполнения запроса приведен в табл. 8.38.

Таблица 8.38. Результат выполнения запроса

MARKAKONFET	VES	CENAZAKILO	CENA
Полет	1	70	70
Арлекино	3	80	240
Мятные	1	50	50
Шоколадные	1	2	2

Если предполагается обрабатывать данные, удалять, изменять и добавлять новые записи, то необходимо перевести их из внешнего набора данных во внутренний. Для этого нужно будет создать новую таблицу, имеющую ту же структуру, что и внешняя. Кроме этого нужно добавить вычисляемое поле SALARY, а в команде COMPUTED BY произвести приведение типов к SMALLINT. Новая таблица задается соответствующим SQL-запросом:

```
CREATE TABLE MYTINT(
MarkaKonfet CHAR(10) NOT NULL,
Ves CHAR(4),
CenaZaKilo CHAR(4),
Salary COMPUTED BY (CAST(Ves AS SMALLINT)*CAST(CenaZaKilo AS SMALLINT)),
NEWLINE CHAR(2));
```

Следующий запрос переносит данные из внешней таблицы во внутреннюю:

```
INSERT INTO MYTINT (MARKAKONFET, VES, CENAZAKILO, NEWLINE)
SELECT MARKAKONFET, VES, CENAZAKILO, NEWLINE FROM MYT
```

Работа с индексами

Индексы предназначены для ускорения получения доступа к записям, отбираемым в соответствии с какими-либо условиями. При выполнении запроса сервер InterBase сначала проверяет, созданы ли для данной таблицы какие-либо индексы. Затем производится анализ, в ходе которого выясняется, что будет более эффективно: просканировать всю таблицу целиком или воспользоваться индексом. Если выгоднее использовать индекс, то сервер сначала производит поиск по его ключевым значениям, а затем, используя указатели на записи, возвращает записи в виде набора данных.

Поиск при использовании индекса осуществляется быстро, так как ключевые значения в индексе упорядочены, а сам индекс относительно невелик. Как только ключевое значение найдено, по указателю на запись определяется ее физическое положение в таблице.

Индекс может быть определен по одному столбцу, тогда его называют простым индексом, либо по нескольким столбцам, которые определяют составной индекс. Команда создания индекса имеет следующий синтаксис:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX IndexName ON table ( col [, col...]);
```

Сервер InterBase автоматически создает уникальный индекс при создании ограничений `PRIMARY KEY` и `UNIQUE` для столбца или группы столбцов. Значения ключевых полей, входящих в уникальный индекс, не могут дублироваться. Используя операторы `ASC[ENDING]` и `DESC[ENDING]`, можно определить порядок сортировки значений в индексе. Константа `IndexName` определяет имя индекса.

Перед созданием уникального индекса следует убедиться, что таблица не содержит повторяющихся сочетаний значений полей, входящих в индекс. Следующий запрос возвращает значения повторяющихся полей для таблицы `MYTINT`:

```
SELECT MARKAKONFET, VES FROM MYTINT
GROUP BY MARKAKONFET, VES
HAVING COUNT(*) > 1;
```

Для создания уникального индекса по полям `MarkaKonfet` и `Ves` для таблицы `MYTINT` с порядком сортировки по возрастанию используется другой запрос:

```
CREATE UNIQUE ASC INDEX MYINDEX ON MYTINT (MARKAKONFET, VES);
```

После некоторого числа изменений, внесенных в базу, индекс становится разбалансированным. Когда это происходит, производительность индекса можно повысить несколькими способами:

- перестроить индекс, используя команду `ALTER INDEX`;
- повторно вычислить селективность индекса, используя команду `SET STATISTICS`;
- пересоздать индекс, используя команды `DROP INDEX` и `CREATE INDEX`;
- произвести резервирование и восстановление базы данных.

Команда ALTER INDEX имеет следующий синтаксис:

```
ALTER INDEX name {ACTIVE | INACTIVE};
```

Параметр INACTIVE переводит индекс в неактивное состояние. При его активации параметром ACTIVE производится перестройка индекса. Для перестройки индекса MYINDEX применяется следующий запрос:

```
ALTER INDEX MYINDEX INACTIVE;
ALTER INDEX MYINDEX ACTIVE;
```

Селективность индекса является вычисляемой величной, которая рассчитывается на основе различных параметров. Селективность используется оптимизатором при составлении плана запроса. Для вычисления селективности индекса следует воспользоваться следующим запросом:

```
SET STATISTICS IndexName;
```

Для удаления индекса используется команда DROP INDEX. В качестве иллюстрации можно привести SQL-запрос, удаляющий индекс MYINDEX:

```
DROP INDEX MYINDEX
```

Работа с представлениями

Представления позволяют возвращать наборы данных, удовлетворяющие нужды конкретных пользователей или групп. После создания представления с ним можно обращаться точно так же, как и с обычной таблицей. Представление может строиться на базе одной или нескольких таблиц, или даже на основе других представлений.

Представления являются временными наборами данных, существующими только во время работы с ними. Важно понимать, что представление является не копией данных, а лишь ссылкой на реальные записи. То есть при изменении данных в представлении будут изменены данные и в физической таблице. И при изменении данных в физической таблице автоматически перестраиваются данные в представлении. Команда создания представления имеет следующий синтаксис:

```
CREATE VIEW name [( view_col [, view_col
AS <select> [WITH CHECK OPTION];
```

Параметр view_col содержит описание имен полей представления, возвращаемых набором данных. Параметр view_col должен содержать такое же количество полей, какое указано в выражении SELECT, либо не содержать их вообще. Простое представление, которое вернет пользователю всю таблицу, создается несложным SQL-запросом:

```
CREATE VIEW MY_VIEW AS
SELECT * FROM EMPLOYEE;
```

Также представление может возвращать поля из нескольких таблиц сразу, как показано ниже:

```
CREATE VIEW ENTRY_LEVEL_WORKERS AS
SELECT JOB_CODE, JOB_TITLE, FIRST_NAME, LAST_NAME
```

```
FROM JOB, EMPLOYEE
WHERE JOB.JOB_CODE = EMPLOYEE.JOB_CODE AND SALARY < 15000;
```

Представления могут быть необновляемыми и обновляемыми. Представление *может быть обновляемым, если выполняется ряд дополнительных условий:*

- Представление строится на базе единственной таблицы или на базе другого обновляемого представления.
- Все столбцы таблицы, на основе которой строится представление, не входящие в представление, должны принимать значение NULL по умолчанию.
- Выражение SELECT не должно содержать подзапросов, предиката DISTINCT, выражения HAVING, агрегатных функций, присоединенных таблиц, пользовательских функций или хранимых процедур.

При несоблюдении приведенных выше условий представление не может быть обновляемым.

Команда WITH CHECK OPTION определяет правила, на основе которых модифицируются данные при помощи представления. Представления, созданные с использованием этой команды, указывают серверу на необходимость проверки вставленных или обновленных записей на соответствие условию проверки, определенному выражением WHERE. Значения могут быть добавлены через представления только в те поля, которые определены в представлении. В остальные поля будет помещено значение NULL. В качестве примера можно привести запрос, создающий обновляемое представление на базе таблицы COMPACT_DISC по всем ее полям:

```
CREATE VIEW COMPACT_VIEW AS
SELECT * FROM COMPACT_DISC
WHERE PROIZVODITEL = 'BASF'
WITH CHECK OPTION
```

После этого к представлению можно добавить новую запись:

```
INSERT INTO COMPACT_VIEW
VALUES ('Музыка света','7','SOFT','BASF');
```

В качестве тренировки можно попробовать добавить в представление строку, в которой условие проверки не соответствует заданному.

Работа с хранимыми процедурами

Хранимая процедура представляет собой программу, хранящуюся на сервере в двоичном виде как часть базы данных. Создав хранимую процедуру, ее можно вызвать в любое время из приложения. Хранимая процедура может принимать входные параметры и возвращать значения и наборы данных.

Существует два типа хранимых процедур:

- Процедуры выборки в качестве результата своей работы возвращают набор данных либо сообщение об ошибке.

- Выполняемые процедуры осуществляют с базой данных какое-либо действие. Опционально могут возвращать некие значения в выходных параметрах.

Оба типа хранимых процедур создаются командой CREATE PROCEDURE. Различие между подобными процедурами заключается лишь в том, как они используются. Фактически, любая процедура может быть использована как процедура выборки либо как исполняемая процедура, но лучше использовать все по назначению, так как сложно предусмотреть возможные ошибки из-за нецелевого использования.

Хранимой процедуре можно назначить определенные привилегии, то есть определить группы пользователей, имеющих право ее выполнять. Команда создания хранимой процедуры имеет следующий синтаксис:

```
CREATE PROCEDURE name
[( param datatype [, param datatype …])]
[RETURNS ( param datatype [, param datatype …])]
AS
<procedure_body>;
< procedure_body> = [<variable_declaration_list>]
< block>
< variable_declaration_list> =
DECLARE VARIABLE var datatype;
[DECLARE VARIABLE var datatype; …]
<block> =
BEGIN
< compound_statement>
[< compound_statement> …]
END
< compound_statement> = {<block> | statement;}
```

В параметре name указывается имя хранимой процедуры. Оно должно быть уникальным в рамках базы данных. В параметре param datatype содержится список входных параметров с указанием их типа. Параметр RETURNS открывает секцию, в которой описываются параметры, возвращаемые хранимой процедурой. Параметр param datatype содержит список возвращаемых параметров с указанием их типа. Процедура возвращает значения параметров, когда достигает в своем теле выражения SUSPEND.

Ключевое слово AS служит разделителем шапки процедуры и ее тела. Команда DECLARE VARIABLE предназначена для объявления локальной переменной определенного типа, указываемого в параметре var datatype. Далее, в блоке, обозначенном ключевыми словами BEGIN и END, помещается тело хранимой процедуры. В табл. 8.39 приведено описание операторов, предназначенных для использования в триггерах и хранимых процедурах.

В качестве примера следует рассмотреть создание нескольких простых хранимых процедур. Сначала нужно разработать хранимую процедуру, которая

будет добавлять записи в таблицу COMPACT_DISC. Каких-либо значений процедура возвращать не будет.

```
CREATE PROCEDURE ADD_COMPACT_PROC(NAZV VARCHAR(50), NOMER SMALLINT)
AS
BEGIN
INSERT INTO COMPACT_DISC (NAZVANIE, NOMERCD)
VALUES (:NAZV, :NOMER);
END;
```

Таблица 8.39. Описание операторов расширения SQL

Выражение	Описание
BEGIN ... END	Определяет блок, в котором заключается выражение
variable = expression	Оператор присваивания переменной какого-либо значения
EXCEPTION exception_name	Вызывает исключение, имя которого содержится в параметре exception_name
EXECUTE PROCEDURE proc_name [var [, var ...]] [RETURNING_VALUES var [, var ...]]	Выполняет хранимую процедуру
EXIT	Выход из тела хранимой процедуры
FOR select_statement DO compound_statement	Оператор цикла. Выполняется до тех пор, пока выражение SELECT возвращает записи. В параметре compound_statement, в блоке BEGIN ... END заключается исполняемое в цикле выражение
IF (condition) THEN compound_statement [ELSE compound_statement]	Оператор ветвления. Условие проверки указывается в параметре condition, выполняемый код заключается в блок BEGIN-END после оператора THEN
POST_EVENT event_name	Возвращает сообщение, ранее зарегистрированное на сервере. Имя сообщения указывается в параметре event_name
SUSPEND	Оператор SUSPEND предназначен для использования в хранимых процедурах выборки. Оператор SUSPEND приостанавливает выполнение процедуры выбора, возвращает пакет данных и ожидает вызова метода FETCH. После того как метод выполнен, процедура продолжает свою работу и возвращает следующий пакет
WHILE (condition) DO compound_statement	Оператор цикла будет выполняться до тех пор, пока проверяемое условие будет соответствовать истине
WHEN {error [, error ...] | ANY} DO compound_statement	Оператор WHEN-DO позволяет организовать обработку возникающих ошибок. Оператор должен размещаться в конце тела хранимой процедуры, после всех других команд и операторов

Вызывать эту процедуру необходимо командой EXECUTE PROCEDURE. Соответствующий запрос выглядит очень просто:

```
EXECUTE PROCEDURE ADD_COMPACT_PROC ('NewCD',11);
```

В таблицу будет добавлена строка, содержащая два значения, переданных процедуре как параметры. Остальные значения будут добавлены по умолчанию.

Для изменения процедуры нужно сначала удалить ее старый вариант:

```
DROP PROCEDURE ADD_COMPACT_PROC;
```

В процедуру можно добавить параметр, возвращающий максимальное значение поля NOMERCD:

```
CREATE PROCEDURE ADD_COMPACT_PROC(NAZV VARCHAR(50), NOMER SMALLINT)
RETURNS (CTS SMALLINT)
AS
BEGIN
INSERT INTO COMPACT_DISC (NAZVANIE, NOMERCD)
VALUES (:NAZV, :NOMER);
SELECT MAX (NOMERCD) FROM COMPACT_DISC INTO CTS;
END;
```

В качестве теста можно добавить новую запись:

```
EXECUTE PROCEDURE ADD_COMPACT_PROC ('NewCD2',1);
```

В ответ на вызов этой процедуры сервер вернет максимальное значение поля. В хранимой процедуре была использована функция MAX, которая и осуществила выборку значения.

Теперь необходимо рассмотреть пример процедуры, возвращающей набор данных. Предположим, что необходимо получить названия дисков, производителем которых является компания BASF. Потребуется составить объявление новой процедуры:

```
CREATE PROCEDURE RETURN_COMPACT_NAZV (PROIZV VARCHAR(10))
RETURNS (NAZV VARCHAR(50))
AS
BEGIN
FOR SELECT NAZVANIE FROM COMPACT_DISC WHERE PROIZVODITEL = :PROIZV INTO :NAZV
DO
SUSPEND;
END
```

Так как хранимая процедура является процедурой выборки, то она возвращает набор данных. Для того чтобы вызвать ее, необходимо использовать выражение SELECT:

```
SELECT * FROM RETURN_COMPACT_NAZV ('BASF')
```

Во входном параметре вызова передается название интересующего производителя. Можно дополнить пример и разработать еще одну хранимую процедуру. Предположим, что необходимо пометить диски из таблицы COMPACT_DISC звездочками. Количество добавленных звездочек будет меняться в зависимости от их типа. В данном случае необходимо использовать оператор ветвления IF:

```
ALTER PROCEDURE MARKCD
RETURNS (NAZV VARCHAR(50), MARK VARCHAR (3))
AS
DECLARE VARIABLE TIP SMALLINT;
DECLARE VARIABLE TEMPTIP VARCHAR(10);
BEGIN
FOR SELECT TIPCD, NAZVANIE FROM COMPACT_DISC INTO :TEMPTIP, :NAZV DO BEGIN
IF (:TEMPTIP = 'Data CD') THEN MARK = '*';
IF (:TEMPTIP = 'Аудио CD') THEN MARK='*';
IF (:TEMPTIP = 'SOFT') THEN MARK='***';
SUSPEND;
END
END
```

Результаты нужно получить при помощи нового запроса:

```
SELECT * FROM MARKCD
```

Следует учитывать, что данные физически не изменяются, а выносятся во временную таблицу, размещенную в памяти. Хранимые процедуры, изменяющие данные, будут построены аналогичным образом. Для изменения данных в них будут использоваться запросы. Для изменения определения хранимой процедуры предназначена команда **ALTER PROCEDURE**, синтаксис которой приведен ниже:

```
ALTER PROCEDURE name
[( var datatype [, var datatype …])]
[RETURNS ( var datatype [, var datatype …])]
AS
procedure_body;
```

В процедуру **MARKCD** можно добавить дополнительный критерий, по которому будет производиться маркировка:

```
ALTER PROCEDURE MARKCD
RETURNS (NAZV VARCHAR(50), MARK VARCHAR (4))
AS
DECLARE VARIABLE TIP SMALLINT;
DECLARE VARIABLE TEMPTIP VARCHAR(10);
BEGIN
FOR SELECT TIPCD, NAZVANIE FROM COMPACT_DISC INTO :TEMPTIP, :NAZV DO BEGIN
IF (:TEMPTIP = 'Data CD') THEN MARK = '*';
IF (:TEMPTIP = 'Аудио CD') THEN MARK='*';
IF (:TEMPTIP = 'SOFT') THEN MARK='***';
IF (:TEMPTIP = 'Видео) THEN MARK='****';
SUSPEND;
END
END
```

Работа с триггерами

Триггер является функцией, выполняющейся при вставке, изменении или удалении записи. Триггеры могут определяться как для таблиц, так и для обновляемых представлений. Триггеры, как и хранимые процедуры, могут использовать механизм обработки исключительных ситуаций. Триггеры обладают некоторыми преимуществами:

- возможностью реализации автоматического поддержания логической целостности данных. Используя данный механизм, можно организовать проверку на допустимость вводимых данных и, в случае несоответствия их заданному критерию, выполнить те или иные действия;
- возможностью ведения истории действий;
- автоматическим оповещением об изменениях, вносимых в базу данных.

Триггер создается командой `CREATE TRIGGER`, имеющей следующий синтаксис:

```
CREATE TRIGGER name FOR { table | view}
[ACTIVE | INACTIVE]
{BEFORE | AFTER} {DELETE | INSERT | UPDATE}
[POSITION number]
AS < trigger_body>
< trigger_body> = [<variable_declaration_list>] < block>
< variable_declaration_list> =DECLARE VARIABLE variable datatype;
[DECLARE VARIABLE variable datatype; …]
< block> =
BEGIN
< compound_statement> [<compound_statement> …]
END
< compound_statement> = {<block> | statement;}
```

В параметре `name` указывается имя триггера. В параметре `table` указывается имя таблицы или представления, с которым будет связан триггер. Параметр `ACTIVE|INACTIVE` определяет активность триггера. Параметры `BEFORE` и `AFTER` определяют, до или после совершения события будет вызван триггер, а параметры `DELETE| INSERT|UPDATE` определяют событие, при наступлении которого триггер сработает.

Параметр `POSITION number` определяет порядок, в котором будут выполняться триггеры. Параметр используется для триггеров, которые имеют одни и те же условия запуска.

Триггеры могут использовать два контекста переменной. Старый контекст переменной связан с текущим или предыдущим значением записи, которая была обновлена или удалена. Новый контекст переменной связан с новым значением записи, которая была вставлена или значения полей которой были обновлены. Новый контекст переменной не может использоваться для удаленных записей. Как правило, контекст переменной используется для сравнения значений записи до и после обновления. Контекст переменной имеет следующий синтаксис:

```
NEW.column
OLD.column
```

Ниже представлен пример использования старого и нового контекстов для ведения истории изменения значений поля:

```
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
IF (old.salary <> new.salary) THEN
INSERT INTO SALARY_HISTORY (EMP_NO, CHANGE_DATE,
UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
VALUES (old.emp_no, 'now', USER, old.salary,
(new.salary - old.salary) * 100 / old.salary);
END !!
SET TERM ; !!
```

Предположим, что при обновлении данных о компакт-диске его номер необходимо увеличивать до максимально возможного значения. Триггер, реализующий данную функцию, будет выглядеть следующим образом:

```
CREATE TRIGGER MYTRIGGER FOR COMPACT_DISC
BEFORE UPDATE AS
DECLARE VARIABLE MAXCOUNT SMALLINT;
BEGIN
SELECT MAX(NOMERCD) FROM COMPACT_DISC INTO MAXCOUNT;
NEW.NOMERCD=MAXCOUNT+1;
END
```

Для тестирования триггера необходимо добавить новую запись в таблицу:

```
UPDATE COMPACT_DISC
SET TIPCD = 'NEW TYPE'
WHERE PROIZVODITEL = 'BASF'
```

В результате выполнения этого запроса всем компакт-дискам, произведенным BASF, будут последовательно увеличены идентификационные номера.

Чтобы изменить триггер, следует воспользоваться командой ALTER TRIGGER, которая имеет следующий синтаксис:

```
ALTER TRIGGER name
[ACTIVE | INACTIVE]
[{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
[POSITION number]
AS < trigger_body>;
```

В качестве примера можно изменить триггер так, чтобы вместо увеличения на единицу, производилось увеличение значения на две единицы:

```
ALTER TRIGGER MYTRIGGER
BEFORE UPDATE AS
```

```
DECLARE VARIABLE MAXCOUNT SMALLINT;
BEGIN
SELECT MAX(NOMERCD) FROM COMPACT_DISC INTO MAXCOUNT;
NEW.NOMERCD=MAXCOUNT+2;
END
```

Для удаления триггера используется команда DROP TRIGGER.

Работа с исключениями

Исключение является именованным сообщением об ошибке и может вызываться в хранимой процедуре или в триггере. Исключение создается командой CREATE EXCEPTION, изменяется командой ALTER EXCEPTION и удаляется командой DROP EXCEPTION. Хранимая процедура или триггер могут вызвать исключение по его имени. При вызове исключения оно возвращает сообщение об ошибке.

Для создания исключения, как уже было отмечено, необходимо использовать команду CREATE EXCEPTION, имеющую следующий синтаксис:

```
CREATE EXCEPTION name ' <message>';
```

В параметре name указывается имя исключения, а в параметре <message> — текст сообщения. В качестве примера можно создать исключение, которое будет использовано позже:

```
CREATE EXCEPTION COMPACT_EXEPT 'Unique key violation! Value in the Field
NomerCD not unique!';
```

Для изменения текста исключения используется команда ALTER EXCEPTION:

```
CREATE EXCEPTION COMPACT_EXEPT 'Unique key violation! Value in the Field
NomerCD not unique! ';
```

Для удаления созданного исключения нужно выполнить следующий запрос:

```
DROP EXCEPTION COMPACT_EXEPT;
```

Исключение можно вызвать в произвольном месте программы при помощи команды EXCEPTION name;, где в параметре name указывается имя исключения.

В триггерах и процедурах могут обрабатываться ошибки трех типов:

- исключения, вызванные оператором EXCEPTION в данной процедуре или в триггере;
- ошибки SQL, имеющие идентификатор SQLCODE;
- ошибки InterBase, возвращаемые в GDSCODE.

Обработка ошибок может производиться при помощи конструкции WHEN ... DO, синтаксис которой приведен ниже:

```
WHEN {< error> [, <error> ...] | ANY}
DO <compound_statement>
< error> =
{EXCEPTION exception_name | SQLCODE number | GDSCODE errcode}
```

Следует отметить, что это выражение должно помещаться в конце тела триггера или хранимой процедуры. Параметр ANY позволяет указывать на то, что будет производиться перехват любых возникающих ошибок.

При возникновении исключения процедура или триггер получают управление назад и могут попытаться исправить ситуацию каким-либо образом. При возникновении исключения выполняется определенная цепочка действий:

1. Производится поиск секции WHEN ... DO. Если она не существует, то прекращается выполнение действий в блоке BEGIN ... END и производится откат до следующего уровня. Поиск повторяется на следующем уровне. Если блок BEGIN ... END какого-либо уровня содержит блок WHEN ... DO, то управление передается этому блоку. В противном случае хранимая процедура завершает свою работу.
2. Последовательно выполняются инструкции блока WHEN ... DO.
3. Управление возвращается блоку процедуры или триггера, следующему за блоком WHEN ... DO.

Следует рассмотреть простой пример, реализующий описанную технологию. В хранимую процедуру DD_COMPACT_PROC нужно внести секцию WHEN ... DO, в которой будем отлавливать ошибку ввода в ключевое поле неуникального значения:

```
ALTER PROCEDURE ADD_COMPACT_PROC(NAZV VARCHAR(50), NOMER SMALLINT)
AS
BEGIN
INSERT INTO COMPACT_DISC (NAZVANIE, NOMERCD)
VALUES (:NAZV, :NOMER);
WHEN SQLCODE -803 DO
EXCEPTION COMPACT_EXEPT;
END;
```

Оператор WHEN будет обрабатывать ошибку SQL с кодом 803. Полный список ошибок SQL можно найти в документе InterBase 7 (6) Language Reference ▸ Error Codes and Messages ▸ SQLCODE error codes and messages. Теперь нужно добавить с помощью процедуры запись, заведомо дублирующую значение первичного ключа:

```
EXECUTE PROCEDURE ADD_COMPACT_PROC ('NewCD2',3);
```

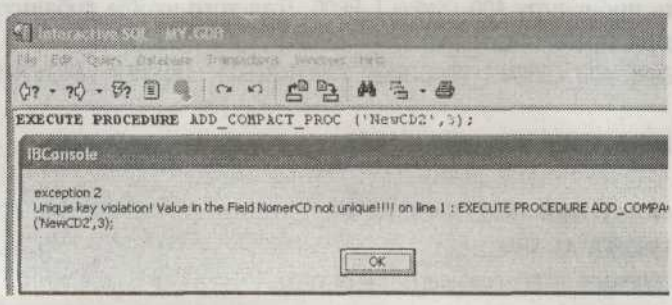


Рис. 8.25. Окно исключения COMPACT_EXEPT, вызванное при добавлении дубликата

В результате будет выведено окно с сообщением об ошибке, определенным в созданном исключении. Внешний вид окна приведен на рис. 8.25.

Точно так же осуществляется обработка исключений в триггерах.

Работа с генераторами

Генератор представляет собой механизм, создающий уникальную последовательность чисел и автоматически заполняющий заданное поле при вставке или обновлении записей. Генераторы, как правило, используются в хранимых процедурах для автоматического заполнения поля (полей), входящих в первичный ключ.

Если программисту необходимо разработать приложение, ведущее историю изменений в базе данных, потребуется гарантия, что ни один порядковый номер не будет повторяться. В этом случае разработчик может обойтись использованием генератора. При этом будет гарантирована уникальность значений номеров.

Генератор является глобальным объектом для всех прочих объектов базы данных. Любая транзакция может использовать генератор для получения уникального значения поля. Для создания генератора необходимо использовать команду `CREATE GENERATOR name;`.

Для того чтобы инициализировать генератор начальным значением, следует использовать команду `SET GENERATOR`, имеющую следующий синтаксис:

```
SET GENERATOR NAME TO int;
```

Следующая команда задает для генератора `MYGEN` стартовое значение 10:

```
SET GENERATOR MYGEN TO 10;
```

При сбросе значения генератора следует отслеживать ситуации, когда генерированное значение будет совпадать с уже существующим. В этом случае, если поле является первичным или уникальным ключом, будет вызвано исключение с сообщением о дублированных значениях.

После того как генератор создан и инициализирован, его можно использовать с помощью функции InterBase `GEN_ID()`. Так как генератор возвращает 64-битное значение, необходимо определить поле, в которое будут помещаться сгенерированные значения соответствующего типа. В качестве базы для примера можно взять процедуру `ADD_COMPACT_PROC`. Для того чтобы добавить генератор этой функции, придется ее переопределить:

```
ALTER PROCEDURE ADD_COMPACT_PROC(NAZV VARCHAR(50), INNOMER VARCHAR(5))
AS
DECLARE VARIABLE NOMER SMALLINT;
BEGIN
IF (INNOMER = '') THEN INNOMER=CAST((GEN_ID(MYGEN,1)) AS VARCHAR (5));
NOMER=CAST(INNOMER AS SMALLINT);
INSERT INTO COMPACT_DISC (NAZVANIE, NOMERCD)
VALUES (:NAZV, :NOMER);
```

```
WHEN SQLCODE -803 DO
EXCEPTION COMPACT_EXEPT;
END;
```

В процедуру добавлена проверка вводимых значений. В некоторых случаях может понадобиться ввести произвольный номер компакт-диска. Поэтому номер вводится как последовательность символов, а затем приводится функцией CAST к типу SMALLINT. Если номер не был указан, он будет сгенерирован автоматически. Примеры соответствующих запросов приведены ниже:

```
EXECUTE PROCEDURE ADD_COMPACT_PROC ('NewGEN','21');
EXECUTE PROCEDURE ADD_COMPACT_PROC ('NewGEN_1','');
```

Теперь следует рассмотреть пример использования генератора из приложения Delphi. После создания нового проекта необходимо добавить в него модуль данных. В модуле данных потребуется разместить компоненты TIBDataBase, TIBTransaction, TIBDataSet и TDataSource. Также поребуется настроить соединение с базой данных и установить кодовую страницу Character Set WIN1251, чтобы корректно работать с русскими символами. После этого можно задать запрос на выборку данных в компоненте TIBDataSet:

```
select * from COMPACT_DISC
```

Этот запрос возвращает все записи из таблицы COMPACT_DISC. Потребуется еще сгенерировать запросы на модификацию данных при помощи редактора DataSet Editor.

Теперь следует разобраться с полем-генератором. Сначала нужно запустить редактор GeneratorField, доступ к которому можно получить при помощи кнопки, расположенной в левой части свойства GeneratorField компонента TIBDataSet. В поле Generator нужно выбрать значение MYGEN, в поле Field — NOMERCD, а в поле IncrementBy задать единичное значение. Далее в группе флажков ApplyEvent нужно выбрать On Post. После установки свойств компонента осталось лишь активировать его.

На форме приложения нужно разместить таблицу и две кнопки. При нажатии на кнопки должны вызываться методы Post и Delete компонента TIBDataSet. Значения нужно выводить в таблицу. Поле NOMERCD будет заполняться автоматически. Окно приложения показано на рис. 8.26.

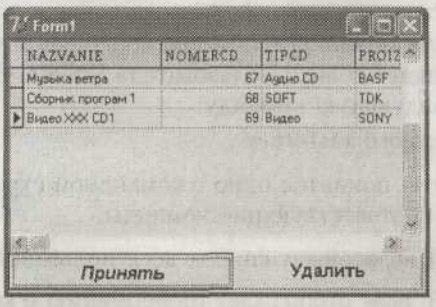


Рис. 8.26. Использование генератора приложением

Для удаления генератора из системной таблицы можно использовать команду DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = 'Имя генератора';.

Так, для удаления генератора MYGEN придется выполнить следующий код:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = 'MYGEN';
```

Администрирование сервера

Администрирование сервера InterBase включает в себя мероприятия по созданию резервных копий баз данных, восстановлению их в случае возникновения каких-либо аварийных ситуаций, чистке базы данных. Также в задачу администрирования входит определение ролей, включение в них пользователей, назначение прав им и другим объектам базы данных.

Создание ролей, учетных записей и определение прав на объекты

Список пользователей хранится в базе admin.ib. В состав базы входят таблицы HOST_INFO и USERS. Структура таблицы USERS приведена в табл. 8.40.

Таблица 8.40. Структура таблицы USERS

Поле	Обязательное для заполнения	Описание
User name	Да	Имя учетной записи, которое пользователи вводят в качестве логина
Password	Да	Пароль, присваиваемый учетной записи. Следует учитывать, что пароль чувствителен к регистру символов. Максимальная длина может составить 32 байта
GROUP_NAME	Нет	Содержит имя группы
UID	Нет	Уникальный идентификатор пользователя
GID	Нет	Уникальный идентификатор группы
PRIVILEGE	Нет	Содержит привилегии данного пользователя
Full name	Нет	Полное имя пользователя

Добавить пользователя можно при помощи консольной утилиты gsec, правом запуска которой обладает только пользователь SYSDBA. Для запуска утилиты нужно выполнить следующую команду:

```
gsec -user SYSDBA -password masterkey
```

После запуска утилиты появится окно с командной строкой, в которой можно будет выполнить соответствующие команды:

- Команда di[splay] возвращает список всех пользователей базы данных.
- Команда di[splay] name отображает информацию о конкретном пользователе.

- Команда a[dd] name -pw password добавляет пользователя в базу данных admin.ib. В этой команде могут использоваться дополнительные параметры, перечисленные в табл. 8.41.
- Команда mo[dify] name [options] позволяет изменить параметры заданного пользователя.
- Команда de[lete] name удаляет указанного пользователя.
- Команда h[elp] возвращает список команд с их описанием.
- Команда q[uit] завершает сессию и закрывает утилиту.

Таблица 8.41. Дополнительные параметры gsec

Параметр	Значение
-password или -pa string	Пользователь, выполняющий модификацию учетной записи
-user string	Имя пользователя, выполняющего модификацию
-pw string	Пароль пользователя
-uid integer	Уникальный идентификатор пользователя
-gid integer	Уникальный идентификатор группы
-fname string	Имя пользователя
-mname string	Фамилия пользователя
-lname string	Имя и фамилия

Команда добавления пользователя имеет следующий синтаксис:

```
a[dd] name -pw password [ options]
```

Если потребуется добавить пользователя Test с паролем mypass, то в командной строке нужно будет ввести соответствующую команду:

```
add Test -pw mypass
```

После этого при помощи команды display можно вывести список всех пользователей базы данных:

```
user name uid gid full name
------------------------------------------------------------------------
SYSDBA 0 0
TEST 0 0
```

Следующая команда добавляет еще одного пользователя с именем Alex и с паролем B2B4. В дополнение к этому будут указаны также имя и фамилия пользователя:

```
add Alex -pw B2B4 -fname Alexander -lname Sokolov
```

После этого потребуется просмотреть результаты изменений:

```
user name uid gid full name
------------------------------------------------------------------------
SYSDBA 0 0
TEST 0 0
ALEX 0 0 Alexander Sokolov
```

Теперь нужно изменить учетную запись пользователя Test, добавив в учетную запись имя, фамилию и идентификатор пользователя:

```
modify Test -uid 8 -fname Konstantin -mname Ivanovich
```

Останется лишь отобразить результаты выполнения команды:

```
user name uid gid full name
------------------------------------------------------------------------------
SYSDBA 0 0
TEST 8 0 Konstantin Ivanovich
ALEX 0 0 Alexander Sokolov
```

Удалить пользователя Test можно при помощи команды delete Test.

Также, учетную запись пользователя можно добавить при помощи утилиты IBConsole. После запуска утилиты нужно выбрать пункт меню Register ▸ User Security. В результате будет отображено диалоговое окно, показанное на рис. 8.27. После нажатия кнопки New поля ввода данных станут доступны для редактирования. В поле User Name нужно ввести имя учетной записи Vlad, в поле Password какой-либо пароль, а в полях First Name, Middle Name и Last Name нужно указать значения Vladislav, Konstantinovich и Kostin соответственно.

После того как пользователь будет добавлен, можно просмотреть общий список учетных записей. Он расположен в правой части окна и показан на рис. 8.28.

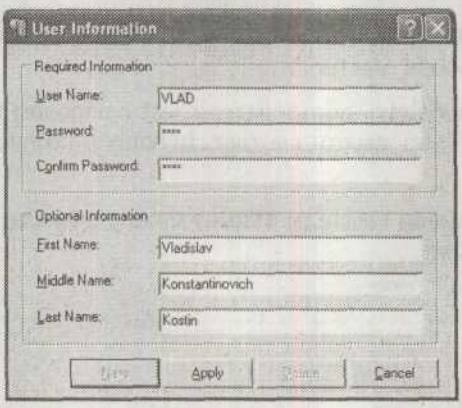


Рис. 8.27. Окно регистрации учетной записи утилиты IBConsole

Рис. 8.28. Список пользователей

Учетная запись пользователя сама по себе не дает ему никаких прав на работу с объектами базы данных. Права доступа устанавливаются командой GRANT. Пользователь, выдающий права командой GRANT, может присвоить другому пользователю только те права, которыми владеет он сам, или более слабый набор прав.

Для работы с таблицей или с представлением пользователь должен обладать правами на выполнение команд SELECT, INSERT, UPDATE, DELETE или REFERENCES. Для того чтобы задать привилегии на все эти команды сразу, можно использовать привилегию ALL. Для вызова хранимой процедуры в приложении пользователь или объект должны обладать правом на выполнение команды EXECUTE. Команда GRANT имеет следующий синтаксис:

```
GRANT < privileges> ON [TABLE] { tablename | viewname}
TO { <object> | <userlist> | GROUP UNIX_group}
| EXECUTE ON PROCEDURE procname TO { <object> | <userlist>}
| < role_granted> TO {PUBLIC | < role_grantee_list>};
< privileges> = {ALL [PRIVILEGES] | < privilege_list>}
<privilege_list> = SELECT
| DELETE
| INSERT
| UPDATE [( col [, col …])]
| REFERENCES [( col [, col …])]
[, < privilege_list> …]
<object> = PROCEDURE procname
| TRIGGER trigname
| VIEW viewname
| PUBLIC
[, <object> …]
<userlist> = [USER] username
| rolename
| Unix_user}
[, <userlist> …]
[WITH GRANT OPTION]
< role_granted> = rolename [, rolename …]
<role_grantee_list> = [USER] username [, [USER] username …]
[WITH ADMIN OPTION]
```

В параметре privilege_list указывается список привилегий, которые указаны в табл. 8.42. В параметре col указывается имя поля, которому присваиваются привилегии. Параметр tablename содержит имя таблицы, а параметр viewname — имя представления. Параметр Userlist содержит список имен пользователей, которые получают указанные права доступа. Параметр WITH GRANT OPTION передает право делегирования прав пользователям, перечисленным в списке Userlist. В параметре rolename указывается имя существующей роли, созданной командой CREATE ROLE. Параметр role_grantee_list содержит список пользователей, которым передаются права роли.

Таблица 8.42. Перечень привилегий

Привилегия	Права пользователей
ALL	Право выполнять над объектом операции SELECT, DELETE, INSERT, UPDATE, и REFERENCES
SELECT	Право производить выборку из таблицы или представления
DELETE	Право удалять записи из таблицы или представления
INSERT	Право добавлять в таблицу или в представление записи
UPDATE	Право обновлять записи в таблице или представлении. Может быть ограничено только определенным набором столбцов
EXECUTE	Право выполнять хранимую процедуру
REFERENCES	Связывает определенные поля с первичным ключом. При работе с первичным ключом должны быть связаны все поля, входящие в него

Следующая команда передает пользователю `Vlad` привилегию на просмотр таблицы `COMPACT_DISC` с правом присваивания привилегий другим пользователям:

```
GRANT SELECT ON COMPACT_DISC TO Vlad WITH GRANT OPTION
```

Для того чтобы войти под этой учетной записью из утилиты IBConsole, нужно выбрать пункт меню **Register ▸ Connect As**. В появившемся диалоговом окне потребуется указать имя пользователя и пароль. После этого можно выполнить SQL-запросы, используя для этого утилиту Interactive SQL:

```
SELECT * FROM COMPACT_DISC
INSERT INTO SELECT COMPACT_DISC VALUES ('111',111,'111','111')
```

При попытке добавить запись будет выведено сообщение об ошибке. То есть пользователь не сможет вставить запись из-за того, что у него нет соответствующих прав. Точно так же можно определить пользователю права только на вставку записей, но не на их просмотр:

```
GRANT INSERT ON COMPACT_DISC TO Vlad WITH GRANT OPTION
```

Перед тем как присвоить пользователю новые права, необходимо забрать у него старые. Это позволяет сделать команда `REVOKE`. Команда `REVOKE` имеет следующий синтаксис:

```
REVOKE [GRANT OPTION FOR] < privileges> ON [TABLE]
{ tablename | viewname}
FROM { <object> | <userlist> | < rolelist> | GROUP UNIX_group}
| EXECUTE ON PROCEDURE procname
FROM { <object> | <userlist>}
| < role_granted> FROM {PUBLIC | < role_grantee_list>}};
< privileges> = {ALL [PRIVILEGES] | < privilege_list>}
<privilege_list> = {
SELECT
| DELETE
| INSERT
```

```
| UPDATE [( col [, col …])]
| REFERENCES [( col [, col …])]
[, < privilege_list> …]}
<object> ={
PROCEDURE procname
| TRIGGER trigname
| VIEW viewname
| PUBLIC
[, <object>]}
<userlist> = [USER] username [, [USER] username …]
< rolelist> = rolename [, rolename]
< role_granted> = rolename [, rolename …]
<role_grantee_list> = [USER] username [, [USER] username …]
```

Следующая команда отбирает все права работы с таблицей COMPACT_DISC у пользователя Vlad:

```
REVOKE ALL ON COMPACT_DISC FROM VLAD
```

Как уже было отмечено, в качестве пользователей базы данных могут выступать другие объекты базы данных. Предположим, что необходимо предоставить привилегию вызова хранимой процедуры ADD_COMPACT_PROC пользователю Vlad:

```
GRANT EXECUTE ON PROCEDURE ADD_COMPACT_PROC TO Vlad
```

Но перед тем как выполнить процедуру ADD_COMPACT_PROC, осуществляющую вставку записей в таблицу COMPACT_DISC, необходимо ей присвоить соответствующие права. Поэтому сначала нужно отменить все права на работу с таблицей пользователю Vlad. А потом можно присвоить необходимые привилегии хранимой процедуре:

```
GRANT INSERT ON COMPACT_DISC TO PROCEDURE ADD_COMPACT_PROC
```

Теперь нужно выполнить хранимую процедуру под учетной записью Vlad:

```
EXECUTE PROCEDURE ADD_COMPACT_PROC ('TestRec','');
```

Можно зайти под учетной записью SYSDBA и убедиться в том, что записи были внесены в таблицу. В данном случае пользователь Vlad не имел прямых прав на работу с таблицей. Но у него было право выполнить хранимую процедуру, которая, в свою очередь, имела право на модификацию таблицы. При делегировании прав следует очень внимательно отслеживать возможные последствия. Для того чтобы отобрать у хранимой процедуры ADD_COMPACT_PROC права на работу с таблицей, следует выполнить следующую команду:

```
REVOKE INSERT ON COMPACT_DISC FROM PROCEDURE ADD_COMPACT_PROC
```

Чтобы задать права всем пользователям сразу, можно использовать ключевое слово PUBLIC. Следует отметить, что права, делегированные как PUBLIC, не распространяются на объекты базы данных, выступающие в качестве пользователей:

```
GRANT SELECT, INSERT, UPDATE ON DEPARTMENTS TO PUBLIC;
```

Как правило, в работе с учетными записями используют более крупные объекты — роли. Роль представляет собой объект базы данных, обладающий неко-

торыми правами и содержащий в себе учетные записи пользователей. Работать с ролью гораздо удобнее, чем с отдельными пользователями. Намного проще переопределить права одной роли, чем каждой учетной записи в отдельности.

Для создания роли используется команда CREATE ROLE rolename;. Права роли определяются командой GRANT, синтаксис которой приведен ниже:

```
GRANT < privileges> ON [TABLE] {tablename | viewname}
TO rolename;
< privileges> = {ALL [PRIVILEGES] | < privilege_list>}
< privilege_list> = {
SELECT
| DELETE
| INSERT
| UPDATE [( col [, col …])]
| REFERENCES [( col [, col …])]
[, < privilege_list> …]}}
```

Следующая команда присваивает роли права на вставку новых записей и просмотр таблицы COMPACT_DISC:

```
GRANT SELECT, INSERT ON COMPACT_DISC TO TESTROLE;
```

Чтобы присвоить права роли конкретной учетной записи, следует также использовать команду GRANT:

```
GRANT { rolename [, rolename …]} TO {PUBLIC
| {[USER] username [, [USER] username …]} }[WITH ADMIN OPTION];
```

Следующая команда добавляет в роль TESTROLE пользователей Vlad и Alex:

```
GRANT TESTROLE TO VLAD, ALEX
```

Одна учетная запись может состоять сразу в нескольких ролях. Но во время одной сессии клиент может работать только под одной ролью. Это может быть удобно в тех случаях, когда один пользователь должен иметь разные права доступа в зависимости от сложившейся ситуации. Еще следует учитывать тот факт, что пользователь, работающий под правами какой-либо роли, наследует собственные права. Например, роль TESTROLE имеет доступ только на чтение и занесение данных, а учетная запись VLAD имеет право на модификацию данных. Таким образом, пользователь VLAD, работая под ролью TESTROLE, будет иметь возможность просматривать, добавлять и изменять записи.

Сборка «мусора»

В ходе своей работы транзакции регистрируются в хранилище *TIP* (Transaction Inventory Page). Каждая стартующая транзакция производит поиск среди зарегистрированных транзакций на предмет занятости того или иного ресурса. Если найдена транзакция, работающая с данным ресурсом, производится определение ее активности. Если транзакция активна, производится отслежи-

вание состояния версии записи. На основании этих данных транзакция берет в работу либо последнюю версию записи, либо исходную.

База данных начинает отслеживать на странице учета транзакций десятки тысяч записей. Соответственно, новая транзакция должна каждый раз обрабатывать огромный объем информации, который постоянно увеличивается. Помимо записей о состоянии транзакции сохраняется версия данных, с которой работала данная транзакция. А значит, увеличивается объем места, занимаемого базой данных.

Процесс удаления старых, неактуальных транзакций и соответствующих им версий записей называется сборкой «мусора». Сборка «мусора» может производиться двумя способами. В автоматическом режиме стартовавшая транзакция удаляет неактуальные транзакции и освобождает занятые ресурсы. В другом режиме база данных самостоятельно периодически производит чистку «мусора». По умолчанию этот период составляет 20 000 транзакций. Для определения интервала между чистками базы данных можно воспользоваться утилитой gfix. Следующая команда задает интервал между чистками в 22 000 транзакций:

```
D:\MyIntDB\gfix.exe -h 22000 D:\MyIntDB\MY.GDB -user SYSDBA
-password masterkey
```

Если количество транзакций установить в нулевое значение, то автоматическая чистка будет отменена. Интервал между чистками можно установить при помощи утилиты IBConsole. Нужно перейти в окно свойств базы данных, выполнив пункт меню Database ▸ Properties. На вкладке General в поле Sweep Interval можно указать необходимое значение. Для того чтобы произвести немедленную чистку базы данных, нужно выполнить команду меню Database ▸ Maintenance ▸ Sweep. В появившемся диалоговом окне потребуется нажать кнопку Yes. После завершения операции будет выведено соответствующее сообщение.

Принудительную чистку базы данных можно выполнить при помощи соответствующей команды gfix:

```
D:\MyIntDB\gfix.exe -Sweep D:\MyIntDB\MY.GDB -user SYSDBA
-password masterkey
```

Работа с механизмом Shadowing

Сервер InterBase предоставляет механизм восстановления баз данных в случае физического повреждения диска, поломки сети или случайного удаления файлов. Метод восстановления называется *Disk Shadowing*. Первым шагом при работе с этим механизмом является создание Shadow-файла, представляющего собой копию базы данных. Как только с базой данных связывается такой файл, все изменения, вносимые в базу данных, немедленно отображаются в нем. Таким образом, файл содержит полную копию базы данных. Использование Shadow-файлов имеет ряд преимуществ:

- Быстрое восстановление базы данных. Активация Shadow-файла автоматически устанавливает его на место файла базы данных.

- Shadow-файлы занимают на диске такой же объем, как и база данных.
- Shadow-файл может представлять собой набор файлов фиксированного размера. Файлы могут размещаться на разных носителях.
- Механизм дублирования во время своей работы не прерывает процесс функционирования базы данных.

Впрочем, использование механизма Shadowing также имеет и некоторые недостатки:

- Механизм Shadowing не является реализацией репликации.
- Shadowing используется только при восстановлении базы данных после физического повреждения носителя или при случайном его удалении. Все ошибки, которые могут привести к логическому крушению базы данных, также сохраняются в Shadow-файле.
- Механизм Shadowing может работать только с локальными дисками. Отображаемые диски и ленточные накопители не поддерживаются.

Для создания Shadow-файла используется команда **CREATE SHADOW**, синтаксис которой приведен ниже:

```
CREATE SHADOW set_num [AUTO | MANUAL] [CONDITIONAL]
' filespec' [LENGTH [=] int [PAGE[S]]]
[ <secondary_file>];
<secondary_file> = FILE ' filespec' [ <fileinfo>] [ <secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int
[ <fileinfo>]
```

Параметр **set_num** содержит положительное уникальное число, идентифицирующее данный Shadow-файл. В параметре **filespec** указывается имя файла или группы файлов с перечислением их параметров.

Нужно рассмотреть пример создания одиночного Shadow-файла. Для этого нужно запустить утилиту **isql** и соединиться с базой **My.GDB**:

```
CONNECT D:\MyIntDB\My.GDB user SYSDBA password masterkey;
```

Затем нужно создать Shadow-файл:

```
CREATE SHADOW 1 'D:\MyShadow.shd';
```

Результат можно посмотреть при помощи команды **SHOW DATABASE**:

```
SHOW DATABASE;
Database: D:\MyIntDB\My.GDB
Owner: SYSDBA
Shadow 1: «D:\MYSHADOW.SHD» auto
PAGE_SIZE 8192
Number of DB pages allocated = 193
Sweep interval = 22000
Default Character set: WIN1251
```

Если база данных довольно велика, имеет смысл разместить Shadow-файлы, входящие в единый набор, на разных дисках. Для создания множественного

Shadow-файла следует перечислить имя, размер и стартовую позицию каждого файла в наборе. Следующая команда добавляет еще один подобный файл:

```
CREATE SHADOW 3 'D:\MyShad.shd'
FILE 'D:\MyShad1.shd' LENGTH 2000 STARTING AT 10000
FILE 'D:\MyShad2.shd' STARTING AT 12000;
```

При помощи команды **SHOW DATABASE** можно просмотреть текущую информацию:

```
Database: D:\MyIntDB\my.gdb
Owner: SYSDBA
Shadow 1: «D:\MYSHADOW.SHD» auto
Shadow 3: «D:\MYSHAD.SHD» auto
file D:\MYSHAD1.SHD length 2000 starting 10000
file D:\MYSHAD2.SHD starting 12000
PAGE_SIZE 8192
Number of DB pages allocated = 193
Sweep interval = 22000
Default Character set: WIN1251
```

По некоторым причинам Shadow-файл может становиться недоступным для базы данных. Если Shadow-файл недоступен и был создан в режиме `Auto`, операции с базой данных продолжают выполняются автоматически, с отключенным механизмом *Shadowing*. Если Shadow-файл недоступен и был создан в режиме `Manual`, доступ к базе данных запрещается до момента исправления ситуации администратором. В табл. 8.43 представлено сравнение режимов функционирования Shadow-файла.

Таблица 8.43. Сравнение режимов функционирования файла Shadow

Режим	Преимущество	Недостаток
Auto	Операции с базой данных не прекращаются	Создается временный период, в течение которого данные не заносятся в файл *Shadow*. Администратор может не знать о том, что база функционирует без файла *Shadow*
Manual	Предотвращает операции с базой данных без дублирования их в файле *Shadow*	Операции с базой данных приостанавливаются до того момента, пока вопрос не будет решен. Требуется вмешательство администратора

Режим `Manual` используется в тех случаях, когда продолжение функционирования механизма Shadowing является более важным, чем функционирование базы данных. Следующая команда создает Shadow-файл, функционирующий в режиме `Manual`.

```
CREATE SHADOW 2 MANUAL 'D:\ManShad.shd';
```

В этом режиме блокирование Shadow-файла останавливает работу с базой данных. В этом случае следует использовать команду `gfix -kill database`. Это команда удаляет из метаданных ссылки на недоступный Shadow-файл, свя-

занный с базой данных. Например, для базы данных MyIntDB команда будет выглядеть следующим образом:

```
gfix.exe -kill D:\MyIntDB\my.gdb -user SYSDBA -password masterkey
```

Эту аварию можно воспроизвести искусственно. Сначала необходимо отключить базу данных:

```
gfix.exe -shut -force 5 D:\MyIntDB\My.gdb -user SYSDBA -password masterkey
```

Данная команда закроет базу данных через пять секунд. После этого можно удалить файл D:\ManShad.shd. Если после этого обратиться к базе данных из утилиты IBConsole, будет выведено сообщение об ошибке, показанное на рис. 8.29.

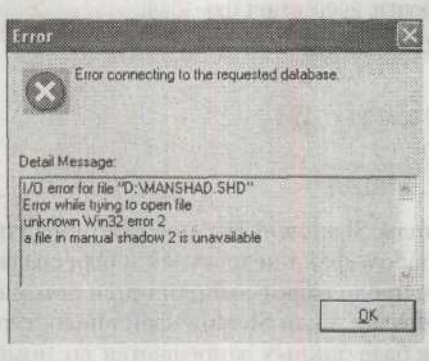


Рис. 8.29. Сообщение о недоступности второго Shadow-файла

Если попробовать соединиться с базой данных из утилиты isql, на экране будет отображено текстовое сообщение:

```
SQL> CONNECT D:\MyIntDB\My.GDB user SYSDBA
Statement failed, SQLCODE = -902
I/O error for file «D:\MANSHAD.SHD»
-Error while trying to open file
-unknown Win32 error 2
-a file in manual shadow 2 is unavailable
```

Для того чтобы исправить ошибку, следует выполнить новую команду:

```
gfix.exe -shut -force 5 D:\MyIntDB\My.gdb -user SYSDBA -password masterkey
```

Когда база данных становится недоступной, операции с ней прекращаются до момента активации Shadow-файла. Для того чтобы произвести активацию, необходимо запустить утилиту gfix с опцией -activate. Для активации набора необходимо указать путь к первому файлу, входящему в него. Для файла MYSHADOW.SHD команда активации будет выглядеть следующим образом:

```
gfix.exe» -activate D:\ MYSHADOW.SHD -user SYSDBA -password masterkey
```

После того как файл будет активирован, необходимо присвоить ему исходное имя базы данных. После этого при необходимости следует создать новый Shadow-файл.

Резервное копирование базы данных

Резервное копирование сохраняет базу данных на жестком диске или другом носителе информации. Для обеспечения максимальной надежности хранения данных следует регулярно проводить резервное копирование на внешние носители. Резервное копирование базы данных можно выполнить при помощи утилит IBConsole и gbak. Сначала следует рассмотреть выполнение копирования при помощи утилиты IBConsole. Для этого надо выполнить команду меню Database ▸ Maintenance ▸ Backup/Restore ▸ Backup... утилиты IBConsole, предварительно выбрав сервер Local Server. В результате будет отображено диалоговое окно, показанное на рис. 8.30.

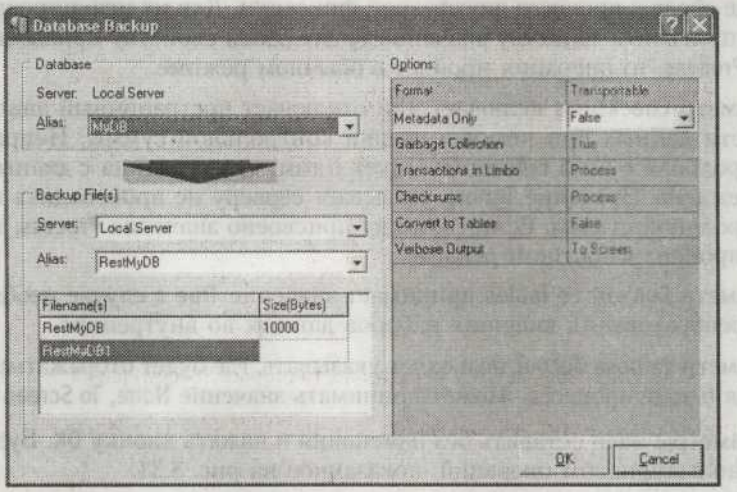


Рис. 8.30. Установка параметров резервного копирования

В поле Database из списка Alias нужно выбрать псевдоним базы данных MyDB. Затем из списка Server выбирается сервер, на котором находится база данных. В этом списке представлены только зарегистрированные серверы. В данном случе нужно выбрать значение Local Server. В списке Alias указывается псевдоним, обозначающий файл, который будет хранить резервную копию базы данных. Для этого примера использовано значение RestMyDB.

В списке Filename(s) следует указать имя файла или файлов с указанием занимаемого ими размера в байтах. Если файл является единственным, то размер указывать не следует, так как часть данных может просто не поместиться в него. Если файлов несколько, то размер не указывается для последнего из них. В списке указаны параметры файла RestMyDB размером 10 000 байт и файла RestMyDB1.

В правой части окна располагается поле Options, в котором можно произвести настройки свойств процесса резервного копирования:

- Параметр Format позволяет сохранять копию базы данных в переносимом формате. Если свойство принимает значение Transportable, база будет ре-

зервироваться в переносимом формате. Если свойство принимает значение Non-transportable — в нетранспортируемом.

- Параметр Metadata Only принимает значение True в том случае, если есть необходимость включить в резервную копию только метаданные, не записывая сами данные. Примером применения данного свойства может послужить необходимость создания пустой копии базы данных.
- Параметр Garbage Collection задает необходимость сборки «мусора» во время резервного копирования. По умолчанию свойство имеет значение True и сборка «мусора» производится.
- Параметр Transactions in Limbo регламентирует порядок обработки тразакций со статусом «Limbo». Как правило, подобные транзакции появляются в ходе сбоев в процессе двухфазной фиксации. Для их игнорирования следует присвоить свойству значение Ignore. Если свойству присвоено значение Process, то операция пройдет в обычном режиме.
- Параметр Checksums включает или отключает постраничный анализ целостности данных при помощи сверки контрольной суммы. Неправильная контрольная сумма свидетельствует о том, что страница с данными была повреждена. Значение Ignore указывает серверу не производить проверку контрольной суммы. Если свойству присвоено значение Process, то операция пройдет в обычном режиме.
- Параметр Convert to Tables принимает значение True в случае необходимости преобразования внешних наборов данных во внутренние.
- Параметр Verbose Output позволяет указывать, где будет отбражаться информация о ходе процесса. Может принимать значение None, To Screen и To File.

Все параметры надо оставить без изменения и нажать кнопку OK. Будет выведено окно с журналом операций, показанное на рис. 8.31.

После завершения процесса в каталоге сервера Bin появятся файлы RestMyDB и RestMyDB1.

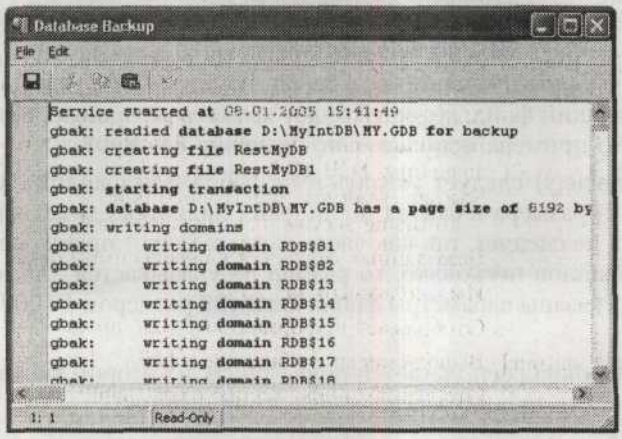


Рис. 8.31. Журнал операций процесса резервного копирования базы данных

Теперь нужно рассмотреть процесс резервного копирования, производимый с помощью утилиты gbak. Команда создания резервной копии имеет следующий синтаксис:

```
gbak [-B] [options] database target
```

В параметре [options] указывается список, перечисленных в табл. 8.44. Параметр target позволяет указать имя файла резервной копии. Если производится резервное копирование в несколько файлов, команда будет иметь следующий синтаксис:

```
gbak [-B] [options] database target1 size1[k|m|g] target2
[ size2[k|m|g] target3
```

В параметре Size указывается размер файла резервной копии.

Таблица 8.44. Список параметров утилиты gbak

Параметр	Описание
-b[ackup_database]	Резервное копирование базы данных в файл
-co[nvert]	Преобразование внешних наборов данных во внутренние
-e[xpand]	Не производится сжатие во время резервирования
-fa[ctor] n	Используется блокировочное число для лентопротяжных механизмов
-g[arbage_collect]	Не производится сборка «мусора» во время резервного копирования
-ig[nore]	Проверка контрольной суммы не производится
-l[imbo]	Транзакции Limbo в процессе резервного копирования игнорируются
-m[etadata]	В резервную копию включаются только метаданные
-nt	Резервная копия создается в не переносимом формате
-ol[d_descriptions]	Производит резервирование метаданных в старом формате
-pa[ssword] text	Пароль на доступ к базе данных
-role name	Имя роли, в которую входит данный пользователь
-se[rvice] servicename	Создает резервную копию базы данных на той машине, на которой расположена база данных, используя Services Manager. Параметр *servicename* вызывает Services Manager на машине с базой данных и может принимать следующие значения: TCP/IP hostname:service_mgr; SPX hostname@service_mgr; Named pipes \\hostname\service_mgr; Local service_mgr;
-t[ransportable]	База данных создается в переносимом формате
-u[ser] name	Имя пользователя
-v[erbose]	Отображает информацию о ходе процесса
-y [file | suppress_output]	Вывод списка сообщений в файл
-z	Отображает информацию о базе данных и движке InterBase

Следующая команда создает резервную копию базы данных My.gdb:

```
gbak.exe -b -user SYSDBA -password masterkey D:\MyIntDb\My.gdb D:\MyDB.gbk
```

Если, например, необходимо создать резервную копию базы данных и поместить ее на клиентской стороне, то команда будет выглядеть иначе:

```
gbak.exe -b -user SYSDBA -password masterkey 127.0.0.1:D:\MyIntDb\My.gdb
D:\MyDB.gbk
```

Адрес 127.0.0.1 является IP-адресом удаленного сервера. Если необходимо произвести резервное копирование базы данных с клиентской машины в файл резервной копии, расположенный на той же машине, команда будет иметь следующий вид:

```
gbak.exe -b -user SYSDBA -password masterkey -service 127.0.0.1:service_mgr
D:\MyIntDb\My.gdb D:\My1.gbk
```

Восстановление базы данных

Восстановление базы данных из резервной копии можно выполнить при помощи утилит IBConsole и gbak. Как правило, резервную копию рекомендуется располагать на внешних носителях информации.

Чтобы восстановить базу данных при помощи утилиты IBConsole, нужно выполнить команду меню Database ▸ Maintenance ▸ Backup/Restore ▸ Restore.... В результате будет отображено окно, внешний вид которого показан на рис. 8.32.

В группе органов управления Backup File(s) расположен список Alias, из которого можно выбрать псевдоним ранее созданной резервной копии базы данных. Для того чтобы получить доступ к окну диалога выбора файла, следует выбрать из списка Alias значение File.... В этом списке необходимо указать созданный ранее псевдоним RestMyDB. В правой части окна расположен список параметров процесса восстановления базы данных:

- Параметр Page Size позволяет выбрать размер страницы.
- Параметр Overwrite разрешает или запрещает перезаписывать существующий файл базы данных. Если свойству присвоено значение True, то файл будет перезаписан.

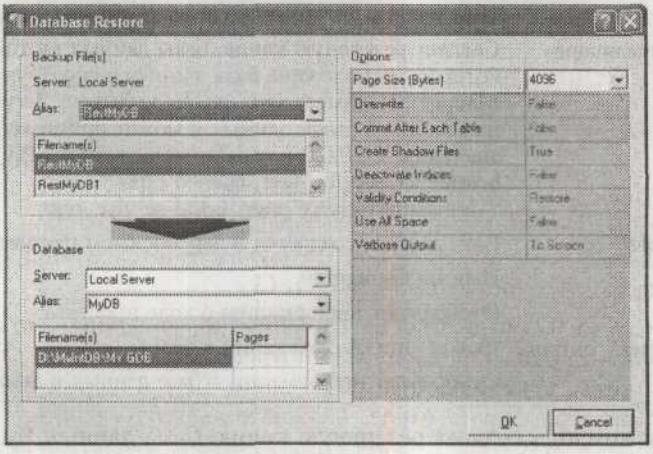


Рис. 8.32. Установка параметров восстановления базы данных

- Параметр Commit After Each Table позволяет начать процесс восстановления данных после восстановления метаданных. Для этого параметр должен получить значение True. Этот параметр используется в тех случаях, когда резервная копия повреждена. Это позволяет восстановить часть метаданных, до того момента, когда встретится таблица со сбойными данными. В этом момент восстановление базы данных будет прервано.
- Параметр Create Shadow Files указывает, что будет создан Shadow-файл.
- Параметр Deactivate Indexes позволяет деактивировать индексы во время процесса восстановления данных.
- Параметр Validity Conditions позволяет отключить на время восстановления проверку условий, наложенных на данные. Чтобы восстановить базу данных с неправильными записями, следует присвоить свойству значение Ingnore. По умолчанию свойство принимает значение Restore.
- Параметр Use All Space используется для восстановления базы данных с абсолютным заполнением страниц данных. По умолчанию свойство имеет значение False, и коэффициент заполнения составляет 80%.
- Параметр Verbose Output позволяет указывать, где будет отображаться информация о ходе процесса. Может принимать значение None, To Screen и To File.

Параметру Overwrite нужно присвоить значение True, а потом останется лишь нажать кнопку OK. База данных будет восстановлена.

Как уже было отмечено ранее, восстановление базы данных можно произвести с помощью утилиты gbak. Команда восстановления базы данных имеет следующий синтаксис:

```
gbak {-C|-R} [options] source dbfile
```

При восстановлении базы данных в несколько файлов из одного резервного команда будет иметь несколько иной синтаксис:

```
gbak {-C|-R} [options] source dbfile1 size1 dbfile2 [ size2 dbfile3 ...]
```

А если потребуется восстанавливать базу из нескольких файлов, то команда запуска процесса снова изменится:

```
gbak {-C|-R} [options] source1 source2 [ source3 ...] dbfile1 size1 dbfile2
[ size2 dbfile3 ...]
```

В параметре source указывается файл базы данных, из которого будет производиться восстановление, в параметре dbfile — имя файла базы данных, а в параметре Size указывается размер файла базы данных либо размер файла резервной копии. В параметре [options] указываются дополнительные свойства процесса, перечисленные в табл. 8.45.

Теперь следует рассмотреть несколько примеров восстановления базы данных. Для проведения испытаний сначала потребуется испортить базу MyDB. Следующая команда восстанавливает базу данных MyDB из резервной копии, расположенной на том же носителе, что и исходная база данных:

```
gbak.exe -r -user SYSDBA -password masterkey D:\MyDB.gdb D:\MyIntDB\My.gdb
```

Таблица 8.45. Список параметров утилиты gbak

Параметр	Описание
-c[reate_database]	Восстановление базы данных в новый файл
-bu[ffers]	Определяет размер кэша для восстанавливаемой базы данных
-i[nactive]	Деактивирует индексы в процессе восстановления базы данных
-k[ill]	Запрет на создание Shadow-файлов
-mo[de] [read_write | read_only}	Определяет режим доступа к восстанавливаемой базе данных
-n[o_validity]	Отключает проверку данных на соответствие ограничениям
-o[ne_at_a_time]	Восстанавливает одну таблицу за один проход. Используется в том случае, если файл резервной копии поврежден
-p[age_size] n	Устанавливает размер страницы. По умолчанию используется размер 1024 байта
-pa[ssword] text	Пароль администратора или владельца базы данных
-r[eplace_database]	База данных восстанавливается в новый файл либо перезаписывает старый
-se[rvice] servicename	Восстанавливает базу данных из резервной копии на той машине, на которой расположена база данных, используя Services Manager. Параметр *servicename* вызывает Services Manager на машине с базой данных и может принимать следующие значения: TCP/IP hostname:service_mgr; SPX hostname@service_mgr; Named pipes \\hostname\service_mgr; Local service_mgr
-user name	Имя пользователя
-use_[all_space]	Восстанавливает базу данных со 100%-ным заполнением страниц. По умолчанию используется значение 80%
-v[erbose]	Отображает информацию о ходе процесса
-y [file | suppress_output]	Вывод списка сообщений в файл
-z	Отображает информацию о базе данных и движке InterBase

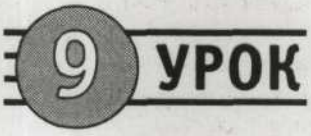
Можно восстановить базу данных из файла резервной копии, расположенного на клиентской машине:

```
gbak.exe -r -user SYSDBA -password masterkey D:\MyDB.gbk
127.0.0.1:D:\MyIntDB\My.gdb
```

Если файл резервной копии находится на той же машине, что и файл с базой данных, но восстановление будет производиться с клиентской машины, то команду придется несколько изменить:

```
gbak.exe -r -user SYSDBA -password masterkey -service 127.0.0.1:service_mgr -p
8192 D:\MyDB.gbk D:\MyIntDB\My.gdb
```

Завершая главу, следует напомнить, что в ней изложены основы работы с сервером InterBase. Однако этой информации уже вполне достаточно для разработки приложений баз данных и обеспечения бесперебойной работы сервера. Последние разделы помогут написать программы, которые будут выполнять резервное копирование базы данных при помощи вызова утилиты gbak с необходимыми параметрами.



Сервер MS SQL Server 2000

SQL Server 2000 является многоцелевым сервером со сложной архитектурой. Этот сервер может использоваться как в промышленных системах, так и в корпоративной среде, сочетая в себе легкость взаимодействия с приложениями с высокой надежностью и отказоустойчивостью. Microsoft SQL Server 2000 является высокопроизводительной клиент-серверной системой управления базами данных. Сервер предназначен для одновременной работы с большим количеством транзакций. Существует несколько вариантов поставки сервера:

- Комплект Enterprise Edition предназначен для использования в качестве корпоративного сервера. В данной конфигурации доступны все возможности сервера. Помимо этого добавлены расширенные возможности масштабирования, использование нескольких процессоров, возможность использования до 64 Гбайт оперативной памяти и многое другое.
- Комплект Standard Edition предназначен для использования в качестве сервера баз данных небольших рабочих групп. Поддерживает работу с восемью процессорами и может использовать до 2 Гбайт оперативной памяти.
- Комплект Personal Edition предназначен для использования в качестве домашнего сервера. Он поддерживает работу с двумя процессорами и использует до 2 Гбайт оперативной памяти. Размер базы данных ограничен 2 Гбайт. Этот вариант сервера оптимизирован на работу максимум с пятью пользователями.
- Комплект Developer Edition предназначен для использования разработчиками приложений баз данных. Эта версия поддерживает все возможности версии Enterprise Edition, но лицензирована только для разработки и тестирования базы данных. Согласно лицензии, сервер в этой комплектации не может использоваться в качестве корпоративного сервера.
- Комплект Desktop Engine является свободно распространяемой версией движка SQL Server 2000, который независимые поставщики могут включать в свои приложения.

- Комплект Windows CE Edition является версией для платформы Windows CE. Этот сервер имеет возможность репликации данных с версий Enterprise и Standard Edition, используя их для синхронизации в качестве первичных серверов.
- Комплект Enterprise Evaluation Edition является полнофункциональной версией сервера, доступной для свободного скачивания из Интернета. Данная версия предназначена для ознакомления с возможностями сервера и не может эксплуатироваться более 120 дней.

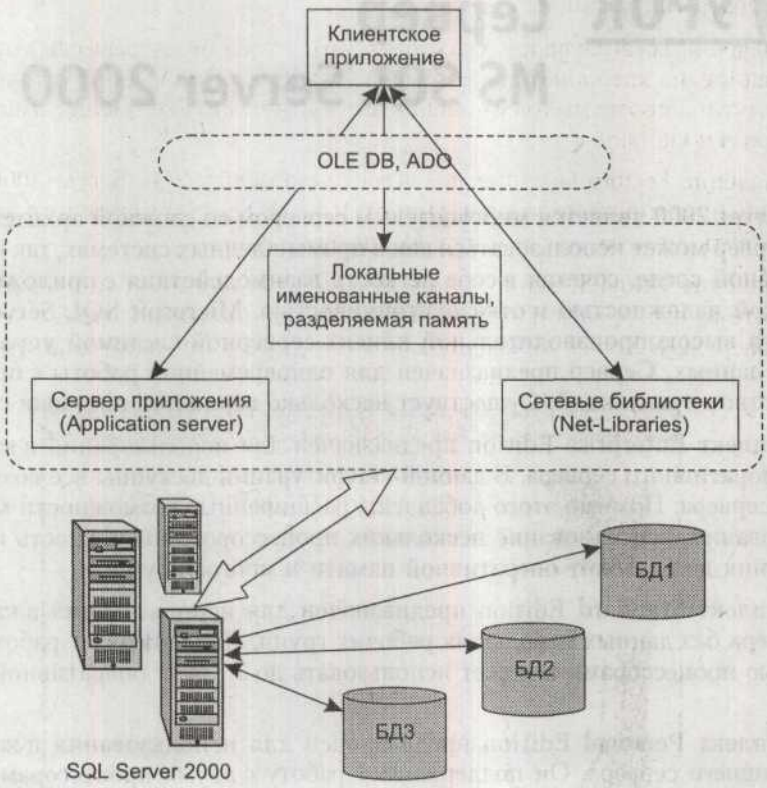


Рис. 9.1. Взаимодействие клиентского приложения и сервера

Как и многие другие продукты Microsoft, сервер использует в своей работе технологию COM, а для связи с приложениями – DCOM. При работе на той же машине, на которой расположен сервер, клиентское приложение взаимодействует с сервером, используя механизмы *InterProcess Communication* (IPC), такие как *локальные именованные каналы*, либо *разделяемую память* (*Shared Memory*). В том случае, если экземпляр SQL Server 2000 и клиентское приложение расположены на разных компьютерах, взаимодействие осуществляется через сетевые IPC, такие как TCP/IP, NWLINK IPX/SPX, именованные каналы и другие сетевые библиотеки (*Net-Libraries*). Поверх них взаимодействие с сервером осу-

ществляется средствами ADO. Также организовать связь сервера с приложением в случае их расположения на разных машинах можно, используя *сервер приложения*. На рис. 9.1 показана упрощенная схема работы с сервером.

Установка SQL Server 2000

Процесс установки сервера начинается с запуска файла \x86\setup\setupsql.exe. В первом окне мастера инсталляции, показанном на рис. 9.2, будет предложено выбрать тип установки:

- Значение Local Computer указывает, что сервер будет размещен на той же машине, на которой была запущена установка. При выборе установки данного типа программа-установщик автоматически подставляет имя компьютера в поле ввода.
- Значение Remote Computer позволяет установить SQL Server 2000 на удаленный компьютер по сети. При этом необходимо указать имя компьютера, путь до него, логин и пароль учетной записи.
- Значение Virtual Server доступно только в том случае, если в операционной системе настроен кластер. Данный тип установки позволяет установить сервер SQL Server 2000 и добавить его в существующий кластер.

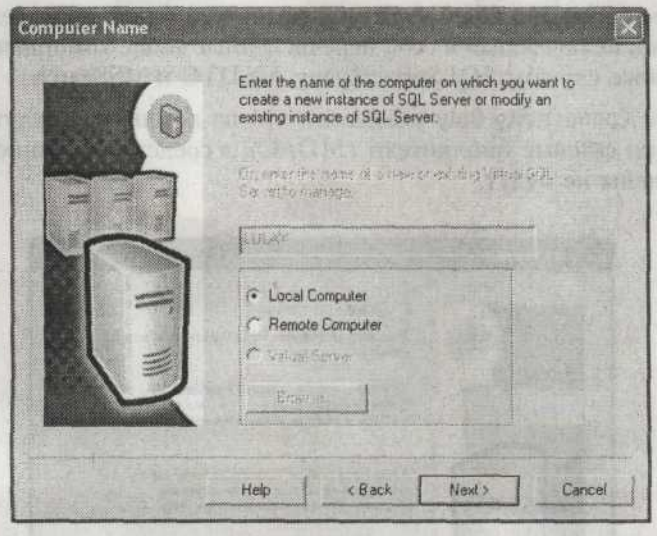


Рис. 9.2. Окно выбора типа установки

В данном случае нужно выбрать значение Local Server и нажать кнопку Next. В результате будет отображено окно, в котором будет предложено ввести информацию о пользователе. После ввода всех необходимых данных нужно нажать кнопку Next. Будет отображено следующее окно мастера установки со списком устанавливаемых компонентов, показанное на рис. 9.3:

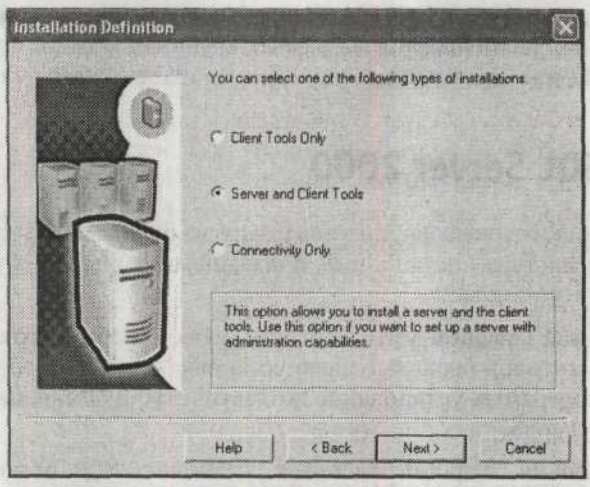


Рис. 9.3. Выбор устанавливаемых компонентов

- Значение Client Tools Only указывает, что будут установлены только инструменты администрирования сервера, такие как Enterprise Manager, Performance Monitor, Query Analyzer и сетевые библиотеки. Также будет установлена документация Books Online. Сам сервер устанавливаться не будет.
- Значение Server and Client Tools указывает, что будет произведена полная установка, включающая в себя перечисленные выше компоненты, сам сервер, а также службы SQLServerAgent, MSDTC и MSSearch.
- Значение Connectivity Only указывает, что на компьютер будут установлены только сетевые библиотеки (MDAC), а средства администрирования установлены не будут.

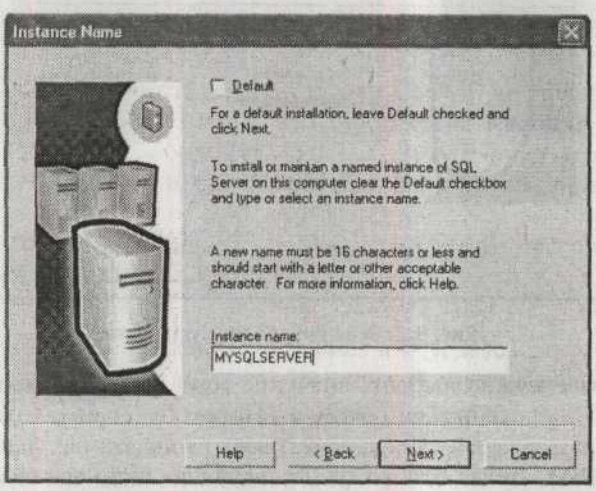


Рис. 9.4. Определение имени экземпляра сервера

На этом этапе необходимо выбрать пункт Server and Client Tools, а потом нажать кнопку Next. В следующем окне мастера, показанном на рис. 9.4, будет предложено указать имя экземпляра сервера.

На одной машине может быть установлено несколько экземпляров SQL Server, причем они не будут зависеть друг от друга. Идентификация экземпляров производится по их имени. По умолчанию, когда установлен флаг Default, серверу не присваивается имя. При обращении к нему по протоколу NetBIOS будет указываться лишь имя компьютера. Если же имя серверу присвоено, то обращение к нему будет производиться по его имени и имени компьютера.

После указания имени сервера нужно снова нажать кнопку Next. Появится окно выбора устанавливаемых модулей, в котором также будет предложено выбрать путь установки сервера. Обычно следует выбрать значение Typical и снова нажать кнопку Next. В результате будет отображено окно выбора учетных записей служб, показанное на рис. 9.5.

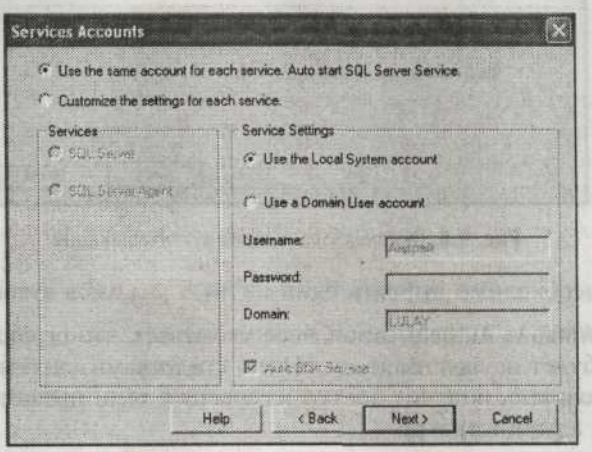


Рис. 9.5. Окно выбора учетных записей служб

В этом диалоговом окне необходимо определить учетные записи Windows, которые будут использоваться двумя основными службами SQL Server – *SQL Server* (ядро сервера) и *SQL Server Agent*. Можно задать отдельную учетную запись для каждой службы с отдельным паролем доступа. А можно оставить общую запись для всех служб. Для того чтобы обеспечить возможность взаимодействия сервера с другими службами, например Microsoft Exchange, необходимо определить ему доменную учетную запись. Таким образом, для выбора доступны два типа учетных записей:

- Значение Use the Local System account означает, что сервер будет запускаться под локальной учетной записью.
- Значение Use a Domain User account означает, что сервер будет запускаться под учетной записью пользователя домена или локального пользователя. Также при выборе этого значения появляется возможность задать логин учетной записи и ее пароль.

Флажок Auto Start Service указывает на необходимость автоматического запуска служб во время запуска операционной системы под зарегистрированной учетной записью. В данном случае необходимо выбрать значение Local System account. В следующем окне, показанном на рис. 9.6, нужно указать используемый режим аутентификации.

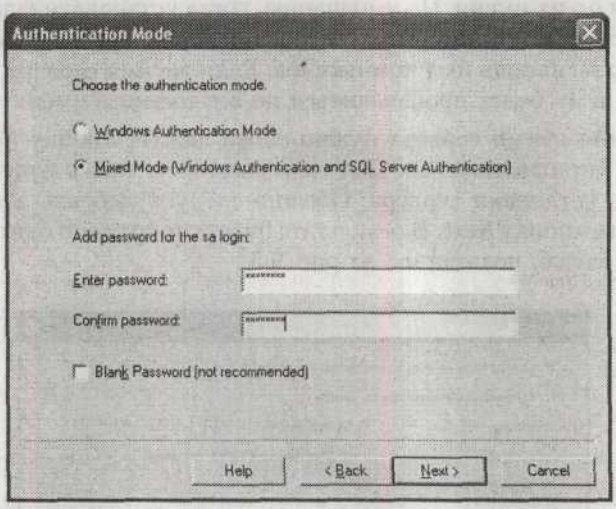


Рис. 9.6. Окно выбора режима аутентификации

В этом окне необходимо выбрать один из двух режимов аутентификации:

- Значение Windows Authentication Mode указывает, что ограничение доступа к серверу будет осуществляться только средствами системы Windows. То есть пользователь получит доступ к серверу с теми правами, которые ему определил администратор.
- Значение Mixed Mode позволяет использовать смешанный режим аутентификации. В этом случае аутентификация производится не только средствами операционной системы, но и самим сервером SQL Server. При выборе этого пункта автоматически становиться доступной возможность установить пароль учетной записи Sa, которая является учетной записью администратора сервера. По умолчанию пароль учетной записи не присваивается, и доступ с ее правами на сервер может получить любой желающий. Поэтому на этом этапе обязательно нужно установить пароль для этой учетной записи.

В данном случае необходимо выбрать режим Windows Authentication Mode и нажать кнопку Next. После этого начнется установка сервера.

Архитектура SQL Server 2000

На рис. 9.7 приведена обобщенная схема архитектуры SQL Server 2000.

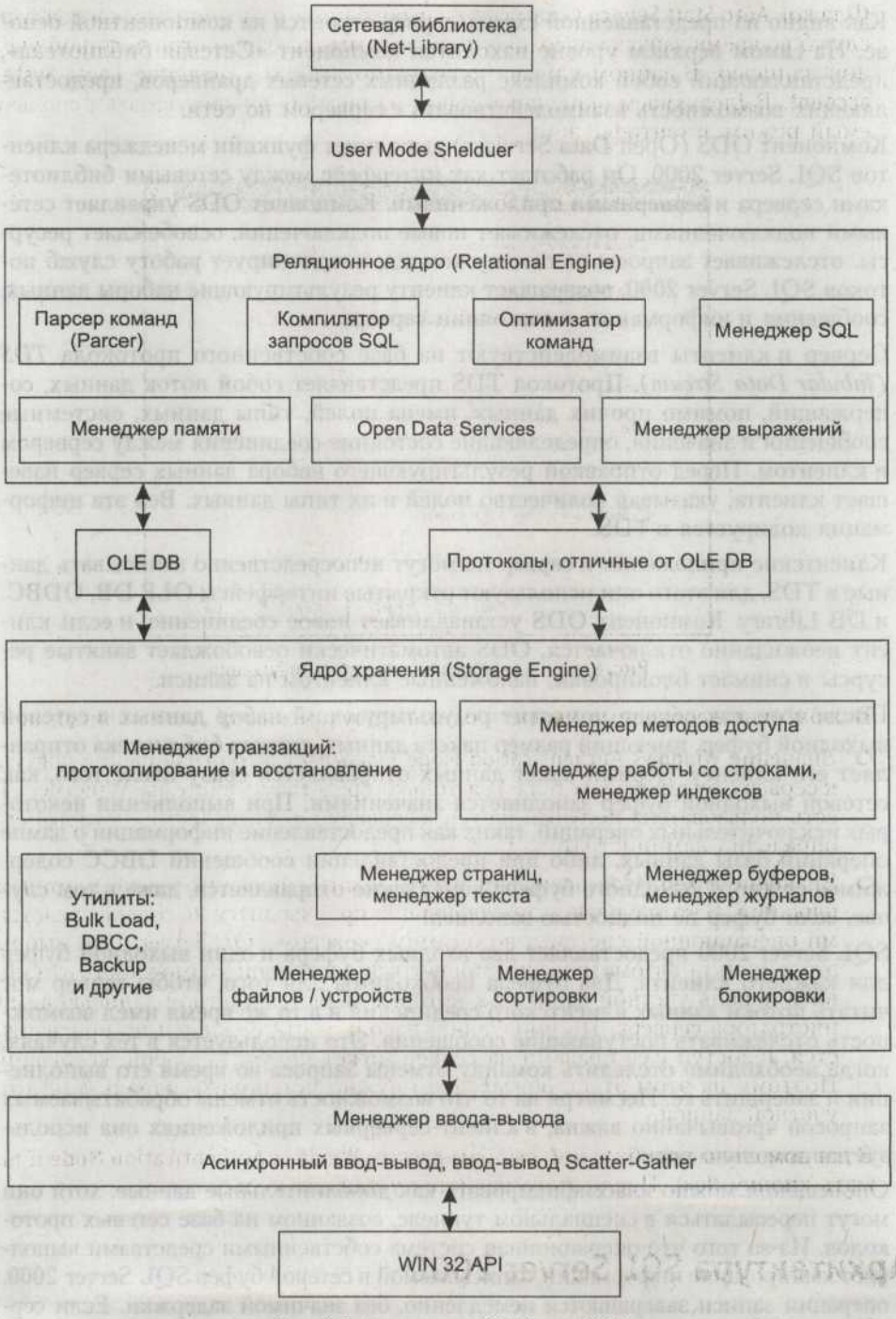


Рис. 9.7. Архитектура SQL Server 2000

Как видно из представленной схемы, сервер строится на компонентной основе. На самом верхнем уровне находится компонент «Сетевая библиотека», представляющий собой комплекс различных сетевых драйверов, предоставляющих возможность взаимодействовать с сервером по сети.

Компонент ODS (Open Data Services) выполняет функции менеджера клиентов SQL Server 2000. Он работает как интерфейс между сетевыми библиотеками сервера и серверными приложениями. Компонент ODS управляет сетевыми подключениями: отслеживает новые подключения, освобождает ресурсы, отслеживает запросы на отмену команд, координирует работу служб потоков SQL Server 2000, возвращает клиенту результирующие наборы данных, сообщения и информацию о состоянии сервера.

Сервер и клиенты взаимодействуют на базе собственного протокола *TDS (Tabular Data Stream)*. Протокол TDS представляет собой поток данных, содержащий, помимо прочих данных, имена полей, типы данных, системные сообщения и значения, определяющие состояние соединения между сервером и клиентом. Перед отправкой результирующего набора данных сервер извещает клиента, указывая количество полей и их типы данных. Вся эта информация кодируется в TDS.

Клиентские приложения и сервер не могут непосредственно записывать данные в TDS, для этого они используют открытые интерфейсы OLE DB, ODBC, и DB Library. Компонент ODS устанавливает новое соединение, и если клиент неожиданно отключается, ODS автоматически освобождает занятые ресурсы и снимает блокировки, наложенные клиентом на записи.

После того как сервер поместит результирующий набор данных в сетевой выходной буфер, имеющий размер пакета данных, сетевая библиотека отправляет его клиенту. Первый пакет данных отправляется сразу после того, как сетевой выходной буфер заполняется значениями. При выполнении некоторых исключительных операций, таких как предоставление информации о дампе операций базы данных, либо при предоставлении сообщений DBCC содержимое сетевого выходного буфера немедленно отправляется, даже в том случае, если буфер не полностью заполнен.

SQL Server 2000 предоставляет два входных буфера и один выходной буфер для каждого клиента. Два буфера необходимы для того, чтобы сервер мог читать потоки данных клиентского соединения и в то же время имел возможность отслеживать поступающие сообщения. Это используется в тех случаях, когда необходимо отследить команду отмены запроса во время его выполнения и завершить ее. Несмотря на то что возможность отмены обрабатываемых запросов чрезвычайно важна, в клиент-серверных приложениях она используется довольно редко .

Оповещения можно классифицировать как дополнительные данные, хотя они могут пересылаться в специальном туннеле, созданном на базе сетевых протоколов. Из-за того что операционная система собственными средствами выполняет кэширование информации, записываемой в сетевой буфер SQL Server 2000, операции записи завершаются немедленно, без значимой задержки. Если сервер произведет несколько операций записи пакетов данных для какого-либо

клиента, а тот не успеет их вовремя считать из собственного сетевого кэша, то кэш в конечном итоге переполнится и операции записи будут заблокированы до момента его освобождения. В то время как клиентское приложение обрабатывает часть результирующего набора данных, сервер продолжает пересылать пакеты в другие буферы и отправлять их по мере возможности.

В то же время приостановленный процесс передачи данных может вызвать эффект блокировки. Это обычно случается, если транзакция заблокировала какой-то блок данных. В случае нормальной работы данные будут отосланы и немедленно разблокированы. Иначе, если клиент не успевает считывать данные из сетевого входного буфера, сервер не разблокирует ресурсы до завершения операции. Следовательно, остальные заинтересованные транзакции будут простаивать в их ожидании.

Размер выходного сетевого буфера существенно влияет на время получения клиентом первого пакета с результирующим набором данных. Как было отмечено ранее, выходной буфер отсылает пакет даже в том случае, если он неполный, но в данный момент больше не существует данных, которые могут быть в него помещены. Если первый запрос возвратил небольшой набор данных и не заполнил выходной буфер полностью, а в то же время выполняется второй запрос, возвращающий больший набор данных, то пакет с данными, содержащимися в сетевом буфере, не будет отправлен до тех пор, пока он не будет заполнен полностью. Если оба SQL-запроса возвращают небольшие наборы данных, то ожидание завершения процесса не продлится долго.

Эту концепцию проще рассмотреть при помощи небольшого примера. Предположим, что одновременно выполняется два запроса. Первый запрос является «быстрым», а второй — «медленным». Первый результирующий набор, к примеру, имеет размер 0,5 Кбайт. В том случае, если размер пакета данных составляет 4 Кбайт, первый запрос вынужден будет ожидать заполнения оставшейся части пакета результатом выполнения второго запроса.

В данном случае было бы разумно отправить результат выполнения первого запроса отдельным пакетом или уменьшить размер пакета данных. Первое решение, вероятно, является более подходящим для этого случая, так как довольно сложно определить оптимальный размер пакета данных для каждой команды. В общем случае передача результатов выполнения запроса от сервера клиенту в отдельном пакете для каждого запроса не является оптимальной. Группировка результатов выполнения запросов в общие пакеты, как правило, более эффективна, потому что снижает затраты на установление соединения между сервером и клиентом и повышает среднюю скорость передачи данных.

На стороне сервера компонент ODS выполняет те же функции, что ODBC, OLE DB и DB-Library на клиентской стороне. Серверное приложение ODS может использовать методы, предназначенные для описания и отправки результирующих наборов данных, преобразования типов данных, назначения контекста безопасности определенному соединению между клиентом и сервером и возвращения клиентскому приложению информации об ошибках и сообщениях.

Компонент ODS использует событийно-управляемую модель программирования. Сообщения, пересылаемые между сервером и клиентом, регистрируются ODS и могут быть переданы в серверное приложение. Используя ODS API, можно создавать собственные процедуры, называемые обработчиками событий, для каждого возможного типа сообщения. Список обрабатываемых событий приведен ниже:

- События, возникающие при подключении (Connect events), позволяют проводить проверку прав пользователя на доступ к серверу. Эти события также инициируются в момент завершения соединения, указывая серверу на необходимость освобождения ресурсов.
- События языка (Language events) возникают в тот момент, когда клиент отсылает строку, содержащую команду. Сервер, в свою очередь, передает полученную команду синтаксическому анализатору.
- События, возникающие при работе с удаленными хранимыми процедурами (Remote stored procedure events), инициируются каждый раз, когда клиент или сервер направляют ODS вызов хранимой процедуры.

Компонент ODS также генерирует события, точно определяющие состояние клиентского и серверного приложений. Эти сообщения позволяют приложению ODS, расположенному на сервере, реагировать на изменение статуса клиентского приложения либо на изменение статуса приложения ODS. Также компонент ODS управляет потоками и нитями SQL Server 2000. Он создает и уничтожает потоки и обеспечивает доступ к ним для компонента User Mode Scheduler (UMS). На рис. 9.8 показана схема взаимодействия клиента и сервера.

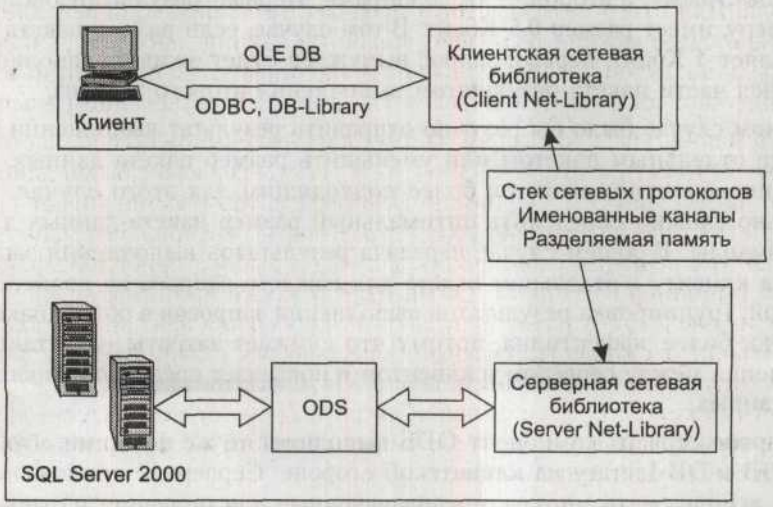


Рис. 9.8. Коммуникация между клиентом и сервером

Ядро SQL Server 2000 состоит из двух главных компонентов: реляционного ядра (Relational Engine) и ядра хранения (Storage Engine). Между собой дан-

ные компоненты взаимодействуют с помощью OLE DB. Реляционное ядро включает в себя все компоненты, необходимые для анализа и оптимизации SQL-запросов. Также реляционное ядро управляет выполнением запросов на получение наборов строк от ядра хранения, а затем возвращает их клиентскому приложению. Ядро хранения включает в себя компоненты, обеспечивающие доступ к данным и их физическую модификацию.

Анализатор команд (парсер) обрабатывает события языка, вызываемые компонентом ODS. Он проверяет правильность синтаксиса и транслирует команды Transact-SQL во внутренний формат, с которым работает сервер. Если парсер не может распознать синтаксис команды, то он немедленно генерирует объект исключения, содержащий сообщение об ошибке и информацию об операторе, вызвавшем ее. Сообщения о несинтаксических ошибках не могут содержать номер строки, вызвавшей ошибку.

Оптимизатор получает дерево запроса от парсера команд и подготавливает его к выполнению. Этот модуль компилирует сразу весь пакет команд, оптимизирует его и производит проверку прав пользователя на доступ к объектам. Результатом оптимизации и компиляции запроса является план его выполнения. Первым шагом в составлении плана запроса является нормализация каждого запроса, в ходе которой производится разложение одного запроса на несколько элементарных подзапросов. После того как оптимизатор нормализовал запрос, он оптимизирует его, определяя план для выполнения запроса. Оптимизация запроса производится в соответствии с его ресурсоемкостью. Оптимизатор выберет наилучший план, руководствуясь наименьшей внутренней единицей варианта плана, включающей вычисленные значения затрат оперативной памяти, загрузки центрального процессора и операций ввода-вывода. Для этого оптимизатор учитывает тип запроса, определяет количество данных в каждой из задействованных таблиц, определяет наличие индексов для каждой из задействованных таблиц, а затем просматривает распределение данных в каждом индексе или поле, задействованном в запросе.

Информация о распределении данных в индексе или в поле является статистической, то есть формируется по ходу работы сервера с базой данных. Опираясь на имеющуюся информацию, оптимизатор рассматривает различные методы доступа и связывает их со стратегиями, которые могут быть использованы для решения выполнения запроса и выбора наиболее эффективного плана выполнения запроса. Оптимизатор также принимает решение о том, какие индексы он будет использовать в запросе для каждой из таблиц и в каком порядке в случае использования нескольких таблиц в одном запросе таблицы будут включены в объединение. Оптимизатор использует упрощенные эвристические методы, снижающие затраты времени, используемые на составление плана запроса. Следует отметить, что оптимизатор не обеспечивает идеальной оптимизации запроса.

После того как нормализация и оптимизация запроса были выполнены, нормализованное дерево запроса, полученное с помощью описанных выше операций, компилируется в план выполнения, представляющий собой структуру данных. Каждая команда, включенная в план, содержит информацию, определяющую таблицу, с которой она будет работать, какой индекс будет исполь-

зован с данной таблицей, какие проверки должны быть выполнены и какому критерию должны соответствовать отбираемые данные.

План выполнения может быть значительно более сложным, чем кажется на первый взгляд. Он включает в себя все необходимые шаги, гарантирующие осуществление всех необходимых проверок. Если в запросе вызывается триггер, то в план выполнения запроса включается вызов соответствующей процедуры, содержащей тело триггера. Триггер хранится в базе данных в откомпилированном виде и обладает собственным планом запроса.

Менеджер SQL (SQL Manager) предназначен для управления хранимыми процедурами и их планами выполнения. Он определяет, когда хранимая процедура должна быть перекомпилирована, и управляет кэшированием плана выполнения процедуры. Менеджер SQL также управляет автопараметризацией запросов. В SQL Server 2000 определенные виды запросов обрабатываются так, как если бы они были параметризованными хранимыми процедурами. В соответствующем формате генерируются и сохраняются их планы. Как правило, так обрабатываются простые запросы, в которых производится сравнение значения поля с константой. Ниже приведен пример простого запроса:

```
SELECT * FROM pubs.dbo.titles
WHERE type = 'business'
```

Этот запрос может быть преобразован в параметризованный запрос, принимающий входной параметр:

```
SELECT * FROM pubs.dbo.titles
WHERE type = @param
```

Следующий запрос будет отличаться только значением параметра, а план выполнения запроса останется тем же.

Менеджер выражений (Expression Manager) управляет ходом вычислений, сравнением и перемещением данных. Его действие лучше продемонстрировать на примере SQL-запроса:

```
SELECT @myqty = qty * 10 FROM mytable
```

В данном запросе менеджер выражений копирует значение поля qty из набора строк, увеличивает его на порядок и записывает результат в переменную @myqty.

Исполнитель запросов выполняет полученный от оптимизатора план выполнения запроса путем последовательного выполнения всех входящих в него команд. Многие команды требуют осуществления взаимодействия с ядром хранения.

Для взаимодействия с подсистемой хранения реляционное ядро использует механизмы OLE DB. Схема взаимодействия показана на рис. 9.9.

Процесс взаимодействия между реляционным ядром и подсистемой хранения показан на примере обработки выражения SELECT, осуществляющего выборку данных из локальных таблиц

Прежде всего реляционное ядро компилирует выражение SELECT в оптимизированный план выполнения запроса. План выполнения определяет последовательность операций с наборами строк, полученных из таблиц, и связанных

с ними индексов. Наборы данных содержат записи, формирующие результирующий набор данных. Например, таблица требует полного сканирования, если она не содержит индексов:

```
SELECT * FROM Northwind.dbo.ScanTable
```

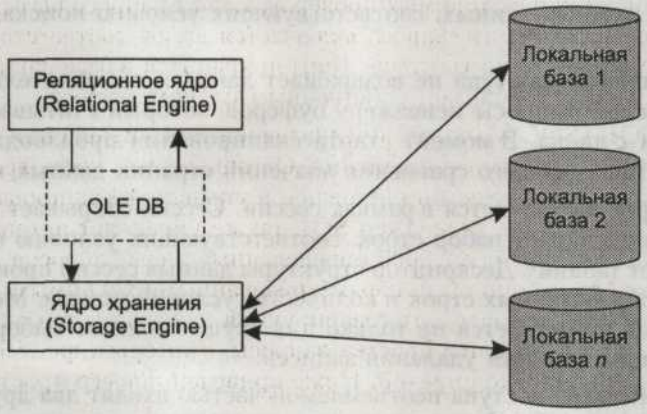


Рис. 9.9. Взаимодействие между реляционным ядром и ядром хранения

Для выполнения запроса реляционная подсистема запрашивает один набор строк, содержащий все строки таблицы ScanTable. Следующий запрос, использующий индекс, будет выполнен гораздо быстрее:

```
SELECT DISTINCT LastName
FROM Northwind.dbo.Employees
```

Для выполнения этого запроса реляционное ядро выполняет сканирование индекса для запроса одного набора строк, содержащего строки, расположенные в листьях индекса, построенного по полю LastName. Следующий запрос возвращает результирующий набор строк, используя два индекса:

```
SELECT CompanyName. OrderID. ShippedDate
FROM Northwind.dbo.Customers AS Cst
JOIN Northwind.dbo.Orders AS Ord
ON (Cst.CustomerID = Ord.CustomerID)
```

Реляционное ядро запрашивает два набора данных. Один строится на основе кластерного индекса, созданного по полю Customers, а другой — на основе некластерного индекса, созданного по полю Orders.

После этого реляционное ядро использует OLE DB API для того, чтобы ядро хранения открыло эти наборы данных. Во время выполнения плана запроса реляционное ядро с помощью OLE DB запрашивает нужные строки у ядра хранения, которое, в свою очередь, открывает наборы данных и копирует нужные строки в буферы данных.

Реляционное ядро по ходу выполнения запроса и получения данных от ядра хранения формирует результирующий набор данных, который отсылает пользователю.

Когда SQL Server 2000 необходимо найти какие-либо данные, он вызывает компонент Менеджер методов доступа (Access Methods Manager). Менеджер методов доступа производит сканирование страниц данных и индексов, а также подготавливает наборы строк для возвращения реляционному ядру. Этот компонент имеет в своем составе службы, предназначенные для открытия таблиц, нахождения нужных данных, соответствующих условию поиска, и обновления данных.

Менеджер методов доступа не возвращает данные самостоятельно. Вместо этого он передает запросы менеджеру буферов, который считывает страницу из кэша или с диска. В момент старта сканирования производится запуск механизма упреждающего сравнения значений страниц данных и индексов.

Каждый запрос выполняется в рамках сессии. Сессия открывает таблицу, запрашивает и оценивает набор строк, соответствующих условию отбора, а затем закрывает таблицу. Дескриптор структуры данных сессии производит учет количества обработанных строк и количества условий поиска. Менеджер методов доступа применяется не только для осуществления выборки данных, но и для обновления или удаления записей.

В состав менеджера доступа неотъемлемой частью входят два других компонента. Это менеджер работы со строками и менеджер индексов. Каждый из них ответственен за управление соответствующими структурами данных на диске. Именно они управляют информацией, содержащейся на страницах данных и индексов.

Результатом работы, выполняемой менеджером методов доступа, менеджером блокировок и менеджером транзакций, является обнаружение строки, наложение на нее блокировки и включение ее в транзакцию. После форматирования или изменения строки в памяти менеджер работы со строками вставляет или удаляет ее.

SQL Server 2000 предлагает три метода обновления строк. Все три метода являются прямыми, то есть не требующими двух проходов журнала транзакций:

- Режим In-place mode используется для обновления кучи или кластерных индексов, когда ключевые поля, входящие в индекс, не изменяются. В этом режиме новые данные записываются на место старых на той же странице.
- Режим Split mode используется для обновления неуникальных индексов при изменении полей, входящих в ключ. Обновление производится в два этапа. Сначала удаляется старое значение индекса, затем производится вставка нового значения. Операции выполняются независимо друг от друга.
- Режим Split with collapse mode используется для обновления уникальных индексов при изменении полей, входящих в ключ. Обновление производится в два этапа. Сначала удаляется старое значение индекса, затем производится вставка нового значения. Если новое значение записывается на место старого, то обе операции объединяются в одну.

Менеджер индексов обеспечивает поиск данных в B-деревьях, на основе которых строятся индексы в SQL Server 2000. Записи с подобными значениями ключевых полей группируются вместе. Это обеспечивает быстрый доступ к данным при поиске по ключевым значениям с использованием индекса.

Ключевой особенностью B-деревьев является сбалансированность индексов. Они создают такую структуру индекса, что доступ к любой его части занимает равный промежуток времени и одинаковое количество уровней страниц индексов. Перемещение по телу индекса начинается с корневой страницы. Затем происходит перемещение через промежуточные уровни. Подобным способом поиск доходит до страниц нижнего уровня, называемых листьями. Листья и являются теми страницами, поиск которых осуществляется в индексе. Они содержат либо саму искомую строку, либо ссылку на строку в таблице базы данных.

SQL Server 2000 поддерживает кластерные и некластерные индексы. В некластерном индексе, схема которого приведена на рис. 9.10, на уровне листьев дерева индекса содержатся ключевые значения данных и закладки для каждого значения индекса.

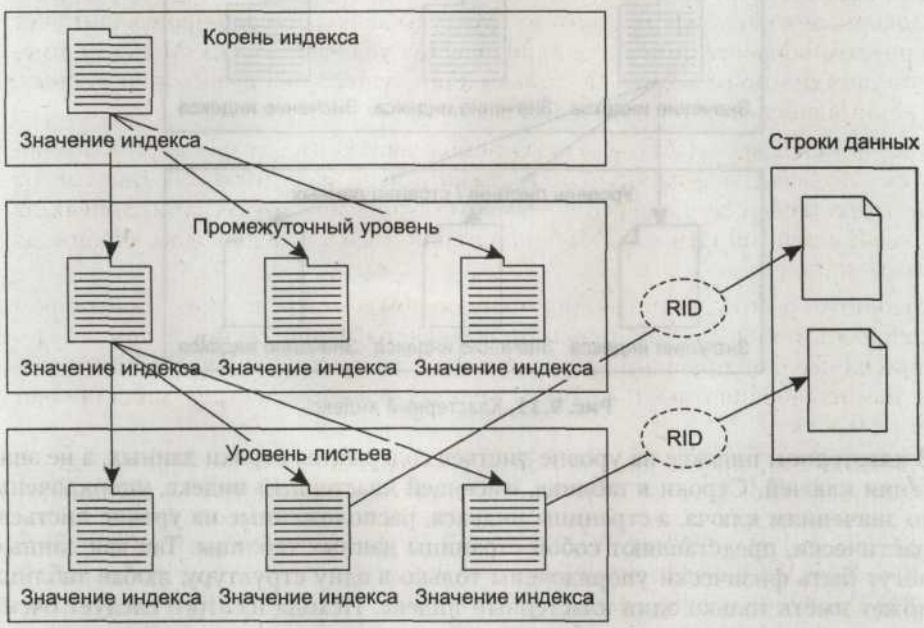


Рис. 9.10. Некластерный индекс

Закладка указывает, где следует искать связанные с ней строки данных. Если таблица имеет некластерный индекс, то закладка рассматривается как «куча», то есть данные в ней хранятся в неупорядоченном виде. Закладка в некластерном индексе, содержащаяся на уровне листьев индекса, указывает на строку данных в куче. Закладка состоит из идентификатора строки, которой составлен из идентификатора номера файла, номера страницы данных и номера строки в странице данных.

Если таблица имеет кластерный индекс, то закладка содержит значение ключа строки из кластерного индекса. После того как искомый уровень листьев некластеризованного индекса будет достигнут, сервер получит информацию о расположении искомых данных. Однако извлечение данных будет произведено

отдельной операцией. За счет того что доступ к данным производится практически напрямую, нет необходимости сканировать всю таблицу, что увеличивает производительность. При использовании кластерного индекса, схема которого приведена на рис. 9.11, доступ к данным осуществляется еще быстрее.

Рис. 9.11. Кластерный индекс

В кластерном индексе на уровне листьев содержатся строки данных, а не значения ключей. Строки в таблице, имеющей кластерный индекс, упорядочены по значениям ключа, а страницы индекса, расположенные на уровне листьев, фактически, представляют собой страницы данных таблицы. Так как данные могут быть физически упорядочены только в одну структуру, любая таблица может иметь только один кластерный индекс. Исходя из этого следует очень внимательно отнестись к выбору полей, входящих в кластерный индекс.

Также индексы используются для обеспечения уникальности значений определенных полей. Определяя полю ограничение `PRIMARY KEY` или `UNIQUE`, сервер, фактически, создает уникальный индекс по входящим в них полям. Оптимизатор может использовать сведения о том, что индекс является уникальным, для формирования более эффективного плана запроса. При использовании кластерного индекса SQL Server 2000 всегда гарантирует уникальность входящих в него значений при помощи добавления к значениям ключевого поля уникального идентификатора. Уникальный идентификатор присваивается ключевым значениям на всех уровнях кластерного индекса и на всех листьях некластерных индексов таблицы.

Менеджер страниц и менеджер текста совместно управляют набором страниц, являющимся базой данных. Каждая база данных представляет собой набор

страниц, размером 8 Кбайт, располагающихся в одном или нескольких файлах. SQL Server 2000 использует восемь типов страниц. Он поддерживает страницы данных, страницы текста и графических изображений, страницы индексов, страницы с информацией о свободном пространстве на других страницах, страницы с информацией о выделенных экстентах, страницы с данными о пользователях экстентов, страницы, хранящие информацию о том, был ли экстент изменен при пакетных обновлениях, и страницы, хранящие информацию о том, был ли экстент изменен дифференциально.

Все пользовательские данные, за исключением данных типа `text`, `ntext` и `image`, хранятся на страницах данных. Эти три типа данных используются для хранения объектов большой размерности и потому могут храниться в виде обособленных наборов страниц. Если запись содержит поля типа `text`, `ntext` и `image`, то они сохраняются в отдельных наборах страниц. Поля остальных типов хранятся на обычных наборах страниц, и рядом со строками хранится указатель на начальную страницу и смещение связанного поля. Если в полях типа `text`, `ntext` и `image` содержатся данные, занимающие небольшой объем, то значение поля сохраняется на основной странице вместе с остальными данными.

Индексные страницы хранят B-деревья, ускоряющие доступ к данным. Менеджер страниц управляет выделением и освобождением всех типов дисковых страниц, организованных в единицы хранения, называемые *экстентами*, по 8 страниц в каждом. В однородных экстентах все восемь страниц выделяются для одного объекта. Смешанные экстенты позволяют использовать свое пространство для хранения разных объектов. Если некоторый объект использует меньше одного экстента, менеджер страниц размещает его в смешанном экстенте. Если размер экстента превышает восемь страниц, пространство для данных выделяется целыми экстентами. Такая схема оптимизации предотвращает излишнее расходование места и снижает дефрагментацию базы данных. Для определения степени фрагментированности данных таблицы можно воспользоваться командой `DBCC SHOWCONTIG`, которая выведет статистику по всей базе данных.

Таблица, для которой много раз выделялось и освобождалось место, может оказаться сильно фрагментированной. В этой случае можно перестроить кластерный индекс при помощи команды `DBCC INDEXDEFRAG`. После дефрагментации скорость доступа к таблице существенно возрастет. Особенно это будет заметно в тех случаях, когда в процессе выборки используются индексы.

Ключевой функцией SQL Server 2000 является гарантия обеспечения соответствия выполняемых транзакций свойству *ACID* (Atomic, Consistency, Isolation, Durability). Транзакция должна быть атомарной (Atomic) — то есть должны быть выполнены или все ее действия, или ни одно из них. Если транзакция была зафиксирована, должна сохраняться возможность ее отката, несмотря на то, что через секунду может произойти сбой системы. Если сбой системы происходит во время выполнения транзакции и транзакция не успела завершить свою работу, при следующем старте сервера производится откат изменений до состояния, которое данные имели до начала работы транзакции.

Перед тем как произвести какие-либо действия в рамках транзакции, сервер протоколирует их в журнале и в последующих шагах вносит изменения в данные. В случае необходимости их можно будет восстановить по записям в журнале.

Менеджер транзакций координирует протоколирование, восстановление данных и управляет работой буферов. Менеджер транзакций определяет команды, которые должны быть сгруппированы вместе для выполнения в рамках одной транзакции. Также он управляет транзакциями, затрагивающими несколько баз данных, и последовательностями вложенных транзакций. Вложенные транзакции выполняются в контексте главной транзакции. При откате главной транзакции автоматически откатываются и вложенные транзакции.

Для выполнения распределенных транзакций, затрагивающих другой сервер или иной менеджер ресурсов, менеджер транзакций взаимодействует со службой Microsoft Distributed Transaction Coordinator, которая, в свою очередь, использует службу удаленного вызова процедур (Remote Procedure Calls, RPC), входящую в состав операционной системы. Менеджер транзакций позволяет использовать точки сохранения, позволяющие программисту определять точки в теле транзакции, до которых в случае какого-либо сбоя будет выполнен откат.

Менеджер транзакций взаимодействует с менеджером блокировок для определения моментов, когда блокировка может быть снята, с учетом уровня изоляции транзакции. Уровень изоляции, в рамках которого выполняется данная транзакция, определяет уровень ее чувствительности к изменениям, внесенным в данные другими транзакциями, и промежуток времени, в течение которого будет производиться защита ресурса от изменения его другими транзакциями.

В SQL Server 2000 определено четыре уровня изоляции транзакции: Uncommitted Read, Committed Read, Repeatable Read и Serializable.

Уровень изоляции транзакции Uncommitted Read позволяет транзакции считывать любые данные на выбранной странице вне зависимости от того, были или нет эти данные зафиксированы. Предположим, пользователь может выполнять некоторую транзакцию, обновляющую данные. Даже в том случае, если он установил монопольную блокировку на данные, текущая транзакция, имеющая уровень изоляции Uncommitted Read, сможет прочесть их. После этого пользователь может произвести откат транзакции. Логически будет считаться, что изменения в базе не были сделаны. Но так как первый пользователь прочитал измененные данные, он будет работать с заведомо неверной информацией, которую и сохранит в базе данных. В однопользовательской системе, в которой работа с базой данных происходит по очереди, грязного чтения произойти не сможет. В многопользовательской системе существует опасность использования незафиксированных данных. Но если использовать уровень изоляции транзакции Uncommitted Read, будет существенно ускорена работа по чтению данных, так как пользователям не придется дожидаться снятия блокировки с ресурса.

Уровень изоляции транзакции Committed Read в SQL Server 2000 используется по умолчанию. На данном уровне обеспечивается гарантия того, что транзакция не сможет прочитать незафиксированные данные, с которыми в данный момент работает другая транзакция. Если текущей транзакции необходимо получить доступ к изменяемым данным, ей будет необходимо дождать-

ся окончания работы с ними. Перед прочтением данных транзакция налагает на них блокировку share lock, при которой другие транзакции тоже могут просматривать данные, но не изменять их. Блокировка снимается после прочтения данных и отправки их клиенту. Хотя транзакция не может прочитать незафиксированные данные, при повторном обращении к тем же данным они могут быть изменены. Следовательно, результат запроса будет расходиться с изначальным положением дел. Если значения запрашиваемых данных были изменены, то это несоотвествие называется неповторяемым чтением, а новые строки называются фантомами.

Уровень изоляции транзакции Repeatable Read гарантирует, что в процессе работы транзакции данные, которые она использует, не смогут быть изменены и повторное их чтение вернет те же результаты, что и изначальное. То есть он обеспечивает больший уровень изоляции, чем Committed Read. Однако при данном уровне изоляции все же возможно появление фантомов. При чтении какой-либо записи транзакция накладывает на нее блокировку share lock и не снимает ее до тех пор, пока транзакция не будет зафиксирована либо пока не будет произведен ее откат. Использовать уровень изоляции транзакции Repeatable Read следует осмотрительно, так как он требует больших затрат ресурсов системы.

На уровне изоляции Serializable обеспечивается повторное чтение одних и тех же данных и гарантируется невозможность появления фантомов. Для обеспечения данного уровня изоляции производится блокировка не отдельных строк набора, а целого диапазона строк. В этот диапазон не могут быть добавлены новые записи, и данные во входящих в него записях тоже не могут быть изменены. Естественно, этот уровень изоляции транзакции требует максимальных затрат ресурсов, так как по ходу выполнения он налагает на записи блокировку share lock и снимает ее только после завершения своей работы.

Блокировка является ключевой функцией в многопользовательском сервере баз данных, таком как SQL Server 2000. SQL Server 2000 осуществляет одновременное управление многими пользователями и гарантирует, что выполняемым транзакциям будет обеспечен именно тот уровень видимости, который определен их уровнем изоляции. При использовании наивысшего уровня изоляции транзакций Serializable можно обеспечить в многопользовательском режиме работы эмуляцию однопользовательского режима. В этом режиме пользователи как будто по очереди получают доступ к ресурсам и данными, и их действия не затрагивают друг друга.

Блокировки защищают данные и внутренние ресурсы, обеспечивая возможность параллельной работы с ними многих пользователей с сохранением логической целостности данных. Менеджер блокировок налагает и снимает с данных различные типы блокировок, такие как Share Locks — для чтения данных и Exclusive Locks — для записи. Менеджер блокировок определят совместимость блокировок различных типов, разрешает взаимоблокировки (Deadlocks) и при необходимости вызывает эскалацию блокировок.

Менеджер блокировок предоставляет два различных механизма наложения блокировок. Первый способ включает механизм блокировки строк, страниц

и таблиц для полностью разделяемых между всеми пользователями таблиц с данными, страниц с данными, строк, страниц с текстом, страниц уровня листьев индекса и строк индекса. Второй механизм блокировки предназначен для внутреннего использования и обеспечивает защиту системных данных. Он обеспечивает защиту от изменения корневых и промежуточных страниц индекса, пока производится его сканирование. Это внутренний механизм использует блокировки «latches», представляющие собой вид блокировок, кратковременно налагаемых на ресурс и не ожидающих завершения транзакции для завершения своей работы. Использование полных блокировок в столь критичных местах могло бы полностью затормозить систему. В дополнение, для обеспечения защиты данных верхнего уровня, входящих в индекс, блокировки «latches» используются для их защиты при пересылке от ядра хранения реляционному ядру.

Сетевые библиотеки

Сетевые библиотеки представляют собой абстрактный программный слой, позволяющий серверу обмениваться информацией с клиентскими приложениями при помощи различных протоколов. Каждому протоколу соответствует свой собственный драйвер. Сетевые библиотеки позволяют относительно просто работать с различными сетевыми протоколами без необходимости изменения ядра сервера или кода приложения.

Фактически, сетевая библиотека является драйвером, реализующим часть механизма межпроцессного взаимодействия (Interprocess Communication, IPC). Весь код SQL Server 2000, включая код сетевой библиотеки, использует вызовы функций подсистемы WIN 32. Сервер управляет сетевыми библиотеками, используя общий внутренний интерфейс между ODS, который управляет соединением и сетевыми библиотеками.

SQL Server 2000 использует абстрактный слой сетевых библиотек на клиентской и серверной сторонах, делая возможным одновременное взаимодействие с сервером нескольких клиентов, использующих различные протоколы. Операционные системы Windows NT/2000 и Windows 98 поддерживают одновременное использование разнородного стека протоколов.

В SQL Server 2000 используются первичные и вторичные сетевые библиотеки. Протоколы TCP/IP, Named Pipes и IPX/SPX относятся к вторичным сетевым библиотекам. По умолчанию взаимодействие между клиентским приложением и экземпляром сервера SQL Server 2000, расположенными на одной машине, осуществляется при помощи разделяемой памяти (Shared Memory). На рис. 9.12 приведен соответствующий пример.

Взаимодействие между сервером и клиентами, расположенными на разных компьютерах, осуществляется через первичную сетевую библиотеку Super Socket. Сетевая библиотека Super Socket может использовать два типа соединений

- При использовании сокетов TCP/IP или протокола NWLINK IPX/SPX сетевая библиотека Super Socket напрямую вызывает методы Windows

Socket 2 API, с помощью которых осуществляется взаимодействие между клиентским приложением и экземпляром SQL Server 2000.

- Если используются Named Pipes, Virtual Interface Architecture (VIA) SAN, Multiprotocol, AppleTalk или Banyan VINES, сетевая библиотека Super Socket использует драйвер, соответствующий данному протоколу. Взаимодействие между сетевой библиотекой и драйвером (вторичной сетевой библиотекой) осуществляется через специальный промежуточный компонент — маршрутизатор сетевой библиотеки (Net-Library router).

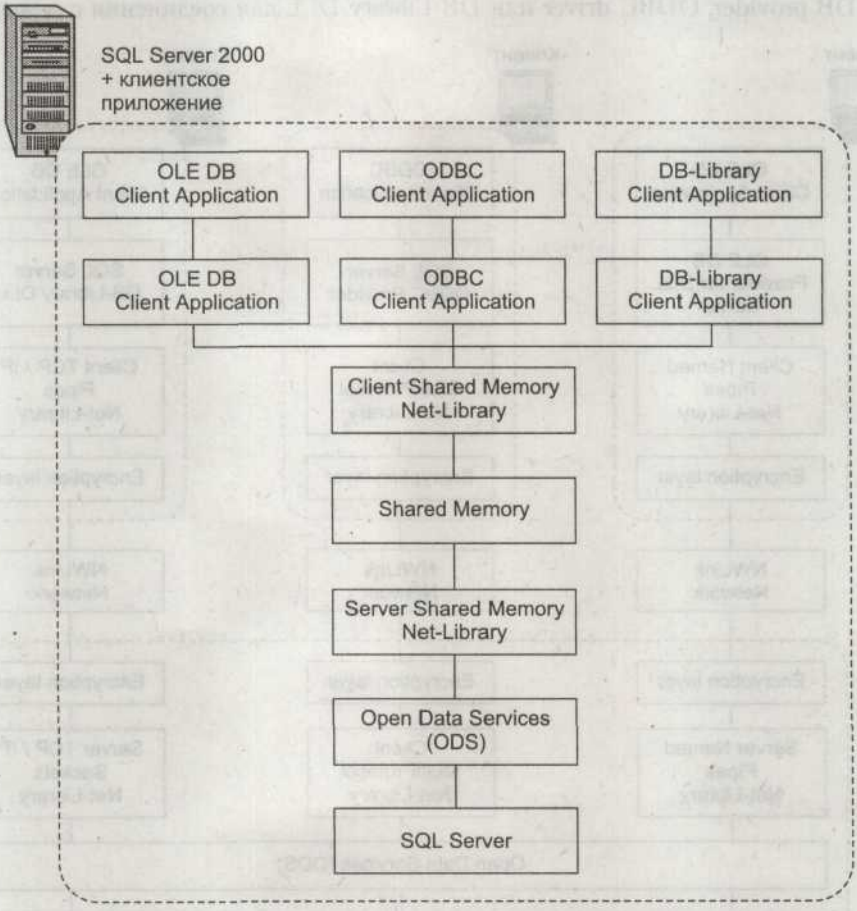


Рис. 9.12. Взаимодействие клиентского приложения и сервера в рамках одной машины

На рис. 9.13 приведена соответствующая схема. Также с сетевой библиотекой может использоваться специальный программный слой, предназначенный для шифрования пересылаемых по сети данных, — Encryption layer.

Шифрование на уровне Encryption layer осуществляется средствами SSL (Secure Sockets Layer) API. Ключ шифрования может занимать 40 или 128

бит в зависимости от версии Windows клиентского и серверного компьютеров. Активизация режима шифрования пересылаемых данных может существенно снизить производительность сетевой базы данных не только за счет существенных нагрузок на систему при шифровании и дешифровании пакетов, но и за счет необходимости постоянного обмена данными между клиентом и сервером. Подводя итог, можно выделить последовательность, с которой происходит установление соединения между сервером и клиентом:

1. Клиентское приложение осуществляет вызов OLE DB, ODBC, DB-Library, или внутренний SQL API. Это достигается при помощи вызовов методов OLE DB provider, ODBC driver или DB-Library DLL для соединения с сервером.

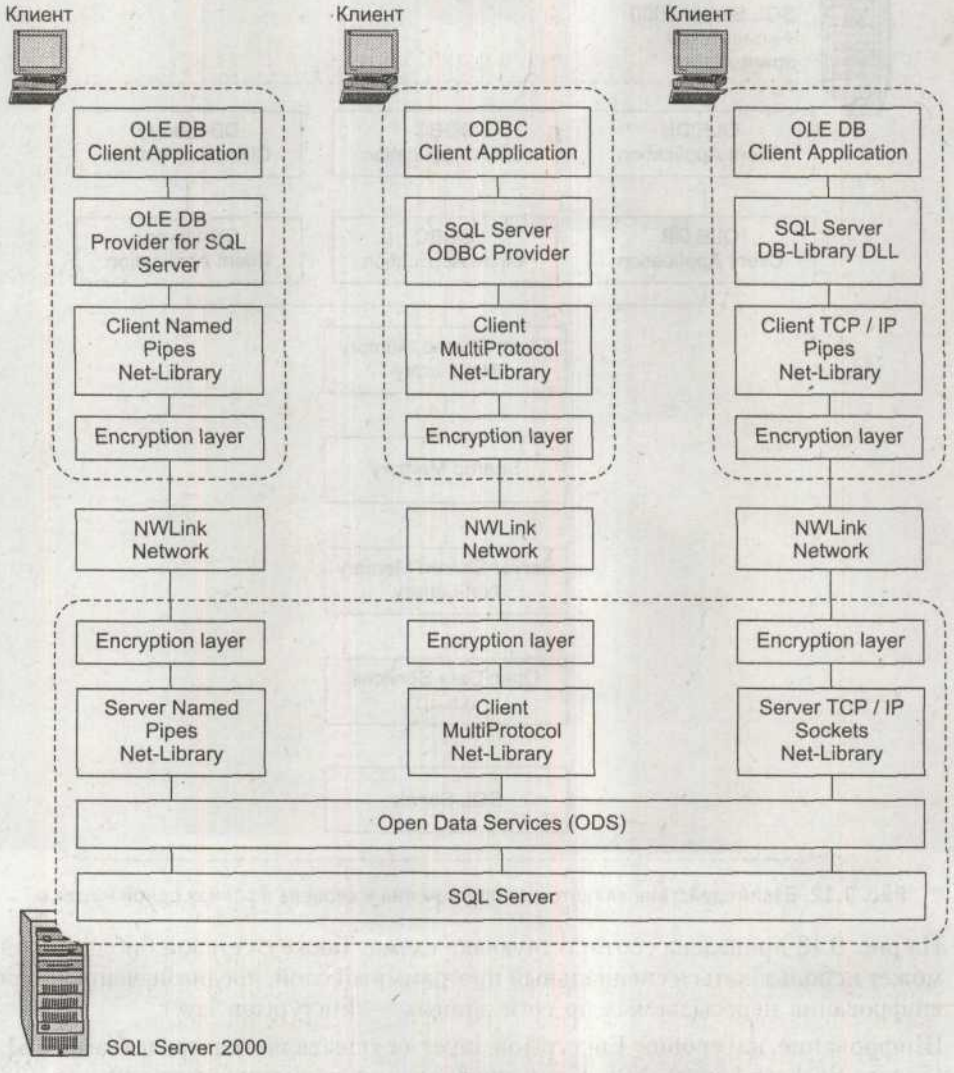


Рис. 9.13. Взаимодействие клиентского приложения и сервера с разных машин

2. OLE DB provider, ODBC driver или DB-Library DLL обращаются к клиентской сетевой библиотеке. Клиентская сетевая библиотека, в свою очередь, обращается к IPC API.
3. Клиентское приложение, используя методы IPC, передает запросы серверной сетевой библиотеке, а через нее — серверному модулю IPC. Если это локальный IPC, то вызовы передаются с использованием средств операционной системы, таких как разделяемая память (Shared Memory) и именованные каналы (local named pipes). Если это сетевой IPC, стек сетевых протоколов клиента взаимодействует со стеком сетевых проколов сервера.
4. Серверная сетевая библиотека передает запросы, поступившие от клиентского приложения серверу SQL Server 2000.

От сервера к клиенту данные передаются в обратном порядке.

Серверные сетевые библиотеки

Настройка сетевых библиотек на стороне сервера начинается с настройки соответствующих сетевых протоколов. После этого уже можно перейти к непосредственной настройке серверных сетевых библиотек. Для управления сетевыми библиотеками на стороне сервера используется утилита SQL Server Network Utility, вызвать которую можно, выполнив команду Svrnetcn. Окно утилиты показано на рис. 9.14.

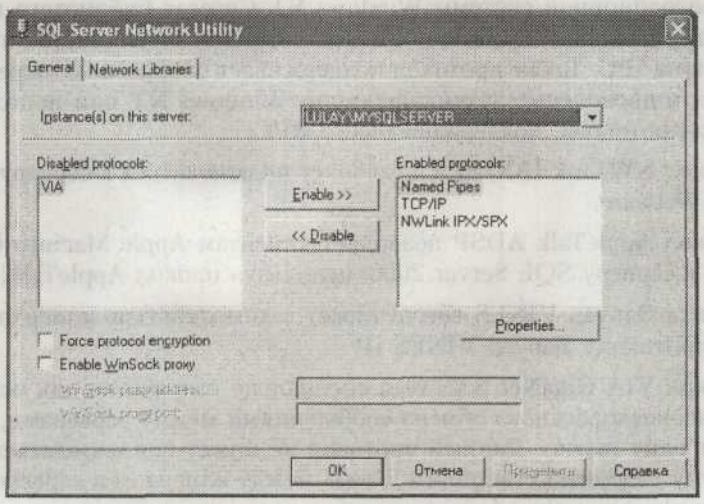


Рис. 9.14. Окно утилиты SQL Server Network Utility, вкладка General

Окно утилиты содержит две вкладки. На вкладке General предоставлены средства для конфигурирования сетевых библиотек на стороне сервера. На вкладке Network Libraries содержится список доступных сетевых библиотек.

Как было отмечено ранее, на одной машине может быть установлено несколько экземпляров SQL Server 2000, имеющих разные имена. Соответственно, каж-

дый экземпляр имеет свои собственные настройки. Выбрать экземпляр SQL Server 2000 можно из списка Instance(s) on this server. В списке Disabled protocols перечислены имена сетевых библиотек, не используемых данным экземпляром сервера, а в списке Enabled protocols указываются используемые протоколы. Добавить или убрать протокол из списка можно при помощи кнопок Enabled и Disabled. При конфигурировании сетевых библиотек на стороне сервера следует учитывать, что некоторые из них не могут работать с именованными экземплярами сервера. Подходящие протоколы перечислены в списке:

- Named Pipes;
- TCP/IP;
- NWLink IPS/SPX;
- Shared Memory.

А теперь следует рассмотреть протоколы подробнее. Впрочем, следует учитывать, что некоторые из них доступны только при использовании неименованного экземпляра SQL Server:

- Протокол TCP/IP использует сокеты TCP/IP. Функционирование осуществляется поверх механизма IPC. Данный протокол используется по умолчанию на всех версиях Windows. SQL Server 2000 обычно использует порт 1433.
- Протокол Multiprotocol использует службу вызова удаленных процедур RPC операционной системы Windows NT. Сетевая библиотека не требует конфигурирования и работает с использованием большинства протоколов механизма IPC. Также протокол поддерживает шифрование пересылаемых данных и проведение аутентификации Windows NT при использовании любого протокола, поддерживающего RPC.
- Протокол NWLink IPX/SPX позволяет подключаться к серверу клиентам Novell Netware.
- Протокол AppleTalk ADSP позволяет клиентам Apple Macintosh подключаться к серверу SQL Server 2000, используя прокол AppleTalk.
- Протокол Banyan VINES обеспечивает взаимодействие клиентов и сервера по протоколу Banyan VINES IP.
- Протокол VIA GigaNet SAN был специально разработан для осуществления высокоскоростного обмена сообщениями между серверами, включенными в одну группу. Данный протокол не может использоваться для связи между рабочими станциями. Связь между клиентом и сервером по данному протоколу может быть осуществлена только в случае взаимодействия систем, основанных на Windows NT Server, Advanced Server, Windows 2000 Server, Advanced Server и Data Center.
- Протокол Named Pipes позволяет серверу использовать именованные каналы и используется в Windows NT и Windows 2000 по умолчанию. Библиотека может работать поверх остальных. По умолчанию для SQL Server 2000 устанавливается канал \\.\pipe\sql\query. Если установлен именован-

ный экземпляр сервера, то перед именем канала указывается его имя: \\.\pipe\MSSQL$instancename\sql\query.

В табл. 9.1 показано, как между собой связаны сетевые библиотеки IPC API и сетевые протоколы.

Таблица 9.1. Связь между сетевыми библиотеками IPC API и сетевыми протоколами

Протокол, используемый сервером SQL Server 2000	Сетевая библиотека IPC API	Сетевой протокол, используемый IPC API
TCP/IP Sockets	Windows Socket 2	TCP/IP
Named Pipes	Windows Named Pipes	File system (local) TCP/IP NetBEUI NWLink
NWLink IPX/SPX	Windows Socket 2	NWLink
VIA GigaNet SAN	Virtual Interface (VIA) Architecture	Virtual Interface Architecture (VIA)
Multiprotocol	Windows RPC	File system (local) TCP/IP NetBEUI NWLink
AppleTalk	AppleTalk ADSP	AppleTalk
Banyan Vines	Banyan VINES SPP	Banyan VINES

Практически все библиотеки имеют параметры, определяющие различные настройки соединения. Для того чтобы получить доступ к настройкам, необходимо выбрать протокол и нажать на кнопку Properties. На рис. 9.15 приведено окно настройки параметров соединения для протокола TCP/IP.

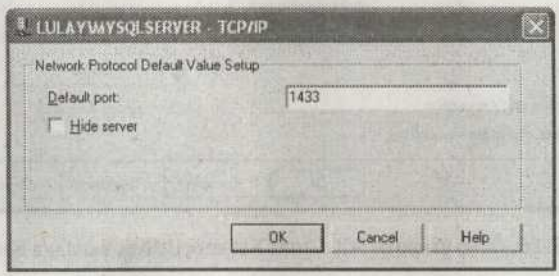


Рис. 9.15. Настройка параметров протокола TCP/IP на стороне сервера

Основным параметром сервера является порт, по которому будет производиться связь с сервером. Установив флажок Hide server, можно «спрятать» экземпляр сервера от сканирования по портам. Для обращения к серверу клиенты должны будут указывать имя сервера и используемый порт.

Также на вкладке General утилиты SQL Server Network Utility находятся флажки Force Protocol Encryption и Enable WinSock proxy. Установка флажка Force Protocol Encryption включает механизм шифрования данных, пересылаемых между клиентом и сервером с использованием протокола SSL. Установка флажка Enable WinSock proxy разрешает использование прокси-сервера для передачи данных. Прокси-сервер может быть использован для обеспечения доступа к серверу

SQL Server 2000, расположенному в локальной сети, из Интернета. Автоматически становятся доступными поля WinSock proxy address и WinSock proxy port, в которых необходимо указать адрес прокси-сервера и порт.

Клиентские сетевые библиотеки

Клиентские сетевые библиотеки используется на стороне клиентских приложений для соединения с сервером. В предыдущих главах для соединения с сервером использовалась трехзвенная архитектура, включающая в себя сервер приложения. В данной главе будет рассмотрен вариант использования стандартных средств коммуникации.

Чтобы соединение могло быть установлено, на стороне сервера и на стороне клиента должны быть зарегистрированы одни и те же протоколы. Для конфигурирования клиентских сетевых библиотек предназначена утилита SQL Client Network Utility, окно которой приведено на рис. 9.16. Вызвать данную утилиту можно при помощи команды cliconfg.

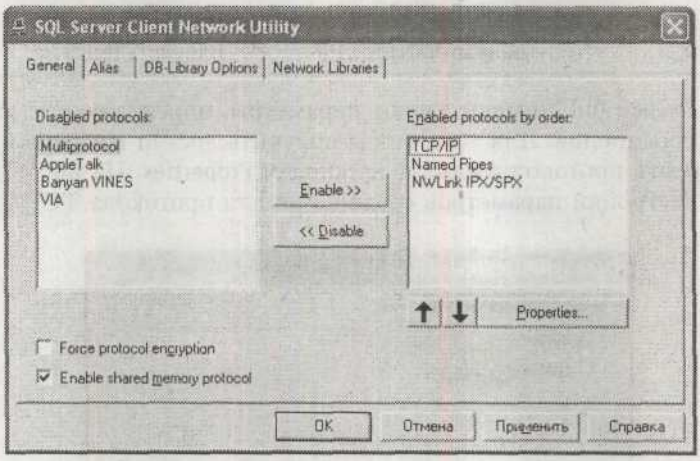


Рис. 9.16. Окно утилиты SQL Client Network Utility, вкладка General

В клиентской утилите есть два списка, в которых указываются используемые и неиспользуемые сетевые протоколы. В списке Disabled Protocols указаны неактивные протоколы. В списке Enabled Protocols by Order указываются протоколы, используемые клиентом для соединения с сервером. Если на клиентской стороне выбрано несколько протоколов, поддерживаемых сервером, связь будет вестись по тому протоколу, по которому будет первым установлено соединение. Первым будет произведена попытка соединения по протоколу, указанному верхним в списке.

Флажок Force Protocol Encryption включает режим шифрования данных, пересылаемых между клиентом и сервером. Шифрование данных будет производиться только в том случае, если оно включено и на сервере. Флажок Enable Shared Memory Protocol активирует режим использования разделяемой памяти

(Shared Memory) при обмене данными между клиентом и сервером, расположенными на одной машине.

Так как каждый экземпляр SQL Server 2000 может иметь свои собственные параметры сетевых библиотек, было бы весьма неудобно всякий раз перенастраивать параметры соединения при работе с нужным экземпляром. Выходом является использование *псевдонимов серверов*, представляющих собой учетные записи, содержащие информацию о подключении. На рис. 9.17 показана вкладка Alias клиентской программы конфигурирования сетевых библиотек, в которой можно конфигурировать псевдонимы серверов.

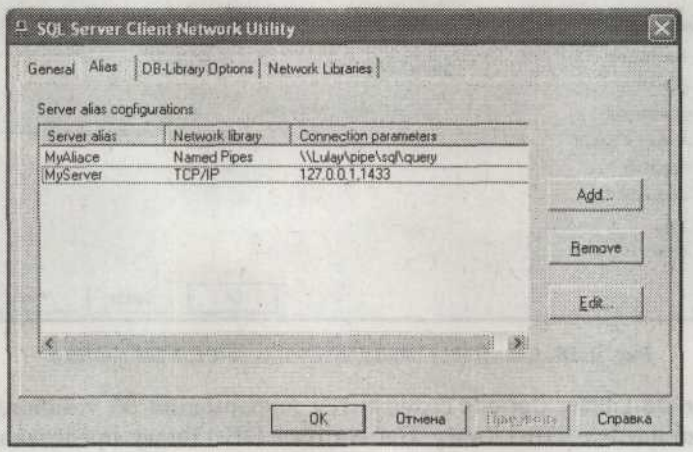


Рис. 9.17. Окно создания нового псевдонима сервера

Сервер может иметь несколько псевдонимов, каждый из которых может содержать различные варианты конфигурирования сетевых библиотек. Имя сервера и имя псевдонима никак не связаны между собой и могут отличаться друг от друга.

На вкладке Alias содержится список зарегистрированных на клиентской стороне псевдонимов с указанием типа сетевой библиотеки и параметров соединения. Для добавления, удаления и изменения параметров псевдонимов предназначены кнопки Add, Remove и Edit. В случае необходимости добавления нового псевдонима необходимо нажать кнопку Add, появится окно, показанное на рис. 9.18.

В поле Server alias необходимо указать уникальное имя псевдонима сервера. Затем следует выбрать сетевую библиотеку из списка Network libraries. В качестве примера можно указать протокол TCP/IP. После этого необходимо указать параметры соединения. В поле Server name вводится IP-адрес сервера. Для локального сервера используется значение 127.0.0.1. В поле Port number указывается порт, через который будет производиться взаимодействие между клиентом и сервером. В данном случае обычно используется порт 1433.

Если установить флажок Dynamically determine port, то клиентское приложение автоматически попытается определить порт, на который настроен сервер

в момент соединения. Использовать данную функцию обычно не рекомендуется, так как каждый раз будет тратиться время на определение порта. В том случае, если на сервере задействован режим Hide server, клиент просто не обнаружит его. Таким образом, для соединения с сервером через Microsoft OLE DB Provider for SQL Server необходимо будет указывать не имя сервера, а имя псевдонима сервера.

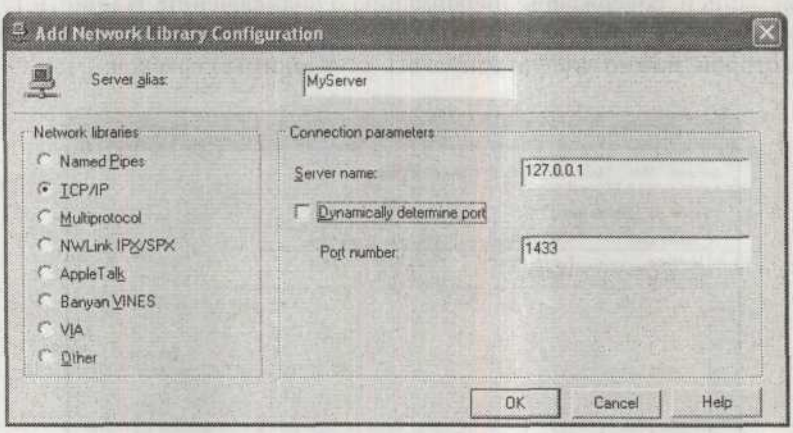


Рис. 9.18. Диалоговое окно добавления сетевой библиотеки

На вкладке DB-Library Options содержится информация об установленной на клиенте библиотеке DB-Library. Библиотека DB-Library представляет собой один из механизмов работы с сервером. Окно, помимо информации о библиотеке, имеет два флажка, позволяющих управлять параметрами соединения. Использование флажка Automatic ANSI to OEM conversion обеспечивает автоматическое преобразование типа данных из формата ANSI в формат OEM и обратно при пересылке между сервером и клиентом. Флажок Use international settings указывает на необходимость использования национальных установок, принятых в операционной системе. В ином случае будут использоваться настройки, установленные в библиотеке.

На вкладке Network Libraries, окно которой приведено на рис. 9.19, содержится список доступных сетевых клиентских библиотек.

При помощи созданного псевдонима нужно соединиться с сервером. После открытия нового проекта на основной форме нужно расположить компонент TADOConnection. После этого необходимо активировать окно настройки соединения с сервером (рис. 9.20).

В поле ввода имени сервера нужно ввести созданный псевдоним, например MyServer. Затем потребуется выбрать аутентификацию средствами Windows NT, в поле выбора базы данных указать Northwind и проверить соединение.

На форме осталось разместить компоненты TADOTable, TDataSource и TDBGrid, которые затем нужно связать друг с другом. В компоненте TADOTable потребуется выбрать ту или иную таблицу. Теперь можно проверить работу приложения, активировав компоненты соединения и компонент таблицы.

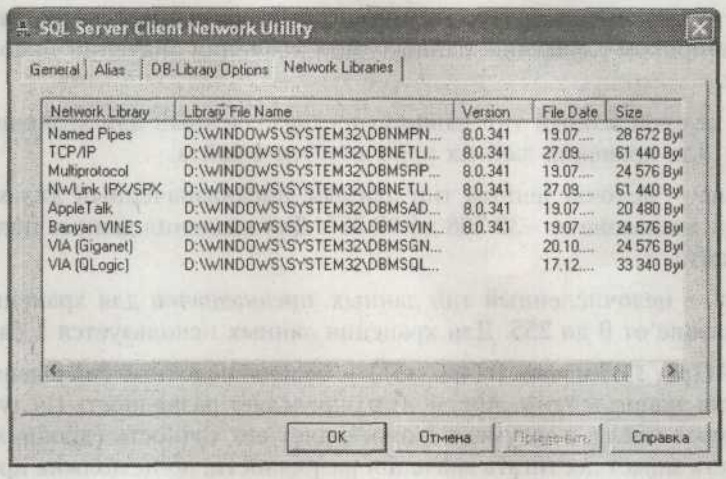


Рис. 9.19. Окно утилиты SQL Client Network Utility, вкладка Network Libraries

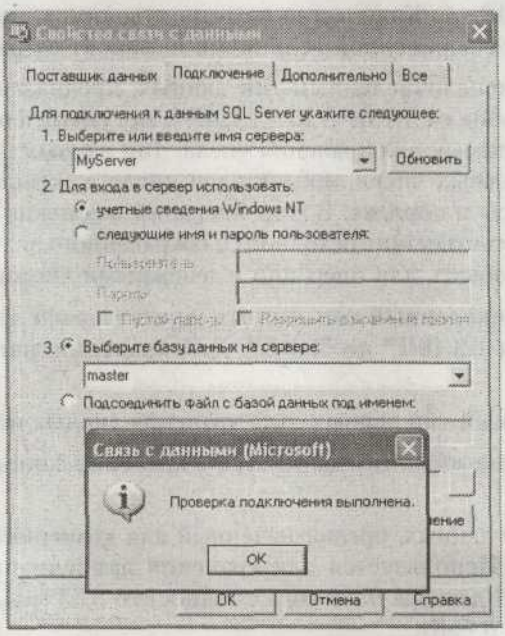


Рис. 9.20. Окно установки связи с сервером

Типы данных SQL Server 2000

SQL Server 2000 имеет 27 типов данных, часть которых эквивалентна типам данных, определенным в стандарте SQL-92. Следует рассмотреть каждый тип данных достаточно детально:

- `bigint` — целочисленный тип данных, предназначенный для хранения чисел в широком диапазоне данных. Для хранения значений используется 8 байт.
- `int` — целочисленный тип данных, предназначенный для хранения целых чисел. Для хранения данных используется 4 байта.
- `smallint` — целочисленный тип данных, предназначенный для хранения чисел в диапазоне от –32 768 до 32 767. Для хранения данных используется 2 байта.
- `tinyint` — целочисленный тип данных, предназначен для хранения чисел в диапазоне от 0 до 255. Для хранения данных используется 1 байт
- `decimal[(p[, s])]` и `numeric[(p[, s])]` — нецелочисленные типы данных. Эти два типа эквивалентны. Аргумент `p` определяет разрядность (целую часть) хранимого числа, а аргумент `s` определяет его точность (дробную часть). Точность может достигать значения разрядности, но не должна превышать ее. По умолчанию точность устанавливается равной нулю, то есть число хранится как целое. Тип данных может хранить число в диапазоне от $-10^{38}+1$ до $10^{38}-1$. Количество памяти, отводимое для хранения одного числа, является переменной величиной и зависит от разрядности.
- `float[(n)]` — нецелочисленный тип данных, предназначен для хранения чисел в диапазоне от $1{,}79E^{308}$ до $-1{,}79E^{308}$. Аргумент `n` определяет количество бит, отводимых для хранения числа. Тип данных производит аппроксимацию хранимых чисел, представляя число в экспоненциальном виде, в виде мантиссы и порядка. В ходе операций возможно округление числа, что вызовет несоотвествие исходного и сохраненного значений. Данный тип нельзя использовать для операций с денежными средствами.
- `real` — нецелочисленный тип данных, предназначен для хранения чисел в диапазоне от $-3{,}40E^{38}$ до $3{,}40E^{38}$. Для хранения данных используется 4 байта.
- `money` — денежный тип данных. Для хранения данных используется 8 байт.
- `smallmoney` — денежный тип данных. Для хранения данных требуется 4 байта.
- `datetime` — тип данных, предназначенный для хранения информации о дате и времени. Используется для хранения дат в диапазоне от 1 января 1753 года до 31 декабря 9999 года с точностью 3,33 мс. Для хранения данных требуется 8 байт.
- `smalldatetime` — аналогичен типу данных `datetime`, но обладает меньшей точностью. Тип данных предназначен для хранения дат в диапазоне от 1 января 1900 года до 6 июня 2079 года с точностью до одной минуты.
- `binary[(n)]` — тип данных, предназначенный для хранения двоичных данных в сегментах определенной длины. В параметре `n` указывается размер хранимого значения. Для представления одного символа используется 1 байт. Может принимать значение размером от 1 до 8000 знаков.

- varbinary[(n)] — тип данных переменной длины, аналогичен binary, за тем исключением, что сохраняется не целый сегмент, а только данные. Работа с этим типом требует от системы несколько большего количества ресурсов, чем работа с типом binary.
- image — тип данных переменной длины, предназначен для хранения двоичных данных. Значения данного типа хранятся на отдельных страницах.
- char[(n)] — строковый тип данных фиксированной длины. Для хранения каждого значения в формате ASCII будет выделен сегмент определенной длины, указанной в параметре n. Тип данных может хранить от 1 до 8000 байт.
- varchar[(n)] — строковый тип данных переменной длины, предназначен для хранения данных в формате ASCII. В целом аналогичен типу char, за исключением того, что хранит данные не в сегментах фиксированного размера, а в равных реальному размеру содержащихся в них данных.
- nchar(n) — строковый тип данных фиксированной длины, предназначенный для хранения данных в формате UNICODE. Тип данных может хранить максимум 4000 символов.
- nvarchar(n) — строковый тип данных переменной длины, предназначенный для хранения данных в формате UNICODE.
- text — тип данных ASCII переменной длины, предназначенный для хранения блоков текстовых данных.
- ntext — аналогичен типу text, но предназначен для хранения данных в формате UNICODE.
- bit — тип данных, принимающий либо нулевое, либо единичное значение.
- cursor — тип данных, используемый в переменных или в хранимых процедурах в качестве выходного параметра. Тип данных содержит указатель на курсор. При создании переменной данного типа ей присваивается нулевое значение.
- sql_variant — тип данных, предназначенный для хранения значений других типов данных. Тип данных может использоваться в столбцах таблиц, переменных, параметрах и возвращаемых значениях пользовательских функций. Значение типа данных может иметь длину до 8016 байт.
- table — специальный тип данных, представляющий собой виртуальный набор данных и предназначенный для использования в качестве переменной. Работа с такой переменной осуществляется, как с обычной таблицей базы данных.
- uniqueidentifier — тип предназначен для хранения уникального глобального идентификатора (GUID). Идентификатор представляет собой шестнадцатеричное число, имеет размер 16 байт и записывается в формате xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. Для генерирования значения идентификатора в SQL Server 2000 предусмотрена функция NEWID.
- timestamp — тип данных, предназначен для автоматического генерирования значения, уникального в пределах базы данных.

Для преобразования значений типов данных из одного в другой могут быть использованы функции CAST(expression AS data_type) и CONVERT(data_type[(length)] expression [style]).

В параметре expression указываются какое-либо выражение или величина, а в параметре data_type указывается тип данных, в который она конвертируется. Применение этих функций иллюстрирует следующий фрагмент кода:

```
SELECT CAST(CAST(0x41 AS nvarchar) AS varbinary);
SELECT CAST('abc' AS varchar(5)) COLLATE French_CS_AS;
```

Возможности обеих функций практически равноценны, однако функция CONVERT предоставляет бóльшие возможности по форматированию значений типа datetime и smalldatetime. Параметр length, который позволяет определить величину параметра, является опциональным. Параметр style позволяет явно задать формат, в котором будут представлены данные типа даты и времени, текстовые и строковые.

Соединение с сервером

Основной утилитой, с помощью которой осуществляется управление сервером, является утилита SQL Server Enterprise Manager. Окно утилиты показано на рис. 9.21.

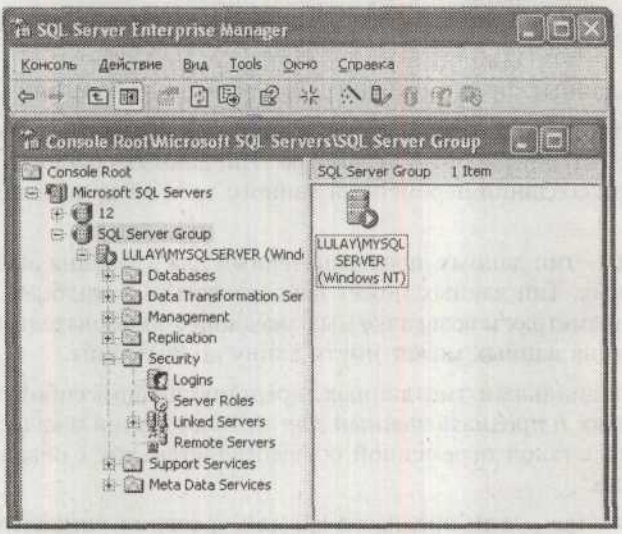


Рис. 9.21. Окно утилиты SQL Server Enterprise Manager

В левой части окна находится список SQL Server Group, в котором можно будет получить доступ к установленному экземпляру SQL Server 2000. Соединиться с сервером можно, дважды щелкнув на названии экземпляра либо выбрав в его контекстном меню команду Connect. В раскрывшемся списке доступны

такие объекты, как Databases — перечень баз данных, установленных в системе, Management — группа объектов, предназначенных для управления сервером, Replication — объекты, осуществляющие репликацию баз данных и подписку на публикуемые базы данных, и Security — группа объектов, определяющих права доступа к системе.

Для того чтобы установить соединение с удаленным сервером по какому-либо протоколу или получить доступ к данному с отличными от администраторских правами, необходимо создать новую регистрацию сервера. Сервер может быть зарегистрирован в текущей группе либо в новой группе. Для того чтобы создать группу, необходимо выбрать пункт главного меню Action ▸ New SQL Server Group. В появившемся окне будет предложено создать корневую группу (Top level group) либо подгруппу какой-либо созданной ранее (Sub-group of:). В качестве примера можно создать подгруппу с именем 12. В контекстном меню созданной группы нужно выбрать пункт New SQL Server Registration. В появившемся диалоговом окне потребуется установить флажок From now on, I want to perform this task without using a wizard и нажать кнопку Next. Появится окно, показанное на рис. 9.22.

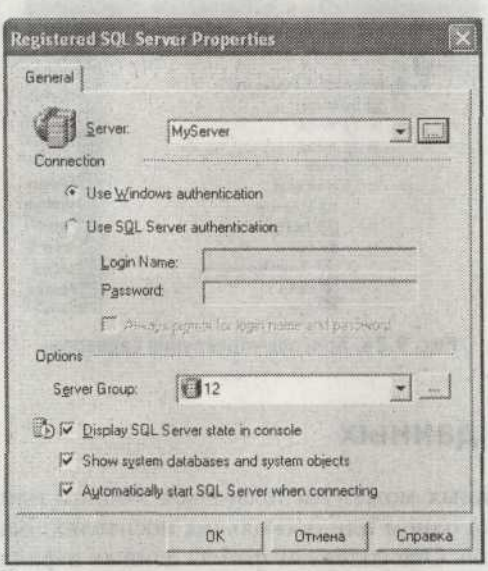


Рис. 9.22. Окно регистрации сервера

В поле Server необходимо указать имя экземпляра сервера, если он располагается на данной машине, либо имя псевдонима сервера, если сервер располагается на удаленной машине. Будем полагать, что сервер располагается на удаленной машине. Поэтому можно использовать ранее созданный псевдоним MyServer, указав его в поле Server. Для ввода имени сервера можно использовать диалоговое окно, показанное на рис. 9.23.

Чтобы активировать это диалоговое окно, следует нажать на кнопку, расположенную справа от поля ввода Server. Так как при установке сервера был

выбран режим аутентификации средствами операционной системы, то нужно выбрать режим аутентификации Use Windows autentification. В списке Server Group остается выбрать группу 12 и нажать кнопку OK. Будет произведено соединение с сервером. Окно консоли показано на рис. 9.24.

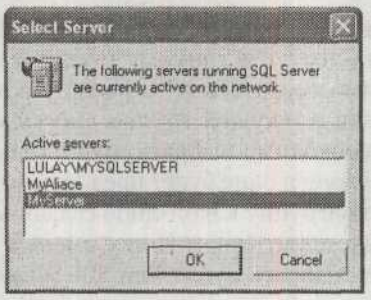


Рис. 9.23. Диалог выбора сервера

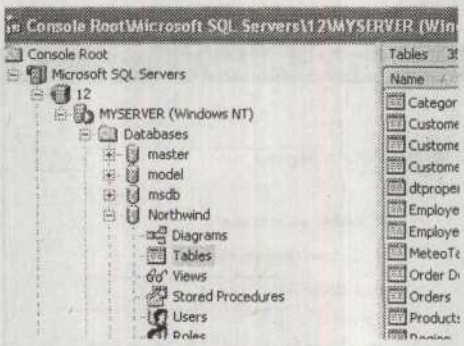


Рис. 9.24. Консоль управления сервером

Создание базы данных

Физически база данных может располагаться в одном или нескольких файлах, размещенных на одном или нескольких носителях. База данных состоит из двух типов файлов. Она содержит файлы данных и файлы журналов транзакций. Файлы данных бывают первичные с расширением .mdf и вторичные с расширением .ndf. По умолчанию база данных имеет только один файл данных — первичный. Использование вторичного файла (или файлов) производится в тех случаях, когда возникает необходимость перемещения базы данных на другой носитель.

Как было отмечено ранее, любая операция в базе данных выполняется в рамках транзакции, а перед выполнением транзакция фиксируется в специальном журнале — файле журнала транзакций. Данный файл имеет расширение .ldf и по умолчанию имеет то же имя, что и первичный файл данных. Файлов журналов транзакций, как и файлов данных, может быть несколько.

Файлы базы данных могут быть объединены в группы. Любой группе файлов можно установить доступ «только для чтения», то есть запретить возможность изменения данных, входящих в таблицу.

Базу данных можно создать либо с помощью утилиты Enterprise Manager, либо при помощи команды CREATE DATABASE. Эта команда имеет следующий синтаксис:

```
CREATE DATABASE database_name
[ ON
[ < filespec > [ ....n ] ]
[ , < filegroup > [ ....n ] ]
]
[ LOG ON { < filespec > [ ....n ] } ]
[ COLLATE collation_name ]
[ FOR LOAD | FOR ATTACH ]
< filespec > ::=
[ PRIMARY ]
( [ NAME = logical_file_name , ]
FILENAME = 'os_file_name'
[ , SIZE = size ]
[ , MAXSIZE = { max_size | UNLIMITED } ]
[ , FILEGROWTH = growth_increment ] ) [ ....n ]
< filegroup > ::=
FILEGROUP filegroup_name < filespec > [ ....n ]
```

В параметре database_name указывается имя базы данных. Этот параметр является обязательным. Остальные нужно указывать по желанию разработчика. Например, следующая команда создаст базу данных с именем TestDB:

```
CREATE DATABASE TestDB
```

В результате выполнения команды в папке \Program Files\Microsoft SQL Server\MSSQL$MYSQLSERVER\Data появятся файлы TestDB.mdf и TestDB_log.LDF. По умолчанию файлу данных и файлу журнала транзакций были присвоены имена, в которые входит имя базы данных. Все параметры базы данных были скопированы из шаблонной базы данных model.

Для выполнения запроса можно воспользоваться утилитой SQL Query Analyzer (рис. 9.25). Утилиту можно запустить, вызвав пункт главного меню Tolls ▸ SQL Query Analayzer либо запустив файл isqlw.exe, расположенный в каталоге \Program Files\Microsoft SQL Server\80\Tools\Binn. В случае прямого запуска файла необходимо будет выбрать имя экземпляра сервера и указать пароль, если используется расширенный режим аутентификации.

В параметрах <filespec> и <filegroup> указываются файлы данных и группы файлов, в которые входят эти файлы. После ключевого слова LOG ON в параметре <filespec> указываются файлы журналов транзакций. Это иллюстрирует следующий запрос, который можно выполнить в утилите SQL Query Analayzer:

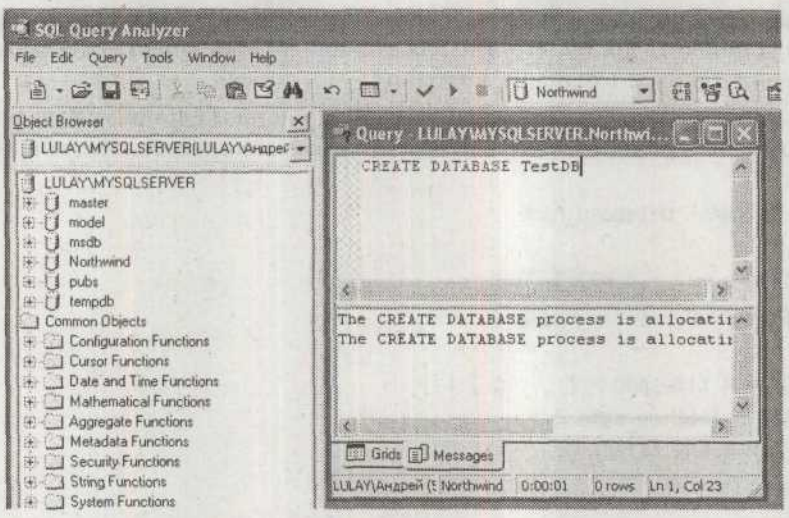


Рис. 9.25. Окно утилиты SQL Query Analayzer

```
USE master
GO
CREATE DATABASE Sales
ON
( NAME = Sales_dat,
FILENAME = 'D:\ MSSQLDB\saledat.mdf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
FILENAME = 'D:\MSSQLDB\salelog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB )
GO
```

Команда `GO` указывает парсеру на необходимость немедленного выполнения блока инструкций. В необязательном параметре `NAME` указывается логическое имя файла. Оно должно быть уникальным в пределах базы данных. В параметре `FILENAME` указывается физическое имя файла базы данных. Параметр `SIZE` определяет начальный размер файла базы данных. По умолчанию размер указывается в мегабайтах. Размер файла базы данных может быть указан только для явно определяемого файла, но не для создаваемого по умолчанию. Параметр `MAXSIZE` позволяет определить максимальный размер файла базы данных. По умолчанию максимальный размер файла базы данных ограничен количеством свободного места, доступного на носителе, что соответствует

значению UNLIMITED. Параметр FILEGROWTH позволяет определить приращение размера файла базы данных.

Приращение представляет собой процесс увеличения размера файла при достижении заносимых в него данных конечного значения свободного места. Увеличение размера файла может производиться в процентах от исходного размера файла либо на определенную величину. По умолчанию приращение устанавливается в 10% от текущего размера файла базы данных. Следующий пример создает базу данных, расположенную в нескольких файлах и использующую несколько файлов журналов транзакций:

```
USE master
GO
CREATE DATABASE Archive
ON
PRIMARY ( NAME = Arch1,
FILENAME = 'D:\MSSQLDB\archdat1.mdf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20),
( NAME = Arch2,
FILENAME = 'D:\MSSQLDB\archdat2.ndf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20),
( NAME = Arch3,
FILENAME = 'D:\MSSQLDB\archdat3.ndf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20)
LOG ON
( NAME = Archlog1,
FILENAME = 'D:\MSSQLDB\archlog1.ldf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20),
( NAME = Archlog2,
FILENAME = 'D:\MSSQLDB\archlog2.ldf',
SIZE = 100MB,
MAXSIZE = 200,
FILEGROWTH = 20)
GO
```

Стоит обратить внимание на то, как объявлены вторичные файлы данных и файлы журналов транзакций. Для этого было использовано простое перечисление.

Теперь нужно создать базу данных при помощи утилиты SQL Server Enterprise Manager. Для этого нужно выполнить команду контекстного меню Databases ▸ New Database. Она активирует диалоговое окно, показанное на рис. 9.26.

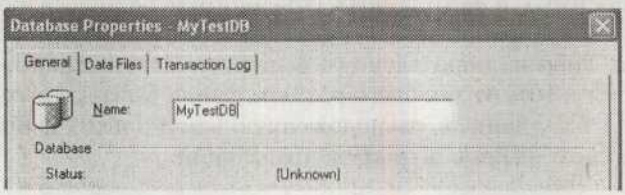


Рис. 9.26. Окно свойств базы данных, вкладка General

В поле Name необходимо указать имя создаваемой базы данных. В поле Collation name указывается имя порядка сортировки символов, используемого по умолчанию. Также на этой вкладке доступна иная информация о базе данных.

На вкладках Data Files и Transaction Log можно изменить присваиваемые по умолчанию имена файлов данных и журналов транзакций, а также добавить новые файлы, определить группы файлов и многое другое. На рис. 9.27 приведено окно вкладки Data Files.

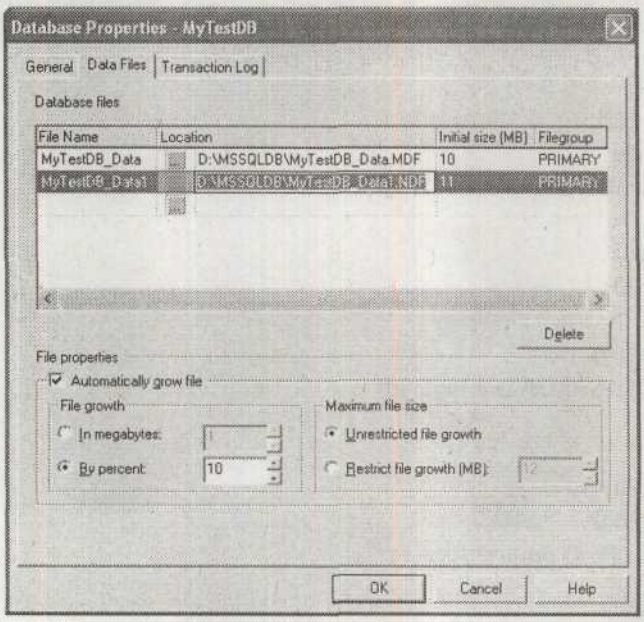


Рис. 9.27. Окно свойств базы данных, вкладка Data Files

В списке Database Files перечисляются имена файлов данных. В поле File Name указывается имя файла данных, в поле Location — путь к файлу данных, в поле Initial size (MB) — начальный размер файла данных, а в поле Filegroup — группа, в которую входит данный файл.

В поле File properties доступны настройки, позволяющие определить размер файла базы данных. Флажок Automatically grow file, установленный по умолчанию, включает режим автоматического увеличения размера файла данных по мере возникновения необходимости. Если режим не включен, системному администратору придется вручную увеличивать размер файла.

В поле File growth доступны два переключателя, позволяющих определить, каким образом будет увеличиваться размер файла данных. В поле In megabytes можно задать жесткую величину приращения файла, а в поле By percent задать увеличение в процентах от текущего размера.

В поле Maximum file size располагаются переключатели, позволяющие определить максимальный размер файла базы данных. Неограниченный размер файла задается при помощи переключателя Unrestricted file growth. Если необходимо четко ограничить размер файла, стоит указать его в поле Restricted file growth.

Аналогичный вид имеет вкладка Transaction Log. В ней также имеется возможность создания нескольких файлов журналов транзакций и определения их размера.

Работа с группами файлов

По умолчанию файлы базы данных и входящие в нее таблицы включаются в первичную группу. Также в первичную группу файлов включаются системные таблицы. В качестве примера можно создать базу данных Test, файлы которой входят в первичную группу Primary:

```
CREATE DATABASE Test ON PRIMARY
(NAME='Test',
FILENAME ='D:\MSSQLDB\Test.mdf')
GO
```

Можно добавить в созданную базу данных собственную группу файлов — MyGroup. Для этого следует воспользоваться командой ALTER DATABASE:

```
ALTER DATABASE Test
ADD FILEGROUP MyGroup
GO
```

Эта команда добавляет в базу данных группу файлов MyGroup. Теперь в созданную группу надо внести файл данных:

```
ALTER DATABASE Test
ADD FILE(NAME='Test1',
FILENAME ='D:\MSSQLDB\Test1.mdf')
TO FILEGROUP MyGroup
GO
```

Следующий код создает базу данных, включающую несколько групп файлов:

```
CREATE DATABASE Sales
ON PRIMARY
( NAME = SPri1_dat,
```

```
FILENAME = 'D:\MSSQLDB\SPri1dat.mdf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
( NAME = SPri2_dat,
FILENAME = 'D:\MSSQLDB\SPri2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
FILENAME = 'D:\MSSQLDB\SG1Fildt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
FILENAME = 'D:\MSSQLDB\SG2Fildt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
FILENAME = 'D:\MSSQLDB\SG2Fi2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
FILENAME = 'D:\MSSQLDB\salelog.ldf',
SIZE = 5MB,
MAXSIZE = 25MB,
FILEGROWTH = 5MB )
GO
```

Чтобы установить для группы SalesGroup1 режим «только для чтения», следует воспользоваться следующей командой:

```
ALTER DATABASE Sales
MODIFY FILEGROUP SalesGroup2 READONLY
GO
```

Нужно обратить внимание на то, что группы создаются вместе с ссзданием базы данных. Те же самые действия можно выполнить при помощи утилиты SQL Server Enterprise Manager. Выполнение команды Properties контекстного меню базы данных активирует соответствующее диалоговое окно свойств базы дан-

ных, показанное на рис. 9.28. Для работы с файлами следует открыть окно свойств ранее созданной базы данных Sales и перейти на вкладку Filegroups.

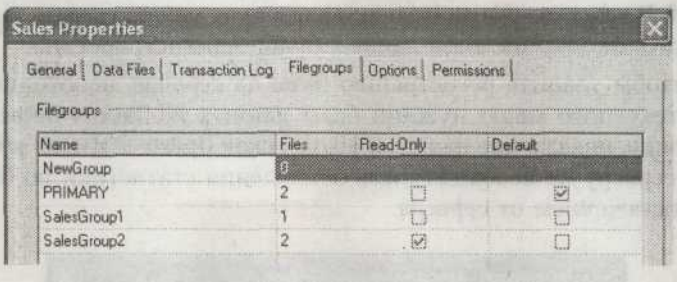


Рис. 9.28. Окно свойств базы данных, вкладка Filegroups

На вкладке Filegroups располагается список групп файлов. В поле Read-Only можно установить режим «только для чтения», активировав соответствующий флажок. Также можно установить группу файлов, используемую по умолчанию, при помощи флажка в поле Default.

Регистрация базы данных

Регистрация существующей базы на сервере является не такой уж и редкой операцией. Проще всего ее выполнить, используя утилиту SQL Server Enterprise Manager. После выполнения команды меню All Tasks ▸ Attach Database будет отображено окно, показанное на рис. 9.29.

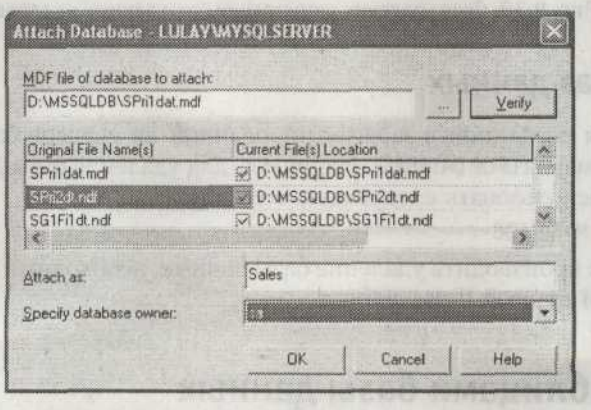


Рис. 9.29. Регистрация базы данных на сервере

В поле MDF file of database to attach необходимо указать имя первичного файла данных и путь до него. Выполнить данную операцию можно вручную либо с помощью диалогового окна, активируемого расположенной рядом кнопкой. В поле Attach as указывается логическое имя базы данных, то есть имя, под которым она будет доступна в системе. Поле Specify database owner позволяет определить владельца базы данных.

Перед тем как зарегистрировать существующую базу данных, необходимо ее отключить. В качестве примера можно выбрать базу данных. После этого можно указать путь к базе данных Sales, выбрав в качестве первичного файла SPri1dat.mdf. В качестве учетной записи владельца можно указать sa, после чего нужно нажать кнопку OK. База данных будет зарегистрирована.

Для того чтобы удалить регистрацию базы на сервере, необходимо вызвать пункт контекстного меню нужной базы данных All Tasks ▸ Detach Database. Появится окно, показанное на рис. 9.30. Флажок Update statistics prior to detach указывает серверу на необходимость обновления статистики по базе данных перед ее отключением от сервера.

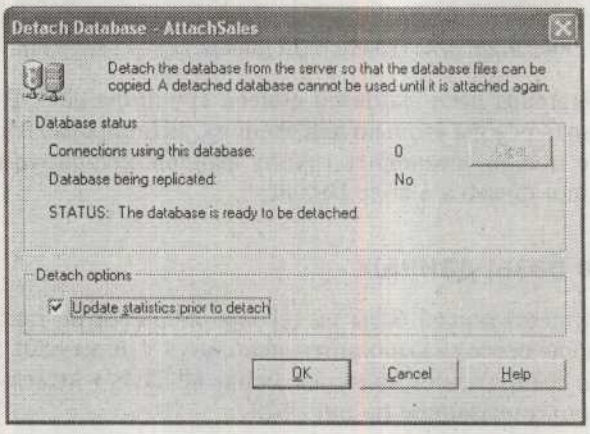


Рис. 9.30. Окно удаления регистрации базы данных на сервере

Удаление баз данных

Для удаления созданной и зарегистрированной на сервере базы данных используется команда DROP DATABASE. Например, для удаления базы данных MyTestDB необходимо использовать следующую команду:

```
DROP DATABASE MyTestDB
```

Перед тем как производить удаление базы данных, необходимо отключить всех пользователей от этой базы данных.

Работа с таблицами базы данных

Помимо обычных таблиц, SQL Server 2000 предоставляет возможность работать с временными таблицами, располагающимися в памяти операционной системы. Временные таблицы могут быть локальными и глобальными.

Локальные временные таблицы перед своим именем имеют префикс #Table_name, а глобальные — ##Table_name. Локальные временные таблицы могут использоваться только создавшими их владельцами.

Каждому экземпляру временной таблицы помимо символьного имени (которое может достигать 116 символов) присваивается уникальный номер, Таким образом, имя временной таблицы складывается из уникального номера, сгенерированного сервером, и символьного имени.

Информация о временных таблицах, созданных на сервере, хранится в таблице sysobjects, в базе данных tempdb. Глобальные таблицы, как и глобальные переменные, доступны всем пользователям системы. Локальные таблицы уничтожаются сразу после того, как их освобождает пользователь. Глобальные таблицы находятся в памяти до тех пор, пока с ними работает хотя бы один объект или пользователь. Работа с временными таблицами не отличается от работы с физическими таблицами, за исключением того, что временные таблицы не могут использовать внешние ключи.

Создание, изменение и удаление таблиц баз данных

Создавать таблицы можно как с помощью команд Transact-SQL, так и с помощью утилиты Enterprise Manager. Команда создания таблицы с помощью Transact-SQL имеет следующий синтаксис:

```
CREATE TABLE
[ database_name.[ owner ] . | owner. ] table_name
( { < column_definition >
| column_name AS computed_column_expression
| < table_constraint > ::= [ CONSTRAINT constraint_name ] }
| [ { PRIMARY KEY | UNIQUE } [ ....n ]
)
[ ON { filegroup | DEFAULT } ]
[ TEXTIMAGE_ON { filegroup | DEFAULT } ]
< column_definition > ::= { column_name data_type }
[ COLLATE < collation_name > ]
[ [ DEFAULT constant_expression ]
| [ IDENTITY [ ( seed , increment ) [ NOT FOR REPLICATION ] ] ]
]
[ ROWGUIDCOL]
[ < column_constraint > ] [ ...n ]
< column_constraint > ::= [ CONSTRAINT constraint_name ]
{ [ NULL | NOT NULL ]
| [ { PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
[ WITH FILLFACTOR = fillfactor ]
[ON {filegroup | DEFAULT} ] ]
]
| [ [ FOREIGN KEY ]
REFERENCES ref_table [ ( ref_column ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
```

```
[ ON UPDATE { CASCADE | NO ACTION } ]
[ NOT FOR REPLICATION ]
]
| CHECK [ NOT FOR REPLICATION ]
( logical_expression )
}
< table_constraint > ::= [ CONSTRAINT constraint_name ]
{ [ { PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
{ ( column [ ASC | DESC ] [ ....n ] ) }
[ WITH FILLFACTOR = fillfactor ]
[ ON { filegroup | DEFAULT } ]
]
| FOREIGN KEY
[ ( column [ ....n ] ) ]
REFERENCES ref_table [ ( ref_column [ ....n ] ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
[ NOT FOR REPLICATION ]
| CHECK [ NOT FOR REPLICATION ]
( search_conditions )
}
```

Выражение database_name.[owner] . | owner.] table_name указывает в параметре database_name имя базы данных, для которой создается таблица, в параметре owner содержит имя владельца базы данных, а в параметре table_name — имя создаваемой таблицы.

Параметр <column_definition> содержит описания создаваемых столбцов. Параметр column_name содержит имя столбца, после которого указывается его тип. Параметры NULL и NOT NULL определяют, обязательно ли поле должно содержать значение.

В параметре <table_constraint> содержится список ограничений, налагаемых на таблицу. Параметры PRIMARY KEY и UNIQUE определяют вхождение поля в первичный ключ и уникальность значений полей. Ключевые слова CLUSTERED | NONCLUSTERED указывают, кластерный или некластерный индекс будет создаваться по уникальным полям или полям, входящим в первичный ключ.

Параметр [WITH FILLFACTOR = fillfactor] определяет степень заполнения значениями страниц индекса. По умолчанию используется нулевое значение, то есть принимается уровень заполнения, используемый в системе. Слишком низкий уровень заполнения вызовет значительную фрагментацию индекса и потребует много места для хранения его значений. Высокий уровень индекса (на уровне 80%) является более предпочтительным, так как обеспечивает допустимый объем свободного места на странице и не потребует частого разбиения сегментов данных по страницам.

Выражение FOREIGN KEY...REFERENCES определяет ограничение ссылочной целостности. Внешний ключ может быть связан только с теми столбцами, которые входят в первичный ключ или являются уникальными, либо с теми, которые входят в уникальный индекс.

Выражение ON DELETE {CASCADE | NO ACTION} определяет, какое действие должно быть предпринято при удалении родительской записи. Выражение ON UPDATE {CASCADE | NO ACTION} определяет, какое действие должно быть предпринято при изменении ключа родительской записи.

Оператор IDENTITY указывает на то, что столбец является идентифицирующим, то есть все значения в нем будут уникальными. В момент добавления новой записи SQL Server 2000 подставляет в поле уникальное значение счетчика, выбирая максимальное значение из всех находящихся в данном поле и увеличивая его на единицу. Данное свойство может быть присвоено полям типа tinyint, smallint, int, bigint, decimal(p,0) или numeric(p,0).

Следующий запрос создает простую таблицу MyTable, включающую в себя три поля:

```
CREATE TABLE MyTable
(
SomeInteger SMALLINT,
SomeValue VARCHAR(5),
SomeText TEXT
)
```

В новом примере создается таблица MyTable1, у которой значения поля SomeValue уникальны и не могут содержать значения типа NULL:

```
CREATE TABLE MyTable1
(
SomeInteger SMALLINT UNIQUE NOT NULL,
)
```

Очередной запрос создает таблицу MyTable2, в которую будет входить уникальный кластерный индекс по полю SomeInteger, и поле SomeValue, в которое будут подставляться значения по умолчанию:

```
CREATE TABLE MyTable2
(
SomeInteger SMALLINT UNIQUE CLUSTERED,
SomeValue VARCHAR(6) DEFAULT 'DefVal'
)
```

Ограничение CHECK используется для осуществления проверки значений добавляемых в поле на соответствие определенному диапазону или иному условию. Например, если необходимо создать таблицу возрастов по определенной категории, в которую могут входить граждане в возрасте от 20 до 30 лет, следует обращаться именно к этому ограничению. Эту проверку можно реализовать в таблице MyTable3:

```
CREATE TABLE MyTable3
(
AGE SMALLINT UNIQUE CLUSTERED CHECK (AGE BETWEEN 20 AND 30),
Citizen VARCHAR(10)
)
```

При помощи оператора CHECK было неявно создано неименованное ограничение, которому автоматически было присвоено имя, состоящее из имени таблицы, имени поля и уникального номера. Можно создать именованное ограничение для той же таблицы:

```
CREATE TABLE MyTable3
(
AGE SMALLINT UNIQUE CLUSTERED CONSTRAINT CK_AGE_id CHECK (AGE BETWEEN 20 AND
30),
Citizen VARCHAR(10)
)
```

Можно к таблице MyTable3 добавить первичный ключ по полю AGE:

```
CREATE TABLE MyTable3
(
AGE SMALLINT CONSTRAINT PK_AGE_id PRIMARY KEY NONCLUSTERED
CONSTRAINT CK_AGE_id CHECK (AGE BETWEEN 20 AND 30),
Citizen VARCHAR(10)
)
```

Точно так же для этой таблицы можно определить уникальное именованное ограничение по полям AGE и Citizen:

```
CREATE TABLE MyTable3
(
AGE SMALLINT CONSTRAINT CK_AGE_id CHECK (AGE BETWEEN 20 AND 30),
Citizen VARCHAR(10),
CONSTRAINT U_Store UNIQUE NONCLUSTERED (AGE, Citizen)
)
```

Следующий запрос создает таблицу с использованием поля типа Uniqueidentifier, в которое будет вставляться сгенерированное по умолчанию значение этого типа при помощи функции NEWID():

```
CREATE TABLE Globally_Unique_Data
(guid uniqueidentifier
CONSTRAINT Guid_Default DEFAULT NEWID(),
Employee_Name varchar(60),
CONSTRAINT Guid_PK PRIMARY KEY (Guid)
)
```

Теперь следует рассмотреть пример создания таблицы с вычисляемыми полями. Она будет состоять из двух полей типа Integer и одного вычисляемого поля. Тип вычисляемого поля определяется компилятором автоматически:

```
CREATE TABLE MyTable4
(
LowValue int,
HighValue int,
myavg AS (LowValue + HighValue)/2
)
```

Оператор IDENTITY используется для указания того, что поле является идентифицирующим полем таблицы. В общем случае поля типа IDENTITY используются в связке с первичными ключами для создания уникальных последовательностей записей. Оператор IDENTITY может быть использован с полями типа tinyint, smallint, int, bigint, decimal(p,0) или numeric(p,0). Оператор принимает параметры seed и increment. В параметре seed указывается начальное значение поля, а в параметре increment указывается значение, на которое будет производиться увеличение максимального значения последовательности при вводе новой записи. В качестве примера можно создать таблицу MyTable5, имеющую генератор значений и первичный ключ по этому полю:

```
CREATE TABLE MyTable5
(
AutoID INT IDENTITY(1,1),
CHECK (AutoID <= 10),
SomeValue CHAR(5),
CONSTRAINT My5_PK PRIMARY KEY (AutoID)
)
```

Для изменения структуры таблицы средствами Transact-SQL следует воспользоваться командой ALTER TABLE. Команда ALTER TABLE позволяет изменять, добавлять и удалять столбцы таблицы и ограничения, активировать и деактивировать ограничения и триггеры. Команда имеет следующий синтаксис:

```
ALTER TABLE table
{ [ ALTER COLUMN column_name
{ new_data_type [ ( precision [ , scale ] ) ]
[ COLLATE < collation_name > ]
[ NULL | NOT NULL ]
| {ADD | DROP } ROWGUIDCOL }
]
| ADD
{ [ < column_definition > ]
| column_name AS computed_column_expression
} [ ....n ]
| [ WITH CHECK | WITH NOCHECK ] ADD
{ < table_constraint > } [ ....n ]
| DROP
{ [ CONSTRAINT ] constraint_name
| COLUMN column } [ ...n ]
```

```
| { CHECK | NOCHECK } CONSTRAINT
{ ALL | constraint_name [ ....n ] }
| { ENABLE | DISABLE } TRIGGER
{ ALL | trigger_name [ ....n ] }
}
< column_definition > ::=
{ column_name data_type }
[ [ DEFAULT constant_expression ] [ WITH VALUES ]
| [ IDENTITY [ (seed , increment ) [ NOT FOR REPLICATION ] ] ]
]
[ ROWGUIDCOL ]
[ COLLATE < collation_name > ]
[ < column_constraint > ] [ ...n ]
< column_constraint > ::=
[ CONSTRAINT constraint_name ]
{ [ NULL | NOT NULL ]
| [ { PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
[ WITH FILLFACTOR = fillfactor ]
[ ON { filegroup | DEFAULT } ]
]
| [ [ FOREIGN KEY ]
REFERENCES ref_table [ ( ref_column ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
[ NOT FOR REPLICATION ]
]
| CHECK [ NOT FOR REPLICATION ]
( logical_expression )
}
< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{ [ { PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ]
{ ( column [ ....n ] ) }
[ WITH FILLFACTOR = fillfactor ]
[ ON {filegroup | DEFAULT } ]
]
| FOREIGN KEY
[ ( column [ ....n ] ) ]
REFERENCES ref_table [ ( ref_column [ ....n ] ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
```

```
[ ON UPDATE { CASCADE | NO ACTION } ]
[ NOT FOR REPLICATION ]
| DEFAULT constant_expression
[ FOR column ] [ WITH VALUES ]
| CHECK [ NOT FOR REPLICATION ]
( search_conditions )
}
```

В параметре table указывается имя таблицы, в структуру которой будут внесены изменения. Команда ALTER COLUMN указывает на то, что будет изменен какой-либо столбец таблицы, а в параметре column_name указывается имя этого столбца. В параметре new_data_type указывается новый тип данных столбца.

Команда ADD позволяет добавить новое поле, вычисляемое поле или ограничение. Команда DROP используется для удаления поля или ограничения. В параметрах constraint_name и column_name указываются имена ограничений и полей.

Конечно же, без рассмотрения примеров изменения структуры таблиц не обойтись. Следующий запрос добавляет поле в таблицу MyTable3:

```
ALTER TABLE MyTable3 ADD NewCol1 VARCHAR(20) NULL
```

Для удаления нового столбца стоит использовать следующий запрос:

```
ALTER TABLE MyTable3 DROP COLUMN NewCol1
```

Теперь можно добавить в таблицу MyTable3 поле NewCol2 с именованным уникальным ограничением Col_Unique:

```
ALTER TABLE MyTable3 ADD NewCol2 SMALLINT NULL
CONSTRAINT Col_Unique UNIQUE
```

Для удаления ограничения Col_Unique следует воспользоваться следующей командой:

```
ALTER TABLE MyTable3 DROP CONSTRAINT Col_Unique
```

Оператор CHECK активирует ограничение, а оператор NOCHECK, в свою очередь, деактивирует его. Приведенный ниже пример демонстрирует применение выражений CHECK CONSTRAINT и NOCHECK CONSTRAINT:

```
CREATE TABLE cnst_example
(id INT NOT NULL,
name VARCHAR(10) NOT NULL,
salary MONEY NOT NULL
CONSTRAINT salary_cap CHECK (salary < 100000)
)
-- Допустимые значения
INSERT INTO cnst_example VALUES (1,'Joe Brown',65000)
INSERT INTO cnst_example VALUES (2,'Mary Smith',75000)
-- Попытка вставки данного значения вызовет ошибку
INSERT INTO cnst_example VALUES (3,'Pat Jones',105000)
-- Деактивация ограничения и повтор попытки. Попытка будет удачной.
```

```
ALTER TABLE cnst_example NOCHECK CONSTRAINT salary_cap
INSERT INTO cnst_example VALUES (3, 'Pat Jones',105000)
-- Активация ограничения и попытка вставки значения. Попытка вызовет ошибку.
ALTER TABLE cnst_example CHECK CONSTRAINT salary_cap
INSERT INTO cnst_example VALUES (4, 'Eric James',110000)
```

В рассмотренном выше примере создается таблица cnst_example с ограничением по полю salary. Производится ряд попыток ввода значений, допустимых и недопустимых, при отключенном и включенном ограничении. Для активации всех ограничений, принадлежащих данной таблице, следует использовать выражение CHECK ALL, как это продемонстрировано в следующем примере:

```
ALTER TABLE cnst_example CHECK CONSTRAINT ALL
```

Для того чтобы деактивировать ограничения, принадлежащие таблице, следует воспользоваться выражением NOCHECK ALL, как это продемонстрировано в следующем примере:

```
ALTER TABLE cnst_example NOCHECK CONSTRAINT ALL
```

Аналогичные операции с таблицами можно выполнять при помощи утилиты SQL Server Enterprise Manager. Для создания таблицы нужно выбрать соответствующий объект Tables из списка базы данных, например TestDB, и выбрать пункт контекстного меню New Table. Та же самая операция может производиться при помощи команды основного меню Action ▸ New Table. В результате будет активировано окно создания таблицы базы данных, приведенное на рис. 9.31.

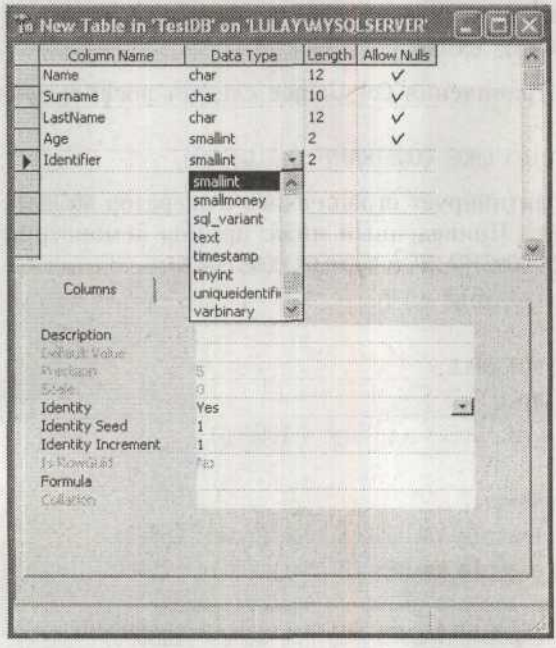


Рис. 9.31. Окно создания таблицы базы данных

В поле Column Name указывается имя столбца таблицы, в поле Data Type — его тип, в поле Length — длина, а в поле Allow Nulls допустимость значения NULL. В колонке Data Type располагается список, в котором можно выбрать тип данных создаваемого поля. В зависимости от типа данных поля будет изменяться состав доступных для изменения элементов, расположенных на панели Columns.

В поле Description можно вести описание создаваемого столбца. В поле Default Value указывается значение, которое будет использоваться по умолчанию для значений создаваемого столбца. Поля Precision и Scale определяют разрядность и точность значений типа Numeric и Decimal. Поле Identity позволяет создавать идентифицирующие столбцы.

В поле Collation можно выбрать тип кодировки символов, используемый в таблице, и определить порядок сравнения символов. Удобнее всего настройку производить при помощи специального редактора, доступ к которому можно получить, нажав на кнопку, расположенную справа от поля. Окно редактора приведено на рис. 9.32.

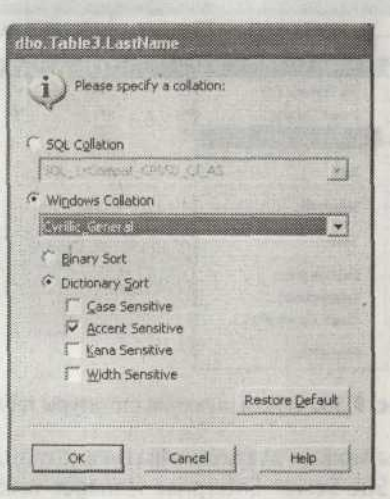


Рис. 9.32. Окно выбора кодировки символов, используемой в таблице

Окно позволяет сделать выбор используемой кодировки. В списке SQL Collation можно выбрать одну из кодировок, поставляющихся с SQL Server 2000. В списке Windows Collation выбираются кодировки операционной системы. При использовании кодировки Windows также доступна возможность выбора порядка сортировки символов. Порядок сортировки Binary Sort используется в тех случаях, когда требуется сортировка только с использованием двоичных кодов символов. Порядок сортировки Dictionary Sort предоставляет более широкие возможности сортировки, позволяя указывать дополнительные параметры:

- Значение Case Sensitive указывает на то, что одинаковые символы верхнего и нижнего регистров трактуются как разные.
- Значение Accent Sensitive указывает на то, что символы с ударением и без ударения должны трактоваться как разные.

- Значение Kana Sensitive используется для сортировки диалектов японских иероглифов.
- Значение Width Sensitive указывает на то, что символы, имеющие половинную и полную длину, являются различными.

Кнопка Restore Default позволяет вернуть все значения в исходное состояние. Чтобы сохранить созданную таблицу, достаточно нажать кнопку Save, расположенную на главной панели SQL Server Enterprise Manager.

Для внесения изменений в созданную таблицу нужно выполнить команду Design Table ее контекстного меню. В результате будет отображено окно, позволяющее работать со структурой таблицы. Для того чтобы удалить столбец, следует вызвать контекстное меню редактора полей на выбранной записи и выполнить команду Delete Column. На рис. 9.33 приведен соответствующий пример.

Рис. 9.33. Окно изменения структуры таблицы

Для удаления таблицы можно воспользоваться командой DROP TABLE TableName. Конечно же, утилита SQL Server Enterprise Manager тоже позволяет выполнить эту операцию. Нужно лишь выделить таблицу и выбрать пункт контекстного меню Delete.

Отношения ссылочной целостности

Для иллюстрации метода применения ссылочной целостности потребуется создать две таблицы. Таблица CITIZEN определяется следующим запросом:

```
CREATE TABLE CITIZEN
(
FirstName VARCHAR (15),
LastName VARCHAR (15),
SurName VARCHAR (15),
CitizenID SMALLINT IDENTITY(1,1)
```

```
CONSTRAINT MyConstr PRIMARY KEY CLUSTERED
)
```

Поле CitizenID задает первичный ключ, по которому будет привязана вторая таблица. Следующий запрос определяет ее структуру:

```
CREATE TABLE Temperature
(
FloorNumber TinyInt,
RoomsNumber TinyInt
CONSTRAINT CK_ROOM CHECK (RoomsNumber < 5),
CitizenID SMALLINT,
FOREIGN KEY (CitizenID) REFERENCES CITIZEN(CitizenID)
)
```

Поле CitizenID было объявлено внешним ключом и связано с одноименным полем таблицы CITIZEN. Того же самого эффекта можно было добиться при помощи другого SQL-запроса:

```
CREATE TABLE Temperature
(
FloorNumber TinyInt,
RoomsNumber TinyInt CONSTRAINT CK_ROOM CHECK (RoomsNumber < 5),
Temperature TinyInt,
CitizenID SMALLINT REFERENCES CITIZEN(CitizenID)
)
```

Если необходимо создать именованное ограничение ссылочной целостности, то запрос потребуется несколько изменить:

```
CREATE TABLE Temperature
(
FloorNumber TinyInt,
RoomsNumber TinyInt,
Temperature TinyInt,
CONSTRAINT CK_ROOM CHECK (RoomsNumber < 5),
CitizenID SMALLINT,
CONSTRAINT FK_Citizen FOREIGN KEY (CitizenID)
REFERENCES CITIZEN(CitizenID)
)
```

Для ограничений ссылочной целостности можно определить действия, которые должны выполняться при изменении значения первичного ключа. При удалении записи вызывается событие ON DELETE {CASCADE | NO ACTION}, в котором можно определить каскадное удаление связанных записей — CASCADE либо вообще не предпринимать каких-либо действий — NO ACTION. При обновлении записей, входящих в ключ, аналогичным образом возникает и обрабатывается событие ON UPDATE {CASCADE | NO ACTION}. Чтобы задать реакцию на события, необходимо переопределить ограничение FK_Citizen таблицы Temperature:

```
ALTER TABLE Temperature DROP CONSTRAINT FK_Citizen
ALTER TABLE Temperature ADD CONSTRAINT FK_Citizen FOREIGN KEY (CitizenID)
REFERENCES CITIZEN(CitizenID) ON DELETE NO ACTION ON UPDATE CASCADE
```

Отношения ссылочной целостности между таблицами можно создать при помощи утилиты SQL Server Enterprise Manager. Для этого необходимо вызвать контекстное меню объекта Diagrams и выполнить команду New Database Diagram. В результате будет отображено диалоговое окно редактора диаграмм, показанное на рис. 9.34. В этом окне отображаются таблицы, между которыми можно устанавливать связи при помощи мыши.

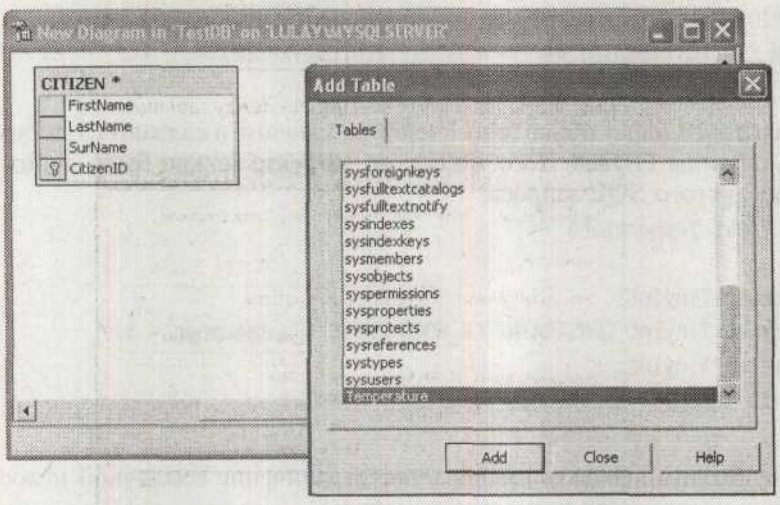


Рис. 9.34. Окно редактора диаграмм со списком добавляемых таблиц

Прежде всего необходимо добавить в диаграмму таблицы. Для этого требуется выполнить команду контекстного меню Add Table. В результате будет отображен список Add Table, из которого можно будет выбрать добавляемые в диаграмму таблицы. В данном случае нужно выбрать в списке таблицы Citizen и Temperature.

После этого необходимо установить связи между таблицами. Для этого достаточно выбрать ключевое поле из таблицы Citizen и перетащить его мышью на ключевое поле таблицы Temperature. Именно это действие и устанавливает связь (рис. 9.35).

Параметры связи определяются в окне параметров, доступ к которому можно получить при создании связи или при помощи команды контекстного меню связи Options. Окно параметров отношения ссылочной целостности приведено на рис. 9.36.

В поле Table указывается имя выбранной таблицы. В поле Relationship name можно указать имя отношения ссылочной целостности. В полях Primary key table и Foreign key table указываются таблицы, между которыми будет создана или уже существует связь «Master—Detail». В столбце Primary key table указы-

ваются поля, входящие в первичный ключ, а в столбце Foreign key table — входящие во внешний.

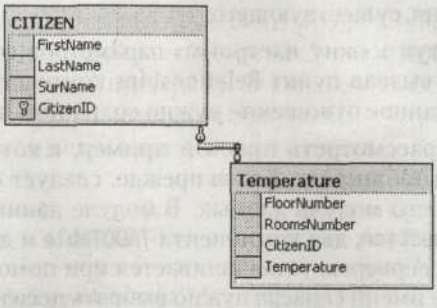


Рис. 9.35. Связь «Master—Detail» между таблицами

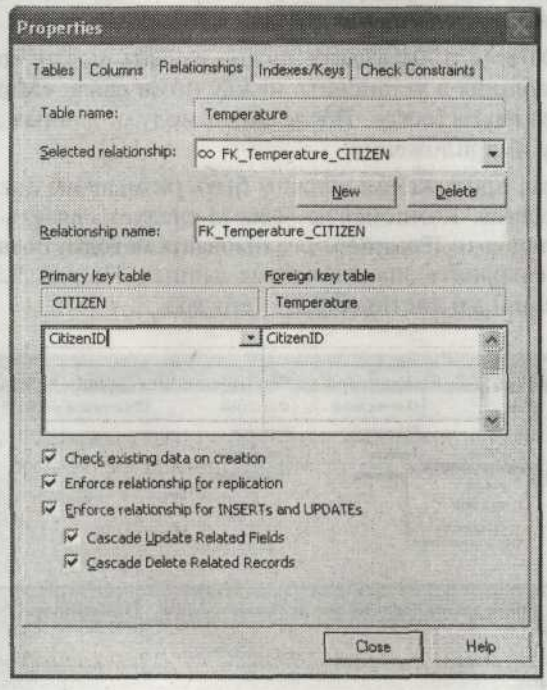


Рис. 9.36. Настройка параметров отношения ссылочной целостности

Флажок Check existing data on creation включает проверку существующих данных на соответствие ограничениям ссылочной целостности. В случае нарушения существующего ограничения будет выдано сообщение об ошибке. Флажок Enforce relationship for Replication разрешает функционирование режима, при котором подчиненная таблица реплицируется так же, как и главная.

Установка флажка Enforce relationship for INSERTs and UPDATEs включает механизм защиты данных от нарушения ограничения ссылочной целостности.

Флажки Cascade Update Related Fields и Cascade Delete Related Fields включают режимы каскадного удаления и каскадного обновления данных. Кнопка New служит для создания нового отношения ссылочной целостности, а кнопка Delete — для удаления существующего.

Также получить доступ к окну настройки параметров отношения ссылочной целостности можно, вызвав пункт Relationships контекстного меню редактора полей таблицы. Созданное отношение нужно сохранить, нажав на кнопку Save.

Теперь необходимо рассмотреть простой пример, в котором будет показана работа с созданными таблицами. Как и прежде, следует создать новое приложение и добавить в него модуль данных. В модуле данных нужно разместить компоненты TADOConnection, два компонента TADOTable и два компонента TDataSource. Соединение с сервером устанавливается при помощи компонента TADOConnection. В качестве имени сервера нужно выбрать псевдоним сервера myServer и указать базу данных TestDB. Свойство IsolationLevel должно получить значение ilReadUncommitted, а свойство LoginPrompt — значение False. После этого можно устанавливать соединение с базой данных.

Компоненты TADOTable и TDataSource нужно связать между собой, определить отображаемые таблицы и установить между ними связь «Master—Detail» при помощи свойства MasterSource. После этого модуль данных нужно объявить в основном модуле приложения.

На главной форме приложения должны быть размещены две таблицы и один компонент TPopupMenu. Компоненты TDBGrid следует связать с компонентами TADOTable. В компоненте TPopupMenu реализованы методы, позволяющие добавлять, удалять и сохранять значения базы данных. На рис. 9.37 приведен вид формы приложения, а в листинге 9.1 — его код.

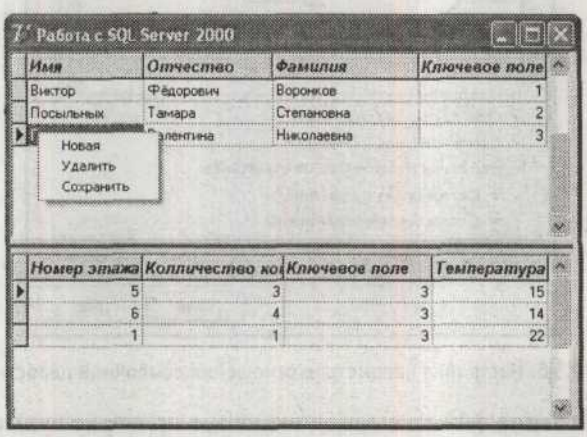


Рис. 9.37. Работа с SQL Server 2000

Листинг 9.1. Работа с SQL Server 2000

```
uses DataModule;

{$R *.dfm}
```

```
procedure TMainForm.InsertClClick(Sender: TObject);
begin
  if (PopupMenu1.PopupComponent.Name = 'DBGrid1') then begin
    DataMod.ADOTable1.Insert;
  end;
  if (PopupMenu1.PopupComponent.Name = 'DBGrid2') then begin
    DataMod.ADOTable2.Insert;
  end;
end;

procedure TMainForm.DeleteClClick(Sender: TObject);
begin
  if (PopupMenu1.PopupComponent.Name = 'DBGrid1') then begin
    DataMod.ADOTable1.Delete;
  end;
  if (PopupMenu1.PopupComponent.Name = 'DBGrid2') then begin
    DataMod.ADOTable2.Delete;
  end;
end;

procedure TMainForm.SaveClClick(Sender: TObject);
begin
  with DataMod do begin
    if ADOTable1.Modified then begin
      ADOConnection1.BeginTrans;
      ADOTable1.Post;
      ADOConnection1.CommitTrans;
    end;
    if ADOTable2.Modified then begin
      ADOConnection1.BeginTrans;
      ADOTable2.Post;
      ADOConnection1.CommitTrans;
    end;
  end;
end;
```

Индексы

Индексы можно создавать как средствами языка Transact-SQL, так и при помощи утилиты SQL Server Enterprise Manager. Индекс создается командой CREATE INDEX, синтаксис которой приведен ниже:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
ON { table | view } ( column [ ASC | DESC ] [ ,...n ] )
```

```
[ WITH < index_option > [ ....n] ]
[ ON filegroup ]
< index_option > :: =
{ PAD_INDEX |
FILLFACTOR = fillfactor |
IGNORE_DUP_KEY |
DROP_EXISTING |
STATISTICS_NORECOMPUTE |
SORT_IN_TEMPDB
}
```

Создаваемый индекс может быть уникальным, кластерным или некластерным. В параметре index_name указывается имя создаваемого индекса. Индекс может быть создан как для таблицы, так и для представления. Операторы ASC | DESC определяют порядок сортировки значений в индексе.

Также индекс можно создать для существующей группы файлов. Для этого используется выражение ON filegroup, в котором указывается имя группы файлов. Оператор FILLFACTOR позволяет определить в процентном отношении количество свободного места на страницах индексов. А оператор PAD_INDEX используется для определения количества свободного места на промежуточных страницах индекса. Оператор PAD_INDEX используется вместе с оператором FILLFACTOR, так как PAD_INDEX использует то же процентное отношение свободного места на промежуточных страницах, которое указано в параметре fillfactor для листьев.

Параметр IGNORE_DUP_KEY указывает на то, что при попытке вставки дублирующей строки она будет проигнорирована и будет выведено сообщение об ошибке. Если опция IGNORE_DUP_KEY не используется, SQL Server 2000 выведет сообщение об ошибке и произведет полный откат транзакции.

Оператор DROP_EXISTING указывает, что данный индекс перезапишет одноименный, если таковой существует. В некоторых случаях пересоздание индекса может быть произведено несколько быстрее за счет того, что старый индекс будет конвертирован в новый.

Параметр STATISTICS_NORECOMPUTE отменяет автоматическое обновление статистики индекса по мере его устаревания. В случае необходимости обновления статистики придется воспользоваться командой UPDATE STATISTICS.

Теперь необходимо перейти к примерам. Следующий запрос создает простой индекс по полю Temperature таблицы Temperature:

```
CREATE INDEX Temp_Ind ON Temperature (Temperature)
```

После этого нужно создать уникальный индекс по полю FirstName таблицы Citizen:

```
CREATE UNIQUE INDEX Temp_Ind1 ON CITIZEN (FirstName)
```

Также потребуется уникальный композитный индекс Temp_Comp_Ind по полям FirstName и LastName таблицы Citizen:

```
CREATE UNIQUE INDEX Temp_Comp_Ind ON CITIZEN (FirstName, LastName)
```

Точно так же можно создать кластерный композитный индекс:

```
CREATE UNIQUE CLUSTERED INDEX Temp_Comp_Ind ON CITIZEN (FirstName, LastName)
WITH DROP_EXISTING
```

В данном примере была использована команда DROP_EXISTING, которая переписала существующий индекс. Следующий запрос создает уникальный кластерный индекс, игнорирующий дубликаты:

```
CREATE UNIQUE CLUSTERED INDEX My_Index ON MyTable (SomeInteger)
WITH IGNORE_DUP_KEY
```

Также индекс можно создать и изменить с помощью утилиты SQL Server Enterprise Manager. Для этого нужно выполнить команду Indexes/Keys контекстного меню редактора полей таблицы. Соответствующее диалоговое окно показано на рис. 9.38.

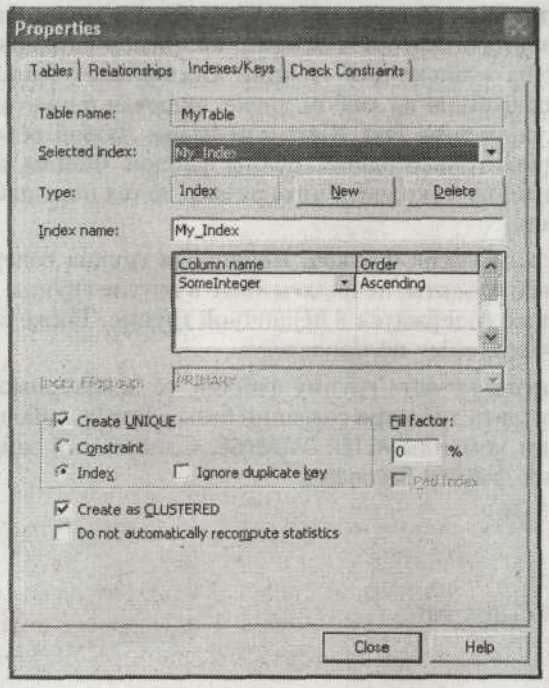


Рис. 9.38. Окно создания и модификации индексов

В поле Table name указывается таблица, с которой в данный момент осуществляется работа. В поле Index name указывается имя текущего индекса. В списке Column name указываются поля, входящие в индекс, а в поле Order — порядок сортировки по каждому полю.

Флажок Create UNIQUE указывает на то, что будет создан уникальный индекс. Поле ввода Fill factor позволяет определить процент заполнения индекса, а флажок Pad Index позволяет задать то же самое значение для промежуточных страниц индекса.

Флажок Ignore duplicate key включает проверку значений на уникальность, не допуская ввода дубликатов. Переключатель Create as CLUSTERED указывает на то, что будет создан кластерный индекс. В свою очередь, флажок Don't automatically recompute statistics определяет, будет ли автоматически обновляться статистика по индексу или нет.

Включение таблиц баз данных в группы файлов

Как уже было отмечено, группа файлов представляет собой набор файлов данных. Каждой группе могут быть назначены свой график резервного копирования и свой режим доступа. При добавлении нового объекта в группу файлов, состоящую из нескольких файлов, SQL Server 2000 самостоятельно выбирает файл, в который будет помещен данный объект.

При грамотном использовании групп файлов можно добиться существенного ускорения работы базы данных за счет разнесения групп файлов по разным дискам и включения некоторых таблиц, например справочников, в группы с режимом доступа «только для чтения». Системные администраторы могут создавать группы файлов на любом диске, включая в них определенные таблицы, индексы, поля типа Text, NText или Image. Любой объект базы данных может быть членом только одной группы файлов. Файлы, входящие в группу, не смогут автоматически увеличить размер до тех пор, пока есть место хотя бы в одном из них.

Различают два типа групп файлов. Первичная группа содержит первичный файл данных и все объекты, не включенные в другие группы файлов. Системные таблицы также содержатся в первичной группе. Также существуют группы файлов, определяемые пользователем.

Перед тем как использовать группу файлов, ее необходимо создать. Группу файлов можно создать либо при создании базы данных, либо изменяя ее структуру при помощи команды ALTER DATABASE. Следующий запрос создает базу данных с группой файлов Secondary:

```
CREATE DATABASE Sales
ON PRIMARY
(NAME = SPri1_dat,
FILENAME = 'c:\SPri1dat.mdf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15%),
(NAME = SPri2_dat,
FILENAME = 'c:\SPri2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 15%),
FILEGROUP SECONDARY
(NAME = SGrp1_dat,
FILENAME = 'c:\SGF1dt.ndf',
```

```
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5),
(NAME = SGrp2_dat,
FILENAME = 'c:\SGF2dt.ndf',
SIZE = 10,
MAXSIZE = 50,
FILEGROWTH = 5)
```

В приведенном выше примере была создана группа файлов Secondary, состоящая из двух файлов. Теперь можно добавить группу файлов к уже существующей базе данных TestDB. Сначала следует добавить объявление группы файлов в базу данных:

```
ALTER DATABASE TestDB
ADD FILEGROUP SecondaryGroup
```

Теперь в созданную группу добавляются файлы данных:

```
ALTER DATABASE TestDB
ADD FILE
( NAME = test1dat3,
FILENAME = 'c:\t1dat3.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB),
( NAME = test1dat4,
FILENAME = 'c:\t1dat4.ndf',
SIZE = 5MB,
MAXSIZE = 100MB,
FILEGROWTH = 5MB)
TO FILEGROUP SecondaryGroup
```

Теперь нужно создать таблицу GR_Table и включить ее в состав только что созданной группы файлов SecondaryGroup:

```
CREATE TABLE GR_Table
(
Value1 VARCHAR (10),
Value2 VARCHAR (20),
Value3 TINYINT,
Value4 TEXT
) ON SecondaryGroup
```

Теперь необходимо создать новую таблицу, часть полей которой будет входить в основную группу — Primary, а часть — в группу SecondaryGroup:

```
CREATE TABLE GR_Mix_Table
(
```

```
Value1 VARCHAR (10),
Value2 TEXT
) TEXTIMAGE_ON SecondaryGroup
```

Оператор TEXTIMAGE_ON используется для указания группы файлов, хранящей «большие» типы данных. Также группу файлов можно создать при помощи средств утилиты SQL Server Enterprise Manager. Для этого нужно выполнить команду контекстного меню Properties базы данных TestDB и перейти на вкладку Filegroups, внешний вид которой показан на рис. 9.39.

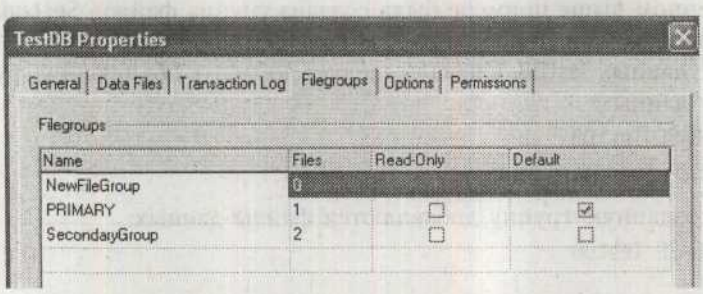


Рис. 9.39. Редактор групп файлов

На рисунке показано, что в редакторе присутствуют созданная ранее группа SecondaryGroup, группа PRIMARY и созданная в редакторе группа NewFileGroup. Выбранной группе можно установить режим «только для чтения», взведя флажок Read-Only, либо выбрать группу, которая будет использоваться по умолчанию, взведя флажок Default. Изначально эту роль выполняет группа Primary.

В нижней части окна располагается кнопка Delete, позволяющая удалить существующую или созданную группу.

Теперь следует обратить внимание на вкладку Data Files, окно которой показано на рис. 9.40. В списке Database files указывается список файлов (File Name), путь к ним (Location), минимальный размер файла данных (Space Allocated (MB)) и используемая группа файлов (Filegroup).

Флажок Automatically grow file включает режим автоматического увеличения размера файла при достижении данными максимума доступного места. Свойство File Growth определяет, как будет произведено приращение данных, — в мегабайтах (In megabytes) или в процентах (By percent). Свойство Maximum File Size позволяет определить максимальный размер файла данных. Для того чтобы изменить группу файлов, в которую входит данная таблица, необходимо перейти в окно редактора полей и вызвать пункт контекстного меню Properties. На рис. 9.41 приведен фрагмент соответствующего окна.

В поле Table Filegroup указывается группа, в которую входит данная таблица, а в поле Text Filegroup указывается группа, в которую входят большие поля. Для того чтобы посмотреть, в какую группу файлов входит данная таблица, можно вызвать пункт контекстного меню Properties или посмотреть в поле Filegroup имя группы.

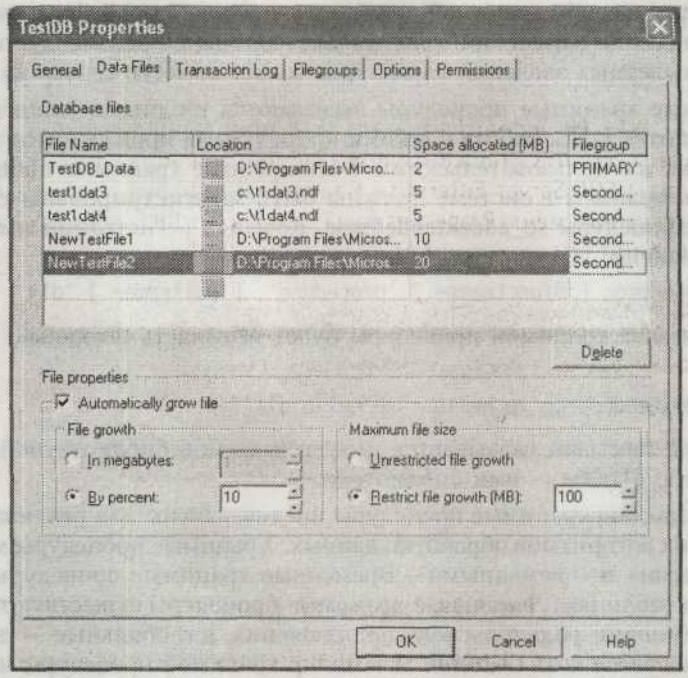


Рис. 9.40. Определение файлов данных группе файлов

Рис. 9.41. Окно смены группы файлов выбранной таблицы

Хранимые процедуры

Хранимая процедура представляет собой набор команд на языке Transact-SQL, хранящихся на сервере в скомпилированном виде. SQL Server 2000 позволяет работать с тремя типами хранимых процедур:

- системными хранимыми процедурами (System Stored Procedures);
- расширенными хранимыми процедурами (Extended Stored Procedures);
- пользовательскими хранимыми процедурами (User-defined Stored Procedure).

Системные хранимые процедуры представляют собой разновидность хранимых процедур, предназначенных для выполнения операций администрирования сервера и получения разнообразной информации о его работе из системных

таблиц. Перед именем хранимой процедуры указывается префикс sp. Самостоятельное внесение изменений в системные таблицы не рекомендуется, так как установка неверных значений параметров может вывести сервер из строя.

Расширенные хранимые процедуры вызываются из динамически подключаемых библиотек DLL. Работа с такими процедурами производится в обычном порядке, как и с пользовательскими. Расширенные хранимые процедуры перед использованием в системе должны быть зарегистрированы с помощью хранимой процедуры sp_addextendedproc. Команда регистрации имеет следующий синтаксис:

```
sp_addextendedproc [ @functname= ] 'procedure', [ @dllname= ] 'dll'
```

Команда вызова хранимой процедуры будет выглядеть несколько иначе:

```
USE master
EXEC sp_addextendedproc xp_hello, 'xp_hello.dll'
```

В параметре functname указывается имя вызываемой библиотечной функции, а в параметре dllname — имя библиотеки.

Пользовательские хранимые процедуры предназначены для реализации пользовательских алгоритмов обработки данных. Хранимые процедуры могут быть «постоянными» и «временными». Временные хранимые процедуры подобны временным таблицам. Локальные хранимые процедуры существуют только во время соединения родительского подключения, а глобальные — до тех пор, пока используется хотя бы один экземпляр хранимой процедуры и не закрыто родительское соединение.

Хранимая процедура создается с помощью команды CREATE PROCEDURE, имеющей следующий синтаксис:

```
CREATE PROC [ EDURE ] procedure_name [ ; number ]
[ { @parameter data_type }
[ VARYING ] [ = default ] [ OUTPUT ]
] [ ,...n ]
[ WITH
{ RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]
[ FOR REPLICATION ]
AS sql_statement [ ...n ]
```

В параметре procedure_name указывается имя хранимой процедуры. Имя хранимой процедуры должно быть уникальным в пределах базы данных. В параметре number указывается номер одноименной хранимой процедуры, состоящей в группе. Параметр parameter содержит имя параметра, а в параметре data_type указывается его тип данных. Перед параметром обязательно должен стоять знак @. Оператор VARYING указывает на то, что данный параметр имеет тип Variant и может возвращать различные типы данных. В хранимых процедурах этот параметр используется только для работы с курсорами.

Ключевое слово OUTPUT используется для обозначения параметров, которые возвращают значения после отработки процедуры. Параметр RECOMPILE позволяет указать серверу на отсутствие необходимости использовать сформированный план выполнения хранимой процедуры. В этом случае хранимую

процедуру нужно будет каждый раз компилировать заново. Параметр ENCRYPTION указывает серверу на необходимость шифрования текста хранимой процедуры, хранящегося в таблице syscomments. Использование данной опции исключает публикацию хранимой процедуры как часть реплицируемой базы данных.

Параметр FOR REPLICATION используется только для выполнения репликации хранимых процедур. Хранимая процедура, созданная с использованием данного параметра, может использоваться только в процессе репликации. В этом случае ее нельзя вызвать напрямую. Подписчику передаются заголовок хранимой процедуры и список ее параметров.

В качестве примера следует создать простую хранимую процедуру, которая выберет несколько значений из таблицы Employees базы данных Northwind:

```
USE Northwind
IF EXISTS (SELECT "name" FROM sysobjects
WHERE "name" = 'au_My_SimpleProc' AND type = 'P')
DROP PROCEDURE au_My_SimpleProc
GO
CREATE PROCEDURE au_My_SimpleProc
AS
SELECT FirstName, LastName FROM Employees
GO
```

Первая часть команды производит проверку на наличие процедуры. Если таковая существует, то она удаляется. Вторая часть команды создает процедуру с именем au_My_SimpleProc. Для выполнения процедуры можно воспользоваться командой EXECUTE или EXEC:

```
EXECUTE au_My_SimpleProc
```

Следующий запрос создает хранимую процедуру au_My_SimpleProc с параметрами.

```
CREATE PROCEDURE au_My_SimpleProc2
@FirstName VARCHAR(15),
@LastName VARCHAR (15)
AS
SELECT FirstName, LastName FROM Employees
WHERE FirstName = @FirstName AND LastName = @LastName
```

Передать хранимой процедуре входные параметры можно следующим образом:

```
EXECUTE au_My_SimpleProc2 'Nancy', 'Davolio'
```

Есть и другой вариант запуска процедуры:

```
EXEC au_My_SimpleProc2 @FirstName = 'Nancy', @LastName = 'Davolio'
```

Следующий запрос изменяет определение процедуры. В нее добавляется возможность использования значений параметров по умолчанию:

```
CREATE PROCEDURE au_My_SimpleProc2
@FirstName varchar(15) = 'Andrew',
```

```
@LastName varchar(15) = 'Fuller'
AS
SELECT FirstName, LastName FROM Employees
WHERE FirstName = @FirstName AND LastName = @LastName
```

Ниже представлены две команды, возвращающие различные результаты:

```
EXECUTE au_My_SimpleProc2 'Nancy', 'Davolio'
EXECUTE au_My_SimpleProc2
```

Первая команда содержит входные параметры, а вторая использует значения по умолчанию.

Теперь следует рассмотреть принципы работы с выходными параметрами. Для этого нужно создать хранимую процедуру **au_My_SimpleProc3**:

```
CREATE PROCEDURE au_My_SimpleProc3
@FirstName varchar(15),
@LastName varchar(15) OUTPUT
AS
SELECT @LastName = LastName FROM Employees
WHERE FirstName = @FirstName
```

Для получения параметра, возвращаемого хранимой процедурой, можно воспользоваться следующим запросом:

```
DECLARE @LastName varchar(15)
EXECUTE au_My_SimpleProc3 'Nancy', @LastName OUTPUT
SELECT @LastName
```

Для изменения созданной хранимой процедуры следует использовать команду **ALTER PROCEDURE**. Команда имеет следующий синтаксис:

```
ALTER PROC [ EDURE ] procedure_name [ ; number ]
[ { @parameter data_type }
[ VARYING ] [ = default ] [ OUTPUT ]
] [ ....n ]
[ WITH
{ RECOMPILE | ENCRYPTION
| RECOMPILE , ENCRYPTION
}
]
[ FOR REPLICATION ]
AS
sql_statement [ ...n ]
```

В качестве примера можно изменить определение хранимой процедуры **au_My_SimpleProc**, включив режим шифрования ее текста:

```
ALTER PROCEDURE au_My_SimpleProc WITH ENCRYPTION
AS
SELECT FirstName, LastName FROM Employees
```

Текст зашифрованной процедуры можно посмотреть при помощи приведенного ниже запроса:

```
SELECT c.id, c.text
FROM syscomments c INNER JOIN sysobjects o
ON c.id = o.id
WHERE o.name = 'au_My_SimpleProc'
```

Теперь нужно применить полученные знания о хранимых процедурах для разработки приложения в Delphi. Как всегда, после создания нового проекта к нему нужно добавить модуль данных. В этом модуле следует разместить компонент `TADOConnection`, два компонента `TADOStoredProc` и компонент `TDataSource`. После настройки соединения с сервером потребуется связать компоненты `TADOStoredProc` с компонентом `TADOConnection`. Один из компонентов `TADOStoredProc` также нужно связать с компонентом `TDataSource`.

В данном приложении будут использованы ранее созданные процедуры `au_My_SimpleProc2` и `au_My_SimpleProc3`. Первая процедура возвращает набор данных, а вторая — отдельное значение в качестве выходного параметра.

В свойствах `ProcedureName` компонентов `TADOStoredProc` нужно задать имена этих процедур. В компоненте, связанном с `TDataSource`, указывается процедура `au_My_SimpleProc2`. В свойство `Parameters` соответствующих компонентов будут загружены сведения о параметрах хранимых процедур.

Модуль данных следует объявить в модуле главной формы приложения. На главной форме нужно установить четыре компонента `TEdit`, компонент `TDBGrid` и две кнопки. При нажатии на кнопки будут вызываться методы, код которых приведен в листинге 9.2.

Компонент `TDBGrid` нужно связать с компонентом `TDataSource`. Первая кнопка будет осуществлять вызов функции `au_My_SimpleProc2`, возвращающей набор данных, а вторая кнопка будет вызывать `au_My_SimpleProc3`, которая будет возвращать значение в параметре. На рис. 9.42 приведено изображение формы приложения.

Рис. 9.42. Работа с хранимыми процедурами

Листинг 9.2. Вызов хранимых процедур и получение результатов выполнения

```
uses StoredMod;

{$R *.dfm}

procedure TStoreForm.Func1BtnClick(Sender: TObject);
begin
  with StoredDM do begin
    ADOStoredProc2.Active:=False;
    ADOStoredProc2.Parameters.ParamByName('@FirstName').Value:=Edit1.Text;
    ADOStoredProc2.Parameters.ParamByName('@LastName').Value:=Edit2.Text;
    ADOStoredProc2.Active:=True;
  end;
end;

procedure TStoreForm.Func2BtnClick(Sender: TObject);
begin
  with StoredDM do begin
    ADOStoredProc1.Active:=False;
    ADOStoredProc1.Parameters.ParamByName('@FirstName').Value:=Edit3.Text;
    ADOStoredProc1.ExecProc;
    Edit4.Text:=VarToStr(ADOStoredProc1.Parameters.ParamByName('@LastName').Value);
  end;
end;
```

Следует обратить внимание на то, что при вызове второй процедуры был использован метод ExecProc, так как эта процедура не возвращает набор данных.

Пользовательские функции

SQL Server 2000 наряду с хранимыми процедурами предоставляет также возможность использования дополнительных функций. Разработчик имеет возможность создавать собственные функции, называемые пользовательскими (User-Defined Functions, UDF). Пользовательская функция является обычным объектом базы данных и имеет владельца. Функция инкапсулирует набор команд Transact-SQL. SQL Server 2000 поддерживает три типа пользовательских функций:

- функции Scalar;
- функции Inline;
- функции Multistatement.

Функции типа Scalar возвращают скалярное, то есть единственное, значение определенного типа, указанное в блоке RETURNS. Могут быть использованы все

скалярные типы данных, включая bigint и sql_variant. Тип данных Timestamp, пользовательские типы данных и нескалярные типы данных, такие как table или cursor, не поддерживаются. Тело функции заключается в блоке BEGIN...END.

Функции типа Inline и Multistatement всегда возвращают значения типа данных Table. Функции типа Inline не имеют блока BEGIN...END, так как возвращаемая ими таблица является результатом выборки единственного выражения SELECT. Функции типа Multistatement имеют блок BEGIN...END, в котором могут быть реализованы сложные запросы.

Пользовательская функция создается командой CREATE FUNCTION. Для функции скалярного типа команда будет иметь следующий синтаксис:

```
CREATE FUNCTION [ owner_name. ] function_name
( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] } [ ....n
] ] )
RETURNS scalar_return_data_type
[ WITH < function_option> [ [,] ...n] ]
[ AS ]
BEGIN
function_body
RETURN scalar_expression
END
```

В параметре function_name указывается имя функции, а в параметре owner_name, при необходимости, имя владельца. Параметр @parameter_name позволяет задавать входной параметр функции, а в параметре scalar_parameter_data_type указывается его тип. Параметр RETURNS scalar_return_data_type указывает тип данных возвращаемого значения.

Дополнительные возможности функции задаются при помощи параметра WITH <function_option>. Опция ENCRYPTION заставляет сервер шифровать тело функции, содержащейся в системной таблице. В свою очередь, опция SCHEMABINDING определяет, что зависимые от данной функции объекты не могут быть изменены командой ALTER или командой DROP.

Команда RETURN scalar_expression указывает точку выхода из функции и возвращаемое значение, имеющее тип, определенный в scalar_return_data_type. Следующий запрос создает функцию MyFunc1, вычисляющую среднее из трех чисел и в случае, если результат больше пяти, увеличивающую его вдвое:

```
CREATE FUNCTION MyFunc1 (@Val1 TINYINT, @Val2 TINYINT, @Val3 TINYINT)
RETURNS DECIMAL (5,3)
AS
BEGIN
DECLARE @Average DECIMAL (5,3)
SET @Average=((@Val1+@Val2+@Val3)/3)
IF @Average > 5 BEGIN
SET @Average=@Average*2
END
```

```
RETURN (@Average)
END
```

Для того чтобы вызвать функцию, необходимо использовать следующую команду:

```
SELECT dbo.MyFunc1 (1,2,3)
```

Для функции типа Inline команда создания функции будет иметь следующий вид:

```
CREATE FUNCTION [ owner_name. ] function_name
( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] } [ ....n
] ] )
RETURNS TABLE
[ WITH < function_option > [ [,] ...n ] ]
[ AS ]
RETURN [ ( ] select-stmt [ ) ]
```

Как видно из синтаксиса команды, функция представляет собой выражение `SELECT` и возвращает только один параметр типа `Table`. Следующий запрос создает функцию `MyFunc2`, которая выбирает из таблицы `Employees` всех работников, проживающих в каком-либо городе. По умолчанию используется Лондон:

```
USE NORTHWIND
ALTER FUNCTION MyFunc2 (@City AS VARCHAR(10) = 'London')
RETURNS TABLE
AS
RETURN
SELECT TitleOfCourtesy, FirstName, LastName FROM Employees
WHERE City = @City
```

Вызов функции будет осуществляться следующим образом:

```
SELECT * FROM dbo.MyFunc2('Tacoma')
```

Для получения значения по умолчанию необходимо выполнить следующий запрос:

```
SELECT * FROM dbo.MyFunc2(Default)
```

Синтаксис функции типа Multistatement приведен ниже:

```
CREATE FUNCTION [ owner_name. ] function_name
( [ { @parameter_name [AS] scalar_parameter_data_type [ = default ] } [ ....n
] ] )
RETURNS @return_variable TABLE < table_type_definition >
[ WITH < function_option > [ [,] ...n ] ]
[ AS ]
BEGIN
function_body
RETURN
END
```

```
< function_option > ::=
{ ENCRYPTION | SCHEMABINDING }
< table_type_definition > :: =
( { column_definition | table_constraint } [ ....n ] )
```

Для изменения пользовательских функций используется команда ALTER FUNCTION. Для скалярных функций она описывается следующим образом:

```
ALTER FUNCTION [ owner_name. ] function_name
( [ { @parameter_name scalar_parameter_data_type [ = default ] } [ ....n ] ] )
RETURNS scalar_return_data_type
[ WITH < function_option> [....n] ]
[ AS ]
BEGIN
function_body
RETURN scalar_expression
END
```

Пользуясь этим синтаксисом, можно изменить функцию MyFunc1 так, чтобы вычислялось не среднее значение, а половина суммы всех параметров:

```
ALTER FUNCTION MyFunc1 (@Val1 TINYINT, @Val2 TINYINT, @Val3 TINYINT)
RETURNS DECIMAL (5,3)
AS
BEGIN
DECLARE @Average DECIMAL (5,3)
SET @Average=((@Val1+@Val2+@Val3)/2)
IF @Average > 5 BEGIN
SET @Average=@Average*2
END
RETURN (@Average)
END
```

Триггеры

Триггер представляет собой специальный тип хранимой процедуры, вызывающейся при наступлении некоторого события. Триггер вызывается автоматически при вставке, обновлении и удалении записей. В теле триггера могут производиться вызовы других таблиц и могут использоваться комплексные запросы на языке Transact-SQL. Триггер выполняется в контексте одной транзакции, и при возникновении каких-либо ошибок производится автоматический откат.

Триггеры в SQL Server 2000 классифицируются по типу отслеживаемого события:

- INSERT TRIGGER – триггеры этого типа вызываются при попытке вставки новой записи в таблицу;

- UPDATE TRIGGER – триггеры данного типа вызываются в том случае, когда производится попытка обновления записи (или набора записей);
- DELETE TRIGGER – данный триггер вызывается при осуществлении попытки удаления записи из таблицы базы данных.

Также триггеры могут классифицироваться по времени выполнения:

- INSTEAD OF – триггеры выполняются взамен пользовательских действий, то есть команда пользователя не выполняется, а вместо нее выполняется триггер;
- AFTER – стандартный тип триггера, который вызывается после того, как были выполнены команды INSERT, UPDATE или DELETE.

Для создания триггера используется команда CREATE TRIGGER, имеющая следующий синтаксис:

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH ENCRYPTION ]
{
{ { FOR | AFTER | INSTEAD OF } { [ INSERT ] [ , ] [ UPDATE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS
[ { IF UPDATE ( column )
[ { AND | OR } UPDATE ( column ) ]
[ ...n ]
| IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
{ comparison_operator } column_bitmask [ ...n ]
} ]
sql_statement [ ...n ]
}
}
```

В параметре trigger_name указывается имя триггера. После оператора ON указывается имя таблицы или модифицируемого представления, с которым будет связан создаваемый триггер. Параметр WITH ENCRYPTION указывает серверу на необходимость шифрования кода триггера, содержащегося в таблице Syscomments.

Группа операторов { {FOR | AFTER | INSTEAD OF} {[INSERT] [,] [UPDATE]} определяет тип триггера и событие, при возникновении которого он вызывается. После ключевого слова FOR указывается тип триггера. Если тип не указан, то по умолчанию принимается определение AFTER. Параметр NOT FOR REPLICATION запрещает выполнение триггера во время процесса репликации.

Конструкция [{IF UPDATE (column) [{ AND | OR } UPDATE (column)] позволяет триггеру отслеживать модификацию определенных столбцов таблицы. В параметре column указывается имя столбца. Используя операторы AND и OR, можно опре-

делить логические условия, при соответствии которым будет производиться выполнение триггера.

Конструкция IF (COLUMNS_UPDATED () {bitwise_operator} updated_bitmask) { comparison_operator } column_bitmask [...n], фактически, выполняет те же функции, что и блок IF UPDATE. Но при изменении указанных столбцов этот блок возвращает битовую маску измененных столбцов, оперируя с которой определим, были ли изменены указанные столбцы. Параметр bitwise_operator указывает на то, что в данном месте должна производиться какая-либо побитовая операция со значением, возвращенным функцией. В параметре updated_bitmask указывается целочисленное значение в десятичном исчислении, представляющее собой маску анализа изменяемости столбцов.

Например, если необходимо выяснить, были ли изменены первый и третий столбцы, битовая маска примет вид 0101 (5 в десятичном исчислении), и команда будет записана следующим образом:

```
(COLUMNS_UPDATED ( ) & 5)
```

Параметр comparison_operator содержит оператор сравнения, предназначенный для проверки реального изменения отслеживаемых полей. Параметр column_bitmask представляет собой значение в десятичном формате, с которым сравнивается результат побитовой операции. Например, для того чтобы выяснить, были ли изменены первый и третий столбцы, необходимо использовать следующее условие:

```
(COLUMNS_UPDATED ( ) & 5) = 5
```

Теперь нужно рассмотреть простой пример создания триггера. Сначала потребуется создать таблицу TrigTable, имеющую несколько полей, и заполнить ее значениями.

```
USE TestDB
CREATE TABLE TrigTable
(
Col1 TINYINT PRIMARY KEY IDENTITY(1,1),
Col2 TINYINT,
Col3 TINYINT,
Col4 TINYINT
)
```

Теперь можно определить триггер, вызывающийся после вставки значения в таблицу и заменяющий жестко устанавливающий нулевое значение для четвертого поля:

```
CREATE TRIGGER MyTRIGGER ON TrigTable
AFTER INSERT
AS
DECLARE @RecID TINYINT
SELECT @RecID = Col1 FROM Inserted
UPDATE TrigTable SET Col4 = 0 WHERE Col1 = @RecID
```

Следующий запрос добавляет в таблицу новую запись.

```
INSERT INTO TrigTable (Col2, Col3, Col4) VALUES (22,22,22)
```

Чтобы обновить значение поля, был получен уникальный номер записи, хранящийся во временной таблице **Inserted**. Помимо таблицы **Inserted**, существует таблица **Deleted**, содержащая удаленные данные. Основываясь на этом, можно создать триггер **MyTRIGGER2**, который будет вызываться при вставке значения в таблицу **TrigTable** и заменять его нулевым.

```
CREATE TRIGGER MyTRIGGER2 ON TrigTable
INSTEAD OF INSERT
AS
INSERT INTO TrigTable (Col2, Col3, Col4) VALUES (0,0,0)
```

После этого следует создать триггер **MyTRIGGER3**, который будет анализировать введенные во второй и третий столбцы значения и на их основе формировать значение четвертого столбца:

```
USE TestDB
CREATE TRIGGER MyTRIGGER3 ON TrigTable
FOR INSERT, UPDATE
AS
DECLARE @RecID TINYINT
DECLARE @RC2 TINYINT
DECLARE @RC3 TINYINT
SELECT @RecID = Col1 FROM Inserted
SELECT @RC2 = Col2 FROM Inserted
SELECT @RC3 = Col3 FROM Inserted
IF ( COLUMNS_UPDATED ()&6)=6 BEGIN
IF (@RC3 > 5 and @RC2 > 5) BEGIN
UPDATE TrigTable SET Col4 = 0 WHERE Col1 = @RecID
END
IF (@RC3 < 5 and @RC2 < 5) BEGIN
UPDATE TrigTable SET Col4 = 1 WHERE Col1 = @RecID
END
END
```

Для анализа изменений значений второго и третьего столбцов была использована функция **COLUMNS_UPDATED**, с которой была использована маска **0110**. При переводе в десятичный формат получится значение **6**. Если второй и третий столбцы были изменены, то будет производиться анализ введенных значений, на основании которых сформируется значение четвертого поля.

Следующие команды тестируют поведение триггера:

```
INSERT INTO TrigTable (Col2, Col3) VALUES (4,4)
UPDATE TrigTable SET Col2 = 6, Col3 = 6 WHERE Col1 = 46
```

Для того чтобы посмотреть связанные с таблицей триггеры, можно воспользоваться утилитой **SQL Server Enterprise Manager**. В результате выполнения ко-

манды контекстного меню таблицы All Tasks ▸ Manage Triggers будет отображено окно, приведенное на рис. 9.43.

Рис. 9.43. Окно администрирования триггеров

В списке Name отображается список триггеров. В нем же можно выбрать шаблон создания триггера. Кнопка Check Syntax инициирует запуск механизма проверки синтаксиса тела триггера, содержащегося в поле Text. Кнопка Delete, соответственно, предназначена для удаления триггера.

Для изменения триггера предназначена команда ALTER TRIGGER, которая имеет следующий синтаксис:

```
ALTER TRIGGER trigger_name
ON ( table | view )
[ WITH ENCRYPTION ]
{
{ ( FOR | AFTER | INSTEAD OF ) { [ DELETE ] [ , ] [ INSERT ] [ , ] [ UPDATE ]
}
[ NOT FOR REPLICATION ]
AS
sql_statement [ ...n ]
}
|
{ ( FOR | AFTER | INSTEAD OF ) { [ INSERT ] [ , ] [ UPDATE ] }
[ NOT FOR REPLICATION ]
AS
{ IF UPDATE ( column )
[ { AND | OR } UPDATE ( column ) ]
[ ...n ]
| IF ( COLUMNS_UPDATED ( ) { bitwise_operator } updated_bitmask )
{ comparison_operator } column_bitmask [ ...n ]
```

```
}
sql_statement [ ...n ]
}
}
```

При помощи этой команды можно зашифровать тело триггера MyTRIGGER.

```
ALTER TRIGGER MyTRIGGER ON TrigTable WITH ENCRYPTION
AFTER INSERT
AS
DECLARE @RecID TINYINT
SELECT @RecID = Col1 FROM Inserted
UPDATE TrigTable SET Col4 = 0 WHERE Col1 = @RecID
```

Для удаления триггера используется команда DROP TRIGGER. В качестве примера можно удалить триггер MyTRIGGER3:

```
DROP TRIGGER MyTRIGGER3
```

Обработка ошибок

Реализуя логику приложения, программист также должен предусмотреть реакцию на возникающие ошибки, например на ошибку в триггере или хранимой процедуре. При помощи команды RAISERROR можно вызвать исключение, предоставляющее код и текст сообщения об ошибке. Данное сообщение может возвращаться приложению в качестве стандартного сообщения об ошибке. Для этого используется объект Error, который обладает свойствами, содержащими информацию об ошибке:

- Свойство Description содержит описание сообщения об ошибке.
- Свойство Number содержит уникальное значение типа Long, идентифицирующее возникшую ошибку.
- Свойство Source возвращает строку, содержащую имя объекта, вызвавшего ошибку.
- Свойство SQLState содержит код ошибки, возникшей при выполнении SQL-запроса.
- Свойство NativeError возвращает код ошибки, определенный провайдером. При помощи этого свойства можно передавать приложению сообщения об ошибках для дальнейшей обработки.

Когда на стороне провайдера возникает ошибка, информация о ней помещается в коллекцию Errors объекта Connection.

SQL Server 2000 предоставляет два механизма возвращения информации о возникающих ошибках. Для этого могут использоваться объекты Errors, которые были рассмотрены ранее, или объекты Messages, которые возвращают лишь сообщения об ошибках.

Следует рассмотреть механизм сообщений об ошибках. Для генерирования сообщения об ошибке, как уже было отмечено, необходимо использовать команду RAISERROR, имеющую следующий синтаксис:

```
RAISERROR ( { msg_id | msg_str } { , severity , state }
[ , argument [ ,...n ] ] )
[ WITH option [ ,...n ] ]
```

В параметре msg_id содержится номер сообщения об ошибке, определенного пользователем и содержащегося в таблице Sysmessages. Пользовательские сообщения об ошибках должны иметь номер больше 50 000, так как с номером до 50 000 размещаются сообщения сервера, а номер 50 000 присваивается любому сообщению об ошибке, генерируемому по ходу выполнения запроса и не зарегистрированному в таблице Sysmessages.

Параметр msg_str содержит сообщение об ошибке. Строка может иметь форматирование в стиле команды PRINTF языка C. Сообщение об ошибке может занимать до четырехсот символов. В параметре severity указывается уровень важности сообщения. Уровень важности сообщения об ошибке в диапазоне от 20 до 25 трактуется как фатальный. При получении сообщения с таким уровнем клиентское приложение разрывает связь с сервером, а в журнал ошибок сервера заносится соответствующая отметка. Параметр severity может принимать значение в диапазоне от 0 до 18 для любого пользователя, а в диапазоне от 19 до 25 — только для членов роли sysadmin.

В параметре state указывается число в диапазоне от 1 до 127, определяющее состояние ошибки. По умолчанию свойству присваивается единичное значение. После оператора WITH указывается список параметров:

- Параметр LOG указывает, что запись с сообщением об ошибке будет помещена в журнал ошибок.
- Параметр NOWAIT указывает, что сообщение об ошибке будет немедленно отправлено клиентскому приложению.
- Параметр SETERROR задает значение параметра msg_id.

Для регистрации сообщения об ошибке в таблице Sysmessages следует воспользоваться хранимой процедурой sp_addmessage. Команда имеет следующий вид:

```
sp_addmessage [ @msgnum = ] msg_id ,
[ @severity = ] severity ,
[ @msgtext = ] 'msg'
[ , [ @lang = ] 'language' ]
[ , [ @with_log = ] 'with_log' ]
[ , [ @replace = ] 'replace' ]
```

Параметр [@lang =] позволяет определить язык сообщения. Если параметр не определен, то будут использованы языковые настройки данной сессии. Параметр [@replace =] используется в тех случаях, когда необходимо изменить текст сообщения об ошибке. Следующий запрос добавляет в таблицу Sysmessages сообщение «Test message» с номером 50005:

```
sp_addmessage 50005, 16, 'Test message'
```

Теперь можно переопределить триггер MyTRIGGER3. В него будет добавлен вызов сообщения об ошибке, когда значения, вводимые в поля Col2 и Col3, будут соответствовать определенным условиям:

```
ALTER TRIGGER MyTRIGGER3 ON TrigTable
FOR INSERT, UPDATE
AS
DECLARE @RecID TINYINT
DECLARE @RC2 TINYINT
DECLARE @RC3 TINYINT
SELECT @RecID = Col1 FROM Inserted
SELECT @RC2 = Col2 FROM Inserted
SELECT @RC3 = Col3 FROM Inserted
IF ( COLUMNS_UPDATED ()&6) = 6 BEGIN
IF (@RC3 = 5 and @RC2 = 5) BEGIN
RAISERROR ('Simple Test Error',16,10)
ROllBACK
END
IF (@RC3 = 6 and @RC2 = 6) BEGIN
RAISERROR (50005,16,10)
ROllBACK
END
END
```

Вызов сообщения об ошибке может производиться при помощи извлечения сообщения по его номеру из таблицы Sysmessages или при помощи прямого создания сообщения. После того как вызывается сообщение об ошибке, производится откат сделанных изменений при помощи команды ROLLBACK. Следующие запросы изменяют значения полей и добавляют новые записи в таблицу. Они позволяют протестировать работу измененного триггера:

```
UPDATE TrigTable SET Col2 = 6, Col3 = 6 WHERE Col1 = 46
INSERT INTO TrigTable (Col2,Col3) VALUES (5,5)
```

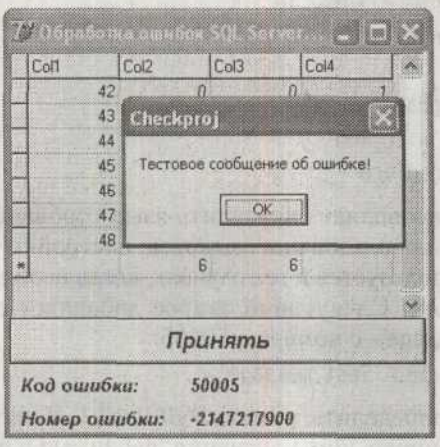


Рис. 9.44. Обработка сообщений об ошибках SQL Server 2000

Теперь следует создать новый проект и добавить в него модуль данных. В модуле данных должны быть размещены компоненты TADOConnection, TADOTable и TDataSource. Затем потребуется настроить соединение с таблицей TrigTable, входящей в состав базы данных TestDB.

На главной форме приложения нужно расположить компонент TDBGrid, который будет связан с компонентом TADOTable. Также потребуется одна кнопка. На рис. 9.44 приведено окно приложения, а в листинге 9.3 — его код.

Листинг 9.3 Обработка сообщений об ошибках

```
procedure TCheckForm.PostBtnClick(Sender: TObject);
var I : Integer;
begin
  with ConnectionDM do begin
    try
      ADOTable1.Post;
    except
      Label2.Caption:=IntToStr(ADOConnection1.Errors.Item[0].NativeError);
      Label3.Caption:=IntToStr(ADOConnection1.Errors.Item[0].Number);
      if ADOConnection1.Errors.Item[0].NativeError = 50000 then
        ShowMessage('Простое тестовое сообщение об ошибке!');
      if ADOConnection1.Errors.Item[0].NativeError = 50005 then
        ShowMessage(Тестовое сообщение об ошибке!');
    end;
  end;
end;
```

Как видно из приведенного в листинге 9.3 кода, будет отслеживаться код сообщения об ошибке, содержащийся в свойстве NativeError объекта Error. После получения кода ошибки он будет соответствующим образом обработан, и на экран будет выведено сообщение об ошибке.

Представления

Представление является виртуальной таблицей, содержание которой определяется результатом выполнения запроса. Данные представления не хранятся физически на диске, а запрашиваются из таблиц каждый раз, когда представление активируется. Представление может состоять из одной или нескольких таблиц. Также в представления могут быть включены данные от связанных серверов.

Представления, как правило, используются для упрощения работы пользователей, предоставляя им только необходимые данные. Также представления могут использоваться как составляющие части механизма безопасности, позволяя пользователям получать доступ к усеченной версии таблиц. Для создания представления используется команда CREATE VIEW:

```
CREATE VIEW [ < database_name > . ] [ < owner > . ] view_name [ ( column [
....n ] ) ]
[ WITH < view_attribute > [ ....n ] ]
AS
select_statement
[ WITH CHECK OPTION ]
< view_attribute > ::=
{ ENCRYPTION | SCHEMABINDING | VIEW_METADATA }
```

В параметре **database_name** указывается имя базы данных, в параметре **owner** задается владелец представления, а в параметре **view_name** — имя представления. Параметр **ENCRYPTION** указывает на необходимость шифрования тела представления, хранящегося в системной таблице. Параметр **SCHEMABINDING** предписывает серверу выполнять связывание структуры представления со структурами объектов, на базе которых создается представление. Таким образом, структура связанных объектов не сможет быть изменена, если вносимые изменения будут оказывать влияние на структуру представления. Например, связанная таблица не сможет быть удалена до тех пор, пока не будет отключен этот параметр.

Параметр **VIEW_METADATA** позволяет возвращать метаданные представлений. Он используется при работе с технологиями DBLIB, ODBC и OLE DB. Параметр **WITH CHECK OPTION** гарантирует, что в представление не будут добавлены записи, которые не соответствуют условию отбора **WHERE**.

В следующем примере создается простое представление **SimpleView** по полям **Col1** и **Col2** таблицы **TrigTable**. Это представление будет содержать записи, имеющие значение столбца **Col1** больше, чем 45:

```
USE TestDB
CREATE VIEW SimpleView (Val1, Val2) AS
SELECT Col1, Col2 FROM TrigTable
WHERE Col1 > 45
```

Представление используется практически так же, как обычная таблица. С помощью команды **SELECT** можно выбрать один или несколько столбцов из представления:

```
SELECT * FROM SimpleView
```

Теперь нужно создать комбинированное представление, объединяющее данные из таблиц **Territories**, **Employees** и **EmployeeTerritories** базы данных **Northwind**. Предположим, что необходимо получить территории, на которых осуществляют свою деятельность работники компании:

```
USE NORTHWIND
CREATE VIEW EmplTerrView AS
SELECT a.TitleOfCourtesy AS Title, a.FirstName AS First_Name,
a.LastName AS Last_Name, c.TerritoryDescription AS Territory
FROM Employees a, EmployeeTerritories b, Territories c
WHERE a.EmployeeID = b.EmployeeID AND b.TerritoryID = c.TerritoryID
```

Для того чтобы выяснить, на каких территориях работают все служащие, следует использовать несложный запрос:

```
SELECT * FROM EmplTerrView
```

Если нужно выяснить, на каких территориях работал Роберт Кинг, запрос следует видоизменить:

```
SELECT Territory FROM EmplTerrView
WHERE First_Name = 'Robert' AND Last_Name = 'King'
```

Для изменения представления используется команда **ALTER VIEW**, имеющая следующий синтаксис:

```
ALTER VIEW [ < database_name > . ] [ < owner > . ] view_name [ ( column [
,...n ] ) ]
[ WITH < view_attribute > [ ,...n ] ]
AS
select_statement
[ WITH CHECK OPTION ]
< view_attribute > ::=
{ ENCRYPTION | SCHEMABINDING | VIEW_METADATA }
```

При помощи этой команды можно ввести контроль вставки новых записей в созданное представление при помощи условия **WHERE**. Если вставляемое значение не будет соответствовать заданному условию, то запись не будет добавляться. Для этого потребуется изменить представление **SimpleView**:

```
USE TestDB
GO
ALTER VIEW SimpleView (Val1, Val2) AS
SELECT Col1, Col2 FROM TrigTable
WHERE Col1 > 45 AND COL2 > 30
WITH CHECK OPTION
GO
```

Следующий запрос пытается добавить новую запись со значением поля **Col2**, которое не удовлетворяет условию. Из-за этого новая запись не будет добавлена в представление:

```
INSERT INTO SimpleView (Val2) VALUES (20)
```

Администрирование сервера

Процесс администрирования сервера включает в себя комплекс мероприятий, направленных на поддержание сервера в работоспособном состоянии. Администратор ограничивает права пользователей на доступ к данным и операции с ними и определяет политики безопасности. Он должен заботиться о проведении резервных сохранений базы данных и правильном хранении архивных записей.

В этом разделе будут рассмотрены ключевые аспекты администрирования сервера базы данных, без владения которыми разрабатывать надежные при-

ложения очень сложно. Речь пойдет о резервном копировании и восстановлении баз данных, о работе с учетными записями пользователей, об их типах и о делегировании прав на объекты базы данных.

Резервное копирование и восстановление

SQL Server 2000 позволяет выполнять операцию резервного копирования, не отключая работающих с базой пользователей. В этом случае процесс осуществляется ступенчато:

1. SQL Server 2000 создает в журнале транзакций контрольную точку и ожидает, пока все завершенные транзакции будут сохранены в базе данных.
2. После этого сервер отмечает последний номер записи журнала транзакций LSN (Log Sequence Number). LSN представляет собой порядковый уникальный номер транзакции в журнале транзакций, присваиваемый ей автоматически.
3. Производится процесс создания резервной копии данных. Все операции с базой данных фиксируются в журнале транзакций.
4. После завершения создания резервной копии данных производится создание резервной копии журнала транзакций, в который включаются все записи, существующие в нем на момент старта процесса.
5. По окончании резервного копирования журнала транзакций процесс резервного копирования завершается.

SQL Server 2000 предоставляет администратору несколько типов резервного копирования:

- полная копия (Database backups);
- дифференциальная копия (Differential backups);
- копия журнала транзакций (Transaction log backups);
- резервное копирование файлов и групп файлов (File and filegroup backups).

Полная копия базы данных (Database backup), как следует из ее названия, является полной копией всей информации, хранящейся в базе. Полное копирование имеет свои преимущества и недостатки. К преимуществам этого метода можно отнести простоту восстановления базы данных, так как вся информация содержится в одном месте. Но есть и недостаток — значительный объем времени, занимаемый процессом резервного копирования. В некоторой степени недостатки полного резервного копирования исправляет дифференциальное копирование.

Дифференциальное резервное копирование создает резервную копию всех данных, которые были добавлены или изменены со времени последнего полного резервного копирования. Как правило, дифференциальное копирование отнимает значительно меньше ресурсов, чем полное. Каждая страница данных имеет специальный флаг, называемый флагом архивирования. При создании полной копии данный флаг сбрасывается. При изменении данных флаг

выставляется вновь. Процесс создания дифференциальной копии выбирает страницы с установленным флагом архивирования и помещает их в дифференциальную копию, но не сбрасывает флаг архивирования.

Суть операции создания резервной копии журнала транзакций состоит в создании резервной копии той части журнала транзакций, в которой были записаны изменения, внесенные в базу со времени создания полной или дифференциальной копии. Резервное копирование журнала транзакций также является инкрементальным, но требует гораздо меньше ресурсов по сравнению с дифференциальным копированием.

В резервную копию журнала транзакций не включаются данные о схеме и файловой структуре базы данных. Когда SQL Server завершает процесс резервного копирования журнала транзакций, он автоматически отсекает и удаляет ненужный блок журнала. Этот блок содержит сведения о завершенных транзакциях, содержащихся в базе данных и более не имеющих необходимости хранится в журнале.

Используя перечисленные выше типы резервного копирования, можно предложить хорошую схему резервного копирования для интенсивно используемой базы данных. Полное резервное копирование производится каждую ночь, дифференциальное резервное копирование — каждый час, а резервное копирование журнала транзакций — каждые десять минут.

При этой схеме в случае любого сбоя будут потеряны данные всего за десять минут работы. В этом случае резервное копирование будет состоять в последовательном восстановлении полной копии данных, дифференциальной копии и копии журнала транзакций.

Последним типом резервного копирования является резервное копирование отдельных файлов и их групп. Как уже было отмечено ранее, часть таблиц базы данных или полей типа Text, NText и Image может храниться в отдельных файлах. Например, различные словари могут храниться в отдельной группе файлов. Часто имеет смысл включать в копию только те таблицы, которые заведомо будут изменены. Как раз для этих целей и предназначено резервное копирование групп файлов.

Резервное копирование базы данных можно выполнять либо SQL-запросами, либо при помощи утилиты SQL Server Enterprise Manager. Команда создания полной резервной копии базы данных на языке Transact-SQL имеет следующий синтаксис:

```
BACKUP DATABASE { database_name | @database_name_var }
TO < backup_device > [ ....n ]
[ WITH
[ BLOCKSIZE = { blocksize | @blocksize_variable } ]
[ [ . ] DESCRIPTION = { 'text' | @text_variable } ]
[ [ . ] DIFFERENTIAL ]
[ [ . ] EXPIREDATE = { date | @date_var }
| RETAINDAYS = { days | @days_var } ]
[ [ . ] PASSWORD = { password | @password_variable } ]
```

```
[ [ . ] FORMAT | NOFORMAT ]
[ [ . ] { INIT | NOINIT } ]
[ [ . ] MEDIADESCRIPTION = { 'text' | @text_variable } ]
[ [ . ] MEDIANAME = { media_name | @media_name_variable } ]
[ [ . ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
[ [ . ] NAME = { backup_set_name | @backup_set_name_var } ]
[ [ . ] { NOSKIP | SKIP } ]
[ [ . ] { NOREWIND | REWIND } ]
[ [ . ] { NOUNLOAD | UNLOAD } ]
[ [ . ] RESTART ]
[ [ . ] STATS [ = percentage ] ]
]
```

В параметре database_name указывается имя базы данных, для которой будет создана резервная копия. В параметре @database_name_var указывается переменная строкового типа, в которую будет помещено имя базы данных. Командой TO < backup_device > определяется логическое или физическое устройство, в котором будет сохранена резервная копия. В качестве физических устройств могут быть указаны жесткий диск или магнитная лента:

```
{ DISK | TAPE } =
{ 'physical_backup_device_name' | @physical_backup_device_name_var }
```

После оператора WITH указываются параметры резервного копирования. В параметре BLOCKSIZE задается физический размер блока данных, сохраняемого на диске. В параметре DESCRIPTION указывается описание создаваемой резервной копии.

Параметр DIFFERENTIAL указывает на то, что должна быть создана дифференциальная резервная копия данных. В параметре EXPIREDATE указывается дата, по истечении которой данные на носителе будут считаться устаревшими и их можно будет перезаписать. Параметр RETAINDAYS определяет количество дней, по истечении которых база данных устареет.

В параметре PASSWORD указывается пароль, ограничивающий доступ к резервному набору данных. Параметры FORMAT/NOFORMAT позволяют форматировать набор резервных носителей. При указании данного параметра на каждый носитель записывается новый заголовок. Параметры INIT/NOINIT указывают серверу на необходимость перезаписи данных, содержащихся на носителе, с сохранением заголовков. В случае, если срок годности данной резервной копии не истек, SQL Server не будет производить процесс резервного копирования. Также процесс не будет начат и в том случае, когда имя носителя не соответствует имени носителя, указанному в команде BACKUP.

При помощи параметра MEDIADESCRIPTION можно ввести описание набора резервных носителей, которое будет указано в их заголовке. Параметр MEDIANAME позволяет определить имя набора резервных носителей. Используя параметр MEDIAPASSWORD, можно назначить пароль для доступа к набору носителей. Если набор носителей защищен паролем, то для проведения с ним каких-либо операций потребуется вводить указанный пароль.

В параметре NAME указывается имя резервного набора данных. Данный параметр имеет смысл использовать в том случае, когда на носителе записано несколько наборов данных. Параметр NO_TRUNCATE указывает серверу, что не следует урезать журнал транзакций после выполнения резервного копирования. Параметры NO_LOG|TRUNCATE_ONLY позволяют урезать журнал транзакций после завершения операции резервного копирования.

Параметры SKIP/NOSKIP позволяют включить или отключить механизм проверки информации, содержащейся на носителе перед записью. Параметр RESTART используется для определения необходимости продолжения записи на ленту в том случае, если запись на нее была прервана по каким-либо причинам. Запись на ленту начинается с того места, на котором она была остановлена ранее. Обычно этот параметр используется при смене лент, когда база не помещается на один носитель. Оператор STATS используется для получения статистической информации о ходе процесса резервного копирования.

Следующий запрос создает полную копию базы данных TestDB:

```
USE TestDB
GO
BACKUP DATABASE TestDB
TO DISK = 'D:\Backup\MSSQLFullBack.dat'
WITH NAME = 'Full Backup of TestDB'
GO
```

Для создания дифференциальной копии базы данных запрос придется несколько изменить:

```
USE TestDB
GO
BACKUP DATABASE TestDB
TO DISK = 'D:\Backup\MSSQLDifBack.dat'
WITH NAME = 'Differential Backup of TestDB', DIFFERENTIAL
GO
```

Для создания резервной копии журнала транзакций следует воспользоваться командой, имеющей следующий синтаксис:

```
BACKUP LOG { database_name | @database_name_var }
{
TO < backup_device > [ ....n ]
```

На ее основе можно создать запрос, выполняющий копирование журнала транзакций базы данных TestDB:

```
USE TestDB
GO
BACKUP LOG TestDB
TO DISK = 'D:\Backup\MSSQLLogBack1.dat'
WITH NAME = 'Log Backup of TestDB'
GO
```

Команда создания копий файлов и групп файлов имеет следующий синтаксис:

```
BACKUP DATABASE { database_name | @database_name_var }
< file_or_filegroup > [ ....n ]
TO < backup_device > [ ....n ]
< file_or_filegroup > ::=
{
FILE = { logical_file_name | @logical_file_name_var }
|
FILEGROUP = { logical_filegroup_name | @logical_filegroup_name_var }
}
```

В параметре FILE указывается имя файла, входящего в группу, а в параметре FILEGROUP — имя группы.

Следующий запрос создает резервную копию группы файлов SECONDARYGROUP базы данных TestDB. В нее будут входить файлы Test1Dat3 и Test1Dat4:

```
BACKUP DATABASE TestDB
FILE = 'Test1Dat3',
FILEGROUP = 'SECONDARYGROUP',
FILE = 'Test1Dat4',
FILEGROUP = 'SECONDARYGROUP'
TO DISK = 'D:\Backup\MSSQLFileGroupBack.dat'
WITH NAME = 'Log Backup of File Group Secondary'
GO
```

Процесс резервного копирования базы данных можно выполнить при помощи утилиты SQL Server Enterprise Manager. Для этого нужно выполнить команду контекстного меню базы данных All Tasks ▸ Backup Database. Появится окно, показанное на рис. 9.45.

В окне расположены вкладки General и Options. На вкладке General находятся элементы, управляющие процессом резервного копирования, а на вкладке Options — дополнительные параметры процесса.

В списке Database выбирается база данных, которая будет резервироваться. В поле Name можно указать логическое имя резервной копии, а в поле Description — ее описание. В поле Backup можно выбрать тип создаваемой резервной копии. Возможные значения этого параметра приведены в списке:

- Значение Database – complete создает полную копию базы данных.
- Значение Database – differential создает дифференциальную копию базы данных.
- Значение Transaction log создает резервную копию журнала транзакций.
- Значение File and filegroup используется для создания резервной копии группы файлов.

Как правило, резервное копирование базы данных производится автоматически по заранее разработанному графику без участия системного администратора.

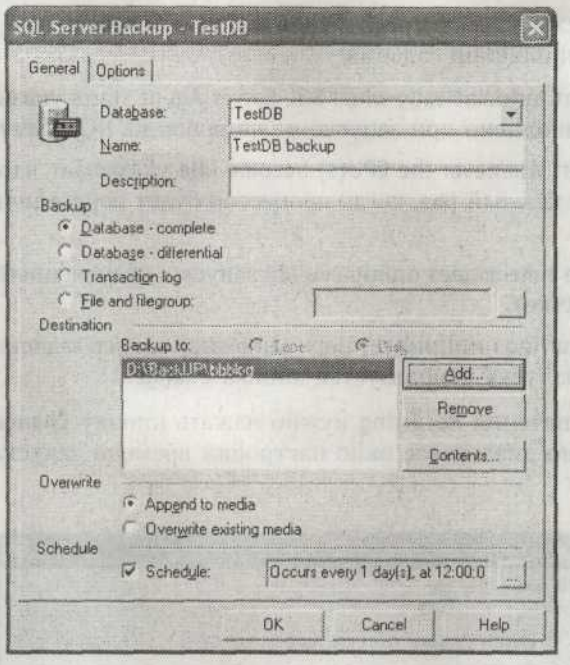


Рис. 9.45. Окно резервного копирования базы данных

В качестве примера использования утилиты можно создать полную копию базы данных. В поле Backup нужно выбрать значение Database – complete, в группе Destination потребуется нажать кнопку Add и указать путь к файлу резервной копии в появившемся диалоговом окне. Также нужно взвести флажок Schedule и нажать на кнопку, расположенную справа от поля ввода. В результате будет отображено окно, приведенное на рис. 9.46.

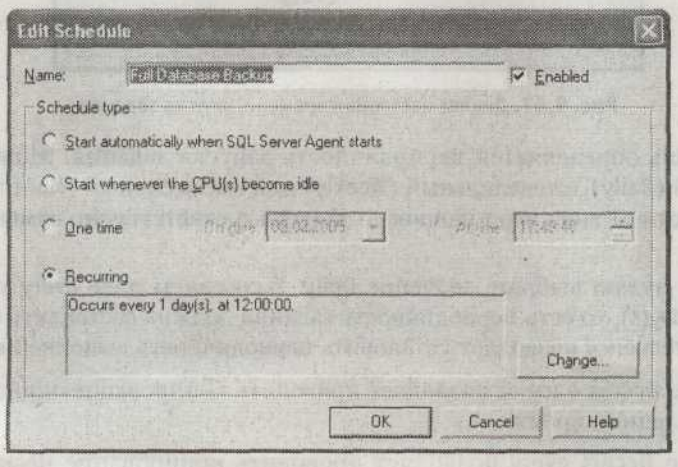


Рис. 9.46. Окно планировщика заданий

В поле Name нужно указать имя задания. Поле Schedule type позволяет выбрать режим выполнения задания:

- Значение Start automatically when SQL Server Agent starts указывает, что задание будет выполнено при запуске планировщика SQL Server Agent.
- Значение Start whenever the CPU(s) become idle указывает, что задание будет выполняться каждый раз, когда процессор будет переходить в режим простоя.
- Значение One time задает одноразовый запуск в назначенный день (On date) и время (On time).
- Значение Recurring инициирует периодический запуск задания. Для настройки времени запуска используется кнопка Change.

После выбора значения Recurring нужно нажать кнопку Change. В результате будет отображено диалоговое окно настройки времени запуска, приведенное на рис. 9.47.

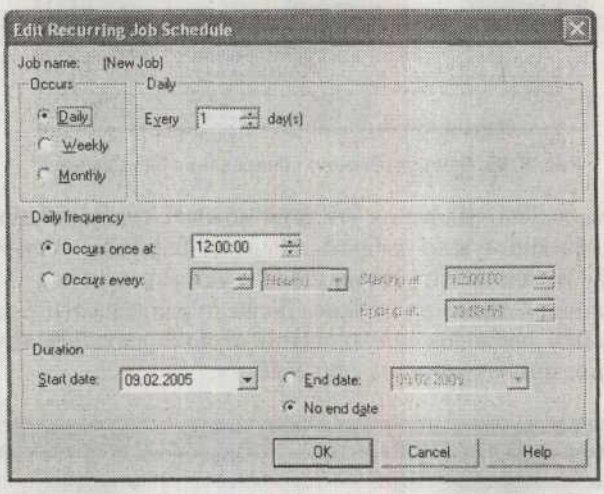


Рис. 9.47. Диалог настройки времени запуска задания

В поле Occurs определяется периодичность запуска задания. Можно задать ежедневный (Daily), еженедельный (Weekly) или ежемесячный (Monthly) запуск. В зависимости от типа периодичности запуска соответственно изменяется вид окна.

Для начала нужно выбрать значение Daily. В соседнем поле Every установите значение 1 day(s), то есть периодичность задания должна составлять один день. Поле Daily frequency позволяет установить периодичность выполнения задания:

- Значение Occurs once at позволяет проводить копирование один раз в день в определенное время.
- Значение Occurs every позволяет проводить копирование несколько раз в день с заданной периодичностью.

После выбора значения Occurs once at нужно установить время запуска, скажем, на 12 часов. После нажатия кнопки OK будет произведен процесс резервного копирования.

Несколько иначе происходит установка параметров для резервного копирования группы файлов. Для этого нужно выбрать значение File and filegroup, после чего станет доступной кнопка, находящаяся справа. Если нажать эту кнопку, то появится окно выбора групп файлов, приведенное на рис. 9.48.

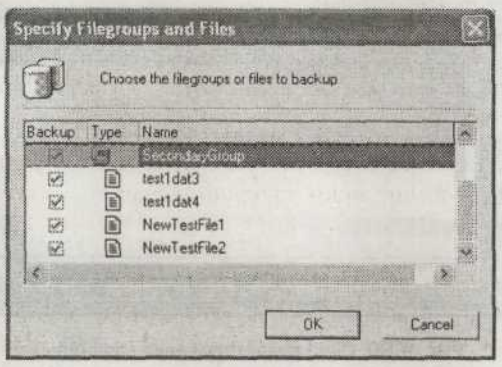


Рис. 9.48. Окно выбора групп файлов

В основном списке следует выбрать группу SecondaryGroup. В нее входит несколько файлов. В резервную копию можно включать как все файлы, входящие в группу, так и лишь некоторые из них. После нажатия на кнопку OK будет создано соответствующее задание.

Для того чтобы посмотреть список созданных заданий, необходимо выполнить команду меню Management ▸ SQL Server Agent ▸ Jobs. В списке Jobs будут представлены существующие задания. Для того чтобы посмотреть тип задания, следует выбрать его. На рис. 9.49 показано окно свойств задания.

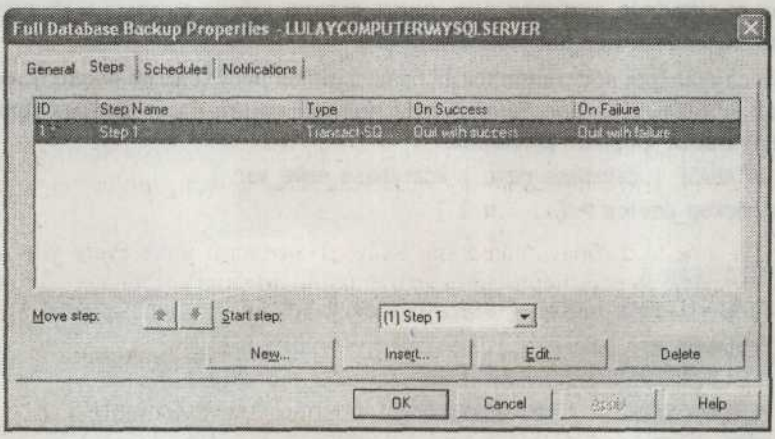


Рис. 9.49. Окно свойств задания, вкладка Steps

Задание может иметь несколько шагов. Если щелкнуть мышью на доступном шаге Step1, то появится окно редактирования задания, показанное на рис. 9.50.

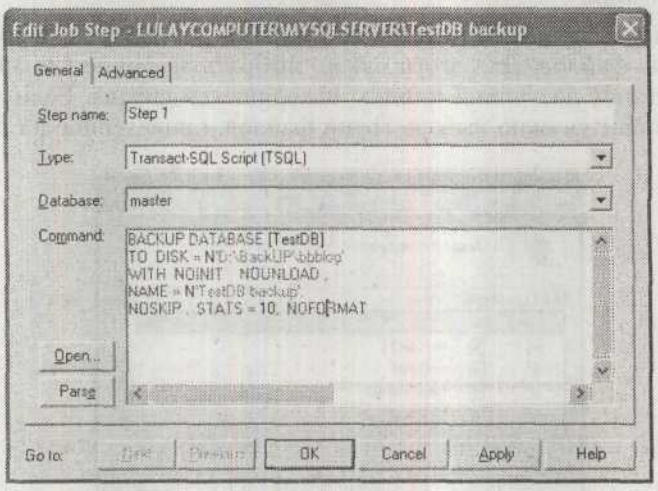


Рис. 9.50. Окно редактирования задания

При помощи этого окна можно проверить правильность сгенерированной команды и исправить что-либо, если возникнет такая необходимость. Настроить процесс резервного копирования довольно просто. Нужно лишь грамотно его спланировать, чтобы не перегружать систему и добиться приемлемого временного диапазона между сеансами резервирования.

В некоторых случаях может понадобиться произвести восстановление базы данных до более раннего состояния. Для восстановления базы данных из резервной копии можно воспользоваться языком Transact-SQL или средствами утилиты SQL Server Enterprise Manager. Перед началом восстановления базы данных необходимо установить в окне ее свойств опцию Members of db_owner, dbcreator, or sysadmin, которая ограничит доступ к базе данных и не позволит изменять ее в процессе восстановления.

В первом случае для восстановления базы данных необходимо воспользоваться командой RESTORE. Для восстановления полной копии базы данных применяется следующий формат команды:

```
RESTORE DATABASE { database_name | @database_name_var }
[ FROM < backup_device > [ ....n ] ]
[ WITH
[ RESTRICTED_USER ]
[ [ . ] FILE = { file_number | @file_number } ]
[ [ . ] PASSWORD = { password | @password_variable } ]
[ [ . ] MEDIANAME = { media_name | @media_name_variable } ]
[ [ . ] MEDIAPASSWORD = { mediapassword | @mediapassword_variable } ]
[ [ . ] MOVE 'logical_file_name' TO 'operating_system_file_name' ]
```

```
[ ....n ]
[ [ . ] KEEP_REPLICATION ]
[ [ . ] { NORECOVERY | RECOVERY | STANDBY = undo_file_name } ]
[ [ . ] { NOREWIND | REWIND } ]
[ [ . ] { NOUNLOAD | UNLOAD } ]
[ [ . ] REPLACE ]
[ [ . ] RESTART ]
[ [ . ] STATS [ = percentage ] ]
]
< backup_device > ::=
{
{ 'logical_backup_device_name' | @logical_backup_device_name_var }
| { DISK | TAPE } =
{ 'physical_backup_device_name' | @physical_backup_device_name_var }
}
```

В параметре database_name указывается псевдоним восстанавливаемой базы данных. Параметр <backup_device> определяет имя резервного носителя, с которого будет произведено восстановление. После ключевого слова WITH следует список опций восстановления базы данных. Ключевое слово RESTRICTED_USER разрешает доступ к восстанавливаемой базе данных только членам групп db_owner, dbcreator или sysadmin, то есть тем пользователям, которые не будут вносить изменения в базу данных во время ее восстановления, что исключает вероятность появления ошибок.

В параметре FILE указывается имя файла, расположенного на резервном носителе, который необходимо восстановить. С помощью параметра PASSWORD указывается пароль, необходимый для доступа к резервной копии, содержащейся на носителе. Параметр MEDIANAME позволяет указать имя резервного носителя, содержащего восстанавливаемую базу данных. Если указанное в данном параметре имя не совпадает с тем, что присвоено физическому носителю, процесс восстановления не будет запущен.

Для указания пароля, необходимого для доступа к резервному носителю, используется параметр MEDIAPASSWORD. По умолчанию база данных восстанавливается в тот же каталог, в котором она находилась на момент создания резервной копии. Параметр MOVE 'logical_file_name' TO 'operating_system_file_name' позволяет указать новое расположение файлов базы данных.

При использовании сложных стратегий резервного копирования восстановление базы данных усложняется. При использовании только полной резервной копии можно воспользоваться командой RESTORE DATABASE. Но если для восстановления используется дифференциальная копия или копия журнала транзакций, то восстановление базы данных потребует несколько большего числа операций.

Каждая резервная копия восстанавливается с помощью отдельной команды RESTORE. После завершения загрузки файлов базы данных и журнала транзакций производится автоматическое восстановление базы данных, сопровожда-

емое откатом всех незавершенных транзакций. Так как дифференциальные резервные копии могут содержать завершенные транзакции, которые не были завершены на момент создания полной резервной копии, то производить процесс полного восстановления базы данных нельзя до восстановления всех резервных копий базы данных.

По умолчанию сервер автоматически восстанавливает базу данных, что равносильно установке параметра RECOVERY. Параметр NORECOVERY запрещает автоматическое восстановление базы данных, то есть откат незавершенных транзакций произведен не будет. При использовании параметра STANDBY автоматического восстановления базы данных тоже не происходит, а база данных остается доступной пользователям только для чтения. Данный параметр используется для восстановления базы данных на резервном сервере.

Параметр REPLACE указывает серверу на необходимость перезаписи восстанавливаемой базы данных поверх существующей копии, если она существует. По умолчанию производится проверка, и если существует база с именем, указанным в параметре database_name, создание резервной копии не производится и выдается сообщение об ошибке.

Чтобы включить возможность продолжения восстановления базы данных с того места, на котором данный процесс был прерван, следует указать параметр RESTART. Параметр STATS указывает серверу на необходимость отображения статистической информации о ходе процесса восстановления базы данных.

Для восстановления резервной копии журнала транзакций используется команда RESTORE LOG, синтаксис которой приведен ниже:

```
RESTORE LOG { database_name | @database_name_var }
[ FROM < backup_device > [ ....n ] ]
[ WITH
. . .
[ [ . ] STOPAT = { date_time | @date_time_var }
| [ . ] STOPATMARK = 'mark_name' [ AFTER datetime ]
| [ . ] STOPBEFOREMARK = 'mark_name' [ AFTER datetime ]
]
]
```

Параметр STOPAT позволяет восстановить базу данных до определенного промежутка времени. Параметр STOPATMARK указывает серверу на необходимость проведения восстановления базы данных до определенной отметки. В восстановленную базу будет включена транзакция, в которой была указана данная метка. Параметр STOPBEFOREMARK указывает, что восстановление производится до точки восстановления, но транзакция, содержащая данную точку, не восстанавливается.

Следующий пример позволяет восстановить удаленную базу данных TestDB. База данных имеет полную, дифференциальную и последовательную копии:

```
RESTORE DATABASE TestDB
FROM DISK ='D:\BackUP\FullCopy'
WITH NORECOVERY
```

```
GO
RESTORE DATABASE TestDB
FROM DISK ='D:\BackUP\DiffCopy'
WITH NORECOVERY
GO
RESTORE LOG TestDB
FROM DISK ='D:\BackUP\LogCopy'
WITH FILE=2, RECOVERY
```

Из кода команды следует, что было произведено последовательное восстановление базы данных. В команде восстановления журнала транзакций был указан файл, из которого производилось восстановление. Дело в том, что в журнале содержались два файла, один из которых был устаревшим. Новый файл находился под индексом 2.

Теперь нужно рассмотреть процесс восстановления базы данных при помощи утилиты SQL Server Enterprise Manager. Для восстановления зарегистрированной базы данных нужно выбрать ее и в контекстном меню выполнить команду All Tasks ▸ Restore Database. Того же самого эффекта можно добиться при помощи команды главного меню Tools ▸ Restore Database. В результате будет отображено диалоговое окно, показанное на рис. 9.51.

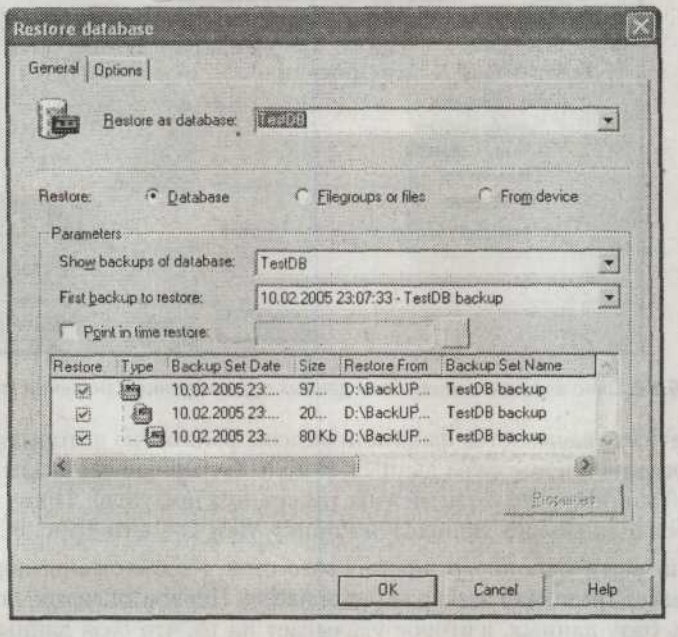


Рис. 9.51. Окно восстановления базы данных

В поле Restore as database указывается имя восстанавливаемой базы данных. В верхней части окна расположены переключатели, позволяющие выбрать расположение и состав источника резервной копии: Database, Filegroups or Files

и From device. По умолчанию выбрана опция Database. Этот вариант реализует самый простой способ восстановления базы данных — откат до определенной точки во времени, устанавливаемой опцией Point in time restore.

Из списка, расположенного в нижней части окна, выбирается ближайшая копия и производится процесс восстановления, для инициации которого необходимо нажать кнопку OK. Если произошел полный крах базы данных, ее восстановление будет необходимо произвести из группы файлов либо с соответствующего логического устройства. Имеет смысл рассмотреть подробно именно этот вариант как наиболее вероятный.

После выбора значения From device диалоговое окно изменит свой внешний вид (рис. 9.52).

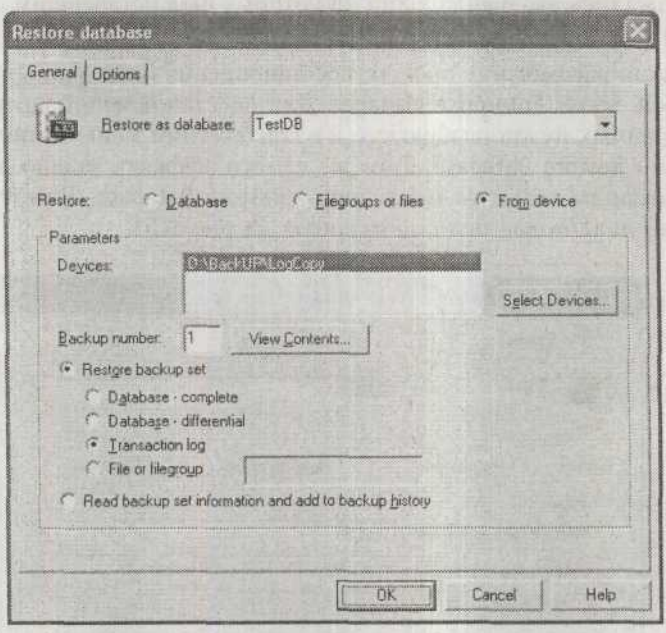


Рис. 9.52. Окно восстановления базы данных при выбранном значении From Device

В поле Devices указывается файл или список файлов, из которых будет произведено восстановление базы данных. В поле Backup number указывается номер резервной копии, содержащейся на резервном носителе. Посмотреть содержание носителя можно, используя кнопку View Contents (рис. 9.53).

В нижней части основного диалогового окна расположены переключатели Restore backup set и Read backup set information. Первое значение позволяет восстановить базу данных, а второе указывает на то, что база данных восстановлена не будет, но информация о ней будет занесена в журнал восстановления.

Как уже было отмечено ранее, при полном восстановлении базы данных восстанавливается полная копия базы данных, затем дифференциальная, а после нее — копия журнала транзакций. В процессе восстановления резервная

копия помещается в список Devices, затем выбирается соответствующий ей тип восстановления базы данных, и после этого нужно нажать кнопку OK. Эти действия следует повторить для каждой резервной копии.

Чтобы добавить резервную копию в список Devices, необходимо нажать на кнопку Select Devices. В результате будет отображено окно, показанное на рис. 9.54.

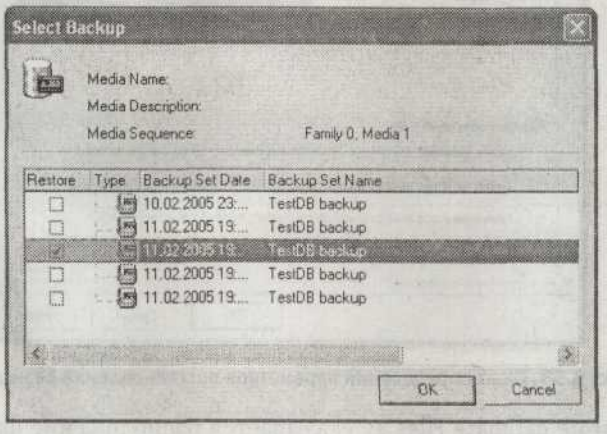


Рис. 9.53. Выбор пакета для восстановления базы данных

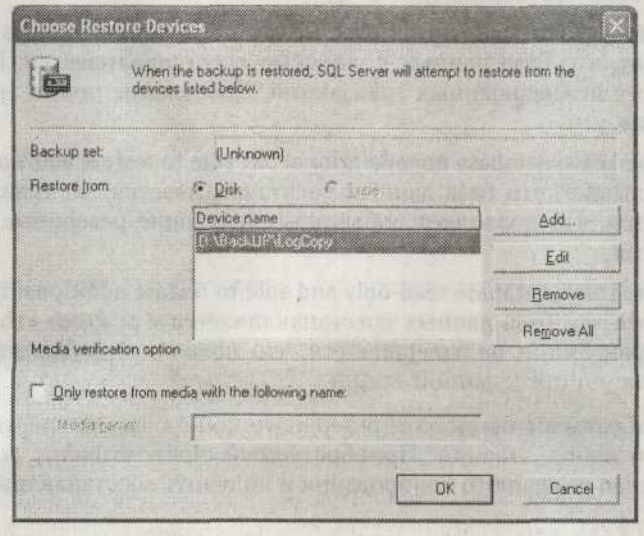


Рис. 9.54. Окно выбора носителя

Используя кнопки Add, Edit, Remove и Remove All, можно добавить, изменить или удалить носитель.

Теперь следует расмотреть вкладку Options, внешний вид которой приведен на рис. 9.55.

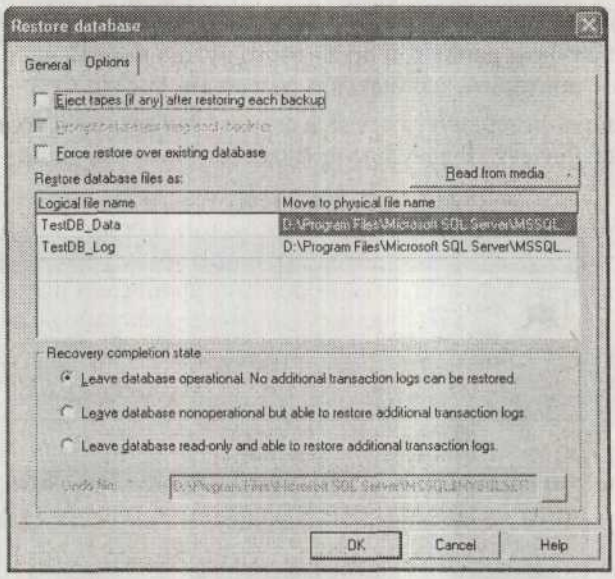


Рис. 9.55. Окно определения параметров восстановления базы данных

В списке Restore database Files as указывается расположение файла с данными и файла журнала транзакции. В нижней части окна можно выбрать порядок восстановления базы данных:

- Значение Leave database operational. No additional transaction logs can be restored указывает, что база данных полностью восстанавливается. Производится откат всех незавершенных транзакций. Добавление новых транзакций невозможно.
- Значение Leave database nonoperational but able to restore additional transaction logs указывает, что база данных восстанавливается, но транзакции не завершаются, что позволяет добавлять следующие резервные копии к данной копии.
- Значение Leave database read-only and able to restore additional transaction logs указывает, что база данных восстанавливается в режиме «только для чтения». Транзакции не завершаются, что позволяет добавлять следующие резервные копии к данной копии.

Чтобы разобраться с процессом резервного копирования, следует потренироваться в его использовании. Приобретенный опыт позволит более грамотно составить план резервного копирования и получить восстанавливаемые резервные копии.

Работа с ролями и учетными записями

Учетная запись пользователя является информацией, определяющей права доступа к объектам системы. Эта запись имеет пароль, ограничивающий ее использование. Учетные записи пользователей хранятся в базе данных Master.

SQL Server 2000 поддерживает два режима аутентификации пользователей. В случае использования режима аутентификации пользователя средствами операционной системы SQL Server 2000 запросит логин текущего пользователя и произведет поиск аналогичного логина в своей базе. Если этот логин будет обнаружен, сервер предоставит ему соответствующий доступ. Если сервер не обнаружит нужный логин, он запросит у операционной системы имена групп, к которым принадлежит пользователь. Если в SQL Server имеются учетные записи, входящие в те же группы, что и пользователь на локальной машине, то ему будет предоставлен доступ. В том случае, если пользователь входит в несколько групп и на сервере тоже прописано несколько групп, будет выбрана учетная запись, предоставляющая максимальные права доступа. В том случае, если нет совпадения по учетным записям и группам, в доступе будет отказано. По умолчанию при установке сервера выбирается именно этот режим.

При включенном режиме аутентификации средствами сервера для получения доступа к нему необходимо ввести логин и пароль. Выбрать режим аутентификации можно на вкладке Security окна свойств сервера (рис. 9.56).

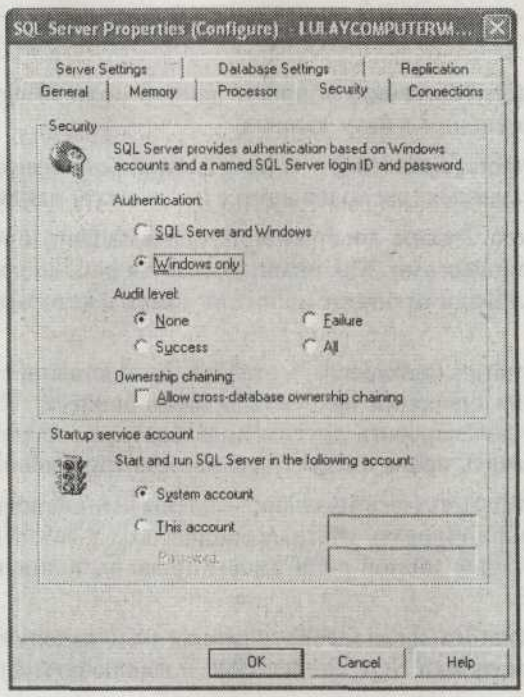


Рис. 9.56. Настройка режимов аутентификации SQL Server 2000

В этом окне нужно выбрать режим SQL Server and Windows. Вместе с созданием сервера создается базовая учетная запись системного администратора, имеющая все разрешения, — Sa. По умолчанию данная учетная запись не имеет пароля, поэтому его необходимо незамедлительно установить.

Роль представляет собой контейнер, содержащий учетные записи пользователей и делегирующий им те права, которыми она сама обладает. Роли могут быть серверными (фиксированными) и определенными на уровне базы данных. На уровне сервера определено несколько типов ролей, делегирующих те или иные административные функции, список которых можно посмотреть в объекте Security ▸ Server Roles (рис. 9.57):

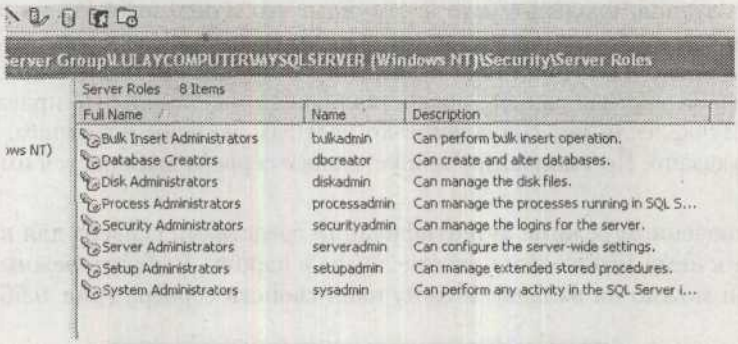


Рис. 9.57. Серверные роли SQL Server 2000

- Database Creators (dbcreator) — члены данной роли имеют право создавать и вносить изменения в базу данных.
- Disk Administrators (diskadmin) — члены данной роли имеют право управлять файлами баз данных, расположенных на жестком диске.
- Серверная роль Process Administrators (processadmin) предоставляет право управления процессами. Как правило, данная роль используется довольно редко из-за сложности определения ситуаций, в которых требуется вмешательство.
- Insert Administrators (bulkadmin) — эта роль предоставляет своим членам право выполнения операций пакетной вставки записей. Члены данной роли имеют право делегировать другим пользователям членство в ней. Исходя из этого выдавать права следует весьма осмотрительно.
- Security Administrators (securityadmin) — члены данной роли имеют право создавать и модифицировать учетные записи пользователей, изменять их пароли, также члены данной роли имеют право выполнять инструкции DENY, GRANT и REVOKE.
- Server Administrators (serveradmin) — данная роль наделяет правом изменять различные настройки SQL Server 2000 и выключать сервер.
- Setup Administrators (setupadmin) — члены данной роли имеют право удалять и добавлять связанные серверы, конфигурировать их.
- System Administrators (sysadmin) — данная пользовательская роль делегирует полный набор прав по работе с сервером, такой же, каким обладает учетная запись Sa. В связи с этим использовать эту роль следует с большой осторожностью.

Как было сказано выше, помимо серверных ролей могут определяться роли баз данных, список которых представлен на рис. 9.58. Эти роли также являются фиксированными, и изменять их нельзя, за исключением роли Public. Роли на уровне базы данных выполняют практически те же функции, что и серверные, но действуют только в пределах данной базы данных.

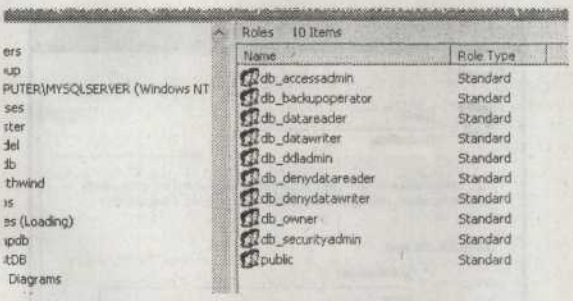


Рис. 9.58. Роли баз данных SQL Server 2000

Для получения доступа к ролям баз данных следует выбрать объект Roles нужной базы данных:

- Public — к данной роли принадлежат все пользователи базы данных. Ее использование очень удобно, так она позволяет просто назначить всем пользователям те или иные права. Использовать данную роль следует осмотрительно, так как назначенные права получат все пользователи.
- db_owner — данная роль предоставляет разрешения на выполнения любых действий с базой данных. По умолчанию такими правами обладает учетная запись dbo.
- db_accessadmin — члены данной роли имеют право добавлять и удалять учетные записи пользователей. Данная роль не дает права изменять разрешения на работу с объектами, поэтому добавляемые пользователи по умолчанию состоят в роли Public.
- db_securityadmin — члены данной роли имеют право управлять всеми разрешениями базы данных, включать пользователей в роли и исключат их них, определять права доступа к объектам.
- db_ddladmin — данная роль предоставляет права на выполнение инструкций языка DDL, таких как, например, CREATE TABLE и DROP TABLE.
- db_backupoperator — в эту роль включаются учетные записи, которым предоставляется право осуществления операций резервного копирования.
- db_datareader — данная роль не является административной, она дает право на просмотр всех таблиц и получения данных из них.
- db_datawriter — члены данной роли имеют право изменять данные в любой пользовательской таблице базы данных.
- db_denydatareader — члены данной роли не имеют права просматривать таблицы базы данных.

- db_denydatawriter — члены данной роли не имеют права вносить изменения в базу данных.

Для создания новой роли на уровне базы данных следует выбрать пункт контекстного меню New Database Role объекта Role. В результате будет отображено диалоговое окно, показанное на рис. 9.59.

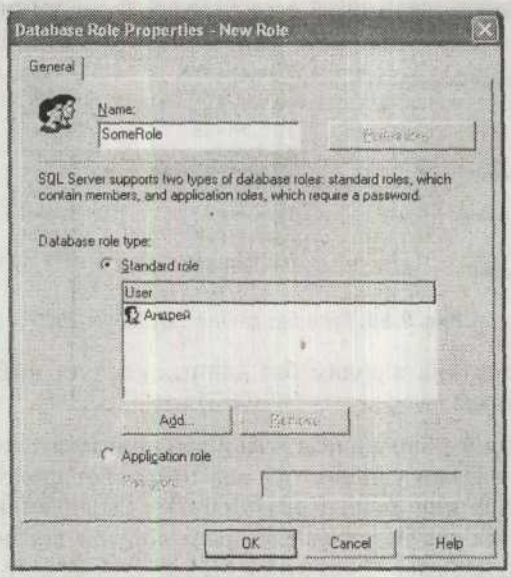


Рис. 9.59. Окно создания роли базы данных

В поле Name указывается имя роли.

На уровне базы данных SQL Server 2000 поддерживает два типа ролей. Стандартная роль включает в себя учетные записи пользователей и предоставляет им определенные права. В отличие от нее, роль приложения не включает в себя пользователей, но тоже может предоставлять свои права входящим в нее учетным записям.

Как правило, роль приложения используется именно приложениями. Роль приложения при активации блокирует все права, присущие данной учетной записи, и заменяет их собственными.

Для создания роли нужно выбрать переключатель Application Role и указать пароль, ограничивающий доступ к ней. Для использования роли ее необходимо активировать из приложения при помощи хранимой процедуры sp_setapprole. Команда активации роли приложения имеет следующий синтаксис:

```
sp_setapprole [@rolename =] 'role' ,
[@password =] {Encrypt N 'password'} | 'password'
[,[@encrypt =] 'encrypt_style']
```

Роль приложения можно создать при помощи Transact-SQL, используя команду sp_addapprole, имеющую следующий синтаксис:

```
sp_addapprole [ @rolename = ] 'role'
, [ @password = ] 'password'
```

Следующий запрос создает и активирует роль AppRole:

```
sp_addapprole AppRole, Pass
sp_setapprole AppRole, Pass
```

Вернемся к рассмотрению стандартных ролей базы данных. После выбора переключателя Standard Role станет активным список User. Если нажать кнопку Add, появится диалог добавления пользователя в роль (рис. 9.60), из которого можно выбрать одну или несколько учетных записей.

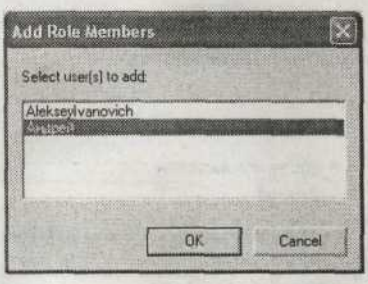


Рис. 9.60. Список пользователей

Выбор пользователей можно подтвердить нажатием кнопки OK. После этого соответствующая роль будет создана.

Для добавления нового пользователя в SQL Server 2000 необходимо активировать объект Security ▸ Logins (рис. 9.61).

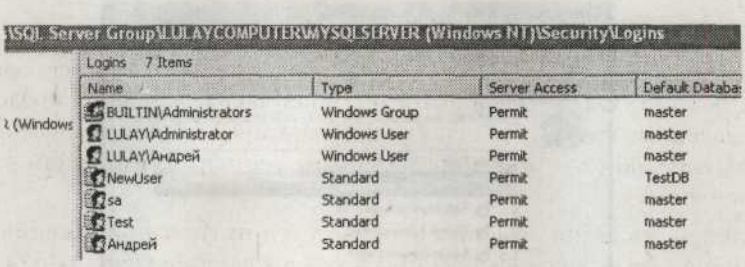


Рис. 9.61. Список учетных записей SQL Server 2000

Легко заметить, что часть учетных записей была импортирована из Windows. Чтобы добавить новую учетную запись или изменить ее настройки, необходимо выбрать пункт New Login контекстного меню объекта Logins или дважды щелкнуть мышью на уже существующей записи. Появится диалоговое окно, приведенное на рис. 9.62.

В поле Name указывается имя учетной записи. В поле Password, соответственно, указывается пароль. В списке Database можно выбрать базу данных, к которой будет принадлежать данный пользователь, а в списке Language — язык, используемый по умолчанию.

Для проверки работы этого окна нужно создать учетную запись MyNewLogin и нажать кнопку OK. На вкладке Server Roles, внешний вид которой показан на рис. 9.63, можно указать, к какой серверной роли принадлежит данная учетная запись. По умолчанию учетная запись не включается ни в одну серверную роль.

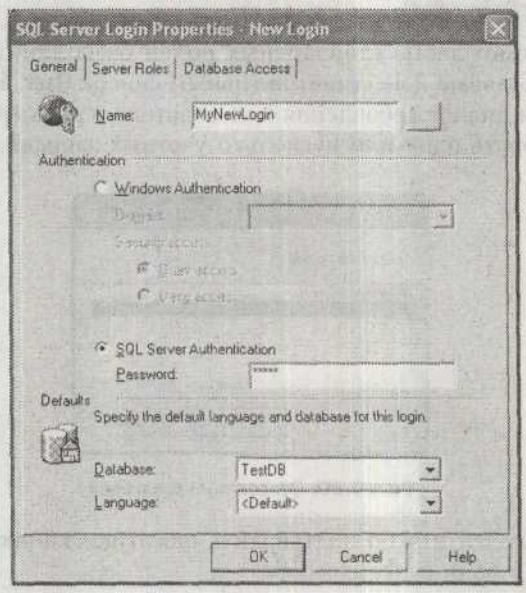


Рис. 9.62. Окно свойств учетной записи. Вкладка General

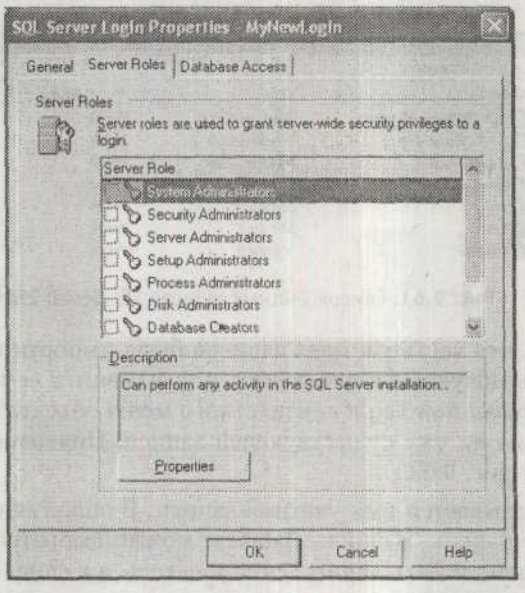


Рис. 9.63. Окно свойств учетной записи. Вкладка Server Roles

В окне свойств данной роли, показанном на рис. 9.64, можно выяснить, какие пользователи принадлежат к данной роли, добавить или удалить пользователей. На вкладке Permissions находится список действий, которые данная роль имеет право выполнять. В поле Name указывается имя роли, а чуть ниже — список учетных записей, входящих в нее. При помощи кнопок Add и Remove можно добавлять и удалять учетные записи из данной роли.

Теперь следует обратить внимание на вкладку Database Access, внешний вид которой показан на рис. 9.65.

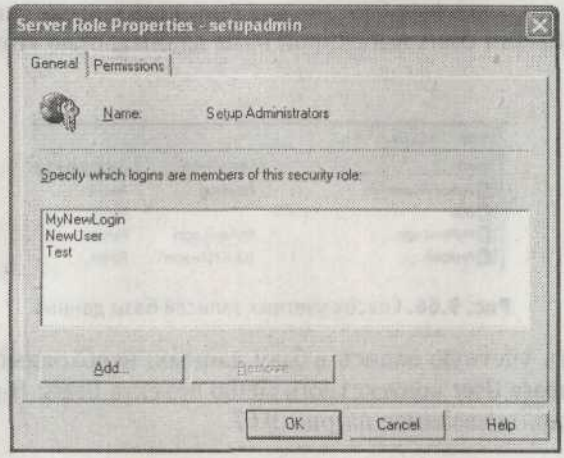


Рис. 9.64. Окно свойств серверной роли

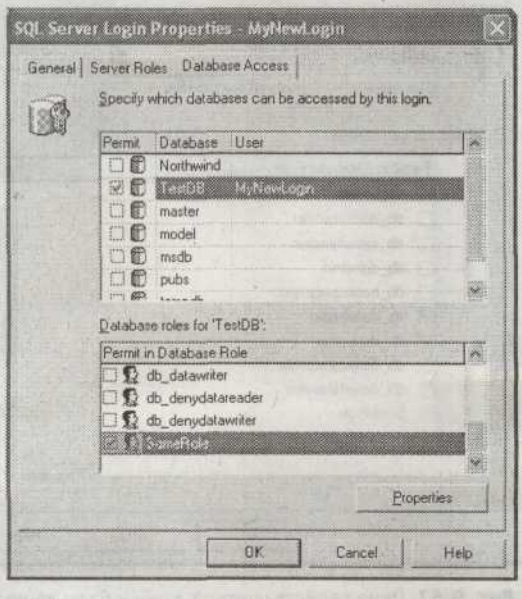


Рис. 9.65. Окно свойств учетной записи. Вкладка Database Access

В верхнем списке выбирается база данных, в которую будет входить данная учетная запись. В нижнем списке Database Roles можно выбрать роль базы данных, к которой принадлежит данная учетная запись. Как видно из рисунка, учетная запись включена в созданную ранее роль SomeRole. Также нужно указать роль db_denydatawriter, чтобы запретить пользователю вносить какие-либо изменения в данные. После нажатия кнопки OK учетная запись будет добавлена в роль.

Учетная запись может не входить ни в одну из зарегистрированных ролей (кроме Public), но может быть зарегистрирована в базе данных. Если учетная запись не была подключена к базе данных описанным выше способом, ее можно назначить из объекта Users выбранной базы данных, окно которого приведено на рис. 9.66.

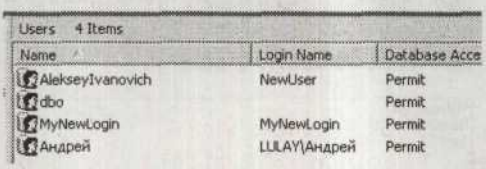


Рис. 9.66. Список учетных записей базы данных

Чтобы добавить учетную запись в базу данных, необходимо выполнить команду New Database User контекстного меню объекта Users. В результате будет отображено окно, показанное на рис. 9.67.

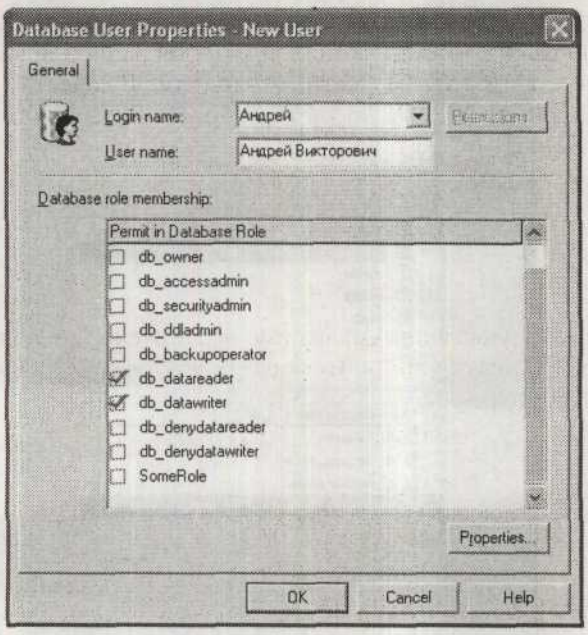


Рис. 9.67. Окно свойств учетной записи базы данных

Из списка Login Name можно выбрать учетную запись пользователя, зарегистрированную в SQL Server 2000, а в поле User Name можно присвоить ей описание. Данной учетной записи нужно предоставить права на работу с базой данных, например возможность просмотра и изменения всех таблиц, которую позволяют получить права db_datareader и db_datawriter.

Определение прав на работу с объектами базы данных

Определять разрешения на работу с объектами базы данных можно как с использованием команд языка SQL, так и при помощи утилиты SQL Server Enterprise Manager. Для начала следует рассмотреть первый вариант. Для присваивания разрешений на работу с объектами группе или отдельным учетным записям используется команда GRANT, синтаксис которой приведен ниже:

```
GRANT
{ ALL [ PRIVILEGES ] | permission [ ....n ] }
{
[ ( column [ ....n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ....n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
TO security_account [ ....n ]
[ WITH GRANT OPTION ]
[ AS { group | role } ]
```

Параметр WITH GRANT OPTION дает данной учетной записи право делегирования данных полномочий другим учетным записям. В качестве примера можно продемонстрировать, как выдать учетной записи MyNewLogin право получения данных из таблицы MyTable1.

```
GRANT SELECT ON MyTable1 TO MyNewLogin
```

Разрешение на получение данных из таблицы MyTable1 можно присвоить отдельной роли.

```
GRANT SELECT ON MyTable1 TO SomeRole
```

Чтобы отменить присвоенные права или ограничения, используется команда REVOKE, имеющая следующий синтаксис:

```
REVOKE [ GRANT OPTION FOR ]
{ ALL [ PRIVILEGES ] | permission [ ....n ] }
{
[ ( column [ ....n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ....n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
{ TO | FROM }
```

```
security_account [ ....n ]
[ CASCADE ]
[ AS { group | role } ]
```

Параметр CASCADE указывает серверу на необходимость отмены всех ограничений, порожденных данной учетной записью или ролью. Следующий запрос отбирает присвоенные права у роли SomeRole:

```
REVOKE SELECT ON MyTable1 TO SomeRole
```

Для запрещения доступа к объекту используется команда DENY, синтаксис которой приведен ниже:

```
DENY
{ ALL [ PRIVILEGES ] | permission [ ....n ] }
{
[ ( column [ ....n ] ) ] ON { table | view }
| ON { table | view } [ ( column [ ....n ] ) ]
| ON { stored_procedure | extended_procedure }
| ON { user_defined_function }
}
TO security_account [ ....n ]
[ CASCADE ]
```

Следующий запрос запрещает группе SomeRole выборку данных из таблицы MyTable1:

```
DENY SELECT ON MyTable1 TO SomeRole
```

Для включения учетной записи или роли в одну из фиксированных ролей используется хранимая процедура sp_addrolemember. Команда имеет следующий синтаксис:

```
sp_addrolemember [ @rolename = ] 'role',
[ @membername = ] 'security_account'
```

Для добавления роли SomeRole в роль db_datareader, определенную на уровне базы данных и предоставляющую право получения данных из всех таблиц, входящих в базу данных, используется следующий запрос:

```
EXEC sp_addrolemember 'db_datareader', 'SomeRole'
```

Для удаления учетной записи из роли используется хранимая процедура sp_droprolemember, синтаксис которой приведен ниже:

```
sp_droprolemember [ @rolename = ] 'role' ,
[ @membername = ] 'security_account'
```

Следующий запрос удаляет роль SomeRole из роли db_datareader:

```
EXEC sp_droprolemember 'db_datareader', 'SomeRole'
```

Теперь следует узнать, как осуществляется определение прав на работу с объектами базы данных при помощи утилиты SQL Server Enterprise Manager. Достаточно выбрать какую-либо роль в объекте Roles, например SomeRole, и дважды щелкнуть на ней мышью. В появившемся окне нужно нажать кнопку Permissions. В результате будет отображено окно, показанное на рис. 9.68.

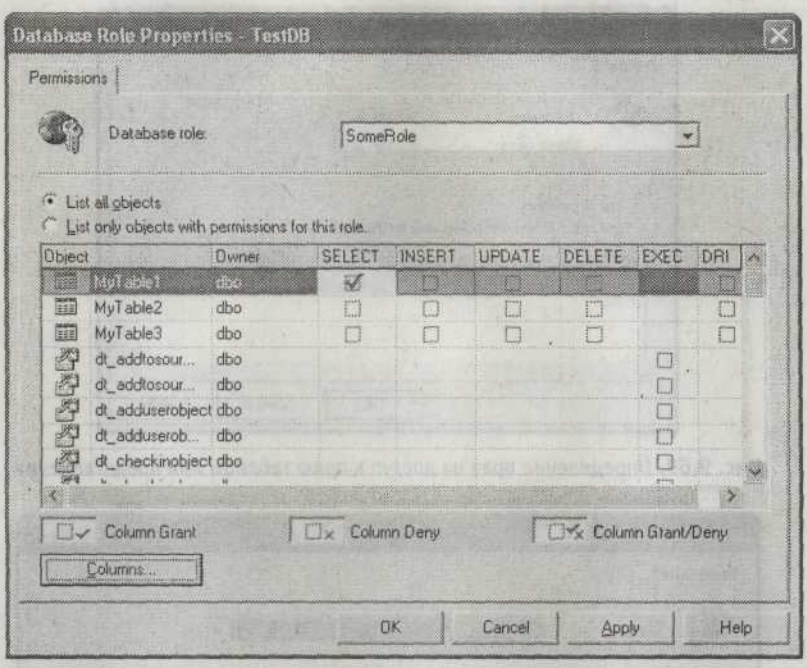


Рис. 9.68. Определение прав роли на работу с объектами базы данных

В списке Database Role можно выбрать роль, для которой устанавливаются права на доступ или ограничения. В списке, расположенном в центральной части окна, перечисляются объекты базы данных и права на работу с ними. Галочка разрешает выполнение той или иной инструкции и соответствует команде GRANT, пустое место соответствует команде REVOKE, а крестик — команде DENY. Как видно из рис. 9.68, роли SomeRole выдано разрешение на запрос данных из таблицы MyTable1.

При выборе таблиц или представлений становится доступной кнопка Colums, активирующая окно, показанное на рис. 9.69. Это окно позволяет определить параметры доступа к отдельным поля таблицы или представления.

Точно так же осуществляется определение прав на работу с объектами для учетных записей пользователей. Чтобы получить доступ к соответствующему диалоговому окну, необходимо активировать учетную запись пользователя и в появившемся окне нажать на кнопку Permissions.

Права на доступ к объекту можно определить в самом объекте. Для этого необходимо выполнить команду контекстного меню All Tasks ▸ Manage Permissions. В результате будет активировано окно, показанное на рис. 9.70.

В списке Object перечисляются все объекты базы данных, которым можно определить те или иные разрешения.

На этом можно завершить рассмотрение возможностей SQL Server 2000. Как видим, этот сервер баз данных предоставляет разработчику и администратору все необходимые возможности для создания мощных приложений.

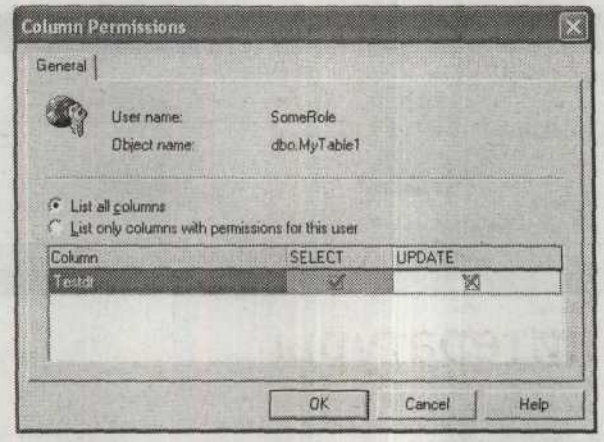


Рис. 9.69. Определение прав на доступ к полю таблицы или представления

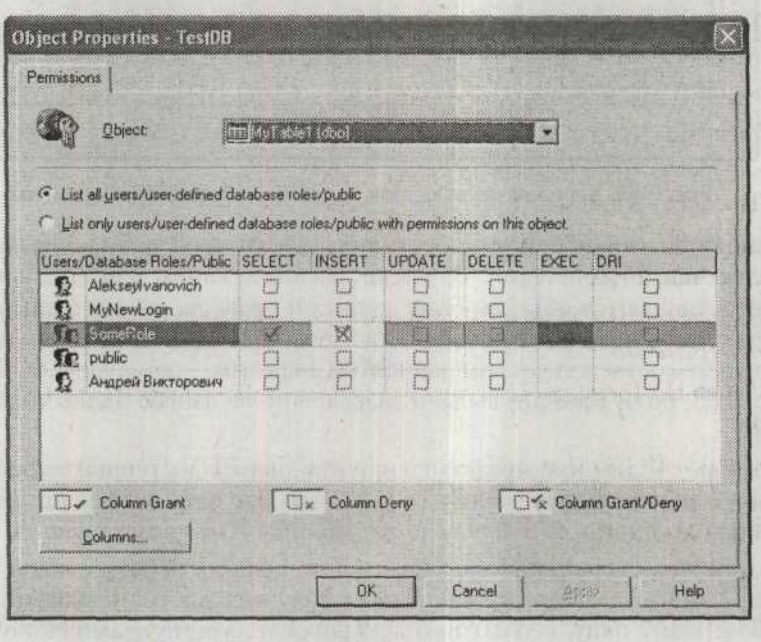


Рис. 9.70. Определение прав на доступ к объекту

Список литературы

1. Крёнке Д. Теория и практика построения баз данных. 8-е изд., СПб.: Питер, 2003.
2. Марков Е., Дарахвелидзе П. Разработка WEB-служб средствами Delphi. СПб.: BHV-Петербург, 2003.
3. Фёдоров А., Елманова Н. ADO в Delphi. СПб.: BHV-Петербург, 2002.
4. Граббер М. Understanding SQL. М.: Лори,1993.
5. Шумаков В., Фаронов П. Delphi 5. Руководство разработчика баз данных. М.: Нолидж, 2000.
6. Шапиро Д., Бойс Д. Windows 2000 Server. Библия пользователя. Киев: Диалектика, 2000.
7. Дарахвелидзе П., Марков Е. Программирование в Delphi 7. СПб.: BHV-Петербург, 2003.
8. Елманова Н., Трепалин С., Тенцер А. Delphi и технология COM. СПб.: Питер, 2003.
9. Понамарёв В. COM и ActiveX в Delphi. СПб.: BHV-Петербург, 2001.
10. Кириллов В., Громов Г. Структуризированный язык запросов (SQL)/ СПбГИТМО. СПб.
11. Скляр А. Введение в InterBase. М.: Горячая линия–Телеком, 2002.
12. Ковязин А., Востриков С. Мир InterBase. 2-е изд. М.: КУДИЦ-ОБРАЗ, 2003.
13. Линзенбардт М., Стиглер М. Администрирование SQL Server 2000. Полное руководство/Пер. с англ. Киев: BHV, 2001.
14. Мамаев Е. Microsoft SQL Server 2000. СПб.: BHV-Петербург, 2004.
15. Kalen Delaney Inside Microsoft SQL Server 2000. Microsoft Press, 2001.
16. InterBase 6 API Guide, Borland.
17. InterBase 6 Data Definition Guide, Borland.
18. InterBase 6 Developer's Guide, Borland.
19. InterBase 6 Embedded SQLGuide, Borland.

20. InterBase 7 Language Reference, Borland.
21. InterBase 7 Operations Guide, Borland.
22. Информационный портал www.ibase.ru.
23. Технологии распределенных баз данных//http://ami.nstu.ru/~vms/lecture/lecture10/lecture10.htm
24. Кузнецов С. Язык баз данных SQL/92//http://www.citforum.ru/database/sqlbook/sqlbook_05.shtml
25. Пржиялковский В. Модели, базы данных и СУБД в информационных системах//http://www.ccas.ru/~prz/book2.htm
26. Ранние подходы к организации БД//http://ami.nstu.ru/~vms/lecture/lecture2/lecture2.HTM
27. Елманова Н. Настройка параметров доступа к данным в C++ Builder//http://www.citforum.ru/programming/application/builder.shtml
28. Настройка BDE//http://www.citforum.tl.ru/programming/32less/les32.shtml
29. John Kaster. dbExpress (Inprise/Borland's new cross-platform data access layer) Draft Specification//http://bdn.borland.com/article/0,1410,22495,00.html
30. Bob Swart. Delphi, dbExpress And MySQL//Delphi Magazine, 2002//http://www.thedelphimagazine.com/Samples/1540/1540.htm

Алфавитный указатель