МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Государственное образовательное учреждение высшего профессионального образования
УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Г. П. Токмаков

БАЗЫ ДАННЫХ

**Концепция баз данных,
реляционная модель данных,
языки SQL и XML**

Учебное пособие

Ульяновск
2010

УДК 681.3 (075)
ББК 73я7
Т 51

Рецензенты: д-р техн. наук, профессор Егоров Ю. П.,
канд. техн. наук, доцент Куприянов А. А.

*Утверждено редакционно-издательским советом университета
в качестве учебного пособия*

Токмаков, Г. П.

Т 51 Базы данных. Концепция баз данных, реляционная модель данных, языки SQL и XML : учебное пособие / Г. П. Токмаков. – Ульяновск : УлГТУ, 2010. − 192 с.

ISBN 978-5-9795-0762-0

Учебное пособие подготовлено по материалам лекционных курсов, посвященных основам теории баз данных, языку SQL, хранимым процедурам и триггерам, которые читались автором в последние десять лет. В пособии также рассмотрены вопросы взаимодействия технологии баз данных с Web-технологиями.

Пособие предназначено для студентов специальностей 522800 и 522300, а также может использоваться студентами других специальностей.

УДК 681.3 (075)
ББК 73я7

ISBN 978-5-9795-0762-0

© Токмаков Г. П., 2010
© Оформление. УлГТУ, 2010

ОГЛАВЛЕНИЕ

ГЛАВА 1. ФАЙЛОВЫЕ СИСТЕМЫ И БАЗЫ ДАННЫХ 9

1.1. ФАЙЛОВЫЕ СИСТЕМЫ 9
1.1.1. История систем управления данными 9
1.1.2. Файловая система как способ отделения логической и физической структуры данных 13
1.1.3. Последовательный и ассоциативный доступ в файловых системах 16

1.2. СТАНОВЛЕНИЕ КОНЦЕПЦИИ БАЗ ДАННЫХ 18
1.2.1. Структуры данных 18
1.2.2. Целостность данных 21
1.2.3. Язык запросов SQL 22
1.2.4. Метаданные, унифицированные процедуры и язык SQL 25

1.3. СУБД КАК НЕЗАВИСИМЫЙ СИСТЕМНЫЙ КОМПОНЕНТ 26
1.3.1. СУБД как средство обеспечения логической и физической независимости данных 27
1.3.2. СУБД в составе информационной системы 28
1.3.3. Выделение СУБД в качестве отдельного компонента информационной системы 29

ГЛАВА 2. ЛОГИЧЕСКИЕ СТРУКТУРЫ РЕЛЯЦИОННОЙ МОДЕЛИ 31

2.1. ОСНОВЫ РЕЛЯЦИОННОЙ АЛГЕБРЫ 31
2.1.1. Объекты и их определения 32
2.1.2. Операторы 35

2.2. ОСНОВНЫЕ ПОНЯТИЯ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ 39
2.2.1. Учебная база данных 39
2.2.2. Первичные ключи 39
2.2.3. Отношение предок/потомок 42
2.2.4. Внешние ключи 43
2.2.5. Индексы 44

2.3. ЦЕЛОСТНОСТЬ ДАННЫХ 45
2.3.1. Условия целостности данных 46
2.3.2. Изменения, способные нарушить ссылочную целостность 47
2.3.3. Правила ссылочной целостности 47

2.4. НОРМАЛИЗАЦИЯ ДАННЫХ 49
2.4.1. Понятие функциональной зависимости 50
2.4.2. Первая нормальная форма: атомарные атрибуты 51
2.4.3. Вторая нормальная форма: отсутствие зависимостей частичного ключа 52
2.4.4. Третья нормальная форма: устранение транзитивных зависимостей 54

2.5. СИСТЕМНЫЙ КАТАЛОГ 54
2.5.1. Назначение системного каталога 55
2.5.2. Структура системного каталога 55
2.5.3. Информация о таблицах 56

2.5.4. Информация о столбцах 56
2.5.5. Информация о представлениях 56
2.5.6. Информация об отношениях между таблицами 57
2.5.7. Информация о пользователях 57
2.5.8. Информация о привилегиях 57
ГЛАВА 3. DDL – ЯЗЫК ОПРЕДЕЛЕНИЯ ДАННЫХ РЕЛЯЦИОННОЙ МОДЕЛИ 60
3.1. СОЗДАНИЕ БАЗЫ ДАННЫХ 60
3.1.1. Общий формат оператора CREATE DATABASE 60
3.1.2. Определение пароля 61
3.1.3. Указание размера страницы БД 61
3.1.4. Указание национальной кодировки символов 62
3.1.5. Типы данных 62
3.2. СОЗДАНИЕ ДОМЕНОВ 64
3.2.1. Общий формат оператора CREATE DOMAIN 64
3.2.2. Ограничения на значения столбцов, ассоциированных с доменом 64
3.2.3. Изменение определения домена 65
3.3. СОЗДАНИЕ ТАБЛИЦ 67
3.3.1. Инструкция CREATE TABLE 67
3.3.2. Инструкция ALTER TABLE 71
3.4. СОЗДАНИЕ ПРЕДСТАВЛЕНИЙ (VIEW) 73
3.4.1. Общий формат оператора CREATE VIEW 75
3.4.2. Горизонтальное представление 75
3.4.3. Вертикальное представление 75
3.4.4. Удаление представления 76
3.4.5. Недостатки представлений 77
3.5. СОЗДАНИЕ ИНДЕКСОВ 77
3.5.1. Общий формат оператора CREATE INDEX 77
3.5.2. Необходимость создания индексов 77
3.5.3. Удаление индекса 78
ГЛАВА 4. DML – ЯЗЫК МАНИПУЛИРОВАНИЯ ДАННЫМИ РЕЛЯЦИОННОЙ МОДЕЛИ 79
4.1. ОПЕРАТОР ВЫБОРКИ SELECT 79
4.1.1. Общий формат оператора SELECT 79
4.1.2. Предложение SELECT 80
4.1.3. Предложение FROM 81
4.1.4. Предложение WHERE 82
4.1.5. Правила выполнения запроса SELECT 87
4.2. АГРЕГАТНЫЕ ФУНКЦИИ 87
4.2.1. Вычисление среднего значения столбца 88
4.2.2. Вычисление суммы значений столбца 88
4.2.3. Вычисление экстремумов 88

4.2.4. Вычисление количества значений в столбце 89
4.2.5. Правила выполнения запросов, в котором участвуют агрегатные функции 89
4.3. ЗАПРОСЫ С ГРУППИРОВКОЙ 90
4.3.1. Предложение GROUP BY 90
4.3.2. Предложение HAVING........ 90
4.3.3. Предложение ORDER BY − определение сортировки........ 91
4.3.4. Правила выполнения запросов с группировкой 91
4.4. ВЛОЖЕННЫЕ ЗАПРОСЫ........ 92
4.4.1. Определение подчиненных запросов........ 93
4.4.2. Условия отбора в подчиненном запросе 95
4.4.3. Подчиненные запросы в предложении HAVING 99
4.4.4. Правила выполнения вложенных запросов........ 99
4.5. МНОГОТАБЛИЧНЫЕ ЗАПРОСЫ 101
4.5.1. Алгоритм выполнения многотабличного запроса........ 101
4.5.2. Внутреннее объединение таблиц........ 101
4.5.3. Внешнее объединение таблиц 105
4.6. ОПЕРАТОРЫ ОБНОВЛЕНИЯ ДАННЫХ 109
4.6.1. Оператор INSERT 109
4.6.2. Оператор UPDATE........ 111
4.6.3. Оператор DELETE 112
ГЛАВА 5. DDS – СРЕДСТВА АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ .. 113
5.1. НАЗНАЧЕНИЕ И ЛИКВИДАЦИЯ ПРАВ 113
5.1.1. Команда GRANT 113
5.1.2. Команда REVOKE........ 115
5.2. НАЗНАЧЕНИЕ ПРАВ ИСПОЛНЕНИЯ ХРАНИМЫХ ПРОЦЕДУР 115
5.3. СОЗДАНИЕ ГРУППЫ УПРАВЛЕНИЯ ПРАВАМИ – РОЛИ........ 116
5.3.1. Команда CREATE ROLE........ 116
5.3.2. Команда DROP ROLE........ 116
5.3.3. Формирование списка прав, связанных с ролью 116
5.3.4. Формирование прав пользователей на основе ролей........ 117
5.3.5. Связывание пользователей с ролями 117
ГЛАВА 6. ИНФОРМАЦИОННЫЕ СИСТЕМЫ С АКТИВНЫМ СЕРВЕРОМ БАЗ ДАННЫХ........ 118
6.1. ХРАНИМЫЕ ПРОЦЕДУРЫ ИЛИ ФУНКЦИИ........ 119
6.1.1. Структура языка........ 120
6.1.2. Команды и выражения........ 121
6.1.3. Переменные 122
6.1.4. Возвращение переменных........ 129
6.1.5. Атрибуты........ 129
6.1.6. Конкатенация........ 131
6.1.7. Передача управления........ 132
6.1.8. Обработка ошибок и исключений........ 142

6.1.9. Вызов функций 144
6.2. ТРИГГЕРЫ 145
6.2.1. Создание триггера 145
6.2.2. Получение информации о триггерах 147
6.2.3. Удаление триггера 148
6.2.4. PL/pgSQL и триггеры 149
ГЛАВА 7. XML КАК СПОСОБ ЛОГИЧЕСКОГО ПРЕДСТАВЛЕНИЯ ИНФОРМАЦИИ 154
7.1. ЯЗЫК HTML И ЕГО НЕДОСТАТКИ 155
7.2. ЯЗЫК XML И ЕГО ОСНОВЫ 156
7.2.1. Объявление XML 157
7.2.2. Элементы и теги 157
7.2.3. Атрибуты 159
7.2.4. Иерархичность структуры XML-документа 159
7.2.5. Комментарии 159
7.3. XML СХЕМЫ И МЕТАДАННЫЕ 160
7.3.1. Структурирование данных и схема XML 160
7.3.2. Типы данных в схеме XML 162
7.3.3. Элементы и атрибуты в XML Схеме 165
7.3.4. Пространство имен 166
7.4. СТИЛИ И ФОРМАТИРОВАНИЕ ДАННЫХ XML 167
7.4.1. Основы XSL 168
7.4.2. Структура таблицы стилей XSL 170
ГЛАВА 8. SQL И XML 172
8.1. XML КАК СРЕДСТВО ПРЕДСТАВЛЕНИЯ СТРУКТУРИРОВАННЫХ ДАННЫХ 173
8.1.1. Представление структурированных данных в XML 173
8.1.2. Сравнение XML и SQL 174
8.2. ИСПОЛЬЗОВАНИЕ XML С БАЗАМИ ДАННЫХ 176
8.2.1. Хранение данных в формате XML 176
8.2.2. Вывод в формате XML 178
8.2.3. Ввод в формате XML 180
8.2.4. Обмен данными в формате XML 181
8.2.5. Интеграция данных в формате XML 182
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 186
ПРИЛОЖЕНИЕ 187
УЧЕБНАЯ БАЗА ДАННЫХ 187

ВВЕДЕНИЕ

Рассматривая трудовую деятельность человека, можно выделить три стадии развития по степени использования орудий труда:

1) механизация труда − когда происходит усиление физических усилий человека с помощью орудий труда или замена физического труда человека механизмами. Здесь предметом и продуктом труда выступает вещество или энергия;

2) автоматизация труда − это замена управленческого труда в управляющей системе автоматом. Предметом и продуктом труда в этом случае является информация. Автомат реализует некоторую программу, алгоритм управления. В зависимости от степени автоматизации управленческого труда выделяют автоматические и автоматизированные системы управления:

– *в автоматических системах* процессы полностью автоматизированы. Роль человека состоит в контроле и настройке управленческих процессов. В этих системах имеется закон управления и критерий эффективности управления. Значит, процесс полностью автоматизирован и может полностью управляться автоматом;

– *в автоматизированных системах* в процессе управления участвует человек и автомат. При этом их функции четко распределены. Автомат, в основном занимается процессом сбора, обработки, хранения и передачи информации. Другими словами, автомат занимается тем, что сейчас принято называть обработкой или управлением данными. Человеку предоставляются функции принятия решений.

3) кибернетизация труда связана с заменой человека в сфере принятия решений. В этом случае машина обеспечивает поддержку принятия решений и заменяет лиц, обеспечивающих их подготовку.

Механизация труда является предметом изучения машиностроительных специальностей, автоматические системы изучаются в рамках теории автоматического управления, автоматизация в сфере принятия решений относится к сфере научной дисциплины «Искусственный интеллект», а вот автоматизированные системы, рассматриваемые в аспекте обработки данных – это и есть предмет нашего изучения.

Сегодня трудно себе представить сколько-нибудь значимую информационную систему, которая не имела бы в качестве основы или важной составляющей базу данных. Концепции и технологии баз данных складывались постепенно и всегда были тесно связаны с развитием систем автоматизированной обработки информации. Создание баз данных после появления реляционного подхода превратилась из искусства в науку, хоатя следует отметить, что элементы искусства все еще имеют место. Тем не менее, сейчас это вполне сложившаяся дисциплина (хотя являющаяся скорее инженерной, чем чисто научной), основанная на достаточно формализованных подходах и включающая широкий спектр приемов и методов создания баз данных.

Сегодня теория баз данных, несмотря на свою молодость, является обязательной для изучения студентами практически всех технических специальностей. В соответствии с новыми стандартами учебная дисциплина «Базы данных» включена в стандарты всех специальностей, связанных с подготовкой специалистов по вычислительной технике: это группа специальностей 22.01, 22.02, 22.03 и 22.04. В остальные технические специальности раздел, посвященный базам данных, включен в общий курс информатики и вычислительной техники.

Часть 1
Базы данных :
Модели и структуры данных

С данными людям приходилось иметь дело с самого начала развития человеческого общества. На заре человечества данные сохранялись в виде наскальных рисунков, затем появились пиктограммы и клинопись. С изобретением алфавита сохранение данных упростилось, и с тех пор технология обмена информацией постоянно совершенствуется.

Независимо от того, какие методы использовались в прошлом или используются в наши дни, основная идея информационных технологий остается неизменной: сохранение, получение и отображение данных остаются основными функциями любой системы. Идея состоит в том, чтобы сохранять данные так, чтобы ими могли воспользоваться другие люди. Сегодня все сводится к работе с информацией, и к этому все будет сводиться в будущем.

Какую бы сферу человеческой деятельности вы не затронули: торговлю, медицину, образование, промышленность, сферу развлечений или управления, – везде главенствующую роль в организационных процессах играют средства накопления, обработки и передачи данных. Сегодня уже невозможно представить торговое предприятие без информационной системы учета операций, банк – без централизованной базы данных, мгновенный доступ к которой может получить любое отделение в любой точке страны.

Для обеспечения эффективного хранения данных, а это означает быстрый поиск, обновление данных, защиту от ошибочных вводов, обеспечение конфиденциальности информации и многое другое, необходима соответствующая их организация. Для быстрого поиска необходимо упорядочение хранимых данных, поддержание связей между ними, контроль на непротиворечивость, обеспечение однократного ввода или изменения при многократном последующем использовании.

Ключевую роль при этом играют методы поддержания логических связей между данными. Именно эти методы рассматриваются в моделях данных, разработанных в рамках развития концепции баз данных, пришедшей на смену файловым системам, в которых преимущественно рассматривались вопросы доступа к данным.

ГЛАВА 1. ФАЙЛОВЫЕ СИСТЕМЫ И БАЗЫ ДАННЫХ

Базы данных представляют собой синтез структур данных и файловых структур. В современных базах данных методы из обеих областей применяются для создания такой системы хранения больших объемов данных, которая может выглядеть как система с множеством видов организаций данных и обслуживать приложения различных типов.

Термин *база данных* относится к совместно используемому набору логически связанных данных (и описанию этих данных), предназначенному для удовлетворения информационных потребностей организации. В случае баз данных на внешнем носителе хранятся данные совместно с их описанием, в то время как в файлах хранятся просто двоичные данные, логическое описание которых содержится в соответствующих программах.

База данных – это единое, большое хранилище данных, которое однократно определяется, а затем используется одновременно многими пользователями – представителями разных подразделений. Вместо разрозненных файлов с избыточными данными здесь все данные собраны вместе с минимальной долей избыточности. База данных уже не принадлежит какому-либо единственному отделу, а является общим корпоративным ресурсом.

1.1. ФАЙЛОВЫЕ СИСТЕМЫ

1.1.1. История систем управления данными

Исторически базы данных развивались как способ интеграции систем хранения данных. С развитием компьютерных технологий для них находилось все больше применений в управлении информацией и начали создаваться информационные системы для автоматизации деятельности бухгалтерий, отделов кадров, отделов закупок и т. д. Каждая такая информационная система информационная систем имела свой собственный набор данных (см. Рис. 1.1.). Сначала эти данные хранились в последовательных файлах, затем появились файлы, основанные на индексировании.

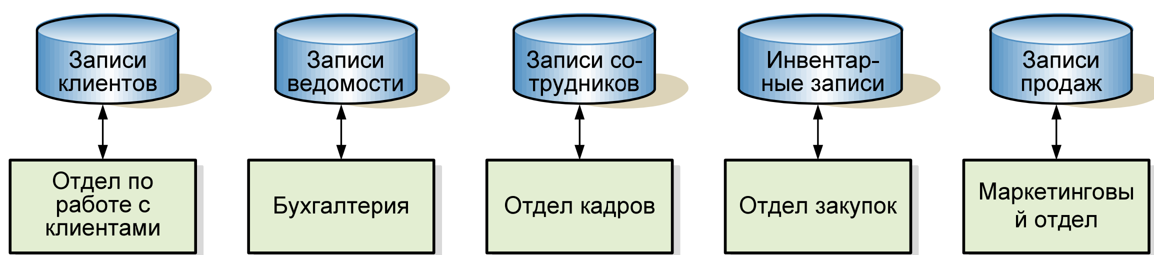


Рис. 1.1. Файловая информационная система

Хотя с появлением индексированных файлов эффективность работы систем существенно повысилась, но набор отдельных информационных систем в пределах одной организации все же является неэффективной тратой ресурсов по сравнению возможностями интегрированной системы базы данных. Например, если в каждом подразделении предприятия использовалась своя файловая система, то большая часть информации, хранимой в организации, дублировалась в этих хранилищах. В результате, например, при смене места жительства сотрудника приходилось корректировать файлы во всех подразделениях, где содержалась информация об адресах сотрудников.

Для решения этой проблемы на первых этапах развития технологии информационных систем для хранения данных создавались файлы коллективного доступа, разрабатываемые в едином центре. Но у файлов коллективного доступа имелись серьезные недостатки. Структура записей в файлах и наборы отношений, реализуемые в прикладных программах, были «жесткими», и их изменение означало перестройку отношений в прикладных программах.

Сложные структуры данных, используемые при решении задачи комплексной автоматизации, связаны между собой не менее сложными взаимосвязями. Но файловая система не поддерживает отношения между данными. Поэтому поддержку связей между данными приходилось реализовывать в прикладных программах. Эти дополнительные средства управления данными составляли существенную часть прикладных программ и практически повторялись от одной системы к другой.

Стремление выделить и обобщить повторяющиеся фрагменты прикладных программ, ответственные за управление сложно структурированными данными коллективного доступа, явилось первой побудительной причиной создания систем управления базами данных (СУБД). В результате СУБД стали мощным средством интеграции данных, хранимых и обрабатываемых внутри определенной организации. С их помощью можно было работать внутри одной интегрированной системы с платежными ведомостями бухгалтерии, записями сотрудников отдела кадров, инвентарными записями отдела закупок и т. д. (см. рРис. 1.2.).

История систем управления данными во внешней памяти началась с появлением магнитных дисков. Первым попытку расширить возможности работы с дисками путем создания СУБД предпринял еще в 1961 г. Чарльз Бахман. Тогда

он работал в фирме `General Electric`, а потому разработка велась на машине этой компании, а созданная им «интегрированная система хранения» могла работать только на этой машине.

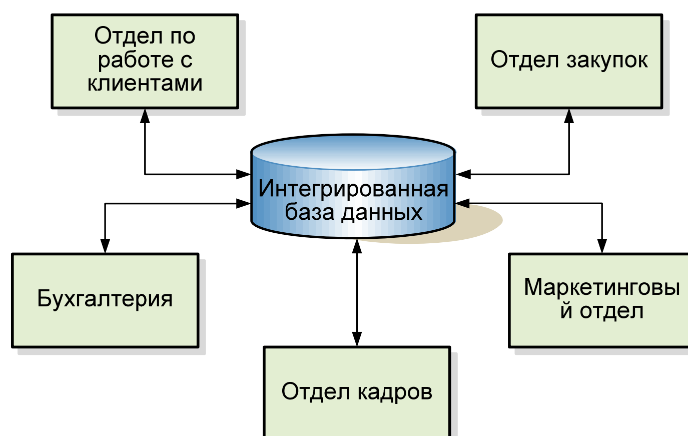


Рис. 1.2. Информационная система с базой данных

Дальнейшее развитие этой идеи предпринял Джон Куллиан. Он был предпринимателем и ему первому пришла в голову оригинальная для того времени идея: продавать ПО для компьютеров. Он не стремился разрабатывать свои программы «с нуля»; наряду с собственными разработками он покупал отдельные программы сторонних организаций, производящих программную продукцию, дорабатывал их соответствующим образом и продавал готовые, более сложные изделия на рынке.

На этом поприще он оказался чрезвычайно успешным, но вскоре столкнулся с тем, что для интеграции приобретенных и разрабатываемых продуктов нужна единая система управления данными. На основании анализа альтернативных решений выбор пал на «интегрированную систему хранения» Бахмана, и это было исключительно верным решением. Система Бахмана, названная Куллианом СУБД, отлично работала на мейнфреймах IBM и совместимых с ними машинах.

Система, созданная на основе «интегрированной системы хранения» Бахмана, была совсем простой: вся СУБД упаковывалась в один файл, а таблицы, содержащие сведения о размещении данных, создавались вручную. Но в этой системе был использован революционный подход: *доступ к данным осуществлялся не непосредственно, как в файловой системе, а через описание данных. Здесь необходимо отметить, что описание файлов, используемое файловой системой касается только способов доступа, а в СУБД Бахмана были использованы описание самих данных, т. е. описание их структуры и взаимосвязей между ними.*

Описание данных Бахмана было представлено в виде иерархической системы «указателей», помогающих осуществить доступ к конкретной ячейке памяти. Эта незамысловатая работа оказала колоссальное влияние на развитие ПО, за что Бахман получил Тьюринговскую премию, а это аналог Нобелевской в области ВТ.

СУБД Бахмана строились с использованием иерархических и сетевых моделей и были названы им навигационными, поскольку для перемещения по записям использовались «указатели» или «пути», что отличает их от реляционных СУБД, где используются принципы логического программирования.

К появлению современных реляционных СУБД привела другая революционная идея, предложенная Коддом. Он в 1970 г. опубликовал статью, содержащую описание простой модели данных, согласно которой база данных составляется из таблиц. Таблицы эти особенные, и их особенность заключается в том, что в них не должно быть одинаковых строк. Поэтому эти таблицы были названы реляциями, а сама модель – реляционной.

Но основное достоинство реляционных моделей данных заключалось не в их простоте, хотя и это имело большое значение, а в том, что в них были применены принципы нечисловой или ассоциативной обработки данных. В соответствии с этими принципами *данные связываются в соответствии с их внутренними логическими взаимоотношениями, а не физическими указателями*, как это делалось в иерархической и сетевой моделях. *В основе этих принципов лежит ассоциативный, а не адресный поиск, и взаимосвязи между элементами данных устанавливаются в соответствии со значениями их атрибутов, а не путем искусственного соединения с помощью указателей*.

При числовой обработке данных содержание данных практически не используется. Допустим, что мы хотим вычислить сумму элементов массива. При этом достаточно указать адрес начального элемента массива, а затем накапливать сумму элементов в некотором регистре, индексируя соответствующим образом адреса. В ходе этого вычисления нас не интересуют текущие значения суммируемых переменных, мы просто должны проверять некоторые переменные, влияющие на ход выполнения программы (например, достигнут или нет конец массива).

При ассоциативной обработке нас в первую очередь интересуют сведения об объектах и их свойствах. При этом мы можем запросить сведения о студенте, указав его фамилию, имя или номер зачетной книжки, или узнать имена всех преподавателей моложе 40 лет, чей оклад выше 300 000 руб.

Для пояснения принципов ассоциативной обработки данных приведем следующий простой пример. Допустим, у нас имеются сведения о студентах и группах, в которых они обучаются, представленные в таблице ГРУППЫ с полями Код, Наименование, и в таблице СТУДЕНТЫ с полями Номер зачетной книжки, Фамилия, Имя, Отчество, Код_группы. По этим данным, используя принципы ассоциативной обработки, а именно сравнивая значения полей Код таблицы ГРУППЫ и Код_группы таблицы СТУДЕНТЫ, мы можем определить в какой группе учится каждый студент, сколько студентов учится в каждой группе, вывести список студентов каждой группы и т. д. Как видно из приведенного примера, при ассоциативной обработке решение задач сводится к сопоставлению или ассоциации значений признаков обрабатываемых данных, т. е. используется содержание данных, а не адрес.

Таким образом, пользователи смогут комбинировать данные из разных источников, если логическая информация, необходимая для такого комбинирования, присутствует в исходных данных. Это открыло новые возможности для информационных систем, поскольку запросы к базе данных теперь не были ограничены физическими указателями и для их реализации уже не приходилось писать программы.

В последующем между Бахманом и Коддом велась активная полемика. Бахман всячески пытался доказывать преимущества навигационных баз данных, но был вынужден уступить перед математической строгостью реляционного подхода и возможностями языка запросов SQL. В модели данных Бахмана содержалась минимальная информация о данных и эта модель, названная навигационной, имела процедурный характер, в то время как реляционная модель содержит обширную декларативную информацию о содержащихся в памяти данных.

Для сравнения навигационного и реляционного (ассоциативного) подходов реализации СУБД можно использовать классический пример, в котором для описания пункта назначения применяются различные способы.

– При использовании навигационного подхода путь до объекта можно указать так: «Едете по шоссе 25 км, поворачиваете направо и продолжаете движение до третьего населенного пункта. Нам нужен 3-й дом с левой стороны».

– Ассоциативный подход, используемый в рамках реляционной СУБД, позволяет просто указать: «Желтый дом в населенном пункте N».

Навигационный подход используется тогда, когда таксист новичок плохо ориентируется на местности, т. е. у него нет сведений о географии местности, и ему нужно подробно описать, как доехать до места назначения.

СУБД, в которой реализована реляционная модель, можно сравнить с опытным таксистом, хорошо ориентирующимся в местности, которому достаточно назвать признаки искомого объекта, а путь к этому объекту он определит сам на основе имеющейся у него информации. Простота доступа к данным в реляционной модели объясняется тем, что СУБД обладает описанием наличных данных и может сама отыскать требуемую информацию без приведения подробной информации относительно пути доступа.

Хотя реляционная модель одержала решительную победу над навигационной, и реляционные базы данных занимают доминирующее положение на рынке, сегодня правота Кодда уже не столь безусловна. С появлением языка XML началась реставрация навигационного подхода. Но это уже тема для другой дисциплины, хотя в нашем курсе лекций мы вкратце рассмотрим основы XML-технологии, так как их роль в современных системах обработки данных чрезвычайно велика, и совместное использование концепции баз данных и XML-технологии позволяет создавать эффективные информационные системы.

1.1.2. Файловая система как способ отделения логической и физической структуры данных

Идеи никогда не возникают на пустом месте. Как правило, любая идея основывается на другой, возникшей на более ранней стадии развития какой-либо отрасли науки или практики. Идея реализации информационных систем на основе концепции баз данных явилась результатом довольно долгого развития файловых систем. В зачаточном виде описание данных присутствовало и в файловой системе. Поэтому для изучения основ концепции баз данных сначала рассмотрим, как в ходе развития файловой системы эволюционировало описание данных, используемое для доступа к ним.

Первые коммерческие информационные системы, основанные на применении файловой системы, использовались в основном для ведения бухгалтерии. При этом необходимо было обеспечить ведение главной бухгалтерской книги, балансовых отчетов, ведомостей заработной платы и т. д. Эту работу обязано делать любое предприятие и поэтому такие компьютерные системы легко и быстро окупались, так как затраты ручного труда на ведения ведомостей по заработной плате или выписывание счетов были очень велики.

Программисты, разрабатывающие эти системы, использовали те термины, которые применялись в бумажном документообороте. Поэтому компьютерные массивы данных, соответствующие пачкам для бумаг (`file folder`) были названы файлами, так как компьютерный файл содержал ту информацию, которая могла бы лежать в обычной папке. В результате сформировалось понятие о *файле* как сущности, позволяющей получить доступ к ресурсам вычислительной системы. С точки зрения прикладной программы, *файл – это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные*.

Так как в результате работы ЭВМ создавалось большое количество различных файлов, требовалось организовать эффективное управление этими объектами. Поэтому со временем было определено еще одно понятие, получившее название файловая система. Термин *файловая система* используется для обозначения программной системы, *управляющей файлами, хранящихся во внешней памяти*. Файловая система берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса внешней памяти и обеспечение доступа к данным. В настоящее время файловая система входит в состав операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на внешних носителях данных, и обеспечить совместное использование файлов несколькими пользователями и процессами.

Программные приложения пишутся на языках высокого уровня, которые предоставляют примитивы обращения к файлам через файловую систему. Эти примитивы позволяют манипулировать файлами с помощью *логических записей*. Как правило, логические записи разделяются на более мелкие блоки, называемые *полями*. Например, каждую логическую запись в файле персонала можно было бы разделить на такие поля, как имя, адрес, идентификационный номер и т. д.

В отличие от логической структуры, хранение файла на запоминающем устройстве предписывает, что файл должен быть разделен на блоки, являющимися *физическими записями*, совместимыми с используемым устройством хранения. Например, файлы, записанные на диски, должны делиться на блоки с размером с сектор.

Если приложению необходимо найти часть файла, измеряемую в логических записях, оно обращается к файловой системе, чтобы та произвела нужное обращение. Файловая система осуществляет управление файлами в терминах физических записей и считывает достаточное для выполнения запроса количество физических записей, размещая полученные записи в буфере, а затем предоставляет этот буфер приложению. Аналогично для записи информации в

файл приложение передает данные файловой системе. Файловая система хранит их в буфере до тех пор, пока не накопится физическая запись, а затем передает эту запись на запоминающее устройство.

Для выполнения доступа к файлам файловая система должна хранить информацию о файле, к которому идет обращение. При этом она должна знать на каком устройстве записан файл, имя файла, расположение буфера, через который передаются данные, и будет ли файл сохранен после того, как приложение завершит работу. Подобная информация хранится в таблице, называемой дескриптором файла. Дескриптор файла создается, когда приложение уведомляет файловую систему, что ей потребуется доступ к файлу, и удаляется, когда приложение сообщает, что файл более не требуется. Процесс создания дескриптора файла называется открытием файла, а его удаление – закрытием файла.

Итак, понятие «файловая система» включает в себя следующие компоненты:

– совокупность всех файлов на диске с их физической организацией;

– наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске, т. е. логическая организация файловых структур;

– комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, поиск и другие операции над файлами.

Файловая система − это дополнительный программный слой, *обеспечивающий возможность работы прикладного программиста на уровне логической организации файлов. При этом файловая система:*

– с одной стороны, предоставляет пользователю интерфейс в терминах логических записей и операций над ними;

– с другой стороны, преобразует логические команды пользователя в детальные операции над физическими данными на внешнем устройстве.

Таким образом, введение промежуточного слоя, составленного из комплекса системных программ, ограждает или абстрагирует программиста от деталей фактической организации низкоуровневых данных и он имеет дело только с логическими данными, так как они представлены в более удобной для него форме.

Абстрагирование данных получило дальнейшее развитие в рамках разработки концепции баз данных, в рамках которой логическое представление данных стал еще ближе человеческим представлениям о действительности.

В ходе создания файловой системы был выработан еще один важный принцип: использование данных, содержащихся в дескрипторе файла, при организации доступа к файлу. Этот принцип, как мы убедимся в дальнейшем, станет основой концепции баз данных, в раках которой эти данные называются метаданными.

1.1.3. Последовательный и ассоциативный доступ в файловых системах

На первых ЭВМ данные хранились на ленте и записи извлекались и обрабатывались последовательно. В программах эти данные были организованы точно также как и на физическом носителе когда еще не существовало понятия логической структуры.

Поэтому при этом использовался самый простой способ локализации записи – сканирование файла до выявления требуемой записи. Для реализации данного способа доступа достаточно было знать начальный адрес файла (см. Рис. 1.3.).

Но уже на этом этапе были выработаны такие важные для дальнейшего изучения понятия как ключевое поле, составной ключ и первичный ключ. Рассмотрим классический пример последовательного файла, содержащего записи, в которых содержится информация об одном сотруднике.

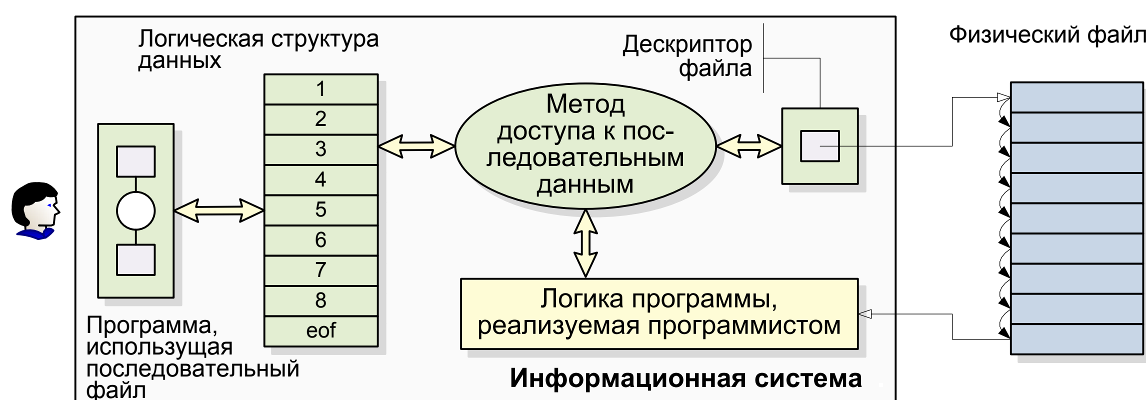


Рис. 1.3. Логическая и физическая структуры файлов с последовательным доступом

Каждая логическая запись делится на поля, такие как имя, адрес, идентификационный номер сотрудника и т. д.

Предположим, что этот файл используется для обработки платежной ведомости. В каждый период платежа этот файл обработатывается целиком, поэтому обработка записей ведется последовательно от начала до конца. Для поддержки этого последовательного процесса физические записи в файле (т. е. на внешнем носителе) должны располагаться в том же порядке, что и в логической структуре данных, которой оперирует прикладная программа. Так как в качестве внешнего носителя в этот период использовались магнитные ленты, такое ограничение легко выполнялось.

Неотъемлемой частью процесса обработки последовательного файла является определение конца файла, а для этого мы должны иметь возможность распознавать записи по какому-нибудь признаку. Логические записи распознаются, как правило, по одному полю в записи. Например, в файле сотрудников это может быть поле, содержащее идентификационный номер сотрудника. Такое поле называется *ключевым*.

Хотя во многих приложениях требуется идентифицировать записи по ключам, которые не являются уникальными (например, Фамилия, Имя, Отчество), но при этом все равно должен существовать один уникальный ключ, используемый для идентификации записи в файле (например, в таблице СТУДЕНТЫ – это номер зачетной книжки). Такой ключ называется *первичным* или *идентификатором*.

Иногда бывает необходимо объединить несколько полей, чтобы обеспечить уникальность ключа, который в этом случае называется *составным ключом*. Эти понятия широко используются в СУБД и понадобятся нам в дальнейшем.

Ограниченные возможности последовательных файлов не помешали им быть эффективным средством для составления один или два раза в месяц счетов, платежных ведомостей и других отчетов. Однако для решения широкого круга задач требуется напрямую обращаться к конкретной записи без предварительной сортировки файла или последовательного чтения всех записей. Поэтому при разработке сложных информационных систем нужно было найти способ организации произвольного доступа. Это стало возможным после появления дисковых систем внешней памяти и разработки индексированных файлов.

Для создания индексированных файлов на основе ключей были реализованы специальные таблицы, переводящие ассоциативный запрос в соответствующий адрес. Эти таблицы были названы списками ссылок или *индексами*. Индекс определяется как таблица, содержащая список ключевых значений, каждому из которых соответствует указатель, локализующий блок записей на носителе данных. Чтобы найти определенный блок информации, сначала необходимо отыскать в индексе его ключ, а потом получить сам блок, который хранится по адресу, связанным с этим ключом.

Классическим примером использования индексированного файла является обслуживание записей сотрудников. За счет создания индекса можно избежать длительных операций поиска для получения отдельной записи. В частности, если файл записей сотрудников индексирован по идентификационным номерам сотрудников, то определенную запись можно быстро получить, если этот номер известен (см. Рис. 1.4.).

Следует обратить внимание на тот факт, что в данном случае структуры логических и физических файлов уже различаются и для доступа к данным требуется больше информации, размещенная в индексных файлах. В индексированных файлах идея использования данных для доступа получила дальнейшее развитие, которая затем была использована при разработке концепции баз данных.

На основе рассмотренных вариантов информационных систем, построенных на основе файловых систем, можно констатировать следующее:

– типовым программным обеспечением обработки данных в этих информационных системах являются методы доступа, а не методы управления данными;

– файловые системы, входящие в состав операционных систем, обеспечивают абстракцию файла для хранения этих записей.

В качестве аналога индекса можно привести предметный указатель, размещаемый в конце книги. Понятие индекса широко используется в СУБД и понадобится нам в дальнейшем.

1.2. СТАНОВЛЕНИЕ КОНЦЕПЦИИ БАЗ ДАННЫХ

Файловые системы обеспечивают хранение слабо структурированной информации, оставляя дальнейшую структуризацию прикладным программам. Типовая информационная система, главным образом, ориентирована на хранение, выбор и модификацию данных соответствующей прикладной области.

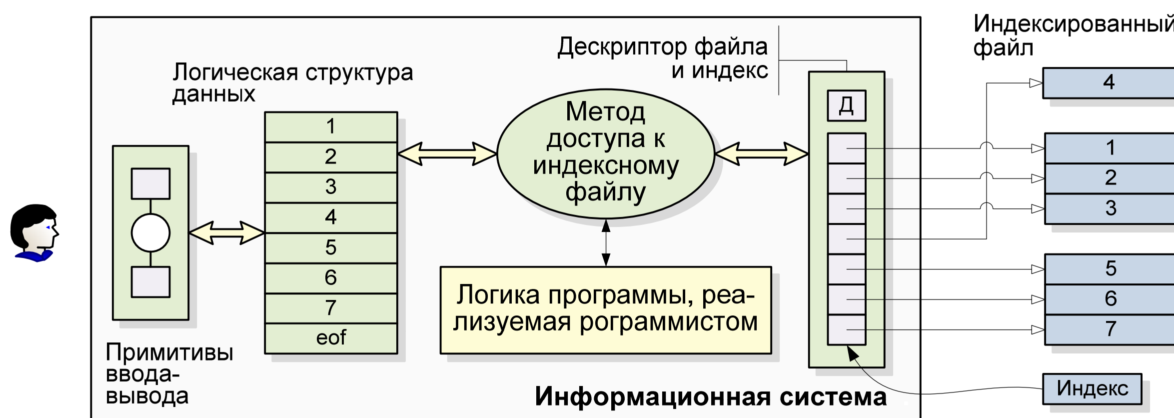


Рис. 1.4. Логическая и физическая структуры файлов с произвольным доступом

Структура таких данных, как правило, гораздо сложнее чем простая последовательность записей, и в информационных системах на поддержку связей сложноструктурированной информации приходилось писать довольно сложный программный код. Этот код, в силу общности правил и закономерностей обработки данных, практически повторялся от одной системы к другой. Рассмотрим это на примере разработки простой информационной системы, поддерживающей учет служащих некоторой организации.

На начальном этапе использования вычислительной техники для построения информационных систем проблемы структуризации данных решались индивидуально в каждой информационной системе на основе традиционных методов обработки данных, рассмотренных выше. При этом для каждого конкретного случая разрабатывалась своя логика внешнего пользователя, которая включала такие понятия, как информационная структура, операции выбора, вставки и удаления информации. Это приводило к возникновению взаимозависимости между данными и программой: поэтому при изменении данных нужно было либо менять программу, либо реорганизовывать данные.

1.2.1. Структуры данных

С целью выделить из прикладной логики общие правила обработки данных в ранних информационных системах производились необходимые надстройки над файловыми системами в виде библиотеки программ, подобно тому, как это делается в компиляторах (см. Рис. 1.5.).

Но очень скоро стало понятно, что с помощью общей библиотеки программ невозможно реализовать более сложные методы хранения данных. Поясним это на примере. Предположим, что требуется реализовать простую информационную систему, поддерживающую учет служащих некоторой организации.

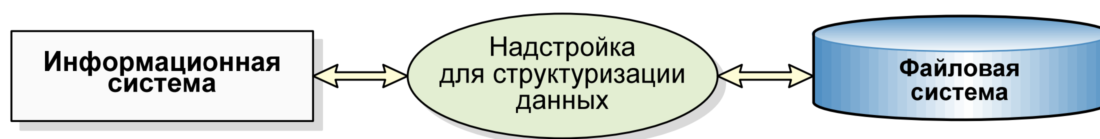


Рис. 1.5. Примитивная схема структуризации данных в информационной системе

Система должна выполнять следующие действия:

– выдавать списки служащих по отделам;

– поддерживать возможность перевода служащего из одного отдела в другой;

– обеспечивать средства поддержки приема на работу новых служащих и увольнения работающих служащих, т. е. ввод и удаление данных о служащих.

Кроме того, для каждого отдела должна поддерживаться возможность получения:

– имени руководителя отдела;

– общей численности отдела;

– общей суммы зарплаты служащих отдела, среднего размера зарплаты и т. д.

Для каждого служащего должна поддерживаться возможность получения:

– номера удостоверения по полному имени служащего (для простоты допустим, что имена всех служащих различны);

– полного имени по номеру удостоверения;

– информации о соответствии служащего занимаемой должности и размере его зарплаты.

Предположим, что мы решили создать эту информационную систему на файловой системе и пользоваться одним файлом `СЛУЖАЩИЕ`, расширив базовые возможности файловой системы за счет специальной библиотеки функций (см. Рис. 1.6.). Поскольку минимальной информационной единицей в нашем случае является служащий, в этом файле должна содержаться одна запись для каждого служащего. Чтобы можно было удовлетворить указанные выше требования, запись о служащем должна иметь следующие поля:

– полное имя служащего (`СЛЖ_ИМЯ`);

– номер его удостоверения (`СЛЖ_НОМЕР`);

– данные о соответствии служащего занимаемой должности (`СЛЖ_СТАТ = {«да», «нет»}`);

– размер зарплаты (`СЛЖ_ЗАРП`);

– номер отдела (`СЛЖ_ОТД_НОМЕР`).

Поскольку мы решили ограничиться одним файлом СЛУЖАЩИЕ, та же запись должна содержать имя руководителя отдела (СЛЖ_ОТД_РУК). Если мы этого поля не введем, то невозможно будет, например, получить имя руководителя отдела в котором работает служащий.

Уникальные ключи

Неуникальный ключ

Файл СЛУЖАЩИЕ

СЛЖ_ИМЯ	СЛЖ_НОМЕР	СЛЖ_СТАТ	СЛЖ_ЗАРП	СЛЖ_ОТД_НОМЕР	СЛЖ_ОТД_РУК
–	–	–	–	–	–
. . .	. . .	. . .	. . .	. . .	. . .
–	–	–	–	–	–

Рис. 1.6. Структура файла СЛУЖАЩИЕ на уровне приложения

Чтобы информационная система могла эффективно выполнять свои базовые функции, необходимо обеспечить многоключевой доступ к файлу СЛУЖАЩИЕ по уникальным ключам СЛЖ_ИМЯ и СЛЖ_НОМЕР. Ключ называется уникальным, если его значения гарантированно различны во всех записях файла. Если мы не сможем обеспечить многоключевой доступ к файлу СЛУЖАЩИЕ, то для выполнения наиболее часто используемых операций получения данных о конкретном служащем понадобится последовательный просмотр в среднем половины записей файла.

Кроме того, система должна обеспечить возможность эффективного выбора всех записей с общим значением СЛЖ_ОТД_НОМЕР, т. е. доступ по неуникальному ключу. Если не поддерживать данный механизм доступа, то для получения данных об отделе в целом, в общем случае потребуется полный просмотр файла.

Но даже в этом случае, чтобы получить численность отдела или общий размер зарплаты, система должна будет выбрать все записи о служащих указанного отдела и посчитать соответствующие общие значения.

Таким образом, мы видим, что при реализации даже такой простой информационной системы на базе файловой системы возникают следующие затруднения:

– требуется создание сложной надстройки для многоключевого доступа к файлам;

– возникает существенная избыточность данных (для каждого служащего повторяется номер и имя руководителя его отдела);

– требуется выполнение массовой выборки и вычислений для получения суммарной информации об отделах.

Кроме того, если в ходе эксплуатации системы потребуется, например, обеспечить операцию выдачи списков служащих, получающих указанную зарплату, то для этого либо придется полностью просматривать файл, либо нужно будет реструктурировать файл СЛУЖАЩИЕ, объявляя ключевым и поле СЛЖ_ЗАРП.

Для улучшения ситуации можно было бы поддерживать два многоключевых файла: СЛУЖАЩИЕ и ОТДЕЛЫ. Первый файл должен был бы содержать поля СЛЖ_ИМЯ,

СЛЖ_НОМЕР, СЛЖ_СТАТ, СЛЖ_ЗАРП и СЛЖ_ОТД_НОМЕР, а второй – ОТД_НОМЕР, ОТД_РУК (номер удостоверения служащего, являющегося руководителем отдела), ОТД_СЛЖ_ЗАРП (общий размер зарплаты служащих данного отдела) и ОТД_ЧИСЛ (общее число служащих в отделе). Структура этих файлов показана на Рис. 1.7. .

Рис. 1.7. Структура файлов СЛУЖАЩИЕ и ОТДЕЛЫ на уровне приложения

Введение этих двух файлов позволило бы преодолеть большинство неудобств, перечисленных выше. При этом:

– каждый из файлов содержал бы только не дублируемую информацию;
– не возникала бы необходимость в динамических вычислениях суммарной информации по отделам.

1.2.2. Целостность данных

Но добавление в информационную систему второго файла создает принципиально иную ситуацию, которая приводит к тому, что наша информационная система должна обладать особенностями, сближающими ее с СУБД.

Эти особенности заключаются в том, что теперь система должна «знать», что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), должна иметь информацию о структуре и смысле каждого поля. Например, системе должно быть известно, что у полей СЛЖ_ОТД_НОМЕР в файле СЛУЖАЩИЕ и ОТД_НОМЕР в файле ОТДЕЛЫ один и тот же смысл – номер отдела.

Кроме того, система должна учитывать, что в ряде случаев изменение данных в одном файле должно автоматически вызывать модификацию второго файла, чтобы общее содержимое файлов было согласованным. Например, если на работу принимается новый служащий, то нужно добавить запись в файл СЛУЖАЩИЕ, а также должным образом изменить поля ОТД_СЛЖ_ЗАРП и ОТД_ЧИСЛ в записи файла ОТДЕЛЫ, соответствующей отделу этого служащего.

Более точно, система должна руководствоваться следующими правилами:

1. Если в файле СЛУЖАЩИЕ содержится запись со значением поля СЛЖ_ОТД_НОМЕР = n, то и в файле ОТДЕЛЫ должна содержаться запись со значением поля ОТД_НОМЕР = n;

2. Если в файле ОТДЕЛЫ содержится запись со значением поля ОТД_РУК = m, то и в файле СЛУЖАЩИЕ должна содержаться запись со значением поля СЛЖ_НОМЕР = m; далее мы увидим, что правила (1) и (2) являются частными случаями общего правила ссылочной целостности: поле СЛЖ_ОТД_НОМЕР содержит «ссылки» на записи таблицы ОТДЕЛЫ, а поле ОТД_РУК содержит «ссылки» на записи таблицы СЛУЖАЩИЕ;

3. Значение поля ОТД_СЛЖ_ЗАРП записи файла ОТДЕЛЫ, для которой значение поля ОТД_НОМЕР = n, должно быть равно сумме значений полей СЛЖ_ЗАРП тех записей файла СЛУЖАЩИЕ, в которых значение поля СЛЖ_ОТД_НОМЕР = n;

4. Значение поля ОТД_ЧИСЛ записи файла ОТДЕЛЫ, для которой значение поля ОТД_НОМЕР = n, должно быть равно числу записей файла СЛУЖАЩИЕ, в которых значение поля СЛЖ_ОТД_НОМЕР = n; далее мы увидим, что правила (3) и (4) представляют собой примеры общих ограничений целостности базы данных.

Понятие согласованности, или целостности, данных является ключевым понятием баз данных. Фактически, если информационная система (даже такая простая, как в нашем примере) поддерживает согласованное хранение данных в нескольких файлах, можно говорить о том, что она поддерживает базу данных. Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных (СУБД).

1.2.3. Язык запросов SQL

Но обеспечение целостности данных – это далеко не все, что требуется от СУБД. Требование поддержания согласованности данных в нескольких файлах не позволяет при построении информационной системы обойтись простой библиотекой функций: такая система должна обладать некоторыми собственными данными (их принято называть метаданными), определяющими целостность данных. Именно наличие метаданных позволяет СУБД описывать и использовать произвольные данные сначала в рамках навигационной, а затем – реляционной модели.

В нашем примере информационная система должна отдельно сохранять метаданные о структуре файлов СЛУЖАЩИЕ и ОТДЕЛЫ, а также правила, определяющие условия целостности данных в этих файлах (принято считать, что правила также составляют часть метаданных).

Таким образом, база данных отличается от простого набора данных тем, что она содержит не только данные, но и план, или модель данных. При создании баз данных разработчик описывает ее логическую структуру и способы ведения данных пользователем. Такое описание базы данных называются моделью данных, схемой или концептуальной схемой. Модель определяет единицы данных, а также специфицирует связи каждой единицы данных с другими единицами данных. Например, число может появиться и в файле, и в базе данных, но в файле это просто обезличенное число, а в базе данных число имеет смысл,

специфицированный для него моделью данных. Это может быть цена товара, номер заказа, индекс продукта и т. д.

Изучение возможностей использования модели данных, начнем с того, что даже в нашем примере пользователю информационной системы будет не слишком просто получить, например, общую численность отдела, в котором работает Петр Иванович Федоров. Для этого ему пришлось бы написать программу, которая работала бы в соответствии со следующим алгоритмом.

1. Установить номер отдела, в котором работает указанный служащий. С этой целью:

– в файле СЛУЖАЩИЕ необходимо отыскать запись, для которой значение поля СЛЖ_ИМЯ = 'ПЕТР ИВАНОВИЧ ФЕДОРОВ'.

– для найденной записи необходимо определить значение поля СЛЖ_ОТД_ НОМЕР. Пусть в файле СЛУЖАЩИЕ это будет запись, для которой значение поля СЛЖ_ОТД_НОМЕР = n.

2. Выбрать из файла ОТДЕЛЫ запись, для которой значение поля ОТД_НОМЕР = n.

3. Определить значение поля ОТД_ЧИСЛ в таблице ОТДЕЛЫ для записи, у которой значение поля ОТД_НОМЕР = n.

И так пришлось бы поступать всякий раз, как только возникала необходимость в извлечении тех или иных данных. Было бы гораздо проще, если бы СУБД предоставляла в распоряжение пользователей язык, который позволял сформулировать соответствующий запрос, в ответ на который СУБД выдавала требуемый результат. В составе современных реляционных СУБД такие языки имеются и называются языками запросов к базам данных.

Самое примечательное то, что эти языки разрабатываются в соответствии со стандартным интерфейсом SQL, который, в настоящее время, де факто является обязательным для всех разработчиков СУБД. Это приводит к тому, что внутренне строение СУБД может быть любым, но ее интерфейс, с которым имеет дело пользователь, будет единым, и пользователю не придется переписывать код всякий раз, когда в информационной системе меняется СУБД.

Например, при наличии языка запросов SQL наш запрос можно было бы выразить в следующей форме (запрос1):

```
SELECT ОТД_ЧИСЛ
FROM СЛУЖАЩИЕ, ОТДЕЛЫ
WHERE СЛЖ_ИМЯ = 'ПЕТР ИВАНОВИЧ ФЕДОРОВ' AND
      СЛЖ_ОТД_НОМЕР = ОТД_НОМЕР;
```

Это пример запроса с «полусоединением», который характеризуется тем, что:

– запрос адресуется к двум файлам – СЛУЖАЩИЕ и ОТДЕЛЫ,

– но данные выбираются только из файла ОТДЕЛЫ.

Условие СЛЖ_ОТД_НОМЕР = ОТД_НОМЕР всего лишь «ограничивает» интересующий нас набор записей об отделах до одной записи, если Петр Иванович Федоров действительно работает на данном предприятии. Если же Петр Иванович Федоров не работает на предприятии, то условие СЛЖ_ИМЯ = 'ПЕТР ИВАНОВИЧ ФЕДОРОВ' не будет удовлетворяться ни для одной записи файла СЛУЖАЩИЕ, и поэтому запрос выдаст пустой результат.

Возможна и другая формулировка того же запроса (запрос2):

```
SELECT ОТД_РАЗМЕР
FROM ОТДЕЛЫ
WHERE ОТД_НОМЕР =
  (SELECT СЛЖ_ОТД_НОМЕР
   FROM СЛУЖАЩИЕ
   WHERE СЛЖ_ИМЯ = 'ПЕТР ИВАНОВИЧ ФЕДОРОВ');
```

Это пример запроса на языке SQL с вложенным подзапросом. Во вложенном подзапросе выбирается значение поля СЛЖ_ОТД_НОМЕР из записи файла СЛУЖАЩИЕ, в которой значение поля СЛЖ_ИМЯ равняется строковой константе 'ПЕТР ИВАНОВИЧ ФЕДОРОВ'. Если такая запись существует, то она единственная, поскольку поле СЛЖ_ИМЯ является уникальным ключом файла СЛУЖАЩИЕ. Тогда результатом выполнения подзапроса будет единственное значение – номер отдела, в котором работает Петр Иванович Федоров. Во внешнем запросе это значение будет ключом доступа к файлу ОТДЕЛЫ, и снова будет выбрана только одна запись, поскольку поле ОТД_НОМЕР является уникальным ключом файла ОТДЕЛЫ. Если же на данном предприятии Петр Иванович Федоров не работает, то подзапрос выдаст пустой результат, и внешний запрос тоже выдаст пустой результат.

Приведенные примеры показывают, что при формулировке запроса с использованием SQL можно не задумываться о том, как будет выполняться этот запрос. Среди метаданных базы данных будет содержаться информация о том, что поле СЛЖ_ИМЯ является ключевым для файла СЛУЖАЩИЕ (т. е. по заданному значению имени служащего можно быстро найти соответствующую запись или убедиться в том, что запись с таким значением поля СЛЖ_ИМЯ в файле отсутствует), а поле ОТД_НОМЕР – ключевое для файла ОТДЕЛЫ (и более того, оба ключа в соответствующих файлах являются уникальными), и система сама воспользуется этим.

Наиболее вероятным способом выполнения запроса в обеих формулировках будет выборка записи из файла СЛУЖАЩИЕ со значением поля СЛЖ_ИМЯ, равным строке 'ПЕТР ИВАНОВИЧ ФЕДОРОВ', взятие из этой записи значения поля СЛЖ_ОТД_НОМЕР и выборка из таблицы ОТДЕЛЫ записи с таким же значением поля ОТД_НОМ.

Если же, например, возникнет потребность в получении списка служащих, должность которых еще не определена, то достаточно обратиться к системе с запросом (запрос3):

```
SELECT СЛЖ_ИМЯ, СЛЖ_НОМЕР
FROM СЛУЖАЩИЕ
WHERE СЛЖ_СТАТ = "НЕТ";
```

и система сама выполнит необходимый полный просмотр файла СЛУЖАЩИЕ, поскольку поле СЛЖ_СТАТ не является ключевым, и другого способа выполнения не существует.

Это говорит о том, что мощность языка SQL такова, что позволяет сформулировать любой запрос для получения необходимой информации, в то время как в навигационных СУБД для этого пришлось бы писать программный код.

1.2.4. Метаданные, унифицированные процедуры и язык SQL

Следует отметить, что возможность разработки языка SQL появилась благодаря наличию в СУБД модели данных, представляющей данные о таблицах базы данных, их столбцах и строках, о взаимосвязях между таблицами и т. д. в фиксированных таблицах системного каталога.

Когда мы используем термин «язык», мы предполагаем некую программу, обеспечивающую интерпретацию текста запроса, и выполнение инструкций данного языка, содержащихся в тексте запроса. Если взять в качестве примера последний запрос (запрос3), то для ее интерпретации и выполнения необходимо выполнить следующие шаги:

– определить процедуру, которая должна выполнить данный запрос на языке SQL. В нашем случае это будет процедура, которую условно назовем SELECT;

– в качестве входного атрибута этой процедуры по ключевому слову FROM определяется имя таблицы СЛУЖАЩИЕ;

– в качестве критерия выборки по ключевому слову WHERE определяется условие СЛЖ_СТАТ = "НЕТ";

– в качестве выходных данных процедуры после ключевого слова SELECT определяются поля СЛЖ_ИМЯ, СЛЖ_НОМЕР выбранных записей;

– после определения всех необходимых данных процедура запускается и после выполнения всех необходимых операций на физическом уровне пользователь получает результат в виде списка значений для полей СЛЖ_ИМЯ, СЛЖ_ НОМЕР.

К процедурам языка предъявляются весьма жесткие требования, так как они должны обеспечить выполнение произвольного синтаксически и семантически правильно сформулированного запроса.

Чтобы описать принципы разработки таких процедур поставим перед собой более простую задачу: *разработать высокоуровневый язык для работы со списками*. С помощью этого языка мы должны иметь возможность создавать и уничтожать списки, а также вставлять в них записи, модифицировать и удалять введенные записи.

Сразу напрашивается решение:

– разработать процедуры для создания и уничтожения списков, а также для вставки, модификации и удаления записей в списках;

– определить спецификации этих процедур в качестве языковых конструкций для работы со списками. Например, эта спецификация могла бы иметь вид:

```
CREATE LIST («Name: СТУДЕНТЫ», «Fields: ФИО, Группа»);
DROP LIST («Name: СТУДЕНТЫ»);
INSERT («Список: СТУДЕНТЫ», «ФИО: Иванов И.И.», «Группа: ЭВМд-31»);
UPDATE («Список: СТУДЕНТЫ», «ФИО: Петров П.П.»);
SELECT (Группа FROM «СТУДЕНТЫ», WHERE «ФИО: Петров П.П.»);
DELETE (FROM «СТУДЕНТЫ», WHERE «ФИО = Петров П.П.»).
```

Эта, на первый взгляд, несложная задача не так просто решается. Если бы эта программа была ориентирована на работу с одним списком, разработка процедур, выполняющих операции с этим списком не представляла бы никакой сложности. Но в нашем случае мы сталкиваемся тем обстоятельством, что эти

операции, зависящие от состава и размеров записей, должны выполняться со списками произвольной структуры. Например, при поиске в списке по некоторому ключевому значению, производится сканирование списка. А для этого после очередного сравнения ключевого значения некоторого поля с текущим значением требуется осуществить смещение адреса, зависящее от размера записи.

Поэтому при разработке унифицированной процедуры поиска записей списков необходимо предусмотреть конструкции для хранения сведений о структуре текущего списка, с тем, чтобы по этой информации она могла автоматически вычислять смещение адреса при сканировании произвольного списка.

Для хранения таких данных мы должны выделить таблицы с фиксированными полями и записать в эти таблицы всю необходимую информацию о структуре списка:

– наименование файла, в котором хранится список;
– наименования полей записей;
– типы данных полей, что и определяет размер каждого поля записи;
– общий размер записи, определяемый по размерам отдельных полей.

Назовем таблицы, в которых хранятся данные о структурах записей списков базой метаданных, а сведения, хранящиеся в этих таблицах, метаданными или моделью данных списка. Далее для полученной базы метаданных разработаем унифицированные процедуры, использующие метаданные при выполнении типовых операций со списками.

В этом случае мы имеем возможность ввести сведения о структурах любых списков, с которыми работает пользователь, так как в нашем распоряжении имеется унифицированная процедура создания списка, обеспечивающая ввод сведений о списке с любой необходимой структурой. После этого с этим списком можно будет выполнять операции, реализуемые унифицированными процедурами.

В случае работы с табличными структурами хранение данных о произвольной базе данных в фиксированных таблицах, т. е. в строго формализованном виде, также позволяет строить унифицированные процедуры для обработки данных базы данных, содержащей произвольное число таблиц, с любым числом атрибутов (в пределах допускаемых СУБД) и строк, между которыми могут иметь место различные взаимосвязи. Эти унифицированные процедуры составляют ядро языка `SQL`, обеспечивают унифицированную обработку произвольных данных, хранящихся в таблицах баз данных, и вызываются с помощью ключевых слов `CREATE`, `DROP`, `SELECT`, `INSERT`, `UPDATE`, `DELETE` и т. д.

1.3. СУБД КАК НЕЗАВИСИМЫЙ СИСТЕМНЫЙ КОМПОНЕНТ

В ходе эволюции файловой системы, а затем и концепции баз данных, фрагмент логики программы, отвечающий за управление данными, был выделен в отдельный компонент, названный СУБД. Одно из важнейших требований, предъявляемых к СУБД – это обеспечение независимости данных от программ; при этом все данные должны храниться в соответствии с некоторой стандартной внутренней структурой, доступ к которым может быть предоставлен всем прикладным программам, нуждающимся в этих данных.

СУБД изолируют данные от прикладных программ таким образом, что при изменении данных не нужно менять программу, либо реорганизовывать данные. Для работы с данными в распоряжение пользователей предоставляется язык запросов, с помощью которого они могут стандартным образом выбирать и изменять данные.

1.3.1. СУБД как средство обеспечения логической и физической независимости данных

Как мы уже отметили, база данных хранит не только рабочие данные организации, но и их описания. По этой причине базу данных еще называют набором интегрированных записей с самоописанием. В современных СУБД описание данных содержится в системном каталоге, а сами элементы описания принято называть метаданными, т. е. «данными о данных». Наличие самоописания данных обеспечивает независимость программ от данных и позволяет СУБД различать и поддерживать *логическую* и *физическую* независимость данных.

Логическая независимость данных означает, что общая структура данных может быть изменена без изменения прикладных программ. Представление данных в приложении не должно зависеть от реальной структуры реляционных таблиц. Если в процессе нормализации одна реляционная таблица разделяется на две, то с помощью соответствующего представления можно так объединить эти данные, чтобы изменение структуры реляционных таблиц не сказывалось на работе приложений.

Например, рассмотрим программу, использующую в своей работе данные таблицы *T*, которую администратор желает реорганизовать путем вертикального разделения на две части, сохраняемые в таблицах T^1 и T^2. Для сохранения работоспособности программы, зависящей от таблицы *T,* администратор может определить представление (`view`) над таблицами T^1 и T^2, соответствующее исходному определению таблицы *T*. Хотя реальная таблица *T* была удалена из базы данных в ходе реорганизации, но с помощью инструкции `CREATE VIEW` можно создать виртуальную таблицу *T*, которая обеспечивает программе доступ к тем же данным, и теми же средствами, что и исходная реальная таблица *T*. Таким образом можно обеспечить корректную работу старых программ при изменении логической схемы базы данных.

Под ***физической независимостью данных*** понимается способность СУБД предоставлять некоторую свободу модификации способов организации базы данных в среде хранения (например, с целью повышения эффективности баз данных), не вызывая необходимости внесения соответствующих изменений в логическое представление и не затрагивая созданных прикладных программ, использующих базы данных. Приложения не должны зависеть от используемых способов хранения данных на носителях, используемых разработчиками различных СУБД.

Таким образом, СУБД находится между «логикой» и «физикой», так же как и файловая система. Но если в первом случае на логическом уровне были уни-

фицированы запись и считывание данных, то во втором случае унифицированы операции с данными, учитывающие структуру и содержание данных.

Этот аспект использования СУБД, связанный с независимостью данных, определяется разным подходом к интерпретации данных в файловой системе и СУБД. В информационных системах, основанных на файловой системе, интерпретирующая информация закладывалась в программы, использующие данные. Это существенно повышает значимость программы, так как вне интерпретации данные представляют не более чем совокупность битов в некотором запоминающем устройстве. В условиях совместного использования данных при множестве различных приложений указанный подход можно применять только до определенного предела, так как написание множества программ, в которые встроены практически одинаковые механизмы интерпретации, становится весьма неэффективным.

В СУБД данные ассоциированы с механизмами их интерпретации. Поэтому не требуется закладывать интерпретирующую информацию в каждое приложение, использующее данные из базы данных. Это обеспечивает однократность представления интерпретирующей информации и существенно меняет роль данных. Данные из баз данных уже нельзя рассматривать как совокупность битов, они приобретают определенную интеллектуальную окраску. В таком качестве их можно расценивать как семантически значимое представление части реального мира.

Существующие на сегодняшний день модели данных дают возможность представить лишь частичную семантику данных. Можно, конечно, считать, что этого недостаточно, и пытаться найти более привлекательную модель, которая полностью отражала бы семантику реального мира. Однако создание такой модели дело будущего, и приходится работать с существующими моделями данных, а недостающую семантику, как и прежде, закладывать в неявном виде в обрабатывающие программы.

1.3.2. СУБД в составе информационной системы

Отделение логической структуры от физической в рамках файловой системы, а затем обеспечение независимости программ от данных в рамках концепции баз данных привело к появлению нескольких уровней представления информации. В соответствии с этим информационные системы, реализованные на основе баз данных, также рассматриваются как сложное многоуровневое программное обеспечение.

На самом верхнем уровне прикладная программа формулирует свои запросы на языке SQL, используя термины логической схемы базы данных, и направляет на вход СУБД.

Интерфейсная составляющая СУБД проводит синтаксический и семантический анализ запроса с использованием метаданных и определяет унифицированную процедуру, отвечающую за выполнение запроса.

Унифицированная процедура СУБД, в соответствии с атрибутами, заданными в запросе, выполняет запрос на уровне физической схемы в памяти машины.

До сих пор мы не вычленяли СУБД из состава информационной системы, имея в виду общую организацию системы, подобную той, которая показана на Рис. 1.8. .

Данной архитектуре информационной системы свойственны два дефекта:

– во-первых, очевидно, что СУБД должна поддерживать достаточно развитую функциональность. Повторять эту функциональность в каждой информационной системе неразумно;

– во-вторых, набор файлов можно назвать базой данных только при наличии метаданных, следовательно, метаданные принадлежат только данной информационной системе. Применительно к нашей информационной системе, в этом случае файлы СЛУЖАЩИЕ и ОТДЕЛЫ можно эффективно использовать только через нашу гипотетическую систему регистрации служащих.

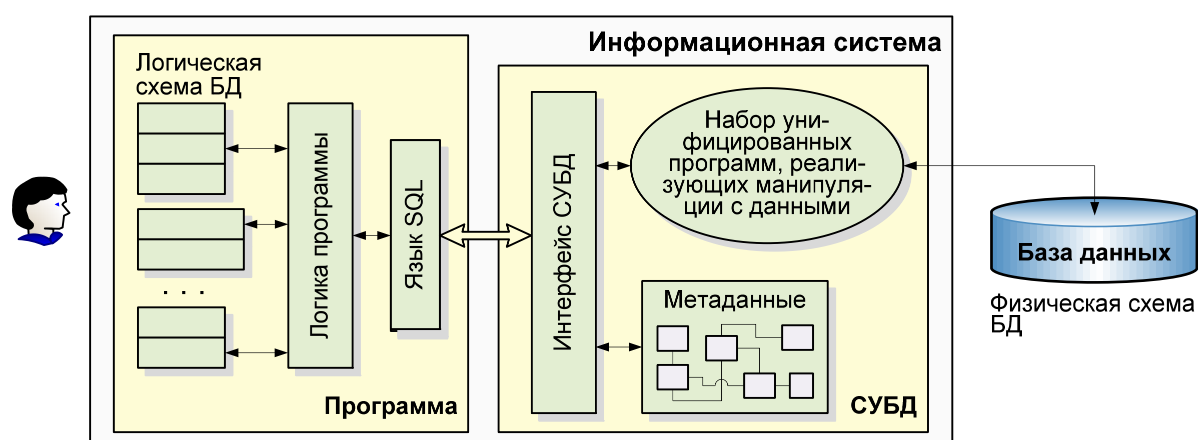


Рис. 1.8. СУБД в составе информационной системы

1.3.3. Выделение СУБД в качестве отдельного компонента информационной системы

Предположим, что предприятию нужна еще и информационная система для бухгалтерского учета. Очевидно, что для ее работы также потребуются данные о служащих и отделах. При показанной выше организации системы возможны два варианта выполнения задачи, ни один из которых не является удовлетворительным.

1. Внедрить бухгалтерскую систему в состав системы регистрации служащих. Но, как правило, бухгалтерские системы покупаются в виде готовых и отдельных продуктов, не приспособленных к подобному «внедрению».

2. Скопировать метаданные системы регистрации служащих в бухгалтерскую систему. Но метаданные (как и данные) не обязательно являются статичными. Структура базы данных может со временем изменяться, могут исчезать одни правила целостности и появляться другие. Поэтому в данном варианте возникает проблема согласования копий метаданных, поддерживаемых независимыми информационными системами.

Так мы приходим к организации системы, показанной на Рис. 1.9. . Здесь мы видим три информационные системы, которые через одну СУБД работают с двумя разными базами данных, причем первая и вторая системы работают с общей базой данных. Это возможно, поскольку метаданные каждой базы данных содержатся в самих базах данных, и достаточно лишь указать СУБД, с какой базой данных желает работать данное приложение.

Поскольку СУБД функционирует отдельно от приложений, и ее работа с базами данных регулируется метаданными, совместное использование одной базы данных двумя информационными системами не вызовет потери согласованности данных, и доступ к данным будет должным образом синхронизироваться.

Заметим, что схема, приведенная на Рис. 1.9. , вплотную приближает нас к наиболее распространенной в последние десятилетия архитектуре «клиент-сервер». СУБД играет роль «сервера», обсуживающего нескольких «клиентов» – прикладных информационных систем.

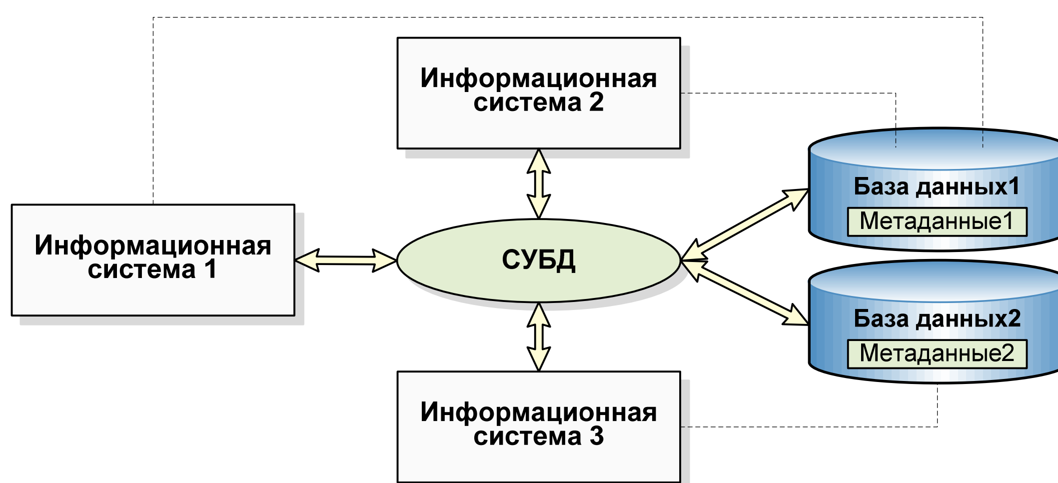


Рис. 1.9. Отдельная СУБД и база данных с метаданными

Таким образом, выделение СУБД в самостоятельный компонент информационной системы решает множество проблем, которые затруднительно или вообще невозможно решить при использовании СУБД, встроенных в информационные системы.

ГЛАВА 2. ЛОГИЧЕСКИЕ СТРУКТУРЫ РЕЛЯЦИОННОЙ МОДЕЛИ

При изучении реляционных баз данных следует иметь в виду, что существует разница между теорией баз данных и их проектированием. Теория включает в себя принципы и правила, которые определяют основу реляционной модели базы данных. Именно это изучается в залах академий, а затем быстро забывается в «реальном мире». Но теория, все-таки, важна: она гарантирует, что реляционные базы данных имеют надежную структуру и что все действия, предпринимаемые над данными, дают предсказуемые результаты.

С другой стороны, проектирование баз данных включает в себя структурированный, организованный набор процессов, которые используются для этого проектирования.

Поэтому в данной главе сначала вкратце рассмотрим теоретические вопросы баз данных и объекты баз данных, вытекающие из теоретических построений. Затем рассмотрим основные понятия реляционных баз данных, методические вопросы проектирования баз данных, обеспечивающие целостность и непротиворечивость данных, отсутствие аномалий при внесении изменений в базы данных, а также приведем описание реализации базы метаданных в виде системного каталога в ряде современных СУБД.

2.1. ОСНОВЫ РЕЛЯЦИОННОЙ АЛГЕБРЫ

Еще в 70-х гг. под влиянием предложенной в то время концепции абстрактных типов понятие типа данных стало трансформироваться таким образом, что в него стали вкладывать не только структурные свойства, но и элементы поведения (изменения данных). Другими словами, инструмент моделирования баз данных должен включать не только средства структурирования данных, но и средства манипулирования данными. Поэтому модель данных в инструментальном смысле стала пониматься как алгебраическая система − множество всевозможных типов данных, а также определенных на них отношений и операций. Позднее в это понятие стали включать еще и ограничения целостности, налагаемые на данные.

Единственным средством структуризации данных в реляционной модели

является отношение. В математике отношение определяется как подмножество декартова или прямого произведения. Формальное определение отношения будет приведено ниже, а пока опишем отношение как таблицу, составленную из m полей и n строк.

Почему мы можем определить отношение как таблицу? Потому что с помощью таблицы можно описать объект, задаваемый m свойствами, некоторые комбинации значений которых задают n экземпляров объекта. Другими словами, объект определяется не как декартово произведение значений свойств, а как некоторая его часть, задаваемая путем наложения некоторых ограничений. Эти ограничения и определяют отношение.

Например, таблица `СТУДЕНТЫ_ГРУППЫ`, составленная из полей `Фамилия`, `Имя`, `Отчество`, `Группа`, может содержать только те записи, которые содержат комбинации значений этих полей, соответствующие студентам, обучающимся в некоторой группе, а не произвольные комбинации их значений.

Теоретической основой табличного представления данных является алгебра отношений или реляционная алгебра, в которой в качестве элементов рассматриваются таблицы, а в качестве операций − операции объединения, вычитания, пересечения, декартова произведения, проекции и селекции.

2.1.1. Объекты и их определения

Перейдем к рассмотрению структурной части реляционной модели данных. Прежде всего, необходимо дать несколько определений. Для понимания истинного смысла термина отношение рассмотрим несколько математических понятий.

Множество

Реляционная МД основана на математическом понятии *отношения*, которое используется для хранения информации об объектах, представленных в базах данных. Понятие отношения выводится из понятия *«множество»*. Множество представляет собой наиболее простую структуру, когда между отдельными изолированными объектами отсутствуют какие-либо внутренние связи. Другими словами, множество не обладает структурой. Оно представляет собой только совокупность данных определенного типа, обладающих некоторым свойством.

Домены

Однако должны быть четко установлены область определения данных, и правила определения принадлежности данных к множеству. Такое множество называется доменом. *Домен* – это некоторое множество элементов, например, множество целых чисел или множество допустимых значений, которые может принимать объект по некоторому свойству. Каждый атрибут реляционной базы данных определяется на некотором домене. Понятие домена имеет большое значение, поскольку благодаря нему пользователь может определять смысл и источник значений, которые могут получать атрибуты. В результате при выполнении реляционной операции системе доступно больше информации, что

позволяет избежать семантически некорректных операций. Например, бессмысленно сравнивать название улицы с номером телефона, даже если для обоих этих атрибутов определениями являются символьные строки.

Обратите внимание на то, что для определения этих отношений необходимо указать множества, или домены, из которых выбираются значения. Таким образом, домен представляет собой семантическое понятие, которое можно рассматривать как подмножество значений некоторого типа данных, имеющих определенный смысл. Домен характеризуется следующими свойствами:

– имеет уникальное имя (в пределах базы данных);
– определен на некотором простом типе данных или на другом домене;
– может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для этого домена.

Декартово произведение

Допустим, у нас есть два множества, D_1 и D_2, где $D_1 = \{2, 4\}$ и $D_2 = \{1, 3, 5\}$. Декартовым произведением этих двух множеств $D_1 \times D_2$ называется набор из всех возможных пар, в которых первым идет элемент множества D_1, а вторым – элемент множества D_2: $D_1 \times D_2 = \{(2,1), (2,3), (2,5), (4,1), (4,3), (4,5)\}$.

Увеличивая количество множеств, можно дать обобщенное определение отношения на *n* доменах. Пусть имеется *n* множеств D_1, D_2, ..., D_n. Декартово произведение этих *n* множеств можно определить следующим образом:

$$D_1 \times D_2 \times \cdots \times D_n = \{(d_1, d_2, ..., d_n) \mid d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n\}.$$

Отношение

Отношение *R* представляет собой двумерную таблицу, содержащую некоторые данные. Математически любое подмножество *n*-арных кортежей декартова произведения, является отношением *n* множеств.

Отношением *R,* определенным на множествах $D_1, D_2, ..., D_n$, называется подмножество декартова произведения $D_1 \times D_2 \times \cdots \times D_n$. При этом:

– множества $D_1, D_2, ..., D_n$ называются доменами отношения;
– элементы декартова произведения $\{d_1, d_2, ..., d_n\}$ называются кортежами;
– число *n* определяет степень отношения;
– количество кортежей называется мощностью отношения.

Атрибут отношения

Атрибут отношения представляет собой пару вида `<Имя атрибута: Имя домена>` (либо $< A : D >$). Имена атрибутов должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов.

Заголовок и тело отношения

Отношение *R,* определенное на множестве доменов, содержит две части: заголовок и тело. *Заголовок отношения* − это фиксированное количество атрибутов отношения, описывающее декартово произведение доменов, на котором задано отношение

$$(< A_1 : D_1 >, < A_2 : D_2 >, ..., < A_n : D_n >).$$

Заголовок статичен: он не меняется во время работы с базами данных.

Тело отношения содержит множество кортежей отношения. Каждый кортеж отношения представляет собой множество пар вида `<Имя атрибута: Значение>` (либо $< A : Z >$)

$$R(< A_1 : Z_1 >, < A_2 : Z_2 >, ..., < A_n : Z_n >)$$

таких, что значение Z_i атрибута A_i принадлежит домену D_i. Тело отношения представляет собой множество кортежей, т. е. подмножество декартова произведения доменов.

Рис. 2.1. Наглядное представление основных понятий реляционной модели

Таким образом, тело отношения собственно и является отношением в математическом смысле слова. Тело отношения может изменяться во время работы с базой данных, так как кортежи могут изменяться, добавляться и удаляться.

Число атрибутов в отношении называется степенью (либо арностью) отношения, а множество кортежей отношения кардинальностью (либо мощностью) отношения.

Таблицы в базах данных

В соответствии с реляционной моделью данные в реляционной базе данных сохраняются в отношениях, которые воспринимаются пользователем как таблицы. Каждое отношение состоит из кортежей (записей) и атрибутов (полей). Пример таблицы приведен на Рис. 2.1. .

Основными структурами в базе данных являются таблицы. При записи отношения в виде таблицы имена атрибутов $A_1, A_2, ..., A_n$ перечисляются в заголовках столбцов, а кортежи образуют строки формата $(d_1, d_2, ..., d_n)$, где каждое значение берется из соответствующего домена.

Таким образом, в реляционной МД отношение можно представить как произвольное подмножество декартова произведения доменов атрибутов, тогда как таблица − это всего лишь представление такого отношения.

Таблицы в реляционной МД являются логическими, а не физическими структурами. На физическом уровне система может использовать любую из существующих структур памяти (последовательный файл, индексирование, хеширование, цепочку указателей и т. п.), лишь бы существовала возможность отображать эти структуры в виде таблицы на логическом уровне.

Таблицы представляют собой абстракцию способа физического хранения данных, в которой множество деталей на уровне памяти скрыто от пользователя. К скрытым деталям относятся: размещение хранимых записей, кодировка хранимых данных, хранимые структуры доступа, такие как индексы, и т. д.

Данные в таблицах удовлетворяют следующим принципам:

– Каждое значение, содержащееся на пересечении строки и колонки, должно быть атомарным (т. е. не расчленяемым на несколько значений).

– Значения данных в одной и той же колонке должны принадлежать к одному типу, доступному для исполнения в данной СУ*БД*.

– Каждая запись в таблице уникальна, т. е. в таблице не существует двух записей с полностью совпадающим набором значений ее полей.

– Каждое поле имеет уникальное имя.

– Последовательность полей в таблице несущественна.

Несмотря на то, что строки таблиц считаются неупорядоченными, любая СУ*БД* позволяет сортировать строки и колонки в выборках из нее нужным пользователю способом. Поскольку последовательность колонок в таблице не существенна, обращение к ним производится по имени, и эти имена для данной таблицы уникальны.

2.1.2. Операторы

Объединение отношений

Объединение отношений $R1$ и $R2$ выражается формулой $R = R1 \cup R2$. Операция объединения отношений применяется только к отношениям одинаковой арности. Результирующее отношение R получается той же арности (см. Рис. 2.2.).

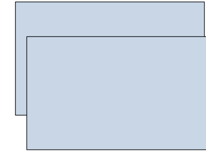
R1

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
3	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
4	Николаев Н.Н.	1977	Ассистент	ИСЭ

R2

№	Ф.И.О.	Год	Должность	Каф.
1	Игонин И.И.	1950	Зав. Каф.	ВТ
2	Пронин П.П.	1953	Проф.	ВТ
3	Федулов Ф.Ф.	1955	Проф.	ИСЭ
4	Петров П.П.	1959	Доцент	ВТ
5	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ

R = R1 U R2



№	Ф.И.О.	Год	Долж-ность	Каф.
1	Игонин И.И.	1950	Зав. Каф.	ВТ
2	Пронин П.П.	1953	Проф.	ВТ
3	Федулов Ф.Ф.	1955	Проф.	ИСЭ
4	Петров П.П.	1959	Доцент	ВТ
5	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
6	Иванов И.И.	1960	Доцент	ВТ
7	Николаев Н.Н.	1977	Ассистент	ИСЭ

Рис. 2.2. Объединение отношений

Разность отношений

Разность отношений R1 и R2 выражается формулой $R = R1 \setminus R2$. Разностью $R1 \setminus R2$ называется множество кортежей, принадлежащих *R1*, но не принадлежащих *R2*. Отношения *R1* и *R2* должны быть одинаковой арности. Отношение *R* – той же арности (см. Рис. 2.3.).

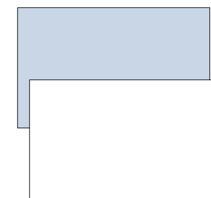
R1

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
3	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
4	Николаев Н.Н.	1977	Ассистент	ИСЭ
5	Пронин П.П.	1953	Проф.	ВТ

R2

№	Ф.И.О.	Год	Должность	Каф.
1	Игонин И.И.	1950	Зав. Каф.	ВТ
2	Пронин П.П.	1953	Проф.	ВТ
3	Федулов Ф.Ф.	1955	Проф.	ИСЭ
4	Петров П.П.	1959	Доцент	ВТ
5	Сидоров П.Ф.	1960	Ст. преп.	ИСЭ

R = R1 / R2



№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
3	Николаев Н.Н.	1977	Ассистент	ИСЭ

Рис. 2.3. Разность отношений

Пересечение отношений

Пересечение отношений $R1$ и $R2$ возвращает отношение, содержащее все кортежи, которые принадлежат одновременно двум заданным отношениям, и выражается формулой $R = R1 \cap R2$ (см. Рис. 2.4.)

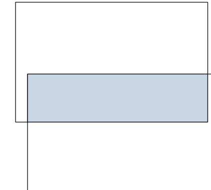
R1

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
3	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
4	Николаев Н.Н.	1977	Ассистент	ИСЭ
5	Пронин П.П.	1953	Проф.	ВТ

R2

№	Ф.И.О.	Год	Должность	Каф.
1	Игонин И.И.	1950	Зав. Каф.	ВТ
2	Пронин П.П.	1953	Проф.	ВТ
3	Федулов Ф.Ф.	1955	Проф.	ИСЭ
4	Петров П.П.	1959	Доцент	ВТ

R = R1 ∩ R2



№	Ф.И.О.	Год	Должность	Каф.
1	Петров П.П.	1959	Доцент	ВТ
2	Пронин П.П.	1953	Проф.	ВТ

Рис. 2.4. Пересечение отношений

Произведение отношений

Декартово произведение – операция, заключающаяся в построении нового отношения на основе двух других путем попарной комбинации всех возможных записей из первого отношения и второго отношения.

R1

Должность
Зав.каф.
Проф.
Доцент
Ст. преп.
Ассистент

R2

Каф.
ВТ
ИСЭ

R = R1 × R2

Должность	Каф.
Зав.каф.	ВТ
Проф.	ВТ
Доцент	ВТ
Ст. преп.	ВТ
Ассистент	ВТ
Зав.каф.	ИСЭ
Проф.	ИСЭ
Доцент	ИСЭ
Ст. преп.	ИСЭ
Ассистент	ИСЭ

Рис. 2.5. Произведение отношений

Если отношение R_1 имеет I записей и арность k_1, а R_2 – J записей и арность k_2, то декартовым произведением отношений R_1 и R_2 является множество $I*J$ кортежей арности $(k_1 + k_2)$ (см. Рис. 2.5.).

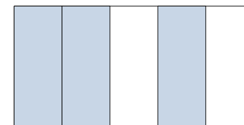
Проекция отношения на компоненты

Операция проекции заключается в том, что из отношения R_1 выбираются указанные столбцы и компонуются в указанном порядке, т.е проекция это операция, заключающаяся в удалении некоторых столбцов в отношении.

R1

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
3	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
4	Николаев Н.Н.	1977	Ассистент	ИСЭ
5	Пронин П.П.	1953	Проф.	ВТ

R



№	Ф.И.О.	Должность
1	Иванов И.И.	Доцент
2	Петров П.П.	Доцент
3	Федоров Ф.Ф.	Ст. преп.
4	Николаев Н.Н.	Ассистент
5	Пронин П.П.	Проф.

Рис. 2.6. Проекция отношений

Смысл операции проекции заключается в выделении из отношения той информации, которая нам нужна. Эта операция используется в операторе `SELECT` языка `SQL` при выборке значений требуемых полей (см. Рис. 2.6.).

Выборка отношения

Операция, заключающаяся в удалении некоторых записей в отношении на основе некоторого условия называется селекцией. Условие определяется как логическое выражение, включающее значения атрибутов. Например, селекция отношения R_1 по формуле F: $R = \sigma_F(R1)$, где F – формула, образованная:

– операндами, являющимися номерами столбцов;
– логическими операторами *И* , *ИЛИ* , *НЕ* ;
– арифметическими операторами сравнения $<, =, >, \leq, \neq, \geq$ (см. 0).

R1

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
3	Федоров Ф.Ф.	1960	Ст. преп.	ИСЭ
4	Николаев Н.Н.	1977	Ассистент	ИСЭ
5	Пронин П.П.	1953	Проф.	ВТ

R

№	Ф.И.О.	Год	Должность	Каф.
1	Иванов И.И.	1960	Доцент	ВТ
2	Петров П.П.	1959	Доцент	ВТ
5	Пронин П.П.	1953	Проф.	ВТ

Рис. 2.7. Выборка отношений

2.2.ОСНОВНЫЕ ПОНЯТИЯ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

2.2.1. Учебная база данных

Дальнейшее изложение основ реляционной модели данных проведем на основе учебной базы данных, содержащей пять таблиц. В каждой таблице содержится информация об одном типе сущности:

– в таблице CLIENTY (Клиенты) хранятся данные о клиентах, которые покупают товары компании (ID_CLN (номер клиента), COMPANY (название компании), ID_SLZH (номер служащего, отвечающего за связь с компанией), LIMIT_CREDIT (лимит кредита));

– в таблице SLUZHASCHIE (Служащие) хранятся данные о служащих, работающих в компании ((ID_SLZH (номер служащего), FAMILY (фамилия служащего), NAME (имя служащего), AGE (возраст), MNGR (номер менеджера), ID_OFC (номер офиса), DLZH (должность), QUOTA (план), SALES (продажи служащего));

– в таблице OFFISY (Офисы) хранятся данные об офисах компании, в которых работают служащие ((ID_OFC (номер офиса), CITY (город), REGION (регион), MNGR (номер менеджера), TARGET (план продаж офиса), SALES (продажи офиса));

– в таблице TOVARY (Товары) хранятся данные о товарах, продаваемых компанией ((ID_MFR (номер производителя), ID_PRD (номер товара), DESCRIPTION (описание), PRICE (цена), COUNT (количество товара на складе));

– в таблице ZAKAZY (Заказы) хранится информация о заказах, сделанных клиентом, (ID_ORDER (номер заказа), DATE_ORDER (дата заказа), ID_CLN (номер клиента), ID_SLZH (номер служащего), ID_MFR (номер производителя), ID_PRD (номер продукта), COUNT (количество заказанного товара), PRICE (стоимость)).

Фрагмент учебной базы данных показан на 0, а более подробное описание приведено в Приложении.

2.2.2. Первичные ключи

Поскольку строки в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру в таблице. В таблице нет «первой», «последней» или «три-

дцатой» строки. Тогда каким же образом можно выбрать в таблице конкретную строку, например, строку для офиса, расположенного в Инзе?

В правильно построенной базе данных в каждой таблице есть один или несколько столбцов, значения которых во всех строках разные. Этот столбец (столбцы) называется первичным ключом таблицы. В нашей учебной базе данных на первый взгляд, первичным ключом таблицы OFFISY могут служить и столбец ID_OFC, и столбец CITY. Однако если компания будет расширяться и откроет в каком-либо городе второй офис, столбец CITY больше не сможет исполнять роль первичного ключа. На практике в качестве первичных ключей таблиц обычно следует выбирать идентификаторы, такие как идентификатор офиса (ID_OFC в таблице OFFISY), служащего (ID_OFC в таблице SLUZHASCHIE) и клиента (ID_CLN в таблице CLIENTY). А в случае с таблицей ZAKAZY нет выбора – единственным столбцом, содержащим уникальные значения, является номер заказа (ID_ORDER).

Таблица CLIENTY

ID_CLN	COMPANY	ID_SLZH	LIMIT_CREDIT
12111	«Заря»	2103	$50000.00
12102	«Гранит»	2101	$65000.00
12103	«Базальт»	2105	$50000.00

Таблица SLUZHASCHIE

ID_SLZH	FAMILY	NAME	AGE	ID_OFC	DLZH	MNGR	QUOTA	SALES
2109	Полев	Андрей	31	311	Брокер	2106	$300000.00	$392725.00
2102	Пронин	Игорь	48	321	Брокер	2108	$350000.00	$474050.00
2106	Петров	Петр	52	311	Гл.Брокер	NULL	$275000.00	$299912.00
2104	Иванов	Иван	33	312	Ст.Брокер	2106	$200000.00	$142594.00
2101	Федоров	Федор	45	312	Брокер	2104	$300000.00	$305673.00
2110	Уткин	Денис	41	NULL	Брокер	2101	NULL	$75985.00

Таблица OFFISY

ID_OFC	CITY	REGION	MNGR	TARGET	SALES
322	Инза	Ульяновская	2108	$300000.00	$186042.00
311	Буинск	Татарстан	2106	$575000.00	$692637.00
312	Тверь	Московская	2104	$800000.00	$735042.00

Таблица TOVARY

ID_MFR	ID_PRD	DESCRIPTION	PRICE	COUNT
УАЗ	2A45C	Деталь кузова	$79.00	210
ВАЗ	4100Y	Деталь двигателя	$2750.00	25
ПМЗ	XK47	Сопло	$355.00	38

Таблица ZAKAZY

ID_ORDER	DATE_ORDER	ID_CLN	ID_SLZH	ID_MFR	ID_PRD	COUNT	PRICE_ALL
312961	12/17/89	12117	2106	УАЗ	2A44L	7	$31500.00
313012	01/11/90	12111	2105	ВАЗ	41003	35	$3745.00
312989	01/03/90	12101	2106	УМЗ	114	6	$1458.00
313051	02/10/90	12118	2108	ПМЗ	XK47	4	$1420.00

Рис. 2.8. Фрагмент учебной базы данных

Таблица `TOVARY` является примером таблицы, в которой первичный ключ представляет собой комбинацию столбцов. Такой первичный ключ называется составным. Столбец `ID_MFR` содержит идентификаторы производителей всех товаров, перечисленных в таблице, а столбец `ID_PRD` содержит номера, присвоенные товарам производителями. Может показаться, что столбец `ID_PRD` мог бы и один исполнять роль первичного ключа, однако ничто не мешает двум разным производителям присвоить своим изделиям одинаковые номера. Таким образом, в качестве первичного ключа таблицы `TOVARY` необходимо использовать комбинацию столбцов `ID_MFR` и `ID_PRD`. Для каждого из товаров, содержащихся в таблице, комбинация значений в этих столбцах будет уникальной. Первичный

ключ для каждой строки таблицы является уникальным, поэтому в таблице с первичным ключом нет двух совершенно одинаковых строк.

2.2.3. Отношение предок/потомок

Одним из отличий реляционной модели от навигационных было то, что в ней отсутствовали явные указатели, используемые для реализации отношений предок/потомок в иерархической модели данных. Но такие отношения существуют и в реляционных базах данных. Например, в нашей учебной базе каждый из служащих закреплен за конкретным офисом, поэтому ясно, что между строками таблицы OFFISY и таблицы SLUZHASCHIE существует отношение. Не приводит ли отсутствие явных указателей в реляционной модели к потере информации?

Ответ на этот вопрос будет отрицательным. На 0 изображено несколько строк из таблиц OFFISY и SLUZHASCHIE. Обратите внимание на то, что в столбце ID_OFC таблицы SLUZHASCHIE содержится идентификатор офиса, в котором работает служащий. Доменом этого столбца (множеством значений, которые могут в нем храниться) является множество идентификаторов офисов, содержащихся в столбце ID_OFC таблицы OFFISY.

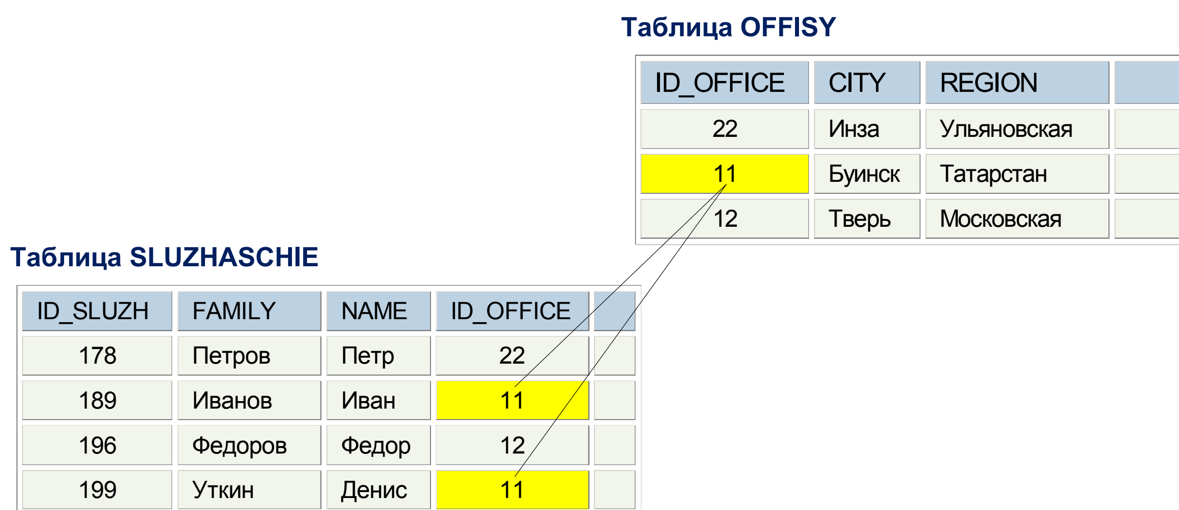


Таблица OFFISY

ID_OFFICE	CITY	REGION	
22	Инза	Ульяновская	
11	Буинск	Татарстан	
12	Тверь	Московская	

Таблица SLUZHASCHIE

ID_SLUZH	FAMILY	NAME	ID_OFFICE	
178	Петров	Петр	22	
189	Иванов	Иван	11	
196	Федоров	Федор	12	
199	Уткин	Денис	11	

Рис. 2.9. Реализация отношения предок/потомок в реляционной базе данных

Узнать, в каком офисе работает Иванов Иван, можно, определив значение столбца ID_OFC в строке таблицы SLUZHASCHIE для Иванов Иван (число 11), а затем отыскав в таблице OFFISY строку с таким же значением в столбце ID_OFC (это строка для офиса в Буинске). Таким же образом, чтобы найти всех служащих Буинского офиса, следует запомнить значение ID_OFC для Буинска (число 11), а потом просмотреть таблицу SLUZHASCHIE и найти все строки, в столбце ID_OFC которых содержится число 11 (это строки для Иванова Ивана и Уткина Дениса).

Отношение предок/потомок, существующее между офисами и работающими в них людьми, в реляционной модели не потеряно; просто оно реализо-

вано в виде одинаковых значений данных, хранящихся в двух таблицах, а не в виде явного указателя. Все отношения, существующие между таблицами реляционной базы данных, реализуются таким способом. Одним из главных преимуществ языка SQL является возможность извлекать данные, связанные между собой, используя эти отношения.

2.2.4. Внешние ключи

Столбец одной таблицы, значения в котором совпадают со значениями столбца, являющегося первичным ключом другой таблицы, называется *внешним (вторичным) ключом*. На 0 столбец ID_OFC представляет собой внешний ключ для таблицы OFFISY. Значения, содержащиеся в этом столбце, представляют собой идентификаторы офисов. Эти значения соответствуют значениям в столбце ID_OFC, который является первичным ключом таблицы OFFISY. Совокупно первичный и внешний ключи создают между таблицами такое же отношение предок/потомок, как и в иерархической базе данных.

Внешний ключ, как и первичный, тоже может быть составным (состоящим из нескольких столбцов). На практике внешний ключ всегда будет составным, если он ссылается на составной первичный ключ в другой таблице. Очевидно, что количество столбцов и их типы данных в первичном и внешнем ключах совпадают.

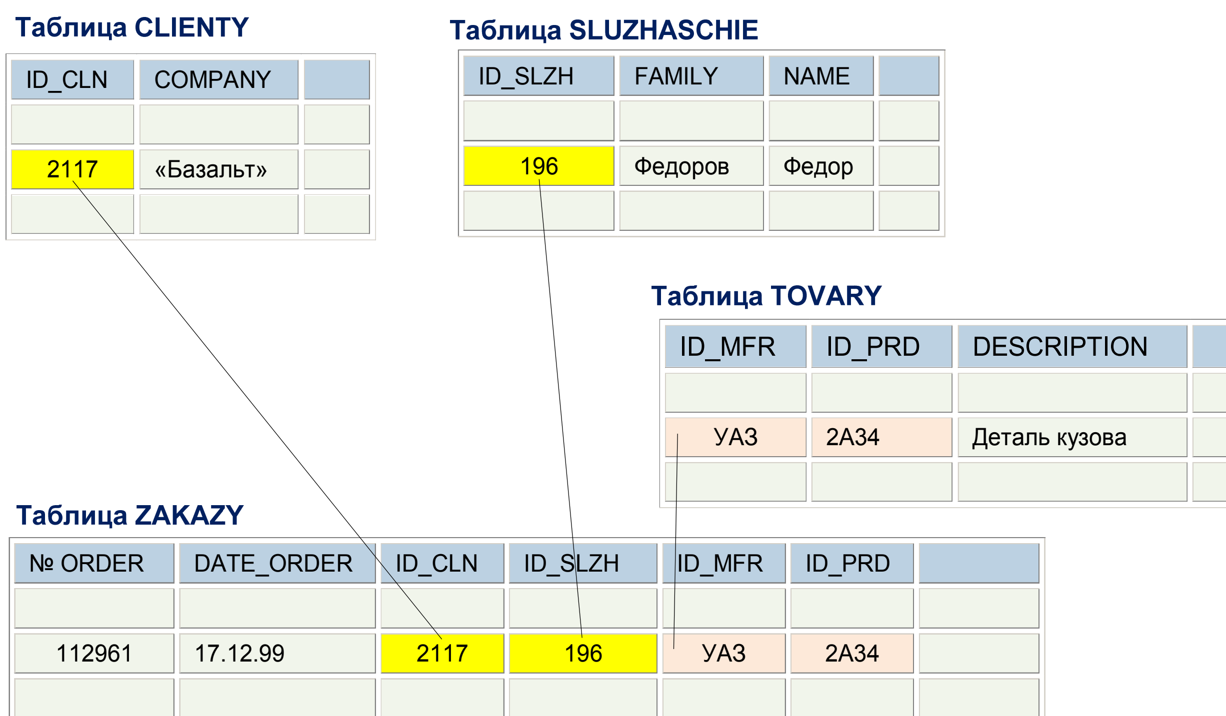


Рис. 2.10. Множественные отношения предок/потомок в реляционной базе данных

Если таблица связана с несколькими другими таблицами, она может иметь несколько внешних ключей. На 0 показаны три внешних ключа таблицы ZAKAZY из учебной базы данных:

– столбец ID_CLN является внешним ключом для таблицы CLIENTY и связывает каждый заказ с клиентом, сделавшим его;

– столбец ID_SLZH является внешним ключом для таблицы SLUZHASCHIE и связывает каждый заказ со служащим, принявшим его;

– столбцы ID_MFR и ID_PRD совокупно представляют собой составной внешний ключ для таблицы TOVARY, который связывает каждый заказ с заказанным товаром.

Внешние ключи являются неотъемлемой частью реляционной модели, поскольку реализуют отношения между таблицами базы данных.

2.2.5. Индексы

Одним из структурных элементов физической памяти, присутствующим в большинстве реляционных СУБД, является индекс. Индекс – это средство, обеспечивающее быстрый доступ к строкам таблицы на основе значений одного или нескольких столбцов. На 0 изображены таблица TOVARY и два созданных для нее индекса. Один из индексов обеспечивает быстрый доступ к таблице на основе столбца DESCRIPTION. Другой обеспечивает доступ на основе первичного ключа таблицы, представляющего собой комбинацию столбцов ID_MFR и ID_PRD.

СУБД пользуется индексом так же, как вы пользуетесь предметным указателем книги. В индексе хранятся значения данных и указатели на строки, где эти данные встречаются. Данные в индексе располагаются в убывающем или возрастающем порядке, чтобы СУБД могла быстро найти требуемое значение. Затем по указателю СУБД может быстро локализовать строку, содержащую ис-

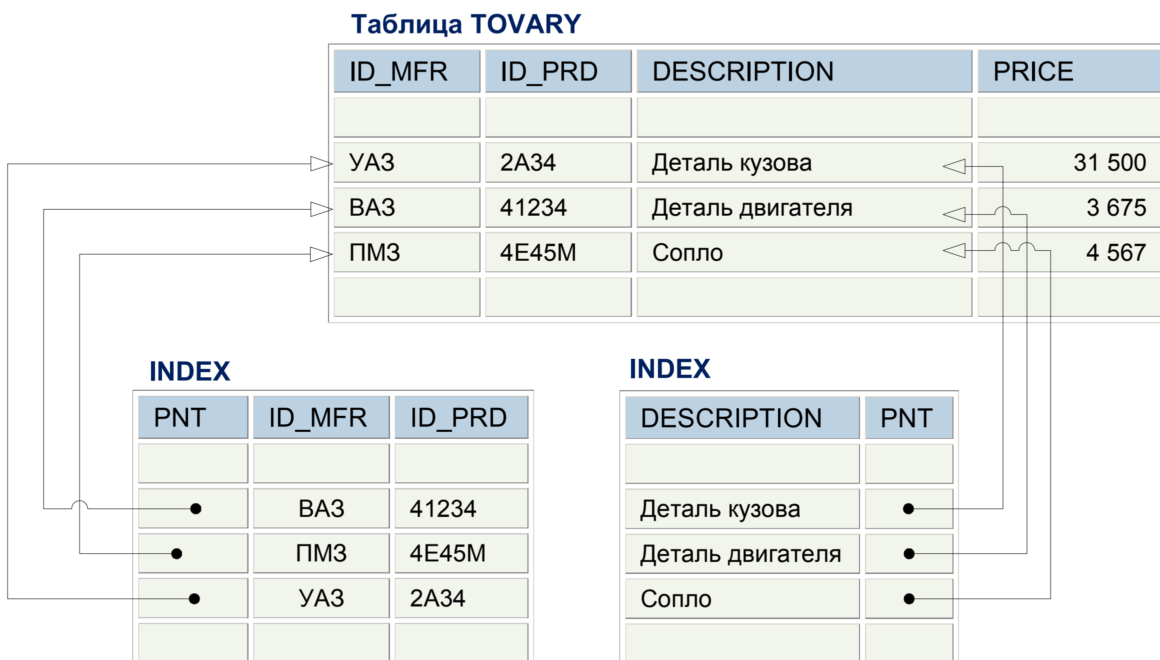


Рис. 2.11. Множественные отношения предок/потомок в реляционной базе данных

комое значение.

Наличие или отсутствие индекса совершенно незаметно для пользователя, обращающегося к таблице. Рассмотрим, например, такую инструкцию SELECT:

Найти количество и цену изделия 2A34.

```
SELECT COUNT, PRICE
FROM TOVARY
WHERE DESCRIPTION = `2A34`
```

В инструкции ничего не говорится о том, имеется ли индекс для столбца DESCRIPTION или нет, и СУБД выполнить запрос в любом случае.

Если бы индекса для столбца DESCRIPTION не существовало, то СУБД была бы вынуждена выполнять запрос путем последовательного «сканирования» таблицы TOVARY, строка за строкой, просматривая в каждой строке столбец DESCRIPTION. Для получения гарантии того, что она нашла все строки, удовлетворяющие условию отбора, СУБД должна просматривать каждую строку таблицы. Если таблица имеет тысячи и миллионы строк, то ее просмотр может занять минуты и даже часы.

Если для столбца DESCRIPTION имеется индекс, СУБД находит требуемые данные с гораздо меньшими усилиями. Она просматривает индекс, чтобы найти требуемое значение (изделие 2A34), а затем с помощью указателя находит требуемую строку (строки) таблицы. Поиск в индексе осуществляется достаточно быстро, так как индекс отсортирован и его строки очень короткие. Переход от индекса к строке (строкам) также происходит очень быстро, поскольку в индексе содержится информация о том, где на диске располагается эта строка.

Как видно из этого примера, индекс имеет то преимущество, что он в огромной степени ускоряет выполнение инструкций SQL с условиями отбора, имеющими ссылки на индексный столбец (столбцы). К недостаткам индекса относится то, что, во-первых, он занимает на диске дополнительное место, и, во-вторых, индекс необходимо обновлять каждый раз, когда в таблицу добавляется строка или обновляется индексный столбец таблицы. Это требует дополнительных затрат на выполнение инструкций INSERT и UPDATE, которые обращаются к данной таблице.

В общем, полезно создавать индекс лишь для тех столбцов, которые часто используются в условиях отбора. Индексы удобны также в тех случаях, когда инструкции SELECT обращаются к таблице гораздо чаще, чем инструкции INSERT и UPDATE. СУБД всегда создает индекс для первичного ключа таблицы, так как ожидает, что доступ к таблице чаще всего будет осуществляться через первичный ключ.

2.3. ЦЕЛОСТНОСТЬ ДАННЫХ

Термин целостность относится к правильности и полноте информации, содержащейся в базе данных. При изменении содержимого базы данных с помощью инструкций INSERT, DELETE или UPDATE может произойти нарушение целостности содержащихся в ней данных:

– в базу могут быть внесены неправильные данные, например, заказ, в котором указан не существующий товар;

– в результате изменения имеющихся данных им могут быть присвоены некорректные значения, например, назначение служащего в несуществующий офис;

– изменения, внесенные в базу данных, могут быть утеряны из-за системной ошибки или сбоя в электропитании;

– изменения, внесенные в базу данных, могут быть внесены лишь частично, например, заказ может быть добавлен без учета изменения количества товара, имеющегося на складе.

2.3.1. Условия целостности данных

Для сохранения непротиворечивости и правильности хранимой информации в реляционных СУБД устанавливается одно или несколько условий целостности данных. Эти условия определяют, какие значения могут быть записаны в базу данных в результате добавления или обновления данных. Как правило, в реляционной базе данных можно использовать следующие условия целостности данных:

– ***Обязательное наличие данных***. Некоторые столбцы в базе данных должны содержать значения в каждой строке; строки в таких столбцах не могут включать псевдозначения `NULL` или не содержать никакого значения. Например, в учебной базе данных для каждого заказа должен обязательно существовать клиент, сделавший этот заказ. Поэтому столбец `ID_CLN` в таблице `ZAKAZY` является обязательным. Для реализации этого условия необходимо указать СУБД, что запись значения `NULL` в такие столбцы недопустима;

– ***Условие на значение***. У каждого столбца в базе данных есть свой домен, т. е. набор значений, которые допускается хранить в данном столбце. В учебной базе данных заказы нумеруются, начиная с числа `100001`, поэтому доменом столбца `ID_ORD` являются положительные целые числа, больше `100000`. Аналогично, идентификаторы служащих в столбце `ID_SLZH` должны находиться в диапазоне от `101` до `999`. Для реализации этого условия необходимо указать СУБД, что запись значений, не входящих в заданный диапазон, недопустима;

– ***Целостность таблицы.*** Первичный ключ таблицы должен в каждой строке иметь уникальное значение, отличное от значений во всех остальных строках. Например, каждая строка таблицы `TOVARY` имеет уникальную комбинацию значений в столбцах `ID_MFR` и `ID_PRD`, которая однозначно идентифицирует товар, представляемый данной строкой. Повторяющиеся значения в этих строках недопустимы, поскольку тогда база данных не сможет отличать один товар от другого. Современные СУБД автоматически обеспечивают это условие для столбцов, объявленных первичными ключами;

– ***Ссылочная целостность.*** В реляционной базе данных каждая строка таблицы-потомка с помощью внешнего ключа связана со строкой таблицы-предка, содержащей первичный ключ, значение которого равно значению внешнего ключа. Если это условие не соблюдается, то речь идет о нарушении ссылочной целостности. Современные СУБД поддерживают ссылочную целостность автоматически.

2.3.2. Изменения, способные нарушить ссылочную целостность

Существует четыре типа изменений в базах данных, которые могут изменить ссылочную целостность отношений «предок – потомок»:

– Добавление новой строки потомка. Когда происходит добавление новой строки в таблицу-потомок SLUZHASCHIE, значение ее внешнего ключа «ID_OFC» должно быть равно одному из значений первичного ключа «ID_OFC» в таблице-предке OFFISY. Если значение внешнего ключа не равно ни одному из значений первичного ключа, то добавление такой строки разрушит базу данных, поскольку появится потомок без предка («сирота»);

– Обновление внешнего ключа в строке-потомке. Это та же проблема, что и в предыдущей ситуации, но выраженная в иной форме. Если внешний ключ «ID_OFC» обновляется инструкцией UPDATE, то его новое значение должно быть равно одному из значений первичного ключа «ID_OFC» в таблице-предке OFFISY. В противном случае обновленная строка окажется сиротой;

– Удаление строки-предка. Если из таблицы-предка OFFISY будет удалена строка, у которой есть хотя бы один потомок (в таблице SLUZHASCHIE), то строки-потомки останутся сиротами. Значения внешних ключей «ID_OFC» в этих строках не будут равны ни одному из значений первичного ключа таблицы-предка OFFISY;

– Обновление внешнего ключа в строке-предке. Если в таблице-предке OFFISY будет обновлено значение внешнего ключа для отдела, у которого есть хотя бы один потомок (в таблице SLUZHASCHIE), то строки-потомки останутся сиротами.

2.3.3. Правила ссылочной целостности

Для решения перечисленных проблем, возникающих при вставке, обновлении и удалении строк связанных таблиц, предусмотрены правила ссылочной целостности (referential integrity, RI). Правила ссылочной целостности – это логические конструкции, которые выражают бизнес-правила использования данных и представляют собой правила вставки, замены и удаления. При генерации схемы базы данных эти логические конструкции будут реализованы в вде правил декларативной ссылочной целостности, которые должны быть предписаны для каждой связи, и триггеры, обеспечивающие ссылочную целостность. Триггеры представляют собой программы, выполняемые всякий раз при выполнении команд вставки, замены или удаления (INSERT, UPDATE или DELETE).

<u>Правило RESTRICT:</u>

– запрещает удаление строки из таблицы-предка, если строка имеет потомков. Инструкция DELETE, пытающаяся удалить такую строку, отбрасывается, и выдается сообщение об ошибке;

– запрещает обновление первичного ключа в строке таблицы-предка, если у строки есть потомки. Инструкция UPDATE, пытающаяся изменить значение первичного ключа в строке-предке, отбрасывается, и выдается сообщение об ошибке;

Правило CASCADE:

– определяет, что при удалении строки-предка все строки-потомки также автоматически удаляются из таблицы-потомка;

– определяет, что при изменении значения первичного ключа в строке-предке соответствующее значение внешнего ключа в таблице-потомке также автоматически изменяется во всех строках-потомках таким образом, чтобы соответствовать новому значению первичного ключа.

Правило SET NULL:

– определяет, что при удалении строки-предка внешним ключам во всех ее строках-потомках автоматически присваивается значение NULL;

– определяет, что при обновлении значения первичного ключа в строке-предке внешним ключам во всех ее строках-потомках автоматически присваивается значение NULL.

Правило SET DEFAULT:

– определяет, что при удалении строки-предка внешним ключам во всех ее строках-потомках автоматически присваивается определенное значение, по умолчанию установленное для данного столбца;

– определяет, что при обновлении значения первичного ключа в строке-предке внешним ключам во всех ее строках-потомках автоматически присваивается определенное значение, по умолчанию установленное для данного столбца.

Правило NONE:

– определяет, что при удалении строки-предка значения внешних ключей во всех ее строках-потомках не меняются;

– определяет, что при обновлении значения первичного ключа в строке-предке значения внешних ключей во всех ее строках-потомках не меняются.

Обычно правило NONE используется в «плоских» таблицах, так как в настольных или файл-серверных системах функциональность, обеспечивающая правила ссылочной целостности, реализуется в клиентском приложении.

Рассмотрим применение правил ссылочной целостности на примере одной из связей предок/потомок учебной базы данных. Относительно таблиц SLUZHASCHIE и OFFISY, связанных отношением предок-потомок, вышеописанные проблемы добавления, обновления и удаления с использованием перечисленных правил решаются следующим образом:

1. Проблема, возникающая при добавлении новой строки потомка, решается путем проверки значений в столбцах внешнего ключа перед выполнением инструкции INSERT. Если они не равны ни одному из значений первичного ключа, то инструкция INSERT отбрасывается и выдается сообщение об ошибке. Другими словами применяется правило RESTRICT.

2. Проблема, возникающая при обновлении внешнего ключа в строке-потомке, решается аналогично: путем проверки нового значения в столбцах внешнего ключа перед выполнением инструкции UPDATE. Если оно не равно ни одному из значений первичного ключа, то инструкция UPDATE отбрасывается и выдается сообщение об ошибке. То есть в данном случае также применяется правило RESTRICT.

3. Проблема удаления строки-предка является более сложной. При этом в зависимости от ситуации можно:

– Не удалять из базы данных отдел до тех пор, пока сотрудники не будут переведены в другой отдел. Другими словами применяется правило RESTRICT;

– Автоматически удалить сотрудников, работавших в данном отделе, т. е. применить правило CASCADE;

– в столбце «ID_OFC» установить для сотрудников, работавших в данном отделе, значение NULL, показывая, что номер их отдела пока не определен. Это означает применение правила SET NULL;

– в столбце «ID_OFC» для сотрудников, работавших в данном отделе, установить значение по умолчанию некоторое значение, указывая, что сотрудники переводятся в этот отдел. Это означает применение правила SET DEFAULT.

4. Проблема обновления внешнего ключа в строке-предке решается аналогично предыдущей:

– не изменять «ID_OFC» до тех пор, пока сотрудники не будут переведены в другой отдел (правило RESTRICT);

– автоматически обновить поле «ID_OFC» для сотрудников, работавших в данном отделе (правило CASCADE);

– в столбце «ID_OFC» установить для сотрудников, работавших в данном отделе, значение NULL, показывая, что номер их отдела пока не определен (правило SET NULL);

– в столбце «ID_OFC» для сотрудников, работавших в данном отделе, установить значение по умолчанию некоторое значение, указывая, что сотрудники переводятся в этот отдел (правила SET DEFAULT).

2.4. НОРМАЛИЗАЦИЯ ДАННЫХ

После определения таблиц, полей, индексов, связей между таблицами и правил ссылочной целостности следует посмотреть на проектируемую базу данных в целом и проанализировать ее с целью устранения логических ошибок. При этом большие отношения, как правило, содержащие большую избыточность, разбиваются на более мелкие логические единицы, группирующие только данные, объединенные «по природе».

В реляционных базах данных схема содержит как структурную, так и семантическую информацию. Структурная информация связана с объявлением отношений, а семантическая выражается множеством известных функциональных зависимостей между атрибутами отношений, объявленных в схеме. Однако некоторые функциональные зависимости могут быть нежелательными из-за побочных эффектов или аномалий, которые они вызывают при модификации баз данных. В связи с этим возникает вопрос о корректности представленной схемы. Корректной считается схема, в которой отсутствуют нежелательные функциональные зависимости. В противном случае приходится прибегать к процедуре, называемой *декомпозицией* (разложением), при которой данное множество отношений заменяется другим множеством отношений (при этом их число возрастет), являющихся проекциями первых. Этот процесс зависит от ин-

туиции и опыта разработчика, однако некоторые его моменты можно формализовать.

Одной из таких формализаций является требование, согласно которому реляционная база данных должна быть нормализована. Окончательная цель нормализации сводится к получению такого проекта базы данных, в котором каждый факт появляется лишь в одном месте, т. е. исключена избыточность информации. Избыточность информации устраняется не только с целью экономии памяти, сколько для исключения возможной противоречивости хранимых данных и упрощения управления ими. Обычно различают следующие проблемы, возникающие при использовании ненормализованных таблиц:

– избыточность данных проявляется в том, что в нескольких записях таблицы базы данных повторяется одна и та же информация. Например, один человек может работать на двух и более должностях. Но если информация о личных данных сотрудника и его должности совмещены, то для сотрудника, занимающего более одной должности, его личные данные будут дублироваться;

– аномалия обновления тесно связана с избыточностью данных. Предположим, что у сотрудника, работающего на нескольких должностях, изменился адрес. Чтобы информация, содержащаяся в таблице, была корректной, необходимо будет внести изменения в несколько записей. В противном случае возникает несоответствие информации, которое и называется аномалией обновления;

– аномалия удаления возникает при удалении записей из ненормализованной таблицы. Пусть в организации некоторые должности аннулируются. При этом следует удалить соответствующие записи в рассматриваемой таблице. Однако удаление приводит к потере информации о сотруднике, занимавшем эту должность. Такая потеря и называется аномалией удаления.

2.4.1. Понятие функциональной зависимости

Для устранения перечисленных аномалий необходима декомпозиция схемы базы данных, гарантирующая отсутствие потерь и сохраняющая зависимости. Сохранение зависимостей подразумевает выполнение исходного множества функциональных зависимостей на отношениях новой схемы.

Термин функциональная зависимость означает следующее: атрибут *B* отношения *R* функционально зависит от атрибута *A* того же отношения, если в каждый момент времени каждому значению атрибута *A* соответствует не более чем одно значение атрибута *B*, связанного с *A* в отношении *R*.

Утверждение, что *B* функционально зависит от *A*, означает то же самое, что *A* однозначно определяет *B*, т. е. если в какой-то момент времени известно значение *A*, то можно получить и значение *B*.

Атрибут может функционально зависеть не от какого-то одного атрибута, а от целой группы атрибутов. Атрибут (или набор атрибутов) *B* из отношения *R* называется полностью зависимым от другого набора атрибутов *A* отношения *R*, если *B* функционально зависит от всего множества *A*, но не зависит ни от какого подмножества *A*. Например, если $A = A_1, A_2, ..., A_k$ и $A_1, A_2 \rightarrow B$, то функциональная зависимость неполная.

2.4.2. Первая нормальная форма: атомарные атрибуты

Первая нормальная форма (1НФ) требует, чтобы каждое поле таблицы было неделимым и не содержало повторяющихся групп. Неделимость поля означает, что содержащиеся в нем значения не должны делиться на более мелкие части. Например, если в поле «Подразделение» содержится название факультета и кафедры, требование неделимости не соблюдается и необходимо выделить название факультета или кафедры в отдельное поле.

Повторяющимися являются поля, содержащие одинаковые по смыслу значения. Например, если требуется получить статистику продаж четырех товаров по месяцам, можно создать поля для хранения данных о продаже по каждому товару. Но что делать, если количество товаров не четыре, а 104, или если количество товаров заранее не известно? Повторяющиеся группы следует устранить, сохранив в таблице единственное поле «Товар». В результате получим запись, содержащую информацию о статистике продаж по одному товару, но этот товар может быть любым: для четырех товаров будем иметь четыре записи, а для 104 товаров – 104 записи.

Рассмотрим пример приведения к 1НФ. Пусть необходимо автоматизировать процесс отпуска товаров со склада по накладной, приведенной 0.

Сначала сведем все имеющиеся в накладной данные в одну таблицу. Приводя ее к 1НФ, учтем, что впоследствии будет необходимо учитывать продажи по разным городам, поэтому из поля «Адрес» выделим часть данных (город) в отдельное поле «Город». Кроме того, известно, что каждый покупатель может закупить в один день различное количество товаров.

Накладная № 123

Дата	Покупатель	Адрес
01.02.2003	АО «Геракл»	г. Москва, ул. Мира 92

Отпущен Товар	Количество	Единица измерения	Цена един. измер.	Общая Стоимость
Тушенка	10 000	банки	25	25 000
Сахар	500	кг	10	5 000
Макароны	300	кг	10	3 000
			Итого	33 000

Рис. 2.12. Накладная на отпуск товаров

Однако, чтобы не создавать повторяющихся групп, фиксируем факт отпуска каждого товара в отдельной записи. В результате получим таблицу «ОТПУСК ТОВАРОВ» (см. 0).

Итак, чтобы привести таблицу к 1НФ, нужно выполнить следующие шаги:

1. Все значения полей необходимо привести к атомарному виду, выделив части сложных значений в отдельные поля.

2. Необходимо свести поля, одинаковые по смыслу в одно поле.

2.4.3. Вторая нормальная форма: отсутствие зависимостей частичного ключа

Следующий шаг в процессе нормализации состоит в удалении всех неключевых атрибутов, которые зависят только от части первичного ключа. Такие атрибуты называются частично зависимыми. Те поля, которые зависят только от части первичного ключа, должны быть выделены в отдельные таблицы. Для приведения к 2НФ в таблице «ОТПУСК ТОВАРОВ» выделим поля, потенциально входящие в первичный ключ. «Дата накладной», «Покупатель» и «Номер накладной» не могут однозначно определять запись, поскольку они будут одинаковыми для всех товаров, отпускаемых по одной накладной. Поэтому введем в первичный ключ еще и поле «Товар».

Нетрудно увидеть, что созданный нами первичный ключ избыточен: поле «Номер накладной» однозначно определяет дату и покупателя. Для данной накладной не может быть иной даты и иного покупателя. А поле «Товар» в комбинации с полем «Номер накладной», напротив, однозначно идентифицирует запись. После уточнения состава первичного ключа получим новую таблицу «ОТПУСК ТОВАРОВ» (см. 0).

Первое требование 2НФ выполнено, чего не скажешь о втором. Некоторые поля зависят только от части первичного ключа. Поля «Ед_измер», «Цена_за_ед_измер» зависят от значения поля «Товар», но не зависят от значения поля «Номер накладной». Поэтому выделяем эти поля в таблицу «ТОВАР» и определяем связь 1:M между таблицами «ОТПУСК ТОВАРОВ» и «ТОВАР», так как один товар может присутствовать во многих накладных (см. 0).

ОТПУСК ТОВАРОВ
Номер накладной
Дата
Покупатель
Город
Адрес
Товар
Един_измер
Цена_за_ед_изм
Отпущ_единиц
Общая_стоим

Рис. 2.13. Таблица «ОТПУСК ТОВАРОВ», *приведенная к* 1НФ

ОТПУСК ТОВАРОВ
Номер накладной
Товар
Дата
Покупатель
Город
Адрес
Един_измер
Цена_за_ед_изм
Отпущ_единиц
Общая_стоим

Рис. 2.14. Таблица «ОТПУСК ТОВАРОВ» *с первичным ключом*

ТОВАР
Товар
Един_измер
Цена_за_ед_изм

ОТПУСК ТОВАР
Номер накладной
Товар (FK)
Дата
Покупатель
Город
Адрес
Отпущ_единиц
Общая_стоим

Рис. 2.15. Таблицы «ТОВАР» *и* «ОТПУСК ТОВАРОВ»

Дальнейший анализ полученных таблиц показывает, что значения поля «Покупатель» не зависят от первичного ключа «Номер накладной» и «Товар», а зависит только от значения «Номер накладной». Поэтому данное поле, и зависящие от его значения поля «Город» и «Адрес» выделим в таблицу «ПОКУПАТЕЛИ» (см. 0).

Анализируя далее структуру таблицы «ОТПУСК ТОВАРОВ», обнаружим, что поле «Дата» зависит только от значения поля «Номер накладной». Поэтому выделяем поля «Дата» и «Номер накладной» в самостоятельную таблицу «НАКЛАДНЫЕ». Установим связи между этими таблицами. Один покупатель может встретиться во многих накладных. Поэтому между таблицами «ПОКУПАТЕЛИ» и «НАКЛАДНЫЕ» имеется связь 1:M по полю «Покупатель». Одной накладной может соответствовать несколько товаров. Поэтому между таблицами «НАКЛАДНЫЕ» и «ОТПУСК ТОВАРОВ» установим связь 1:M по полю «Номер накладной». В итоге получим схему базы данных, приведенную на 0.

Рис. 2.16. Таблицы «ТОВАР»*,* «ОТПУСК ТОВАРОВ» *и* «ПОКУПАТЕЛИ»

Итак, чтобы перейти от 1НФ к 2НФ, нужно выполнить следующие шаги:

1. Определить, на какие части можно разбить первичный ключ так, чтобы некоторые из неключевых полей зависели от одной из этих частей.

2. Создать новую таблицу для каждой части ключа и группы зависящих от нее полей и переместить их в эту таблицу. Часть бывшего первичного ключа при этом станет первичным ключом новой таблицы.

3. Удалить из исходной таблицы поля, перемещенные в другие таблицы, кроме тех из них, которые станут вторичными ключами.

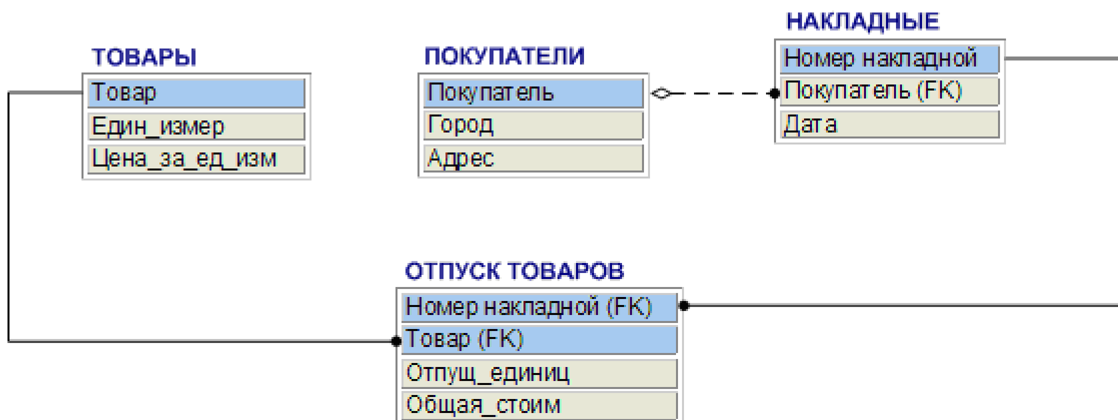


Рис. 2.17. База данных, приведенная к 2НФ

2.4.4. Третья нормальная форма: устранение транзитивных зависимостей

Третья нормальная форма (3НФ) требует, чтобы в таблице не имелось транзитивных зависимостей между неключевыми полями, т. е. чтобы значение любого поля, не входящего в первичный ключ, не зависело от другого поля, также не входящего в первичный ключ.

Пусть *A, B, C* − три атрибута или три набора атрибутов отношения *R*. Если *C* зависит от *B*, а *B* − от *A*, то *C* зависит от *A*. Если при этом обратное соответствие неоднозначно (т. е. *A* не зависит от *B*, или *B* не зависит от *C*), то говорят, что *C* транзитивно зависит от *A*.

В нашем примере можно увидеть, что в таблице «ОТПУСК ТОВАРОВ» имеется зависимость значения поля «Общая стоимость» от значения поля «Отпущено единиц». Поэтому поле «Общая стоимость» из таблицы «ОТПУСК ТОВАРОВ» удаляем. Следует отметить, что здесь рассматривается частный случай правила приведения к 3НФ, так как удаляемое поле является вычисляемым. Схема базы данных, приведенной к 3НФ, показана на 0.

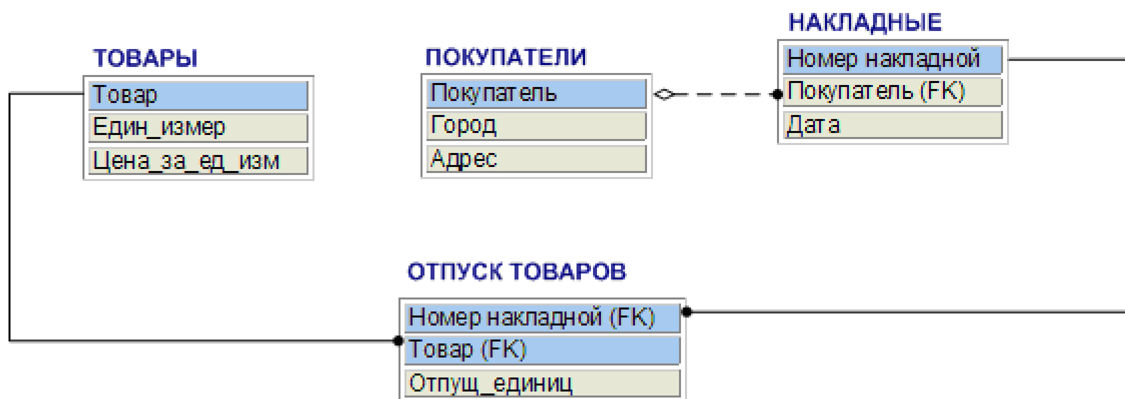


Рис. 2.18. Нормализованная база данных, приведенная к 3НФ

Переход от 2НФ к 3НФ в соответствии с общим правилом содержит следующие шаги:

1. Определить поля (или группы полей), от которых зависят другие поля.
2. Создать новую таблицу для каждого такого поля (или группы полей) и группы зависящих от него полей и переместить их в эту таблицу. Поле, от которого зависят все остальные перемещенные поля, станет при этом первичным ключом новой таблицы.
3. Удалить перемещенные поля из исходной таблицы, оставив лишь те из них, которые станут вторичными ключами.

2.5. СИСТЕМНЫЙ КАТАЛОГ

Как уже было отмечено выше, для управления данными СУБД должна иметь в своем составе базу метаданных, содержащую сведения о структуре базы данных. В реляционных СУБД для этой цели предназначены так называе-

мые системные каталоги, представляющие собой совокупности системных таблиц, используемых для внутренних целей. В этих таблицах содержатся описания таблиц, представлений, столбцов и других структурных элементов баз данных.

2.5.1. Назначение системного каталога

Таблицы системного каталога создаются автоматически при создании базы данных и сопровождаются самой СУБД в ходе ведения базы данных.

При обработке инструкций SQL СУБД постоянно обращается к данным системного каталога. Например, чтобы обработать двухтабличную инструкцию SELECT, СУБД должна:

– проверить, существуют ли указанные в запросе таблицы;

– убедиться, что пользователь имеет разрешение на доступ к ним;

– проверить, существуют ли столбцы, на которые имеются ссылки в данном запросе;

– установить к каким таблицам относятся имена столбцов;

– определить типы данных каждого столбца.

Так как информация о структуре базы данных хранится в системных таблицах, СУБД использует свои собственные методы и алгоритмы для быстрого доступа к информации, необходимой для выполнения перечисленных задач.

Системные таблицы доступны и для пользователей, но пользователи могут только извлекать информацию из системного каталога. СУБД запрещает модифицировать системные таблицы, так как это может нарушить целостность базы данных. СУБД сама вставляет, удаляет и обновляет строки системных таблиц во время модифицирования структуры базы данных.

Изменения в системных таблицах происходят в качестве побочного результата выполнения таких инструкций как CREATE, ALTER, DROP, GRANT и REVOKE.

2.5.2. Структура системного каталога

Каждая таблица системного каталога содержит информацию об отдельном структурном элементе базы данных. Входящие в состав системного каталога таблицы описывают один из следующих пяти элементов:

– *Таблицы*. В каталоге описывается каждая таблица базы данных: указывается ее имя, владелец, число содержащихся в ней столбцов, их размер и т. д.

– *Столбцы*. Каждый столбец базы данных полностью описан в каталоге. При этом приводится имя столбца и таблицы, которой он принадлежит, тип данных столбца, его размер, разрешены ли значения NULL и т. д.

– *Пользователи*. Каждый зарегистрированный пользователь базы данных в каталоге представлен своим именем, паролем в зашифрованном виде и другими данными.

– *Представления*. В каталоге описываются и представления, содержащиеся в базе данных. При этом указывается имя представления, имя владельца, запрос, являющийся определением представления и т. д.

– *Привилегии*. В системном каталоге описывается каждый набор привилегий. Это описание включает имена тех, кто предоставил привилегии, и тех, кому они

предоставлены, указываются сами привилегии, объекты, на которые они распространяются и т. д.

2.5.3. Информация о таблицах

Во всех реляционных СУБД имеется системная таблица, где отслеживается состояние всех таблиц базы данных. Например, в СУБД DB2 эта таблица называется SYSCAT.TABLES.

С помощью запросов SQL можно получить информацию о таблицах в базе данных DB2. Например, запрос

```
SELECT DEFINER, TABNAME
FROM SYSCAT.TABLES
WHERE TYPE = ‘T’
```

выводит имена всех таблиц базы данных, а также имена владельцев этих таблиц.

DEFINER – это поле таблицы SYSCAT.TABLES, где хранятся идентификаторы владельцев таблиц и представлений. В поле TYPE хранятся символы, обозначающие типы объектов базы данных: T – таблица, V – представление, A – псевдоним (это особый объект СУБД DB2).

В других СУБД информация о таблицах может храниться в таблице под другим именем. Например, в СУБД SQL Server аналогичная информация хранится в таблице SYSOBJECTS. Эта таблица хранит информацию о таблицах, представлениях, хранимых процедурах, правилах и триггерах.

2.5.4. Информация о столбцах

Во всех реляционных СУБД имеется таблица, содержащая сведения о столбцах базы данных. В этой таблице для каждого столбца базы данных отведена одна строка. Большая часть информации в этой строке относится к определению столбца. Здесь указывается его имя, тип данных, размер, возможность значения NULL и т. д.

В СУБД DB2 информация о столбцах хранится в таблице SYSCAT.COLUMNS. Как и в случае с таблицей SYSCAT.TABLES, к таблице также можно обратиться с запросом о предоставлении необходимых сведений о столбцах. Например, можно запросить все столбцы, тип данных которых DATE. Для этого необходимо написать запрос

```
SELECT TABSCHEMA, TABNAME, COLNAME
FROM SYSCAT.COLUMNS
WHERE TYPESCHEMA = ‘SYSIBM’ AND
              TYPENAME = ‘DATE’
```

Здесь TABSCHEMA – это схема к которой относится таблица, содержащая столбец, TABNAME – имя таблицы, содержащей столбец, COLNAME – имя столбца, TYPESCHEMA – схема, которой принадлежит домен столбца, TYPENAME – название типа данных или домена столбца.

2.5.5. Информация о представлениях

Определения представлений, созданных в базе данных, также хранятся в системном каталоге. В системном каталоге СУБД DB2 содержится две системные таблицы, в которых содержатся сведения о представлениях:

– таблица SYSCAT.VIEWS содержит SQL-определения всех представлений в текстовом виде. Если длина определения превышает 3600 символов, то оно хранится в нескольких строках с последовательными номерами;

– таблица SYSCAT.VIEWDEP содержит информацию о зависимости представления от других таблиц и представлений. Для каждого отношения зависимости отводится одна строка, поэтому представление с тремя исходными таблицами будет занимать в этой таблице три строки.

С помощью этих двух таблиц можно посмотреть определения представлений базы данных и быстро найти исходные таблицы любого представления.

2.5.6. Информация об отношениях между таблицами

Системный каталог содержит также информацию о первичных и вторичных ключах и создаваемых ими отношениях предок-потомок. В DB2, которая была одной из первых СУБД, поддерживающих ссылочную целостность, эта информация находится в системной таблице SYSCAT.REFERENCES.

Каждое отношение предок-потомок между двумя таблицами базы данных представлено одной строкой. В этой строке содержатся имена таблицы-предка и таблицы-потомка, имя отношения, а также правила обновления и удаления этого отношения. Чтобы получить информацию об отношениях в базе данных, следует выполнять запрос именно к этой таблице.

Приведем пример такого запроса, для вывода списка всех отношений предок-потомок между таблицами ‘USER’, включая имя отношения, имя таблицы-предка, имя таблицы-потомка и правило удаления для каждого отношения.

```
SELECT CONSTNAME, REFTABNAME, TABNAME, DELETERULE
FROM SYSCAT.REFERENCES
WHERE DEFINER = ‘USER’
```

Имена столбцов вторичных ключей и соответствующих им столбцов первичных ключей перечислены в текстовом виде в столбцах FK_COLUMNES и PK_COLUMNES таблицы SYSCAT.REFERENCES.

Информация о первичных ключах и отношениях предок-потомок, в которых они участвуют, содержится также в системных таблицах SYSCAT.TABLES и SYSCAT.COLUMNS описанных выше.

2.5.7. Информация о пользователях

В большинстве случаев системный каталог содержит таблицу, в которой перечислены все пользователи, имеющие санкционированный доступ к базе данных. СУБД может использовать эту системную таблицу для проверки имени и пароля пользователя, когда он первый раз устанавливает соединение с базой данных.

2.5.8. Информация о привилегиях

Помимо информации о структуре базы данных, системный каталог хранит информацию, которая необходима СУБД для обеспечения безопасности базы данных.

Часть 2
Язык SQL
Унификация доступа к данным

Появление новых технических и программных решений влечет необходимость изменения алгоритмов поиска и обновления данных, но при этом необходимо обеспечить и возможность использования ранее разработанных программных продуктов. Кроме того, разработка информационных систем не должна зависеть от выбора той или иной СУБД для организации данных. Все это подталкивает к необходимости стандартизации, если не самих алгоритмов доступа к данным, что едва ли возможно, а хотя бы спецификаций процедур доступа.

Если мы на каком-либо языке сумеем описать данные, а сам алгоритм будет строить компилятор или интерпретатор соответствующей СУБД, то описание будет одним и тем же для всех СУБД, понимающих этот язык. Именно эту проблему решает язык SQL – язык структурированных запросов. Этот язык принят в качестве стандарта всеми фирмами, разрабатывающими СУБД. Благодаря этому разработчики могут не заботиться о том, в среде какой СУБД будет работать его задача.

Язык SQL – это язык нечисловой обработки данных, предназначен для работы с содержанием данных. Например, если на алгоритмическом языке для обращения к массивам *A* и *B* нужно определить адрес массива и воспользоваться значением индекса *I* для выбора конкретного элемента. Аналогично для выборки из памяти из памяти значения переменной *X* достаточно знать ее имя, которое указывает на ее местоположение в памяти (см. Пример 1).

Пример 1.

```
for (i = 1; i <= 10; i++)
{ A(I) = A(I) + X*B(I); }
```

В другом примере (см. Пример 2), написанном на языке SQL, имена служащих выбираются из файла не по адресу, а по содержимому полей AGE и QUOTA.

Пример 2____________________________.

```
SELECT FAMILY
FROM SLUZHASCHIE
WHERE AGE < 35 AND QUOTA = 4000
```

Этот способ адресации отличается от способа обращения к элементам массивов A и B. Способ адресации, используемый в языке SQL, называется ассоциативным обращением или ассоциативной адресацией.

В отличие от реляционной алгебры, где были представлены только операции запросов к базе данных, SQL является полным языком. В нем присутствуют не только операции запросов, но и операторы соответствующие DDL (Data Definition Language), то есть языку описания данных. Кроме этого, язык содержит операторы, предназначенные для администрирования базы данных. В коммерческих СУБД набор операторов SQL расширен путем включения операторов определения хранимых процедур и операторов определения триггеров.

Говоря о языке SQL, нужно помнить о его главном назначении:

– во-первых, это средства описания хранимых данных, их структуры, правил доступа к ним. Эту часть будем называть *языком определения данных* (Data Definition Language – DDL);

– во-вторых, это описание запросов на поиск и изменение данных в существующей базе. Эту часть будем называть *языком манипулирования данными* (Data Manipulation Language – DML);

– в-третьих, это средства *управления доступа к данным.* Эти средства образует группа операторов управления ограничения доступа к данным. Часто их относят к языку определения данных.

В SQL существует более 40 инструкций, каждая из которых требует выполнить определенное действие, например, извлечь данные, создать таблицу и т. д. Все инструкции SQL имеют одинаковую структуру.

Каждая инструкция начинается с команды, т. е. с ключевого слова, описывающего действие, выполняемое инструкцией. Командами, например, являются CREATE, INSERT, SELECT и т. д.

После команды идет одно или несколько предложений, описывающих данные с которыми работает инструкция, либо содержится уточняющая информация о действии, выполняемой инструкцией. Каждое предложение также начинается с ключевого слова (например, WHERE, FROM, INTO и т. д.).

ГЛАВА 3. DDL – ЯЗЫК ОПРЕДЕЛЕНИЯ ДАННЫХ РЕЛЯЦИОННОЙ МОДЕЛИ

Для того чтобы работать с данными, сначала необходимо их описать, т. е. указать их структуру, формы представления, связи, методы их контроля и многое другое. Действий с описаниями весьма немного, точнее их всего три. Это ввод новых описаний, модификация существующих и удаление ненужных. каждому из этих действий существует своя команда:

CREATE – ввод новых описаний;

ALTER – модификация существующих описаний;

DROP – удаление ненужных описаний.

Каждая из этих команд имеет множество вариантов, связанных как с вариантом описаний, так и с тем фактом, что в описании нуждается множество различных информационных объектов.

3.1. СОЗДАНИЕ БАЗЫ ДАННЫХ

3.1.1. Общий формат оператора CREATE DATABASE

В стандарте SQL1 задается спецификация оператора описания схемы базы данных, но не указывается способ создания собственно базы данных, поэтому в различных СУ*БД* используются неодинаковые подходы к этому вопросу.

Для создания базы данных используется оператор SQL, имеющий следующий формат:

```
CREATE {DATABASE | SHEMA } «<имя_файла>»
[USER «имя пользователя» [PASSWORD «пароль»]]
        [PAGE_SIZE [=] целое]
        [LENGTH [=] целое [PAGE[S]]]
        [DEFAULT CHARACTER SET набор_символов]
```

Здесь:

– «<имя_файла>» – указывает спецификацию файла, в котором будет храниться создаваемая база данных;

– USER «имя пользователя» – имя пользователя, которое вместе с паролем будет проверяться при соединении пользователя с сервером;

– PASSWORD «пароль» – пароль, который вместе с именем пользователя будет проверяться при соединении пользователя с сервером;

– PAGE_SIZE [=] целое – размер страницы базы данных в байтах. Допустимые размеры: 1024 (по умолчанию), 2048, 4096 или 8192;

– LENGTH [=] целое [PAGE[S]] – длина файла в страницах. По умолчанию 75 страниц. Минимум 50 страниц. Максимум ограничен имеющимся дисковым пространством;

– DEFAULT CHARACTER SET набор_символов – определяет набор символов, применимый в базе данных. Если не указан, по умолчанию берется NONE;

Почти все параметры, кроме имени являются необязательными. Пример оператора:

```
CREATE DATABASE «D:\BD\Library»
```

3.1.2. Определение пароля

Пароль при создании базы данных указывается для того, чтобы сервер смог идентифицировать владельца БД по паре значений «имя пользователя = пароль»:

```
CREATE DATABASE «D:\BD\Library»
USER «xxx» PASSWORD «xxx»
```

Это имя и пароль принадлежат пользователю, создающему базу данных, и служат для его идентификации: после создания базы данных ни один другой пользователь, кроме системного администратора, не имеет прав доступа к базе. Впоследствии системный администратор может предоставить другим пользователям те или иные права доступа к базе данных.

3.1.3. Указание размера страницы БД

Размер страницы указывается в байтах и может быть 1024 (по умолчанию), 2048, 4096 или 8192 байт, например:

```
CREATE DATABASE «D:\BD\Library»
PAGE_SIZE 4096;
```

Увеличение размера страницы может привести к ускорению работы с базой данных, поскольку:

– уменьшается глубина индексов (число шагов, за которое при помощи индекса будут найдены требуемые записи);

– уменьшается количество операций чтения при считывании длинных записей, поскольку за одну операцию чтения всегда считывается одна страница, запись, расположенная на одной странице, будет считана за один раз; при малом объеме страницы запись располагается на нескольких страницах и соответственно считывается за несколько операций чтения.

Увеличение размера страницы не оправдано в том случае, если запросы на чтение к базе данных возвращают небольшое количество записей, не занимаю-

щих всю страницу. Поскольку за операцию чтения считывается одна страница базы данных, будет считываться много лишних записей.

3.1.4. Указание национальной кодировки символов

Для символьных столбцов в составе таблиц базы данных используются различные национальные кодировки. При создании баз данных национальный набор символов устанавливается предложением

```
DEFAULT CHARACTER SET набор_символов
```

Например:

```
CREATE DATABASE «D:\BD\Library»
DEFAULT CHARACTER SET WIN1251;
```

В дальнейшем всем создаваемым символьным столбцам таблиц баз данных (типы CHAR, VARCHAR) ставится в соответствие указанный набор символов, например, объявление в одной из таблиц баз данных Library столбца

```
FAMILIA VARCHAR(25);
```

интерпретируется как объявление

```
FAMILIA VARCHAR(25) CHARACTER SET WIN1251;
```

Изменить кодировку, принятую для базы данных по умолчанию, можно при определении конкретных доменов и столбцов.

3.1.5. Типы данных

Типы данных различных СУБД различаются. В таблице 3.1 показаны основные типы данных, включенные в стандартную дистрибуцию СУБД PostgreSQL, обладающей обширным набором собственных типов данных, доступных пользователям.

Таблица 3.1

Типы данных в СУБД PostgreSQL

Имена типов данных	Псевдонимы	Описание
bigint	int8	восьмибайтное целое число со знаком
bigserial	serial8	автоинкрементное восьмибайтное целое число
bit		строка битов фиксированной длины
bit varying(n)	varbit(n)	строка битов переменной длины
boolean	bool	логическая булева переменная (true/false)
box		прямоугольное поле в двухмерной плоскости
bytea		двоичные данные
character(n)	char(n)	строка символов фиксированной длины

Окончание табл. 3.1

Имена типов данных	Псевдонимы	Описание
character varying(n)	varchar(n)	строка символов переменной длины
cidr		сетевой IP-адрес
circle		круг в двухмерной плоскости
date		календарная дата (год, месяц, день)
double precision	float8	число с плавающей запятой удвоенной точности
inet		адрес IP-хоста
integer	int, int4	четырехбайтное целое число со знаком
interval(p)		полезное время общего использования
line		прямая в двухмерной плоскости
lseg		отрезок в двухмерной плоскости
macaddr		адрес MAC
money		денежная единица США
numeric [(p, s)]	decimal[(p, s)]	число с выбираемой точностью
oid		объектный идентификатор
path		открытый и закрытый геометрический путь в двухмерной плоскости
point		геометрическая точка в двухмерной плоскости
polygon		закрытый геометрический путь в двухмерной плоскости
real	float4	число с плавающей запятой обычной точности
smallint	int2	двухбайтное целое число со знаком
serial	serial4	автоинкрементное четырехбайтное целое число
text		строка символов переменной длины
time [(p)] [without time zone]	time	время дня
time [(p)] with time zone	timetz	время дня, включая временной пояс
timestamp [(p)] without time zone	timestamp	дата и время
timestamp [(p)][with time zone]	timestamptz	дата и время, включая временной пояс

Пользователи могут добавлять в него новые типы с помощью команды CREATE TYPE.

Большинство альтернативных имен, перечисленных в столбце «Псевдонимы», исторически используется в других СУБД.

Каждый тип данных имеет внешнее представление, определяемое его входными и выходными функциями. Многие встроенные типы имеют внешние форматы, однако некоторые существуют только в СУБД PostgreSQL (открытые и закрытые пути), а другие имеют несколько возможных форматов (типы даты и времени). Большинство входных и выходных функций, передающихся в базовые типы (т. е. целые числа и числа с плавающей точкой), производят проверку

ошибок, но некоторые функции необратимы, то есть результат выходной функции при сравнении с входной теряет точность.

3.2.СОЗДАНИЕ ДОМЕНОВ

Если в таблице базы данных или в нескольких таблицах присутствуют столбцы, обладающие одними и теми же характеристиками, можно предварительно описать тип такого столбца и его поведение с помощью домена, а затем поставить в соответствие каждому из одинаковых столбцов имя домена.

3.2.1. Общий формат оператора CREATE DOMAIN

Домен определяется оператором CREATE DOMAIN, имеющий следующий формат

```
CREATE DOMAIN домен [AS] <тип данных>
[DEFAULT {литерал | NULL | USER}]
[NOT NULL]
[COLLATE collation];
[CHECK (<огранич_домена>)]
```

Предложение DEFAULT определяет выражение, которое по умолчанию заносится в колонку, ассоциированную с доменом, при создании записи таблицы. Это значение будет присутствовать в соответствующем столбце записи до тех пор, пока пользователь не изменит его каким-либо образом. Значения по умолчанию могут быть выражены как литерал-значение (числовое, строковое или дата), NULL − специфицирует пустое значение или USER − имя текущего пользователя.

Предложение NOT NULL указывает, что столбцы, ассоциированные с доменом, обязательно должны содержать какое-либо значение, отличное от пустого.

Предложение COLLATE задает порядок сортировки символов.

3.2.2. Ограничения на значения столбцов, ассоциированных с доменом

Предложение CHECK определяет требования к значениям каждого столбца, ассоциированного с доменом. Столбцу не могут быть присвоены значения, не удовлетворяющие ограничениям, наложенным в предложении CHECK. Формат ограничения, накладываемого на значения полей, ассоциированных с доменом:

```
<огранич_домена> = {
  VALUE <оператор> <значение> |
  VALUE [NOT] BETWEEN < значение1> AND <значение2> |
  VALUE [NOT] LIKE < значение> [ESCAPE <значение>] |
  VALUE [NOT] IN < значение1> [, <значение2> ...] |
  VALUE IS [NOT] NULL |
  VALUE [NOT] CONTAINING <значение> |
  VALUE [NOT] STARTING [WITH] <значение> |
  NOT (<ограничение домена>) |
  (<ограничение домена>) OR (<ограничение домена>) |
  (<ограничение домена>) AND (<ограничение домена>)}
```

– <оператор> = { = | < | > | <= | >= | !< | !> | <> | != };

– ключевое слово VALUE означает все правильные значения, которые могут быть присвоены столбцу, ассоциированному с доменом.

– <оператор> <значение> – значение домена находится с параметром <значение> во взаимоотношениях, определяемых параметром оператор. Например,

```
CHECK (VALUE >= 100);
```

– BETWEEN <значение1> AND <значение2> – значение домена должно находиться в промежутке между значение1 и значение2, включая их.

– LIKE < значение> [ESCAPE <значение>] – значение домена должно «походить» на параметр значение. При этом символ «%» употребляется для указания любого значения любой длины и символ подчеркивания «_» – для указания любого единичного символа. Например, LIKE «%USD» – вводимое значение должно оканчиваться символами «USD» независимо от того, какие символы и сколько расположены перед ними; LIKE «__94» – вводимое значение может содержать 4 символа, из которых первые два – любые и последние два – «94». ESCAPE <значение> используется, если в операторе LIKE служебные символы «%» или «_» должны использоваться в шаблоне подобия. В этом случае выбирается некоторый символ, например, «!», после которого служебные символы теряют свой статус и входят в поисковую строку как обычные символы. Символ «!» указывается после слова ESCAPE. Например, CHECK LIKE «%!%» ESCAPE «!»); Согласно приведенному ограничению значения домена должны заканчиваться символом «%».

– IN <значение1> [, <значение2> ...] – значение домена должно совпадать с одним из приведенных в списке параметров значениеN, например: CHECK (VALUE IN («Муж», «Жен»));

– CONTAINING <значение> – значение домена должно содержать вхождение параметра значение, не важно, в каком месте. Например, в наименовании отдела вхождение «041» может встретиться где угодно «00304107», «Отдел - 041002»:

```
CHECK (VALUE CONTAINING «041»);
```

– STARTING [WITH] <значение> – значение домена должно начинаться параметром значение. Например, название отдела должно начинаться с «041»:

```
CHECK (VALUE STARTING WITH «041»);
```

Может быть задана комбинация условий, которым должно соответствовать значение домена. В этом случае отдельные условия соединяются операторами AND или OR. Например:

```
CHECK (VALUE STARTING WITH «041» AND VALUE CONTAINING «-12»)
```

Для большинства условий можно указать слово NOT, которое изменяет условие с точностью до наоборот: CHECK (VALUE NOT BETWEEN 1 AND 10);

3.2.3. Изменение определения домена

Оператор ALTER DOMAIN, имеющий следующий формат

```
ALTER DOMAIN имя
 { [SET DEFAULT {литерал | NULL | USER}] |
   [DROP DEFAULT] |
   [ADD [CONSTRAINT] CHECK (<ограничен_домена>)]
   [DROP CONSTRAINT]  };
```

позволяет изменит параметры домена, определенного ранее оператором CREATE DOMAIN. Однако нельзя изменить тип данных и определение NOT NULL. Следует помнить, что все сделанные изменения будут учтены для всех столбцов, определенных с использованием данного домена (в том случае, если параметры домена не были переопределены при создании столбцов таблицы или впоследствии).

– SET DEFAULT устанавливает значения по умолчанию подобно тому, как это делается в операторе CREATE DOMAIN.

– DROP DEFAULT отменяет текущие значения по умолчанию.

– [ADD [CONSTRAINT] CHECK (<ограничен_домена>)] добавляет условия, которым должны соответствовать значения столбца, ассоциированного с доменом. При этом возможно определение условий, рассмотренных выше для предложения CHECK оператора CREATE DOMAIN.

– DROP CONSTRAINT – удаляет условия, определенные для домена в предложении CHECK оператора CREATE DOMAIN или предыдущих операторов ALTER DOMAIN.

Например, пусть определен домен ID_TYPE:

```
CREATE DOMAIN ID_TYPE AS INTEGER
CHECK(VALUE >= 100);
```

и в дальнейшем использован при создании таблицы AAA:

```
CREATE TABLE AAA
  ( ID ID_TYPE NOT NULL,
    FIO VARCHAR(20),
    Primary KEY(ID)   );
```

Изменить условие CHECK так, чтобы значение было больше или равно 100 и меньше или равно 500, можно за два шага:

– сначала нужно удалить старое условие

```
ALTER DOMAIN ID_TYPE
DROP CONSTRAINT;
```

– затем добавить новое (которое есть модифицированное старое)

```
ALTER DOMAIN ID_TYPE
CHECK(VALUE >= 100 AND VALUE <= 500);
```

Заметим, что изменять определение таблицы AAA нет необходимости, и отныне столбец ID этой таблицы можно занести значения, большие или равные 100 и меньшие или равные 500.

3.3. СОЗДАНИЕ ТАБЛИЦ

3.3.1. Инструкция CREATE TABLE

Инструкция `CREATE TABLE` определяет новую таблицу и подготавливает ее к приему данных. Перед созданием таблиц базы данных необходимо продумать определения всех столбцов таблицы и характеристик каждого столбца (таких как тип, длина, обязательность для ввода, ограничения, накладываемые на значения и т. д.), индексов, ограничений целостности по отношению к другим таблицам. Если при определении столбцов используются домены, то эти домены должны быть предварительно созданы оператором `CREATE DOMAIN`.

Создание таблицы базы данных осуществляется оператором

```
CREATE TABLE ИмяТаблицы
  (столбец тип_данных ¦ домен [DEFAULT значение NOT NULL,]
   PRIMARY KEY (поле, ... ),
[CONSTRAINT <имя отношения>]
FOREIGN KEY (<список столбцов внешнего ключа>)
REFERENCES <имя таблицы-предка> [<список столбцов таблицы-предка>]
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
UNIQUE (поле, ... ),
CHECK (условие_отбора) );
```

Определение столбцов

Определения столбцов представляют собой заключенный в скобки список, элементы которого отделены друг от друга запятыми:

– `столбец` – имя столбца, которое используется для ссылки на столбец в инструкциях `SQL`. Каждый столбец в таблице должен иметь уникальное имя, но в разных таблицах имена столбцов могут совпадать;

– `тип_данных` – показывает, данные какого вида хранятся в столбце;

– `домен` – имя домена, т. е. ранее описанного типа столбца;

– `DEFAULT` – определяет значение, которое по умолчанию заносится в столбец, ассоциированный с доменом, при создании записи таблицы;

– `NOT NULL` – указывает на то, что столбец обязательно должен содержать значение.

Ниже приведен пример инструкции `CREATE TABLE` для таблицы `OFFISY` из учебной базы данных.

```
CREATE TABLE OFFISY
       ( ID_OFC INTEGER NOT NULL,
         CITY VARCHAR(15) NOT NULL,
         REGION VARCHAR(10) NOT NULL,
         MNGR INTEGER,
         TARGET MONEY,
         SALES MONEY NOT NULL);
```

Предложения PRIMARY KEY и FOREIGN KEY

Кроме определений столбцов таблицы, в инструкции CREATE TABLE указывается информация о первичном ключе таблицы и ее связях с другими таблицами базы данных. Эта информация содержится в предложениях PRIMARY KEY и FOREIGN KEY.

Предложением PRIMARY KEY задается столбец или столбцы, которые образуют первичный ключ и служат в качестве уникального идентификатора строк таблицы. СУБД автоматически следит за тем, чтобы первичный ключ был уникален. Кроме того в определениях столбцов первичного ключа должно быть указано, что они не могут содержать значения NULL (имеют ограничения NOT NULL).

В предложении FOREIGN KEY задается внешний ключ таблицы и определяется связь, которую он создает для нее с другой таблицей (таблицей-предком). Итак, внешний ключ строится в дочерней таблице для соединения родительской и дочерних таблиц базы данных. В предложении FOREIGN KEY содержатся (или могут содержаться) следующие определения:

– CONSTRAINT определяет необязательное имя отношения; оно не используется в инструкциях SQL, но может появляться в сообщениях об ошибках и потребуется в дальнейшем, если будет необходимо удалить внешний ключ;

– список столбцов внешнего ключа – определяет столбцы дочерней таблицы, по которым строится внешний ключ; столбец или столбцы создаваемой таблицы, которые образуют внешний ключ;

– имя таблицы-предка – определяет таблицу, в которой описан первичный ключ. На этот ключ должен ссылаться внешний ключ дочерней таблицы для обеспечения ссылочной целостности; таблица, связь с которой создает внешний ключ; это таблица-предок, а определяемая таблица в данном отношении является потомком;

– список столбцов таблицы-предка – необязателен при ссылке на первичный ключ родительской таблицы;

– ON DELETE или ON UPDATE – определяют способы изменения подчиненных записей дочерней таблицы при удалении или изменении поля связи в записи родительской таблицы. Перечислим эти способы:

- ✧ NO ACTION – запрет удаления/изменения родительской записи при наличии подчиненных записей в дочерней таблице;
- ✧ CASCADE – для оператора ON DELETE: при удалении записи родительской таблицы происходит удаление подчиненных записей в дочерней таблице; для оператора ON UPDATE: при изменении поля связи в записи родительской таблицы происходит изменение на то же значение поля внешнего ключа у всех подчиненных записей в дочерней таблице;
- ✧ SET DEFAULT – в поле внешнего ключа у записей дочерней таблицы заносится значение этого поля по умолчанию, указанное при определении поля (параметр DEFAULT); если это значение отсутствует в первичном ключе, инициируется исключение;
- ✧ SET NULL – в поле внешнего ключа заносится значение NULL.

Ниже приводится расширенная инструкция CREATE TABLE для таблицы ZAKAZY, в которую входит определение первичного ключа и трех внешних ключей, имеющихся в таблице:

```
CREATE TABLE ZAKAZY
 (ID_ORDER INTEGER NOT NULL,
  DATE_ORDER DATE NOT NULL,
  ID_CLN INTEGER NOT NULL,
  ID_SLZH INTEGER,
  ID_MFR CHAR(3) NOT NULL,
  ID_PRD CHAR(5) NOT NULL,
  COUNT INTEGER NOT NULL,
  PRICE_ALL MONEY NOT NULL,
  PRIMARY KEY (ID_ORDER),
  CONSTRAINT PLACEDBY
  FOREIGN KEY (ID_CLN) REFERENCES CLIENTY ON DELETE CASCADE,
  CONSTRAINT TAKENBY
  FOREIGN KEY (ID_SLZH) REFERENCES SLUZHASCHIE ON DELETE CASCADE,
  CONSTRAINT PLACEDBY
  FOREIGN KEY (ID_MFR, ID_PRD) REFERENCES TOVARY ON DELETE CASCADE
 )
```

На Рис. 3.1. изображены три созданные этой инструкцией связи с присвоенными им именами. В общем случае связи следует давать имя, поскольку оно помогает лучше понять, какая именно связь создана внешним ключом. Например, каждый заказ делается клиентом, идентификатор которого находится в столбце CUST таблицы ZAKAZY. Связь с этим столбцом получила имя PLACEDBY.

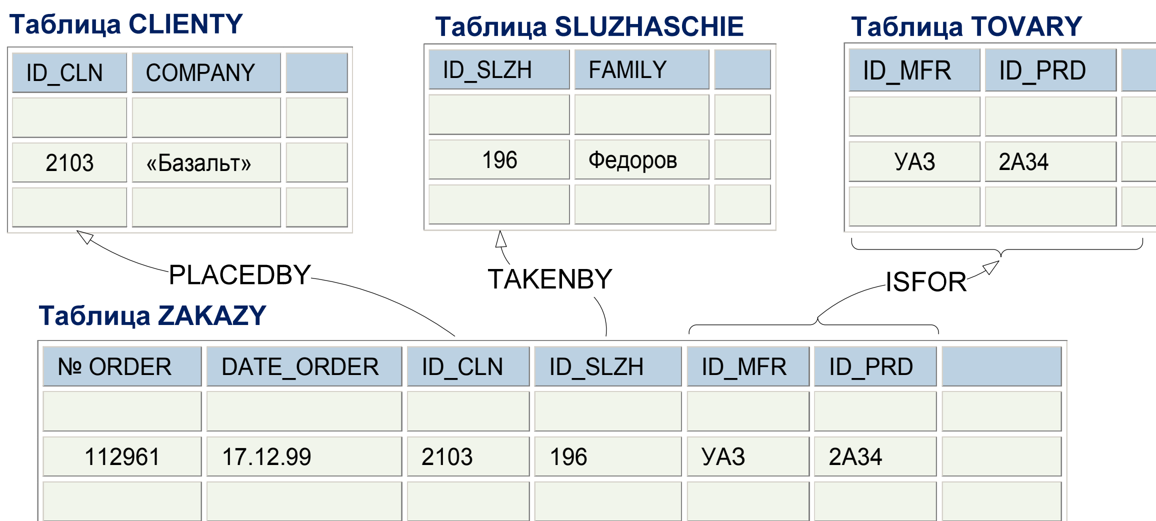


Рис. 3.1. Имена связей в инструкции CREATE TABLE

Когда СУБД выполняет инструкцию CREATE TABLE, она сравнивает определение каждого внешнего ключа с определениями связанных таблиц. СУБД проверяет, соответствуют ли друг другу внешний и первичный ключи в связанных

таблицах как по числу столбцов, так и по типу данных. Для того чтобы такая проверка была возможна, связанная таблица уже должна быть определена.

Обратите внимание на то, что в предложении FOREIGN KEY задаются также правила удаления и обновления, которым будет подчиняться создаваемое отношение таблиц.

Предложение UNIQUE

Для того чтобы сервер автоматически проверял и поддерживал уникальность для некоторого поля, надо для данного поля ввести ограничитель на уникальность. Для этого используется предложение UNIQUE инструкции CREATE TABLE.

Ниже приведена модифицированная инструкция CREATE TABLE для таблицы OFFISY с включением в нее условием уникальности для столбца CITY:

```
CREATE TABLE OFFISY
     (
       ID_OFC INTEGER NOT NULL,
       CITY VARCHAR(15) NOT NULL,
       REGION VARCHAR(10) NOT NULL,
       MNGR INTEGER,
       TARGET MONEY,
       SALES MONEY NOT NULL,
       PRIMARY KEY (OFFICE),
       CONSTRAINT HASMGR
       FOREIGN KEY (MNGR) REFERENCES SLUZHASCHIE ON DELETE SET NULL,
       UNIQUE (CITY) );
```

Если первичный или внешний ключ включают в себя только один столбец, либо если условие уникальности или условие на значения касаются одного столбца, то разрешается использовать «сокращенную» форму ограничения, при которой оно просто добавляется в конец определения столбца, как это показано в нижеследующем примере:

```
CREATE TABLE OFFISY
    (ID_OFC INTEGER NOT NULL,
     CITY VARCHAR(15) NOT NULL UNIQUE,
     REGION VARCHAR(10) NOT NULL,
     MNGR INTEGER,
     TARGET MONEY,
     SALES MONEY NOT NULL,
     PRIMARY KEY (OFFICE),
     CONSTRAINT HASMGR
     FOREIGN KEY (MGR) REFERENCES SLUZHASCHIE ON DELETE SET NULL);
```

Предложение CHECK

Когда создается таблица, то для каждого поля задается тип его значения. Это может быть INTEGER, CHAR и т. п. Тип определяет допустимое множество значений для данного поля. Но в некоторых случаях это множество значений много шире реально используемого множества. В SQL есть средства для более тонкого описания множества допустимых значений поля (в теории это множе-

ство называется доменом). Задать домен для того или иного поля можно с помощью ограничений на значение. Этот ограничитель указывается при создании таблицы. После типа поля или значения по умолчанию надо указать ключевое слово CHECK и логическое выражение в скобках

```
CREATE TABLE OFFISY
     (
       ID_OFC INTEGER NOT NULL,
       CITY VARCHAR(15) NOT NULL,
       REGION VARCHAR(10) NOT NULL,
       MNGR INTEGER,
       TARGET MONEY,
       SALES MONEY NOT NULL,
       PRIMARY KEY (OFFICE),
       CONSTRAINT HASMGR
       FOREIGN KEY (MGR) REFERENCES SLUZHASCHIE ON DELETE SET NULL,
       CHECK (TARGET >= 0.00) );
```

В ограничителе на значение можно использовать сколь угодно сложное логическое выражение. Выражение должно иметь тип, совместимый с типом столбца. Не допускается использование вложенных подзапросов, агрегатных функций и вызовов хранимой процедуры.

3.3.2. Инструкция ALTER TABLE

В процессе работы с таблицей у пользователя может возникнуть необходимость в изменении таблицы. В языке SQL имеются средства изменения схемы таблиц. Например, в учебной базе данных может потребоваться:

– добавить в каждую строку таблицы CUSTOMERS имя и номер телефона служащего компании клиента, через которого поддерживается контакт;

– добавить в таблицу PRODUCTS столбец с указанием минимального количества товара, чтобы иметь возможность предупреждения о том, что запас какого-либо товара стал меньше допустимого предела;

– сделать столбец REGION в таблице OFFISY внешним ключом для вновь созданной таблицы REGIONS, первичным ключом которой является название региона;

– удалить определение внешнего ключа для столбца CUST таблицы ZAKAZY, связывающего ее с таблицей CLIENTY, и заменить его определениями двух внешних ключей, связывающих столбец CUST с двумя вновь созданными таблицами CUST_INFO и ACCOUNT_INFO.

Для модифицирования таблиц используется оператор ALTER TABLE, который позволяет выполнить следующие операции изменения таблицы:

– добавить новый столбец в уже существующую и заполненную таблицу;
– удалить столбец из существующей таблицы;
– изменить значение по умолчанию для какого-либо столбца;
– добавить или удалить первичный ключ таблицы;
– добавить или удалить внешний ключ таблицы;
– добавить или удалить условие уникальности;

– добавить или удалить условие проверки для любого столбца или для таблицы в целом.

Однако оператором ALTER TABLE можно провести только одно из перечисленных изменений, например, за один раз можно добавить 1 столбец.

Добавление столбца

Чаще всего инструкция ALTER TABLE применяется для добавления столбца в существующую таблицу. Предложение с определением столбца в инструкции ALTER TABLE имеет точно такой же вид, что и в инструкции CREATE TABLE, и выполняет ту же самую функцию. Новое определение добавляется в конец определений столбцов таблицы, и в последующих запросах новый столбец будет крайним справа. СУБД обычно предполагает, что новый столбец во всех существующих строках содержит значения NULL. Поэтому нельзя объявлять новый столбец как NOT NULL.

Но если такое объявление все же необходимо, то необходимо определить этот столбец как NOT NULL WITH DEFAULT. При этом СУБД считает, что этот столбец содержит значение по умолчанию, и не будет автоматически добавлять значение NULL.

Пример. *Добавить контактный телефон и имя служащего компании клиента в таблицу* CLIENTY .

```
ALTER TABLE SLUZHASCHIE
ADD CONTACT_NAME VARCHAR(30)

ALTER TABLE SLUZHASCHIE
ADD CONTACT_PHONE CHAR(10)
```

Пример. *Добавить в таблицу* TOVARY *столбец с данными о минимальном допустимом количестве товара на складе.*

```
ALTER TABLE TOVARY
ADD MIN_QTY INTEGER NOT NULL WITH DEFAULT 0
```

В первом примере новые столбцы будут иметь значения NULL для существующих клиентов. Во втором примере столбец MIN_QTY для существующих товаров будет содержать нули, что вполне уместно.

Удаление столбца

С помощью инструкции ALTER TABLE можно удалить из существующей таблицы один или несколько столбцов, если в них больше нет необходимости. Ниже приведен пример удаления столбца QUOTA из таблицы SLUZHASCHIE:

```
ALTER TABLE SLUZHASCHIE
DROP QUOTA
```

Следует отметить, что операция удаления столбца вызывает проблемы с целостностью данных, описанные в разделе 2.3. Например, при удалении столбца, являющегося первичным ключом в каком либо отношении, связанные с ним внешние ключи становятся недействительными.

Эти проблемы в стандарте SQL2 решены так же, как и в случае инструкций DELETE и UPDATE, с помощью правил удаления RESTRICT и CASCADE.

В случае применения правила RESTRICT инструкция ALTER TABLE завершится выдачей сообщения об ошибке и столбец не будет удален. Во втором случае внешние ключи, связанные с удаляемым столбцом будут удалены. Однако правило CASCADE может вызвать целую «лавину» изменений, поэтому применять его следует с осторожностью. Лучше указывать правило RESTRICT, а связанные внешние ключи обрабатывать с помощью дополнительных инструкций типа ALTER.

Изменение первичных и вторичных ключей

Инструкция ALTER TABLE чаще всего применяется для изменения или добавления определений первичных и вторичных ключей таблицы. Предложения, добавляющие определения первичного и внешнего ключей, являются точно такими же, как в инструкции CREATE TABLE, и выполняет те же функции.

Пример. *Сделать столбец* REGION *таблицы* OFFISY *внешним ключом для вновь созданной таблицы* REGIONS*, первичным ключом которой является название региона.*

```
ALTER TABLE OFFISY
ADD CONSTRAINT IN REGION
FOREIGN KEY (REGION) REFERENCES REGIONS
```

Предложения, удаляющие первичный или внешний ключи, являются довольно простыми. Однако следует отметить, что удалить внешний ключ можно только тогда, когда создаваемая им связь имеет имя.

Пример. *Изменить первичный ключ таблицы* OFFISY.

```
ALTER TABLE SLUZHASCHIE
DROP CONSTRAINT WORKSIN
FOREIGN KEY (REP_OFFICE) REFERENCES OFFISY
```

```
ALTER TABLE OFFISY
DROP PRIMARY KEY (OFFICE)
```

Если имя присвоено не было, то задать эту связь в инструкции ALTER TABLE невозможно. В этом случае для удаления внешнего ключа необходимо удалить таблицу и воссоздать ее в новом формате.

3.4. СОЗДАНИЕ ПРЕДСТАВЛЕНИЙ (VIEW)

Представление (view) является логической (виртуальной) таблицей, записи в которую отобраны с помощью оператора SELECT. Представлением называется запрос на выборку, которому присвоили имя, а затем сохранили в базе данных. Представление позволяет пользователю увидеть результаты сохраненного запроса, а SQL обеспечивает доступ к этим результатам таким образом, как если бы они были простой таблицей.

Представления используются по нескольким причинам:

– они позволяют сделать так, что разные пользователи базы данных будут видеть ее по-разному;

– с их помощью можно ограничить доступ к данным, разрешая пользователям видеть только некоторые из строк и столбцов таблицы;

– они упрощают доступ к базе данных, показывая каждому пользователю структуру хранимых данных в наиболее подходящем виде.

Представление является «виртуальной таблицей», содержимое которой является запросом. Для пользователя базы данных представление выглядит обычной таблицей, состоящей из строк и столбцов. Однако, в отличие от таблицы, представление как совокупность значений в базе данных реально не существует. Строки и столбцы данных, которые пользователь видит с помощью представления, являются результатом запроса, лежащего в его основе. При создании представление получает имя, и его определение сохраняется в базе данных.

Преимущество представления заключается в том, что можно один раз отобрать записи и использовать их в дальнейшем без повторного выполнения оператора `SELECT`. Это выгодно при частом использовании одинаковых запросов, особенно тех, в которых реализованы сложные условия отбора.

Представления могут использоваться для описания внешних моделей в реляционной модели. В действительности представление содержит не данные, а лишь `SQL`-запрос типа `SELECT`, указывающий, какие именно колонки и из каких таблиц нужно взять при обращении к этому представлению. Задание представлений входит в описание схемы базы данных в реляционных СУБД. Представления позволяют скрыть ненужные детали для разных пользователей, модифицировать реальные структуры данных в удобном для приложений виде, разграничить доступ к данным и тем самым повысить защиту данных от несанкционированного доступа.

В отличие от реальной таблицы, представление в том виде, как оно сконструировано, не существует в базе данных, это действительно только виртуальное отношение, хотя все данные, которые представлены в нем, действительно существуют в базе данных, но в разных отношениях. Они скомпонованы для пользователя в удобном виде из реальных таблиц с помощью некоторого запроса.

Однако пользователь может этого не знать, он может обращаться с этим представлением как со стандартной таблицей. Представление при создании получает некоторое уникальное имя, его описание храниться в описании схемы базы данных, и СУБД в любой момент времени при обращении к этому представлению выполняет запрос, соответствующий его описанию, поэтому пользователь, работая с представлением, в каждый момент времени видит действительно реальные, актуальные на настоящий момент данные. Оно формирует их как бы на лету, в момент обращения.

При необходимости в представлении может задаваться новое имя для каждого столбца виртуальной таблицы. При этом надо помнить, что если указывается список столбцов, он должен содержать ровно столько столбцов, сколько содержит их `SQL`-запрос.

Для создания представления мы можем использовать `SQL`-предложение `CREATE VIEW`, для его модификации – предложение `ALTER VIEW`, а для удаления – предложение `DROP VIEW`. Предложение `CREATE VIEW` используется для создания представлений, позволяющих извлекать данные, удовлетворяющие некоторым

требованиям. Представление создается в текущей базе данных и хранится как отдельный объект. Наилучший способ создания представления – создать запрос `SELECT` и, проверив его, добавить недостающую часть `CREATE VIEW`.

Если список имен столбцов в представлении не задан, то каждый столбец представления получает имя соответствующего столбца запроса. Рассмотрим типичные виды представления и их назначение.

3.4.1. Общий формат оператора CREATE VIEW

Создание просмотра базы данных осуществляется оператором

```
CREATE VIEW Имя_Представления
AS <Оператор SELECT>;
```

– `Имя_Представления` – имя просмотра после его создания можно использовать как имя физической таблицы.

– `Список столбцов` – определяет состав столбцов просмотра. Если список не задан, то в просмотр отбираются все столбцы таблиц, указанных в операторе `SELECT`.

– `WITH CHECK OPTION` – для редактируемого просмотра запрещает добавление записей, не удовлетворяющих условиям отбора, заданным в операторе `SELECT`.

На Рис. 3.2. изображено представление, определенное в соответствии с запросом

```
CREATE VIEW SLUZHASCHIE_OFFISY AS
      SELECT C.FAMILY, C.NAME, O.CITY, O.REGION, C.QUOTA, C.SALES
      FROM SLUZHASCHIE C, OFFISY O
      WHERE C.ID_OFC = O.ID_OFC
```

3.4.2. Горизонтальное представление

Этот вид представления широко применяется для уменьшения объема реальных таблиц в обработке и ограничения доступа пользователей к закрытой для них информации. Так, например, правилом хорошего тона считается, что руководитель подразделения фирмы может видеть оклады и результаты работы только своих сотрудников, в этом случае для него создается горизонтальное представление, в которое загружены строки общей таблицы сотрудников, работающих в его подразделении. Пример горизонтального представления:

```
CREATE VIEW SAL_DEPT AS
SELECT *
FROM  EMPLOYEE
WHERE DEPARTMENT = «отдел продаж»
```

3.4.3. Вертикальное представление

Этот вид представления практически соответствует выполнению операции проектирования некоторого отношения на ряд столбцов. Он используется в основном для скрытия информации, которая не должна быть доступна в конкретной внешней модели. Например, для работника табельной службы, который учитывает присутствие сотрудников на работе, информация об окладе и над-

бавке должна быть закрыта. Для него можно создать следующее вертикальное представление:

```
CREATE VIEW TABLE AS
SELECT T_NUM, NAME, POSITION, DEPRT
FROM   EMPLOYEE
```

Таблица SLUZHASCHIE

ID_SLZH	FAMILY	NAME	AGE	…	QUOTA	SALES
105	Болгов	Виктор	37	…	$350 000.00	$367 911.00
109	Майоров	Олег	31	…	$300 000.00	$392 725.00
102	Сергеев	Игорь	48	…	$350 000.00	$474 050.00
106	Санкин	Петр	52	…	$275 000.00	$299 912.00
104	Бобров	Иван	33	…	$200 000.00	$142 594.00
101	Данилов	Сергей	45	…	$300 000.00	$305 673.00

Представление SLUZHASCHIE_OFFISY

FAMILY	NAME	CITY	REGION	QUOTA	SALES
Болгов	Виктор	Инза	Ульяновская	$350 000.00	$367 911.00
Майоров	Олег	Буинск	Татарстан	$300 000.00	$392 725.00
Сергеев	Игорь	Тверь	Московская	$350 000.00	$474 050.00
Санкин	Петр	Буинск	Татарстан	$275 000.00	$299 912.00
Бобров	Иван	Тверь	Московская	$200 000.00	$142 594.00
Данилов	Сергей	Инза	Ульяновская	$300 000.00	$305 673.00

Таблица OFFISY

ID_OFFICE	CITY	REGION	MGR	
22	Инза	Ульяновская	104	
11	Буинск	Татарстан	106	
12	Тверь	Московская	108	

Рис. 3.2. Типичное представление с двумя исходными таблицами

В большинстве случаев представления используются для обеспечения безопасности данных. Например, некоторые категории пользователей могут иметь доступ к представлению, но не к таблицам, данные которых его формируют; кроме того, SQL-запрос может содержать параметр USER (имя, под которым зарегистрировался пользователь), и в этом случае данные, доступные при обращении к представлению будут зависеть от имени пользователя.

3.4.4. Удаление представления

Удалить просмотр можно следующим оператором:

```
DROP VIEW <ИмяПросмотра>;
```

Например,

```
DROP VIEW vStore;
```

3.4.5. Недостатки представлений

Наряду с перечисленными выше преимуществами, представления обладают и двумя существенными недостатками.

– ***Производительность***. Представление создает лишь видимость существования соответствующей таблицы, и СУБД приходится преобразовывать запрос к представлению в запрос к исходным таблицам. Если представление отображает многотабличный запрос, то простой запрос к представлению становится сложным объединением и на его выполнение может потребоваться много времени;

– ***Ограничения на обновление***. Когда пользователь пытается обновить строки представления, СУБД должна установить их соответствие строкам исходных таблиц, а также обновить последние. Это возможно только для простых представлений (созданных на основе одной таблицы); сложные представления обновлять нельзя, они доступны только для выборки.

Указанные недостатки означают, что не стоит без разбора применять представления вместо исходных таблиц. В каждом конкретном случае необходимо учитывать перечисленные преимущества и недостатки представлений.

3.5. СОЗДАНИЕ ИНДЕКСОВ

3.5.1. Общий формат оператора CREATE INDEX

Индекс может быть создан оператором:

```
CREATE [UNIQUE] [ASC[ENDING]|DESC[ENDING]]
INDEX ИмяИндекса ON ИмяТаблицы (столбец1 [, столбец2 ...]);
```

– `UNIQUE` – требует создания уникального индекса;

– `ASC[ENDING]` – указывает на необходимость сортировки значений индексных полей по возрастанию (по умолчанию);

– `DESC[ENDING]` – указывает на необходимость сортировки значений индексных полей по убыванию;

– `ИмяИндекса` – имя создаваемого индекса;

– `ИмяТаблицы` – имя таблицы, для которой создается индекс;

– `столбецN` – имена столбцов, по которым создается индекс.

Ниже дан пример инструкции `CREATE INDEX`, которая создает индекс для таблицы `ZAKAZY` на основе столбцов `MFR` и `PRODUCT` и содержит требование уникальности для комбинации этих столбцов:

```
CREATE UNIQUE INDEX ORD_PROD_IDX
ON ZAKAZY (MFR, PRODUCT);
```

3.5.2. Необходимость создания индексов

Индексы необходимо создавать в том случае, когда по столбцу или группе столбцов:

– часто производится поиск в базе данных (столбец или группа часто перечисляются в предложении WHERE оператора SELECT);

– часто строятся объединения таблиц;

– часто производится сортировка (т. е. столбец или столбцы часто используются в предложении ORDER BY оператора SELECT.

Не рекомендуется строить индексы по столбцам или группам столбцов, которые:

– редко используются для поиска, объединения и сортировки результатов запросов;

– часто меняют значение, что приводит к необходимости часто обновлять индекс и способно существенно замедлить скорость работы с базой данных;

– содержат небольшое число вариантов значения.

3.5.3. Удаление индекса

Для удаления индекса, созданного оператором CREATE INDEX, используется оператор

```
DROP INDEX <имя_индекса>;
```

Нельзя удалить индекс, созданный в результате определения первичного и внешнего ключей. Для этой цели следует использовать оператор ALTER TABLE.

ГЛАВА 4. DML – ЯЗЫК МАНИПУЛИРОВАНИЯ ДАННЫМИ РЕЛЯЦИОННОЙ МОДЕЛИ

С точки зрения человека, пользующегося тем или иным хранилищем данных, существуют всего четыре действия над данными: поиск и выборка запрошенных данных, ввод новых данных, обновление существующих данных и удаление данных, ставших ненужными. В соответствии с этим в SQL для решения этих задач и предусмотрены четыре команды:

– SELECT – выборка данных, удовлетворяющих заданным условиям;
– INSERT – ввод новых данных;
– UPDATE – обновление существующих данных;
– DELETE – удаление данных.

Каждая из этих команд имеет множество вариантов, которые заслуживают отдельного рассмотрения.

4.1. ОПЕРАТОР ВЫБОРКИ SELECT

4.1.1. Общий формат оператора SELECT

Инструкция SELECT, используемая для построения SQL-запросов, является наиболее мощной из всех инструкций SQL, и реализует все операции реляционной алгебры.

Синтаксис оператора SELECT имеет вид:

```
SELECT [ALL | DISTINCT] (<Список полей>)
FROM <Список таблиц>
[WHERE <Предикат – условие выборки или соединения>]
[GROUP BY <Список полей результата>]
[HAVING <Предикат – условие для группы>]
[ORDER BY < список_столбцов>]
```

Инструкция состоит из шести предложений:

– предложения SELECT, содержащего список столбцов, которые должны быть возвращены инструкцией;

– предложения FROM, перечисляющего список имен таблиц, содержащих элементы данных, извлекаемые запросом;

– предложения WHERE, содержащего условия отбора записей из перечисленных таблиц;

– предложения GROUP BY, позволяющего создать итоговый запрос. Обычный запрос включает в результаты запроса по одной записи для каждой строки из таблицы. Итоговый запрос вначале группирует строки базы данных по определенному признаку, а затем включает в результаты запроса одну итоговую строку для каждой группы;

– предложения HAVING, указывающего что в результаты запроса следует только некоторые из групп, созданных с помощью предложения GROUP BY. В этом предложении, как и в предложении WHERE, для отбора включаемых групп используются условия отбора;

– предложение ORDER BY сортирует результаты запроса на основании данных, содержащихся в одном или нескольких столбцах.

4.1.2. Предложение SELECT

С предложения SELECT начинаются все инструкции SELECT. Наличие этого предложения обязательно, и оно используется для определения столбцов, которые вы хотите получить в наборе данных для своего запроса. Эти элементы задаются в виде списка возвращаемых столбцов, разделенных запятыми. Для каждого элемента из этого списка в таблице результатов будет создан один столбец, которые будут расположены в том порядке, что элементы списка.

Возвращаемый столбец может представлять собой:

– имя столбца, идентифицирующее один из столбцов, содержащихся в таблицах, которые перечислены в предложении FROM;

– константу, показывающую, что в каждой строке результатов запроса должно содержаться одно и то же значение. Это может пригодиться для создания таблицы результатов запроса, которая более удобна для восприятия, как в следующем примере

```
SELECT CITY, ‘имеет объем продаж’, SALES
FROM OFFISY
```

В результате выполнения этого запроса получаем таблицу

CITY	ИМЕЕТ_ОБЪЕМ_ПРОДАЖ	SALES
Инза	имеет объем продаж	$186 000.00
Буинск	имеет объем продаж	$567 000.00
Тверь	имеет объем продаж	$735 000.00

– выражение, показывающее, что СУБД должна помещать в результирующую таблицу значение, вычисляемое по формуле, заданной в выражении. В примере в результирующую таблицу добавляется столбец, содержащий значение, превышающее плановое задание по продажам:

```
SELECT CITY, REGION, SALES-TARGET
FROM OFFISY
```

В результате выполнения этого запроса получаем таблицу

CITY	REGION	SALES - TARGET
Инза	Ульяновская	- $389 000.00
Буинск	Татарстан	- $118 000.00
Тверь	Московская	- $065 000.00

В предложении SELECT могут быть использованы следующие обозначения:

* – означает, что в результирующий набор строк включаются все столбцы из исходных таблиц запроса;

ALL – в результирующий набор строк включаются все строки, удовлетворяющие условиям запроса, то есть могут иметь место одинаковые строки;

DISTINCT – в результирующий набор включаются только различные строки, то есть дубликаты строк результата не включаются в набор.

Смысл использования этих обозначений следующий. Если в списке возвращаемых столбцов запроса на выборку указан первичный ключ таблицы, то каждая строка запроса будет уникальной. В противном случае результаты запроса могут содержать повторяющиеся строки.

Повторяющиеся строки из таблицы результатов можно удалить, если в инструкции SELECT перед списком возвращаемых столбцов указать предикат DISTINCT.

Наоборот, если в результирующий запрос нужно включить все записи, после SELECT указывают слово ALL (во многих СУБД это делается по умолчанию).

Столбцам можно присвоить псевдонимы с помощью предиката AS:

```
SELECT имя_столбца AS новое_имя_столбца
```

4.1.3. Предложение FROM

Предложение FROM содержит список имен таблиц, разделенных запятыми. Каждое имя определяет таблицу, содержащую данные, извлекаемые данным запросом. Такие таблицы называются исходными таблицами запроса (инструкции SELECT), поскольку все данные, содержащиеся в таблице результатов запроса, берутся из них.

При составлении много табличных запросов может, что в разных таблицах имеются одноименные столбцы. При этом необходимо перед именем столбца через точку указать имя таблицы. Использование имен таблиц при написании имен столбцов может привести к громоздким записям. Намного лучше присвоить каждой таблице какое-нибудь краткое имя. Такие имена называются псевдонимами таблиц. Формат задания псевдонимов таблиц следующий:

```
SELECT  . . .
FROM <таблица1 псевдоним> [, <таблица1 псевдоним> ...]
WHERE   . . .
```

Например, запрос

```
SELECT SLUZHASCHIE.FAMILY, SLUZHASCHIE.NAME, OFFISY.CITY
FROM SLUZHASCHIE, OFFISY
WHERE SLUZHASCHIE.ID_OFC = OFFISY.ID_OFC
```

после введения в него псевдонимов выглядит так

```
SELECT C.FAMILY, C.NAME, O.CITY
FROM SLUZHASCHIE C, OFFISY O
WHERE C.ID_OFC = O.ID_OFC
```

4.1.4. Предложение WHERE

Предложение WHERE используется для включения в набор данных лишь нужных записей. В этом случае оператор SELECT имеет следующий формат:

```
SELECT {* | <Список_полей>}
FROM <Список_таблиц>
WHERE <условие_выборки>
```

В наборе данных, возвращаемых оператором SELECT, будут включены только те записи, которые удовлетворяют условиям поиска.

В SQL используется множество условий отбора, позволяющих создавать различные типы запросов. Мы рассмотрим пять основных условий отбора:

сравнение – значение одного выражения сравнивается со значением другого выражения. Например, такое условие отбора используется для отбора всех офисов, находящихся в Московской области, или всех служащих, фактические объемы продаж которых превышают плановые;

проверка на принадлежность диапазону – проверяется, попадает ли указанное значение в определенный диапазон. Например, такое условие отбора используется для нахождения служащих, чей возраст больше 30, но меньше 50 лет;

проверка на членство в множестве – проверяется, совпадает ли значение выражения с одним из значений заданного множества. Например, такое условие отбора используется для выбора офисов, расположенных в городах Москва, Пенза или Самара;

проверка на соответствие шаблону – проверяется, соответствует ли значение, содержащееся в столбце, определенному шаблону. Например, такое условие используется для выбора клиентов, чьи имена начинаются с буквы «A»;

проверка на равенство на значение NULL – проверяется, содержится ли в столбце значение NULL. Например, такое условие отбора используется для нахождения всех служащих, которым не был назначен руководитель.

Сравнение

Наиболее общим типом условия выборки является сравнение. При сравнении СУБД вычисляет и сравнивает значения двух выражений для каждой строки данных. Синтаксис сравнения можно записать следующим образом:

```
<выражение1> <оператор> <выражение2>
оператор – {= | < | > | <= | >= | !< | !> | <> | !=}.
```

Сравнение столбца с константой. Чаще всего выражения бывают простыми и содержат в качестве выражения1 имя столбца, а в качестве выражения2 – константу. При этом условие поиска имеет такой вид:

```
<имя_столбца> <оператор> <константа>.
```

Здесь в качестве константы явно указываются строковые или числовые значения.

Сравнение столбца с результатом вычисления выражения. В более сложных ситуациях условие поиска в предложении WHERE может быть сформулировано при помощи выражения:

```
<выражение1> <оператор> <имя_столбца>
```

Или может использоваться и другой вариант написания условия поиска

```
<имя_столбца> <оператор> <выражение2>
```

Последний способ чаще применяется при использовании механизма вложенных подзапросов, речь о которых пойдет ниже. В обоих случаях результат вычисления выражения сравнивается с содержимым указанного столбца.

Использование логических выражений. Сложные логические выражения строятся при помощи операторов AND, OR и NOT. Их использование, а также построение из них сложных выражений подчиняется стандартным правилам, принятым для большинства алгоритмических языков, с одним исключением: операции отношения в них имеют больший приоритет, чем логические операции, что избавляет от необходимости расстановки многочисленных скобок.

Пример: получить фамилию и имя служащих из таблиц SLUZHASCHIE, плюс город в котором он работает из таблицы OFFISY. При этом возраст служащего должен быть не меньше 30 и не больше 50 лет.

```
SELECT S.FAMILY, S.NAME, O.CITY
FROM SLUZHASCHIE S, OFFISY O
WHERE (S.ID_OFC = O.ID_OFC) AND
      (S.AGE >= 30 AND S.AGE <= 50)
```

Когда СУБД сравнивает значения двух выражений, могут получиться следующие результаты:

– если значение истинно, то результат проверки имеет значение TRUE;
– если значение ложно, то результат проверки имеет значение FALSE
– если хотя бы одно из выражений имеет значение NULL, то результат проверки имеет значение NULL.

Следует помнить, что в трехзначной логике SQL в результат запроса попадают только те строки, для которых условие отбора рано TRUE. Поэтому строки, содержащие NULL-значения, «исчезают» при выполнении запроса.

Проверка на принадлежность диапазону

Предикат BETWEEN A AND B – принимает значения между A и B. Предикат истинен, когда сравниваемое значение попадает в заданный диапазон, включая границы диапазона. Одновременно в стандарте задан и противоположный предикат Not Between A and B, который истинен только тогда, когда сравниваемое значение не попадает в заданный интервал, включая его границы.

В условии поиска можно указать, что некоторое значение должно находиться в интервале между значениями:

```
<проверяемое выражение> [NOT] BETWEEN <A> AND <B>
```

Оператор BETWEEN ... AND проверяет, находится ли проверяемое выражение между двумя заданными значениями A и B. При этом типы данных выражений проверяемое выражение, A и B должны быть сравнимыми.

Проверяемое выражение, заданное в операторе BETWEEN ... AND, может быть любым допустимым выражением, однако обычно оно представляет собой имя столбца.

Пример: вывести сведения обо всех заказах, сделанных между 1 и 31 октябрем 1989 года.

```
SELECT ID_ORDER, DATE_ORDER, ID_MFR, ID_PRD
FROM   ZAKAZY
WHERE  DATE_ORDER BETWEEN `01-OCT-89` AND `31-OCT-89`
```

В результате выполнения этого запроса получаем таблицу

ID_ORDER	DATE_ORDER	ID_MFR	ID_PRD
112961	17.10.89	УАЗ	2A34
113888	21.10.89	ВАЗ	41234
122777	29.10.89	ПМЗ	4E45M

Следует помнить, что проверку на принадлежность диапазону можно выразить в виде двух операций сравнения.

Проверка на членство в множестве

Еще одним распространенным условием отбора является проверка на членство в множестве, реализуемое оператором IN. Если нужно, чтобы значение какого-либо столбца совпадало с одним из дискретных значений, в условии вызывается оператор проверки *вхождения в множество* IN, который истинен только тогда, когда сравниваемое значение входит в множество заданных значений. При этом множество значений может быть задано простым перечислением или встроенным подзапросом.

В условии поиска можно указать, что некоторое значение должно совпадать с одним из значений заданного множества:

```
<проверяемое_выражение> [NOT] IN список_констант
```

Пример: вывести список служащих, которые работают в Инзе (22) и Твери (12)

```
SELECT FAMILY, NAME, QUOTA, SALES*
FROM   SLUZHASCHIE
WHERE  ID_OFC IN (22, 12)
```

В результате выполнения этого запроса получаем таблицу

FAMILY	NAME	QUOTA	SALES
Петров	Петр	$350 000.00	$367 991.00
Федоров	Федор	$350 000.00	$476 456.00

Одновременно существует обратный предикат NOT IN (множество). В этом случае в результирующий набор данных будут включены только те записи, для

которых <значение>, стоящее слева от IN, равно одному из значений, указанных в списке (<значение1> [, <значение2> ...]).

Проверку IN также можно выполнить через проверку на сравнение.

Проверка на соответствие шаблону

Для выборки строк, в которых содержимое некоторого текстового столбца совпадает с заданным текстом, можно использовать простое сравнение.

Однако очень легко можно забыть, какое именно название носит интересующая нас компания: «Грант», «Гранит» или «Гранат». Проверка на соответствие шаблону позволяет выбрать из базы данных строки на основе частичного соответствия текстовых строк.

Предложение LIKE определяет шаблоны сравнения строковых значений. Если необходимо, чтобы сравниваемое значение (значение столбца или результат вычисления строкового выражения) удовлетворяло шаблону, в условии поиска необходимо указать

```
<значение> [NOT] LIKE <шаблон> [ESCAPE <подшаблон>]
```

Шаблон представляет собой строку, в которую может входить один или более подстановочных знаков, которые интерпретируются особым образом:

– символ «%» (или «*») обозначает строку любой длины;
– символ «_» (или «?») используется для указания любого единичного символа.

Подстановочные знаки можно помещать в любое место строки шаблона, и в одной строке может содержаться несколько подстановочных знаков.

При проверке строк на соответствие шаблону может оказаться, что подстановочные знаки входят в строку символов в качестве литералов (например, знак %). Для проверки наличия в строке литералов, использующихся в качестве подстановочных знаков, применяются *символы пропуска*.

Например, нельзя проверить, содержится ли знак процента в строке, просто включив его в шаблон, поскольку СУБД считает этот знак подстановочным. Но когда в шаблоне встречается символ пропуска, то символ, следующий за ним непосредственно, считается не подстановочным знаком, а литералом. Другими словами, происходит пропуск символа.

Символ пропуска определяется с помощью ключевого слова ESCAPE. Ниже приведен пример использования знака доллара ($) в качестве символа пропуска.

Найти товары, коды которых начинаются с букв A%BC

```
SELECT ID_ORDER, ID_PRD
FROM   ZAKAZY
WHERE  ID_PRD LIKE `A$%BC%` ESCAPE $
```

Первый символ % в шаблоне, следующий за символом пропуска $, считается литералом, второй − подстановочным знаком.

Проверка на значение NULL

Значения NULL обеспечивают возможность применения трехзначной логики в условиях отбора. Иногда бывает необходимо явно проверять значения столбцов на равенство NULL и непосредственно обрабатывать их.

Если требуется выдать все записи, в которых некоторый столбец имеет значение NULL (т. е. не имеет никакого значения), достаточно в условии поиска указать предложение

```
<значение> IS [NOT] NULL
```

Неопределенное значение интерпретируется в реляционной модели как значение, неизвестное на данный момент времени. Это значение при появлении дополнительной информации может быть заменено некоторым конкретным значением. Введение NULL – значений вызвало необходимость модификации классической двухзначной логики и превращения ее в трехзначную. Все логические операции, производимые с неопределенными значениями, подчиняются этой логике в соответствии с заданной таблицей истинности.

A	B	NotA	A(B)	A(B)
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	NULL	FALSE	NULL	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	NULL	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL	TRUE
NULL	FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL	NULL

Наличие неопределенных значений повышает гибкость обработки информации, хранящейся в баз данных.

Пример: найти служащих, которые еще не закреплены за офисом

```
SELECT FAMILY, NAME
FROM SLUZHASCHIE
WHERE ID_OFC IS NULL
```

Результат:

FAMILY	NAME
Скворцов	Петр
Семшов	Иван
Аршавин	Федор

Может показаться странным, но нельзя проверить значение на равенство NULL с помощью операции сравнения, например,

```
SELECT FAMILY, NAME
FROM SLUZHASCHIE
```

```
WHERE ID_OFC = NULL
```

Ключевое слово NULL здесь нельзя использовать, поскольку NULL это не значение, а просто сигнал о том, что значение неизвестно.

4.1.5. Правила выполнения запроса SELECT

Назначение простого оператора SELECT состоит в выборке и отображении данных одной таблицы базы данных. Это очень мощный оператор, способный выполнять действия, эквивалентные операциям реляционной алгебры, таким как выборка, проекция и объединение.

Результаты запроса, возвращаемые инструкцией SELECT, получаются в результате поочередного применения входящих в инструкцию предложений, в следующем порядке.

1. Берется таблица, указанная в предложении FROM.
2. Если имеется предложение WHERE, применить заданное в нем условие отбора к каждой строке таблицы. Если при этом получается значение TRUE, то текущая строка добавляется в результирующую таблицу, если получается значение FALSE, то строка отбрасывается.
3. Далее выполняется предложение SELECT, которое из указанных в нем столбцов создает строки результирующей таблицы.
4. Если в предложении SELECT указано ключевое слово DISTINCT, то повторяющиеся строки из результирующей таблицы удаляются.
5. Если в запросе имеется предложение ORDER BY, результирующая таблица сортируется.

4.2. АГРЕГАТНЫЕ ФУНКЦИИ

Агрегатные функции предназначены для вычисления итоговых значений операций над всеми записями набор данных. Многие запросы к базе данных требуют узнать всего одно или несколько значений, которые подытоживают информацию, содержащуюся в базе данных. Например:

– Какова общая сумма плановых продаж для всех служащих?
– Каковы наибольший и наименьший объемы продаж?
– Сколько служащих перевыполнили план?
– Какова средняя стоимость заказа?
– Какова средняя стоимость заказа в каждом офисе?
– Сколько служащих закреплено за каждым офисом?

В SQL запросы такого типа можно создавать с помощью агрегатных функций.

Агрегатные функции используются подобно именам полей в операторе SELECT, но с одним исключением: они берут имя поля как аргумент. С функциями SUM и AVG могут использоваться только числовые поля.

К агрегатным относятся следующие функции:

Функция	Результат
AVG	Среднеарифметическое значение выбранных значений данного поля
SUM	Сумма всех выбранных значений данного поля
MIN	Наименьшее из всех выбранных значений данного поля

```
MAX....Наибольшее из всех выбранных значений данного поля
COUNT..Количество строк или непустых значений полей, которые выбрал запрос
```

4.2.1. Вычисление среднего значения столбца

Агрегатная функция `AVG()` вычисляет среднее всех значений столбца. Аргументом агрегатной функции может быть простое имя столбца как показано ниже. Данные, содержащиеся в этом столбце должны иметь числовой тип.

Пример. *Каковы плановый и средний фактический объемы продаж компании?*

```
SELECT AVG(QUOTA), AVG(SALES)
FROM SLUZHASCHIE
```

Аргументом агрегатной функции может быть также выражение, как показано в следующем примере.

Пример. *Какой средний процент выполнения плана в компании?*

```
SELECT AVG(100 * (SALES/QUOTA))
FROM SLUZHASCHIE
```

После выполнения этого запроса СУБД создает временный столбец, содержащий значения `100 * (SALES/QUOTA)` для каждой строки таблицы `SLUZHASCHIE`, а затем вычисляет среднее значение временного столбца.

4.2.2. Вычисление суммы значений столбца

Агрегатная функция `SUM( )` вычисляет сумму всех значений столбца. При этом столбец должен иметь числовой тип данных (целые числа, десятичные числа, числа с плавающей запятой, или денежные величины). Результат, возвращаемый этой функцией, имеет тот же тип данных, что и столбец.

Ниже приведен пример, в котором используется функция `SUM( )`.

Пример. *Каковы общий плановый и общий фактический объемы продаж в компании?*

```
SELECT SUM(QUOTA), SUM(SALES)
FROM SLUZHASCHIE
```

4.2.3. Вычисление экстремумов

Агрегатные функции `MIN( )` и `MAX( )` позволяют найти соответственно наименьшее и наибольшее значения в столбце. При этом столбец может содержать числовые или строковые значения, либо значения даты/времени. Результат, возвращаемый этими функциями, имеет точно такой же тип данных, что и сам столбец.

Ниже приведен пример, показывающий использование данных функций.

Пример. *Каковы наибольший и наименьший плановые объемы продаж в компании?*

```
SELECT MIN(QUOTA), MAX(QUOTA)
FROM SLUZHASCHIE
```

В случае применения функций MIN() и MAX() к числовым данным числа сравниваются по арифметическим правилам (среди двух отрицательных чисел меньше то, у которого модуль больше; нуль меньше любого положительного числа и больше любого отрицательного). Сравнение дат происходит последовательно (более ранние значения дат считаются меньшими, чем более поздние). Сравнение интервалов времени выполняется на основании их продолжительности.

В случае применения функций MIN() и MAX() к строковым данным результат сравнения двух строк зависит от используемой таблицы кодировки. На персональных компьютерах используется таблица кодировки ASCII, где установлен порядок сортировки, при котором цифры идут перед буквами, а все прописные буквы – перед строчными.

4.2.4. Вычисление количества значений в столбце

Агрегатная функция COUNT() подсчитывает количество значений в столбце. При этом тип данных столбца может быть любым. Функция COUNT() всегда возвращает целое число независимо от типа данных столбца.

Пример. *Подсчитать количество клиентов компании?*

```
SELECT COUNT(ID_CLN)
FROM CLIENTY
```

4.2.5. Правила выполнения запросов, в котором участвуют агрегатные функции

Таблица результатов запроса, в котором участвуют агрегатные функции, генерируется следующим образом.

1. Берется таблица, указанная в предложении FROM.

2. Если имеется предложение WHERE, применить заданное в нем условие отбора к каждой строке таблицы. Если при этом получается значение TRUE, то текущая строка добавляется в результирующую таблицу, если получается значение FALSE, то строка отбрасывается.

3. Для каждой из оставшихся строк вычислить значение каждого элемента в списке возвращаемых столбцов и создать одну строку таблицы результатов запроса.

Смысл запроса, в котором участвуют агрегатные функции, легче понять, если разбить процесс выполнения на два этапа.

Сначала представим выполнение запроса без агрегатных функций (п.п. 1-5 раздела 4.1.5).

Затем представим, как СУБД применяет агрегатные функции к результатам запроса, формируя при этом итоговую строку.

Все агрегатные функции возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым полям, так и к нечисловым полям, тогда как функции SUM и AVG могут применяться только к числовым полям.

4.3. ЗАПРОСЫ С ГРУППИРОВКОЙ

4.3.1. Предложение GROUP BY

Иногда требуется получить агрегированные значения (минимум, максимум, среднее) не по всему результирующему набор данных, а по каждой из входящих в него групп записей, характеризующихся одинаковыми значениями какого-либо столбца. Для применения агрегатных функций предполагается предварительная операция группировки. В чем состоит суть операции группировка? При группировке все множество кортежей отношения разбивается на имеющие одинаковые значения атрибутов, которые заданы в списке группировки.

Эту возможность предоставляет предложение `GROUP BY` инструкции `SELECT`. Назначение предложения `GROUP BY` лучше всего можно понять на примере.

Пример. *Какова средняя стоимость заказа для каждого сотрудника компании?*

```
SELECT ID_SLZH, AVG(PRICE)
FROM ZAKAZY
GROUP BY ID_SLZH
```

На логическом уровне запрос выполняется следующим образом:

1. Заказы делятся на группы, по одной для каждого служащего. В каждой группе все заказы имеют одно и то же значение в столбце `ID_SLZH`.
2. Для каждой группы вычисляется среднее значение столбца `PRICE` по всем строкам, входящим в группу, и генерируется одна итоговая строка результатов. Эта строка содержит значение столбца `ID_SLZH` для группы и среднюю стоимость заказа для данной группы.

4.3.2. Предложение HAVING

Если в результирующих наборах данных нужно выдавать агрегацию не по всем группам, а только по тем из них, которые отвечают некоторому условию, после предложения `GROUP BY` указывают предложение

```
HAVING <агрегатная функция> <отношение> <значение>
```

`агрегатная функция` – одна из функций `MIN`, `MAX`, `AVG`, `SUM`;

`отношение` – одна из операций отношения `=`, `<>`, `<`, `>`, `<=`, `>=`;

`значение` – константа, результат вычисления выражения или единичное значение, возвращаемое вложенным оператором `SELECT`.

Таким образом, после `HAVING` указываются условия, которые отличаются от условий, определяемых в предложении `WHERE`, одним важным обстоятельством: в них обязательно должна быть указана одна из агрегатных функций, в то время как в предложении `WHERE` такие функции указывать нельзя.

Агрегатные функции могут применяться как в выражении вывода результатов строки `SELECT`, так и в выражении условия обработки сформированных групп `HAVING`. В этом случае каждая агрегатная функция вычисляется для каждой выделенной группы. Значения, полученные при вычислении агрегатных функций, могут быть использованы для вывода соответствующих результатов или для условия отбора групп.

Пример. *Построить запрос, который выводит среднюю стоимость заказа для каждого служащего из числа тех, у кого общая стоимость заказа превышает* `$30 000`?

```
SELECT ID_SLZH, AVG(PRICE)
FROM ZAKAZY
GROUP BY ID_SLZH
HAVING SUM(PRICE) > 30 000.00
```

4.3.3. Предложение ORDER BY – определение сортировки

Строки результатов запроса, как и строки таблицы базы данных, не имеют определенного порядка. Но включив в инструкцию `SELECT` предложение `ORDER BY`, можно отсортировать результаты запроса.

Результирующий набор данных можно отсортировать с помощью предложения

```
ORDER BY <список_столбцов>
```

`список_столбцов` – содержит имена столбцов, по которым будет производиться сортировка. Если указаны два и более столбцов, первый столбец будет использован для глобальной сортировки, второй столбец для сортировки внутри группы, определяемой единым значением первого столбца, и т. д. Например, результаты следующего запроса отсортированы по двум столбцам, `REGION`, `CITY`.

Пример. *Показать физические объемы продаж для каждого офиса, отсортированные в алфавитном порядке по названиям регионов и в каждом регионе – по названиям городов.*

```
SELECT  CITY, REGION, SALES
FROM OFFISY
ORDER BY REGION, CITY
```

В предложении `ORDER BY` можно выбрать возрастающий или убывающий порядок сортировки. По умолчанию данные сортируются в порядке возрастания. Чтобы сортировать их по убыванию, следует включить в предложение сортировки ключевое слово `DESC`, как это сделано в следующем примере.

Пример. *Показать список офисов, отсортированный по фактическим объемам продаж в порядке убывания.*

```
SELECT CITY, REGION, SALES
FROM OFFISY
ORDER BY SALES DESC
```

4.3.4. Правила выполнения запросов с группировкой

На запросы, в которых используется группировка, накладываются ограничения. В список возвращаемых столбцов запроса с группировкой всегда входят столбец (столбцы) группировки и одна или несколько агрегатных функций. Таким образом, возвращаемым столбцом может быть:

– константа;

– агрегатная функция, возвращающая одно значение для всех строк, входящих в группу;

– столбец группировки, который по определению имеет одно и то же значение во всех строках группы;

– выражение, включающее в себя перечисленные выше элементы.

Запросы с группировкой и с условиями отбора групп выполняются следующим образом.

1. Взять таблицу, указанную в предложении FROM.

2. Если имеется предложение WHERE, применить заданное в нем условие отбора к каждой строке таблицы. Если при этом получается значение TRUE, то текущая строка добавляется в результирующую таблицу, если получается значение FALSE, то строка отбрасывается.

3. Если имеется предложение GROUP BY, разделить строки таблицы таким образом, чтобы строки в каждой группе имели одинаковые значения в столбцах группировки (т. е. в столбцах, указанных за ключевым словом GROUP BY).

4. Если имеется предложение HAVING, применить заданное в нем условие к каждой группе и оставить в результирующей таблице те группы, для которых это условие выполняется.

5. Для каждой из оставшихся групп строк вычисляются значения агрегатных функций, указанных в предложении SELECT. При этом для каждой группы формируется одна строка, содержащая значения столбцов группировки и значения агрегатных функций, указанных в предложении SELECT.

4.4. ВЛОЖЕННЫЕ ЗАПРОСЫ

В SQL существует механизм вложенного запроса, позволяющий использовать результаты одного запроса в другом запросе. Этот механизм играет важную роль в SQL по следующим причинам:

– вложенные запросы соответствуют словесному описанию запроса и поэтому являются самым естественным способом выражения запроса;

– вложенные запросы позволяют структурировать запрос путем разбиения на части (на главный запрос и вложенные запросы);

– существуют ситуации, когда невозможно обойтись без вложенных запросов.

Вложенным называется запрос, содержащийся в предложении WHERE или HAVING другого запроса.

Рассмотрим следующую ситуацию. *Требуется вывести список офисов, для которых плановый объем продаж* (поле TARGET) *превышает сумму плановых объемов продаж всех служащих* (поле QUOTA).

Первая часть этого запроса должна выглядеть как

```
SELECT CITY
FROM OFFISY
WHERE TARGET > ???
```

Во второй части заказа нужно ответить на вопрос: как определить сумму плановых объемов продаж служащих для отдельного офиса (скажем офиса с идентификатором 22).

Это можно сделать с помощью запроса, использующего агрегатную функцию SUM:

```
SELECT SUM(QUOTA)
FROM SLUZHASCHIE
WHERE ID_OFC = 22
```

А теперь объединим эти запросы путем замены знаков ??? на второй запрос и получим следующий структурированный запрос

```
SELECT CITY
FROM OFFISY
WHERE TARGET > (SELECT SUM(QUOTA)
               FROM SLUZHASCHIE
               WHERE SLUZHASCHIE.ID_OFC = OFFISY.ID_OFC)
```

В приведенном запросе вложенный (внутренний) запрос выполняет для каждого офиса вычисление суммы плановых продаж всех служащих, работающих в данном офисе. Главный (внешний) запрос сравнивает плановый объем продаж офиса с вычисленной суммой и, в зависимости от результата сравнения, либо добавляет текущую запись в результирующую, либо нет. В результате такой согласованной работы главный и вложенный запросы извлекают из базы данных требуемую информацию и формируют результирующую таблицу.

4.4.1. Определение подчиненных запросов

Вложенные запросы всегда входят в предложение WHERE или HAVING и заключаются в круглые скобки. В предложении WHERE они отбирают из таблицы отдельные строки, а в предложении HAVING – группы строк. Подчиненные запросы имеют ту же структуру, что и инструкция SELECT, содержащая предложение FROM и необязательные предложения WHERE, GROUP BY и HAVING. Однако между вложенным запросом и инструкцией SELECT имеются отличия:

– таблица результатов вложенного запроса всегда состоит из одного столбца, поэтому в предложении SELECT вложенного запроса всегда следует указывать только один возвращаемый столбец;

– во вложенный запрос не может входить предложение ORDER BY, так как результаты вложенного запроса используются только внутри главного запроса и для пользователя остаются невидимыми. Поэтому нет смысла их сортировать.

Чаще всего вложенные запросы используются в предложении WHERE и участвуют в процессе отбора строк. В простейшем случае вложенный запрос возвращает значение, позволяющее проверить истинность или ложность условия отбора.

Рассмотрим следующий пример: *вывести список служащих, чей плановый объем продаж составляет менее 10% от планового объема продаж всей компании*.

```
SELECT NAME
FROM SLUZHASCHIE
WHERE QUOTA < (0.1 * (SELECT SUM(TARGET)
                     FROM OFFISY))
```

В приведенном запросе вложенный запрос вычисляет одну и ту же сумму плановых объемов продаж всех офисов, которая затем умножается на 0,1 (10%). Полученное значение используется в условии отбора при сканировании таблицы SLUZHASCHIE на предмет поиска нужных строк.

Далее рассмотрим более сложный пример, приведенный в предыдущем разделе

```
SELECT CITY
FROM OFFISY
WHERE TARGET > (SELECT SUM(QUOTA)
               FROM SLUZHASCHIE
               WHERE SLUZHASCHIE.ID_OFC = OFFISY.ID_OFC)
```

На Рис. 4.1. приведена схема выполнения этого запроса, в котором вложенный запрос возвращает различные результаты для каждого офиса.

1. Главный запрос извлекает данные из таблицы OFFISY.

2. Условие предложения WHERE обеспечивает отбор офисов, включаемых в таблицу результатов запроса. Это условие поочередно применяется ко всем строкам таблицы OFFISY.

3. В предложении WHERE сравнивается значение текущей строки в столбце TARGET со значением, возвращаемым вложенным запросом.

4. Для каждой строки результирующей таблицы выполняется свой вложенный запрос, вычисляющий сумму плановых объемов продаж для служащих текущего офиса.

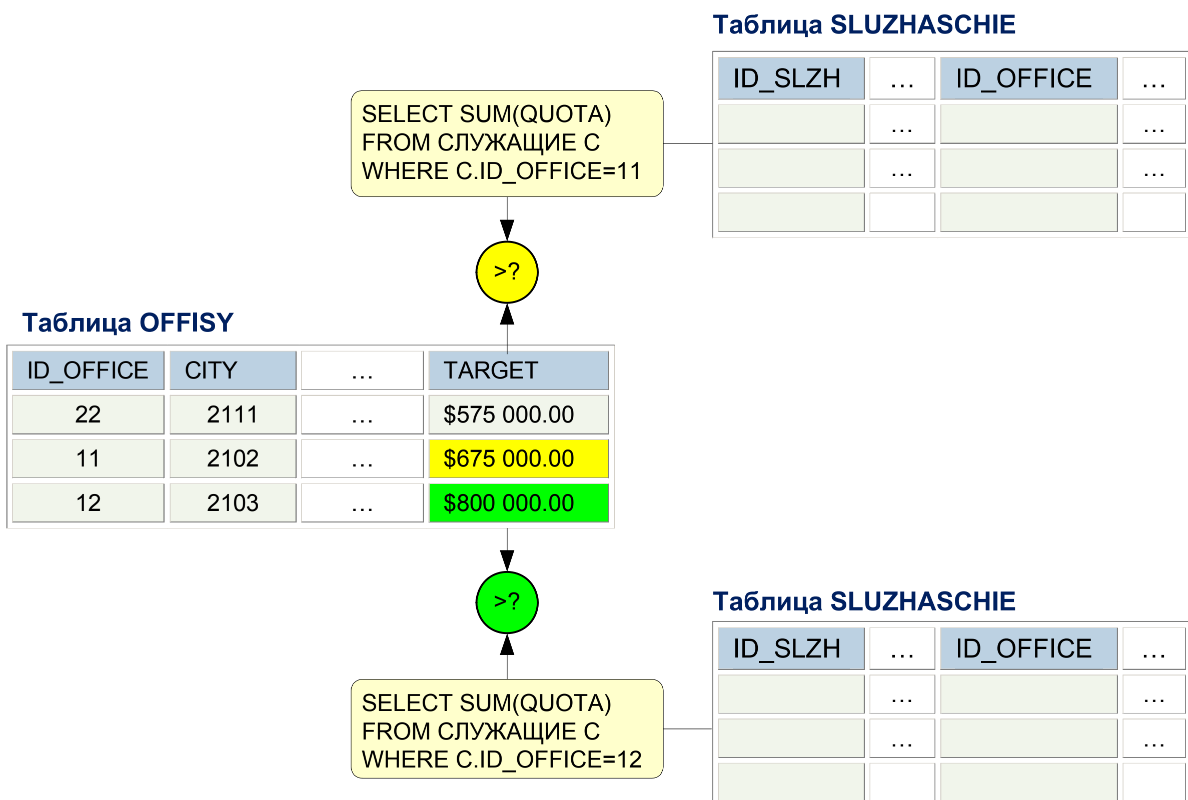


Рис. 4.1. Выполнение вложенного запроса в предложении WHERE

5. Результатом выполнения вложенного запроса является одно число, и предложение WHERE сравнивает его значение со значением столбца TARGET, выбирая или отбрасывая текущий офис на основании результата сравнения.

4.4.2. Условия отбора в подчиненном запросе

Вложенный запрос всегда является частью условия отбора в предложении WHERE и HAVING. В SQL используются следующие условия отбора вложенного во вложенном запросе:

– *сравнение с результатом подчиненного запроса*: значение выражения сравнивается с одним значением, вычисленным вложенным запросом. Эта проверка представляет собой простое сравнение;

– *проверка на принадлежность результатам вложенного запроса*: значение выражения проверяется на равенство одному из множества значений, возвращаемых вложенным запросом. Эта проверка представляет собой проверку на членство в множестве;

– *проверка на существование*: проверяется наличие строк в таблице результатов вложенного запроса;

– *многократное сравнение*: значение выражения сравнивается с каждым из множества значений, возвращаемых вложенным запросом.

Сравнение с результатом полученного запроса

В данном условии отбора значение выражения сравнивается со значением, вычисленным вложенным запросом. В случае совпадения значений проверка дает результат TRUE, а в случае несовпадения – FALSE. Этот вид условия отбора используется для сравнения значения из проверяемой строки с одним значением, полученным от вложенного запроса.

Приведем пример использования описываемого вида условия отбора: *вывести список служащих, у которых плановый объем продаж равен или больше планового объем продаж офиса, расположенного в Инзе*.

```
SELECT FAMILY, NAME
FROM SLUZHASCHIE
WHERE QUOTA >= (SELECT TARGET
                FROM OFFISY
                WHERE CITY = ‘Инза’)
```

FAMILY	NAME
Филатов	Петр
Полев	Андрей
Пронин	Игорь

В описываемом примере вложенный запрос считывает плановый объем продаж для офиса в Инзе. Затем это значение используется для отбора тех служащих, у которых плановый объем продаж выше, чем у этого офиса.

При этом следует иметь в виду, что вложенный запрос должен возвращать единичное значение, т. е. одну строку, содержащую один столбец. Если в ре-

зультате выполнения вложенного запроса выводится несколько строк или столбцов, то сравнение теряет смысл и СУБД выводит сообщение об ошибке.

В случае возвращения вложенным запросом значения NULL или если не будет выведено ни одной строки, то операция сравнения вернет значение NULL.

Проверка на принадлежность результатам вложенного запроса

Проверка на принадлежность результатам вложенного запроса, включенного в состав предиката IN, является аналогом проверки на членство в множестве. Текущее значение проверяемого выражения главного запроса сравнивается со столбцом значений, возвращаемых вложенным запросом. Если это значение равно одному из элементов столбца, то проверка дает результат TRUE и текущая строка включается в результирующую таблицу.

Приведем пример запроса, использующего проверку на принадлежность результатам вложенного запроса: *вывести список служащих тех офисов, где фактический объем продаж превышает плановый*.

```
SELECT FAMILY, NAME
FROM SLUZHASCHIE
WHERE SLUZHASCHIE.ID_OFC IN (SELECT OFFISY.ID_OFC
                FROM OFFISY
                WHERE SALES > TARGET)
```

FAMILY	NAME
Филатов	Петр
Петров	Петр

Вложенный запрос возвращает список идентификаторов офисов, где фактический объем продаж превышает плановый (в учебной базе данных есть два таких офиса – с идентификаторами 12 и 22). Главный запрос проверяет каждую строку таблицы SLUZHASCHIE, чтобы определить, работает ли данный служащий в одном из отобранных офисов.

В приведенном примере вложенный запрос возвращает в качестве результата множество значений, а предложение WHERE главного запроса проверяет равенство значения из строки таблицы главного запроса одному из значений полученного множества. Таким образом, проверка IN с вложенным запросом выполняется аналогично проверке IN в инструкции SELECT.

Проверка на существование

С помощью проверки на существование (предикат EXISTS) можно выяснить, содержится ли в таблице вложенного запроса хотя бы одна строка, удовлетворяющая условию отбора. Аналогичной проверки в простой инструкции SELECT не существует. Эта проверка возможна только во вложенном запросе.

Допустим, что требуется *вывести список товаров, на которые получен заказ стоимостью* $20 000.00 *или больше*. Чтобы решить эту задачу путем выполнения вложенного запроса с проверкой на существование, перефразируем эту задачу следующим образом: вывести список товаров, для которых в табли-

це ZAKAZY существует, по крайней мере, один заказ, удовлетворяющий условиям: является заказом на данный товар и имеет стоимость не менее $20 000.00.

Инструкция SELECT, используемая для решения поставленной задачи имеет вид

```
SELECT DESCRIPTION
FROM TOVARY
WHERE EXISTS (SELECT ID_ORDER
              FROM ZAKAZY
              WHERE ZAKAZY.ID_PRD = TOVARY.ID_PRD AND
                    ZAKAZY.ID_MFR = TOVARY.ID_MFR AND
                    ZAKAZY.PRICE >= 20000.00)
```

DESCRIPTION
Деталь кузова

В данном случае главный запрос последовательно просматривает все строки таблицы TOVARY и для каждой строки выполняется вложенный запрос. Результатом вложенного запроса является набор данных, содержащий номера всех заказов текущего товара на сумму не меньше чем $20 000.00.

Если такие заказы есть (т. е. набор данных не пустой), то проверка EXISTS возвращает TRUE. Если вложенный запрос не возвращает ни одной строки, то проверка EXISTS возвращает значение FALSE. Эта проверка никогда не возвращает NULL.

Многократное сравнение

При проверке на принадлежность результатам вложенного запроса с помощью предиката IN осуществляется многократное сравнение значения таблицы главного запроса с набором значений из таблицы вложенного запроса.

В SQL имеются еще две разновидности множественной проверки, осуществляемые с помощью предикатов ANY и ALL. С помощью этих предикатов проверка осуществляется не только на совпадение, но и на «больше» или «меньше».

Предикат ANY сравнивает проверяемое значение с набором данных, выбираемых вложенным запросом, используя операторы «=», «<>», «<», «<=», «>», «>=». Проверяемое значение поочередно сравнивается с каждым значением из набора данных. Если любое из этих сравнений дает значение TRUE, то и проверка ANY возвращает значение TRUE.

Приведем пример использования предиката ANY: *вывести список служащих, принявших заказ на сумму, большую, чем на* 10% *от плана*.

```
SELECT FAMILY, NAME
FROM SLUZHASCHIE S
WHERE (0.1*QUOTA < ANY (SELECT PRICE_ALL
                        FROM ZAKAZY Z
                        WHERE S.ID_SLZH = Z.ID_SLZH))
```

FAMILY	NAME
Ганин	Сергей
Петров	Петр
Нилов	Лев

Главный запрос последовательно проверяет все строки таблицы SLUZHA-SCHIE. Вложенный запрос находит все запросы, принятые текущим служащим, и возвращает набор данных, содержащий стоимости этих заказов. Предложение WHERE главного запроса вычисляет 10% от плана текущего служащего и использует это число в качестве проверяемого значения, сравнивая его со стоимостью каждого заказа, выбранного вложенным запросом.

Если есть хотя бы один заказ, стоимость которого превышает вычисленное значение, то проверка < ANY возвращает значение TRUE, а имя служащего заносится в результирующую таблицу. Если таких заказов нет, имя служащего в результирующую таблицу не заносится.

Предикат ALL, как и предикат ANY, использует один из шести операторов («=», «<>», «<», «<=», «>», «>=») для сравнения проверяемого значения с набором данных, выбранных вложенным запросом. В данном случае проверяемое значение последовательно сравнивается с каждым значением из набора данных. Если все сравнения дают положительный результат, то предикат ALL возвращает значение TRUE.

Пример. *Вывести список офисов с их плановыми объемами продаж, все служащие которых превысили плановый объем продаж на 50% от плана офиса.*

```
SELECT CITY, TARGET
FROM OFFICY O
WHERE (0.50*TARGET < ALL (SELECT SALES
                          FROM SLUZHASCHIE S
                          WHERE O.ID_OFC = S.ID_OFC))
```

CITY	TARGET
Инза	$300 000.00
Буинск	$575 000.00
Орел	$350 000.00

Главный запрос последовательно проверяет строку за строкой таблицы OFFICY. Вложенный запрос находит всех служащих, работающих в текущем офисе, и возвращает значение с фактическими объемами продаж для каждого служащего.

Предложение WHERE главного запроса вычисляет 50% от плана продаж офиса и сравнивает полученное значение со всеми объемами продаж, выдаваемыми вложенным запросом. Если все объемы продаж превышают вычисленное значение, то предикат < ALL возвращает значение TRUE и текущий офис включается в результирующую таблицу.

4.4.3. Подчиненные запросы в предложении HAVING

Вложенные запросы могут использоваться также в предложении HAVING, когда требуется отобрать группу строк. Рассмотрим следующий пример, содержащий вложенный запрос в предложении HAVING. *Вывести список служащих, у которых средняя стоимость заказов на товары, изготовленные компанией* ВАЗ, *выше, чем общая средняя стоимость заказов*.

```
SELECT FAMILY, NAME, AVG(PRICE_ALL)
FROM SLUZHASCHIE S, ZAKAZY Z
WHERE S.ID_SLZH = Z.ID_SLZH AND Z.ID_MFR = `ВАЗ`
GROUP BY FAMILY, S.ID_SLZH
HAVING AVG(PRICE_ALL) >= (SELECT AVG(PRICE_ALL)
           FROM ZAKAZY Z WHERE S.ID_SLZH = Z.ID_SLZH))
```

FAMILY	NAME	AVG(PRICE_ALL)
Ганин	Сергей	
Петров	Петр	
Нилов	Лев	

Алгоритм выполнения запроса

1. Вложенный запрос вычисляет среднюю стоимость по всем заказам.

2. Этот простой вложенный запрос вычисляет среднюю стоимость один раз, а затем многократно используется в предложении HAVING.

3. Главный запрос просматривает строки таблицы ZAKAZY, отыскивая все заказы на товары компании ВАЗ, и группирует их по именам служащих.

4. Предложение HAVING сравнивает среднюю стоимость по каждой группе товаров со средней стоимостью по всем заказам, вычисленной ранее.

5. Если средняя стоимость по группе больше, чем общая средняя стоимость, то данная группа строк сохраняется, если нет, то данная группа строк исключается.

6. Предложение SELECT создает для каждой группы итоговую строку, содержащую фамилию, имя и среднюю стоимость принятых ими заказов.

Описанный алгоритм выполнения запроса приведен на Рис. 4.2. .

4.4.4. Правила выполнения вложенных запросов

А теперь подведем итоги изучения вложенных запросов и сформулируем правила, позволяющие использовать результаты одного запроса для получения результатов другого.

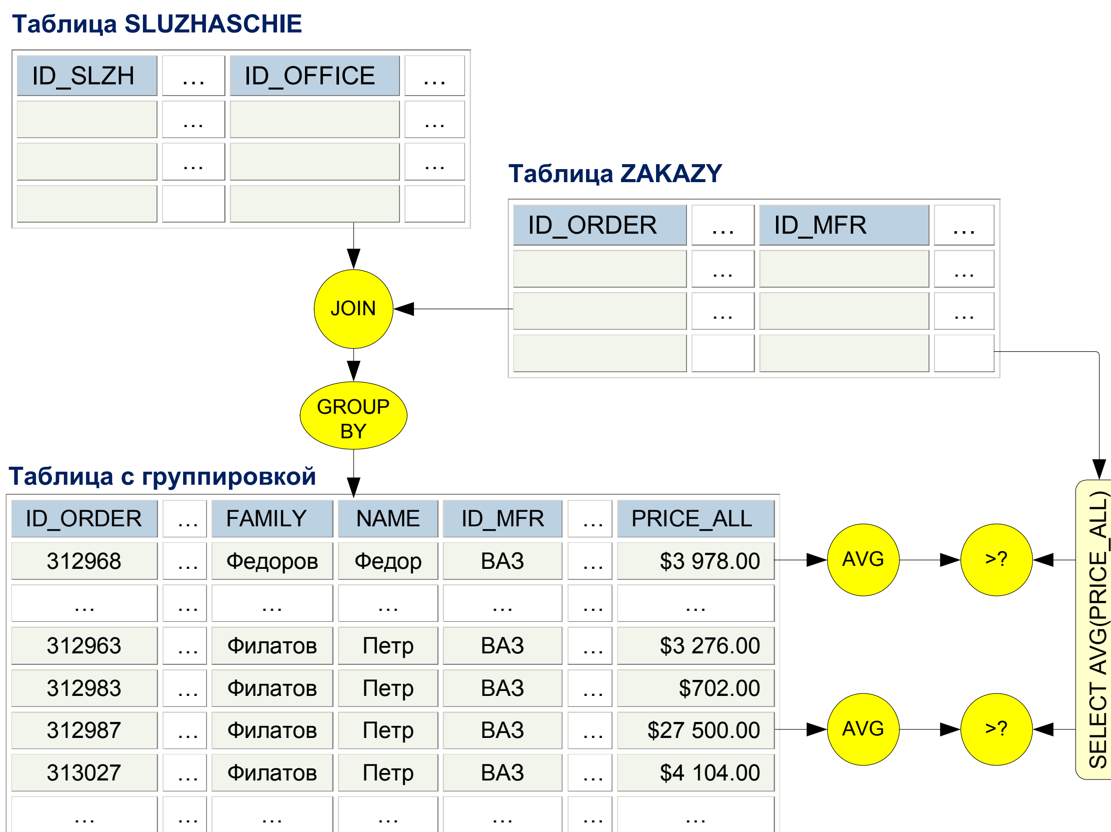


Рис. 4.2. Выполнение вложенного запроса в предложении HAVING

1. Если вложенный запрос содержится в предложении `WHERE`, его результаты используются для отбора отдельных строк, данные из которых заносятся в таблицу результатов запроса.

2. Если вложенный запрос содержится в предложении `HAVING`, его результаты используются для отбора групп строк, данные из которых заносятся в таблицу результатов запроса.

3. Вложенные запросы могут иметь несколько уровней вложенности.

4. При сравнении с результатом вложенного запроса проверяемое значение сравнивается с единственным значением, которое возвращается вложенным запросом.

5. При проверке на принадлежность результатам вложенного с помощью предиката `IN` значение выражения проверяется на равенство одному из множеств значений, которые возвращаются вложенным запросом.

6. При проверке на существование с помощью предиката `EXISTS` выясняется, возвращает ли вложенный запрос какие-либо значения.

7. При многократном сравнении с помощью предикатов `ANY` и `ALL` значение выражения сравнивается со всеми значениями, отобранными подчиненным запросом, чтобы выяснить, выполняется ли условие сравнения для некоторых, либо для всех значений.

4.5. МНОГОТАБЛИЧНЫЕ ЗАПРОСЫ

До сих пор мы рассматривали однотабличные запросы, однако на практике довольно часто приходится иметь дело с многотабличными запросами. Например, предположим, что требуется выполнить запрос, реализующий вывод списка служащих и офисов, в которых они работают (таблицы SLUZHASCHIE и OFFISY) или вывод списка заказов, выполненных за заданный период, включая следующую информацию: наименование заказанного товара, стоимость заказа и имя клиента (таблицы ZAKAZY, CLIENTY и TOVARY).

Для ответа на эти вопросы SQL обеспечивает возможность выполнения многотабличных запросов, объединяющих данные из нескольких таблиц.

4.5.1. Алгоритм выполнения многотабличного запроса

Рассмотрим порядок выполнения многотабличного запроса на примере запроса, объединяющего данные из двух различных таблиц (см. Рис. 4.3.). Допустим, что требуется вывести список всех заказов, включая номер и стоимость заказа, фамилию и имя клиента. Перечисленные данные содержатся в следующих таблицах:

– Номер (ID_ORDER) и Стоимость (PRICE) заказа содержатся в таблице ZAKAZY;

– Имя клиента (COMPANY) и лимит кредита (LIMIT_CREDIT) содержатся в таблице CLIENTY .

Рассмотрим умозрительный порядок выполнения данного заказа.

1. Сначала построим результирующую таблицу, содержащую четыре перечисленные выше колонки (ID_ORDER, COMPANY, PRICE, LIMIT_CREDIT).

2. Найдите в таблице ZAKAZY Номер и Стоимость первого заказа (первой записи) и перепишите полученные значения в поля ID_ORDER и PRICE результирующей таблицы.

3. Запомните Номер клиента (ID_CLN) для первой записи таблицы ZAKAZY.

4. Перейдите к таблице CLIENTY и в столбце ID_CLN найдите значение, соответствующее запомненному значению поля ID_CLN для первой записи таблицы ZAKAZY.

5. Отыщите Имя клиента и Лимит кредита для найденной записи таблицы CLIENTY и перепишите их в поля COMPANY и LIMIT_CREDIT результирующей таблицы.

6. Таким образом, мы создали одну строку результирующей таблицы. Далее повторите шаги 2 – 5 до тех пор, пока не будут перечислены все заказы.

В данном алгоритме можно зафиксировать два важных момента:

– каждая строка результирующей таблицы формируется из пары строк: одна строка находится в таблице ZAKAZY, а другая – в таблице CLIENTY .

– любая пара строк определяется по совпадению значений ключевых полей.

4.5.2. Внутреннее объединение таблиц

Описанный выше процесс формирования пар строк для записи в результирующую таблицу называется *объединением таблиц*.

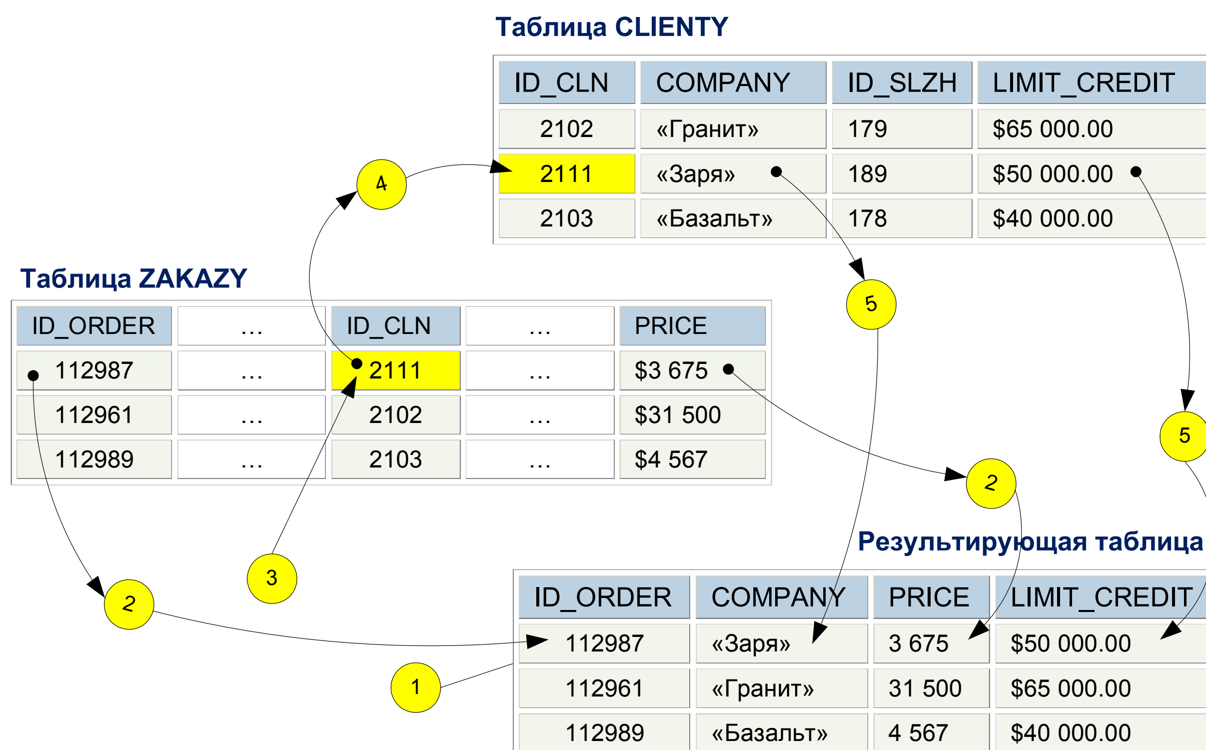


Рис. 4.3. Алгоритм выполнения многотабличного запроса

Объединение таблиц по равенству

Объединение на основе точного совпадения значений двух и более столбцов называется *объединением по равенству*. Объединения могут быть сформированы и на других видах сравнения значений столбцов (например, «больше» или «меньше»).

В реляционной базе данных информация между таблицами формируются путем сопоставления значений соответствующих столбцов. Таким образом, объединения являются мощным средством выявления отношений, существующих между данными. По этой причине оператор `SELECT` для многотабличного запроса должен содержать условие отбора, определяющее взаимосвязь между столбцами таблиц.

Приведем пример многотабличного запроса для описанного выше алгоритма: вывести список всех заказов, включающий номер и стоимость заказа, наименование компании-клиента и лимит кредита клиента.

```
SELECT ID_ORDER, COMPANY, PRICE, CREDIT_LIMIT
FROM ZAKAZY, CLIENTY
WHERE ZAKAZY.ID_CLN = CLIENTY.ID_CLN
```

Обратите внимание на то, что в приведенном запросе нет указаний о том, как должен выполняться запрос. В запросе ничего не сказано, с какой таблицы начинать выполнение процесса выборки, какую запись таблицы рассматривать первой, а какую – второй. В запросе указывается лишь то, каким должен быть результат выполнения запроса: из каких таблиц и значения каких полей должны содержать-

ся в результирующей таблице. Порядок выполнения запроса определяет СУБД. Этим и отличается язык SQL от алгоритмических языков типа C++ и Pascal.

Приведенный пример многотабличного запроса похож на однотабличные запросы за исключением следующих различий:

– Предложение FROM содержит две таблицы, а не одну.
– В предложении WHERE сравниваются связанные столбцы разных таблиц.

Запросы с использованием отношенияпредок-потомок

Как правило, многотабличные запросы выполняются для двух таблиц, связанных отношением предок-потомок. Запрос на извлечение данных о заказах и клиентах, приведенный выше, является примером такого запроса. У каждого заказа из таблицы-потомка существует соответствующий ему клиент из таблицы-предка. Таким образом, пары строк, из которых формируются строки результирующей таблицы, связаны отношением предок-потомок.

Чтобы использовать в многотабличном запросе отношение предок-потомок, необходимо задать в нем условие отбора, в котором значение первичного ключа сравнивается со значением вторичного ключа. Приведем пример такого запроса, который выводит список всех служащих с указанием городов и регионов, в которых они работают.

```
SELECT FAMILY, NAME, CITY, REGION
FROM SLUZHASCHIE, OFFISY
WHERE SLUZHASCHIE.ID_OFC = OFFISY.ID_OFC
```

Результат выполнения этого запроса будет иметь вид

FAMILY	NAME	CITY	REGION
Иванов	Иван	Буинск	Татарстан
Полев	Андрей	Буинск	Татарстан
Уткин	Денис	Буинск	Татарстан
Петров	Петр	Инза	Ульяновская
Филатов	Петр	Инза	Ульяновская
Пронин	Игорь	Тверь	Московская
Федоров	Федор	Тверь	Московская

Таблица SLUZHASCHIE (потомок) содержит столбец ID_OFC, который является вторичным ключом для таблицы OFFISY (предок). Это отношение предок-потомок используется с целью поиска в таблице OFFISY для каждого служащего соответствующей строки, содержащей город и регион, и включения ее в результаты запроса.

Рассмотрим еще один запрос, использующий отношение предок-потомок, но здесь роли предка и потомка меняются. Таблица OFFISY (потомок) содержит столбец MNGR, представляющий собой вторичный ключ для таблицы SLUZHASCHIE (предок).

Воспользуемся этой связью, чтобы для каждого офиса найти в таблице SLUZHASCHIE строку, содержащую фамилию и имя руководителя этого офиса, и включить ее в результирующую таблицу.

Создадим запрос, который выводит список офисов с указанием города, в котором он расположен, а также фамилию и имя руководителя офиса.

```
SELECT CITY, FAMILY, NAME
FROM OFFISY, SLUZHASCHIE
WHERE OFFISY.MNGR = SLUZHASCHIE.MNGR
```

Результат выполнения этого запроса приведен в следующей таблице.

CITY	FAMILY	NAME
Буинск	Полев	Андрей
Инза	Филатов	Петр
Тверь	Пронин	Игорь

Как правило, связанные столбцы в результирующую таблицу многотабличного запроса не вводятся, так как первичные и вторичные ключи представляют собой идентификаторы, трудно поддающиеся запоминанию, тогда как наименования (городов, офисов, сотрудников, должностей и т. д.) запоминаются гораздо легче.

Поэтому в предложении WHERE для объединения двух таблиц используются идентификаторы, а в предложении SELECT для создания столбцов результирующей таблицы – более удобные для восприятия имена.

Запросы на основе составных ключей

Таблицы ZAKAZY и TOVARY в учебной базе данных связаны с помощью составных ключей ID_MFR и ID_PRD. Поля ID_MFR и ID_PRD в таблице ZAKAZY составляют вторичный ключ для таблицы TOVARY и связаны с ее полями ID_MFR и ID_PRD соответственно. Чтобы объединить таблицы на основе составных ключей, в условии отбора необходимо задать все пары связанных полей, как показано в нижеприведенном примере.

Создадим запрос, который выводит список всех заказов с указанием их стоимости и наименования товаров.

```
SELECT ID_ORDER, DESCRIPTION, PRICE
FROM ZAKAZY, TOVARY
WHERE ZAKAZY.ID_MFR = TOVARY.ID_MFR AND
      ZAKAZY.ID_PRD = TOVARY.ID_PRD
```

ID_ORDER	DESCRIPTION	PRICE
112961	Деталь кузова	31 500
112987	Деталь двигателя	3 675
112989	Сопло	4 567

Условие отбора в данном запросе показывает, что связанными парами строк таблиц ZAKAZY и TOVARY являются те, в которых пары связанных столбцов содержат одни и те же значения. В SQL количество связанных столбцов не огра-

ничивается, но, как правило, отношения между таблицами создаются с помощью одной пары столбцов, реже с помощью двух или трех.

Правила выполнения многотабличных запросов на выборку

Написать правильную инструкцию SELECT для простых многотабличных запросов не сложно. Но если многотабличный запрос составлен из многих таблиц с использованием сложных условий отбора, то инструкция многотабличного запроса становится трудной для понимания. Поэтому сначала приведем более точное определение понятия «объединения».

Объединение – это подмножество декартова произведения двух таблиц. Произведение двух таблиц представляет собой таблицу, состоящую из всех возможных пар строк обеих таблиц, составленной из всех столбцов первой таблицы, за которыми следуют все столбцы второй таблицы. На основе понятия декартова произведения, определим формальное определение правил выполнения многотабличных запросов на выборку.

А теперь приведем правила, раскрывающие смысл любого многотабличного запроса на выборку и позволяющие определять процедуру.

1. Сформировать произведение таблиц, перечисленных в предложении FROM.
2. Если имеется предложение WHERE, применить заданное в нем условие отбора к каждой строке таблицы произведения и оставить в ней те строки, для которых это условие выполняется.
3. Для каждой из оставшихся строк сформировать строку результирующей таблицы, включающей столбцы, указанные в предложении SELECT.
4. Если в предложении SELECT указано ключевое слово DISTINCT, то повторяющиеся строки из результирующей таблицы удаляются.
5. Если в запросе имеется предложение ORDER BY, результирующая таблица сортируется.

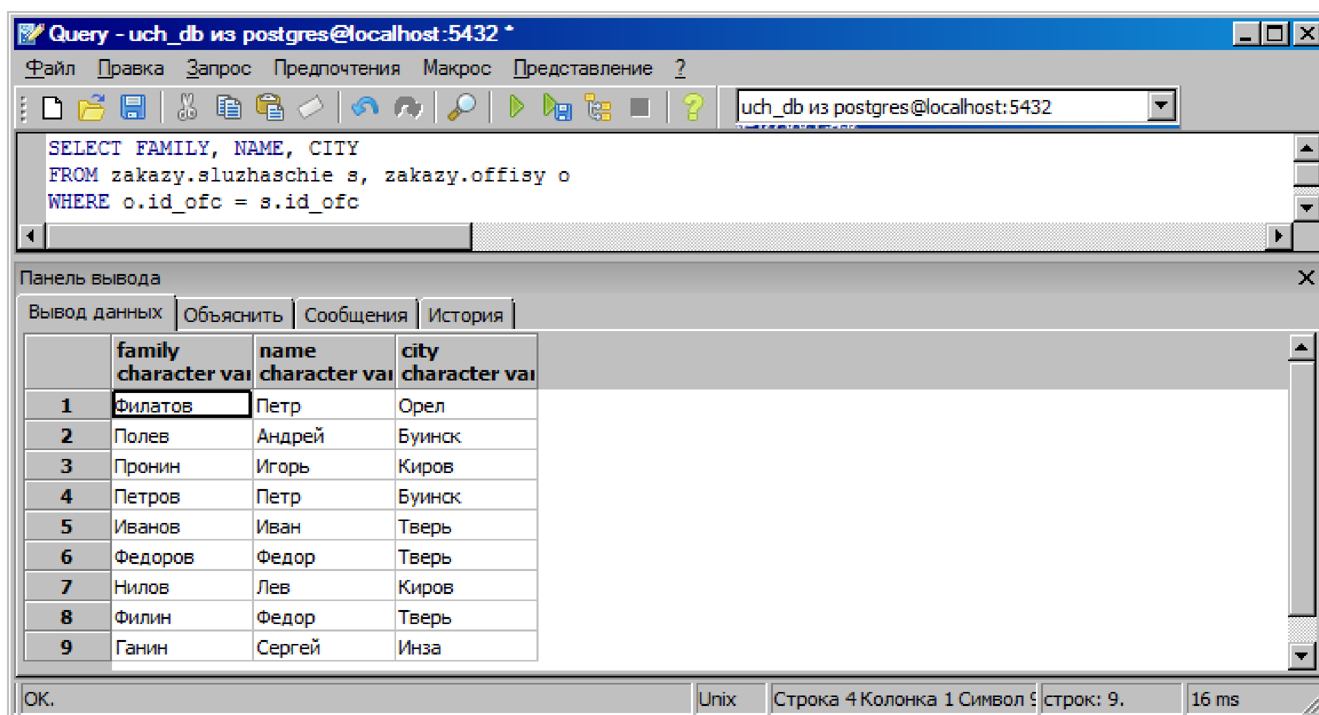
4.5.3. Внешнее объединение таблиц

Как уже было описано выше, операция объединения соединяет данные из двух таблиц, формируя пары связанных строк из этих таблиц. При внутреннем объединении все записи, для которых не находится пары в другой таблице, просто игнорируются. Поэтому если строка одной из таблиц не имеет пары, то такое объединение (внутреннее) может привести к неожиданным результатам.

Рассмотрим следующий пример. *Вывести список служащих и городов, где они работают:*

```
SELECT FAMILY, NAME, CITY
FROM zakazy.sluzhaschie s, zakazy.offisy o
WHERE o.id_ofc = s.id_ofc
```

Результат выполнения этого запроса приведен на Рис. 4.4. .



	family character var	name character var	city character var
1	Филатов	Петр	Орел
2	Полев	Андрей	Буинск
3	Пронин	Игорь	Киров
4	Петров	Петр	Буинск
5	Иванов	Иван	Тверь
6	Федоров	Федор	Тверь
7	Нилов	Лев	Киров
8	Филин	Федор	Тверь
9	Ганин	Сергей	Инза

Рис. 4.4. Результат выполнения запроса внутреннего объединения

Из приведенного результата видно, что в результирующую таблицу не вошли записи из таблицы offisy об офисе, расположенном в г. Омск, для которого еще не набраны служащие, и таблицы sluzhaschie об Уткине Денисе, который еще не получил назначение ни в один офис. Таким образом, если в таблицах объединения содержатся несвязанные (непарные) строки, то стандартный SQL приведет к потере информации.

Если мы хотим вывести сведения обо всех офисах, независимо от того, набраны в него служащие или еще нет, то правильный результат можно получить, выполнив запрос, приведенный ниже:

```
SELECT *
FROM zakazy.offisy o
LEFT JOIN zakazy.sluzhaschie s on o.id_ofc = s.id_ofc
```

Запрос, приведенный в этом примере, называется внешним (в данном случае левым) объединением таблиц. Результат выполнения запроса показан на Рис. 4.5. .

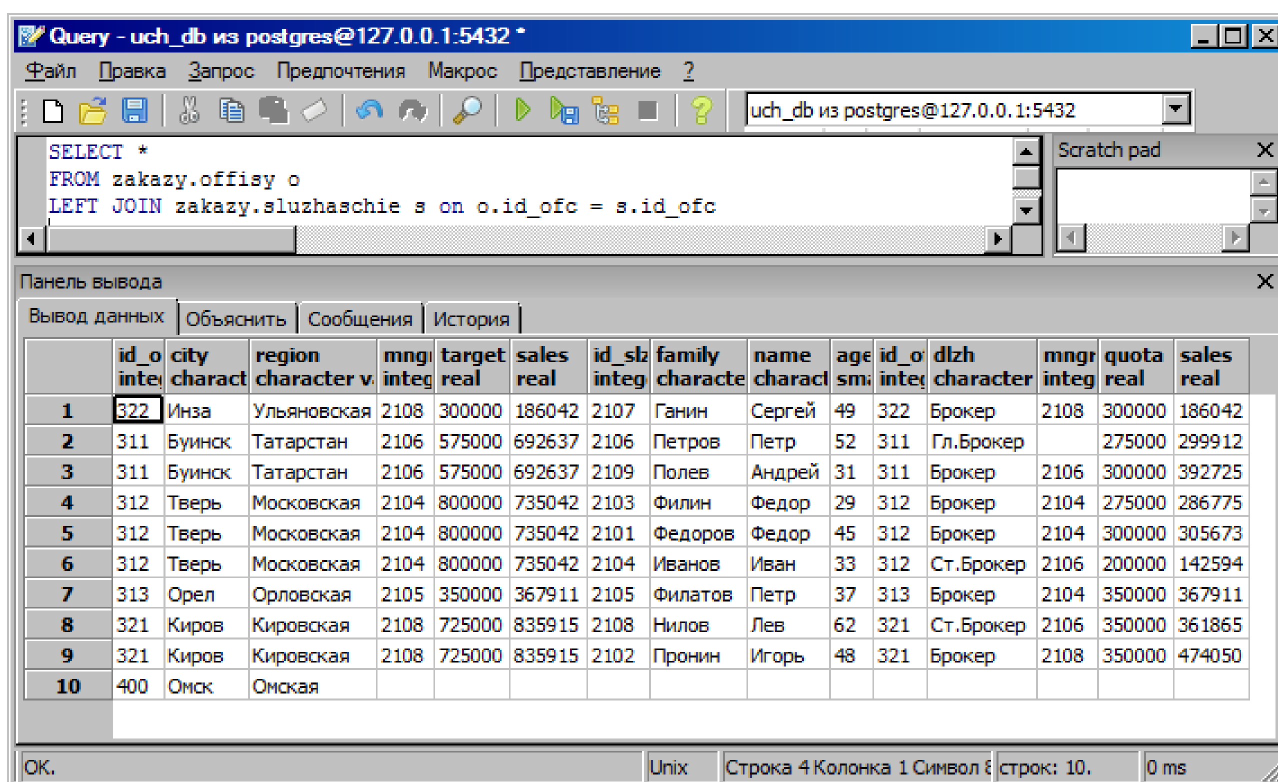
Как видно из приведенного примера, внешнее объединение может сохранить записи, для которых не находится соответствия в других наборах. При этом недостающие поля заполняются значениями NULL.

Правила выполнения внешних объединений

Можно привести следующее правило построения внешнего объединения.

1. Создать внутреннее объединение двух таблиц обычным способом.

2. Каждую строку первой таблицы, не имеющую связи ни с одной строкой второй таблицы, добавить в результаты запроса, присваивая столбцам второй таблицы значение NULL.



	id_o inte	city charact	region character v	mngı integ	target real	sales real	id_slı integ	family characte	name charact	age sma	id_o integ	dlzh character	mngr integ	quota real	sales real
1	322	Инза	Ульяновская	2108	300000	186042	2107	Ганин	Сергей	49	322	Брокер	2108	300000	186042
2	311	Буинск	Татарстан	2106	575000	692637	2106	Петров	Петр	52	311	Гл.Брокер		275000	299912
3	311	Буинск	Татарстан	2106	575000	692637	2109	Полев	Андрей	31	311	Брокер	2106	300000	392725
4	312	Тверь	Московская	2104	800000	735042	2103	Филин	Федор	29	312	Брокер	2104	275000	286775
5	312	Тверь	Московская	2104	800000	735042	2101	Федоров	Федор	45	312	Брокер	2104	300000	305673
6	312	Тверь	Московская	2104	800000	735042	2104	Иванов	Иван	33	312	Ст.Брокер	2106	200000	142594
7	313	Орел	Орловская	2105	350000	367911	2105	Филатов	Петр	37	313	Брокер	2104	350000	367911
8	321	Киров	Кировская	2108	725000	835915	2108	Нилов	Лев	62	321	Ст.Брокер	2106	350000	361865
9	321	Киров	Кировская	2108	725000	835915	2102	Пронин	Игорь	48	321	Брокер	2108	350000	474050
10	400	Омск	Омская												

Рис. 4.5. Результат выполнения запроса левого внешнего объединения

3. Каждую строку второй таблицы, которая не имеет связи ни с одной строкой первой таблицы, добавить в результаты запроса, присваивая столбцам первой таблицы значение NULL.

4. Результирующая таблица является внешним объединением двух таблиц.

Внешнее объединение, полученное при выполнении п.п. 1, 2, 3, называется полным внешним объединением. Оно симметрично по отношению к обеим таблицам. Однако существуют еще два типа внешних объединений, которые не симметричны относительно двух таблиц. Эти объединения называются левыми и правыми внешними объединениями.

На практике левые и правые объединения более полезны, чем полное объединение, особенно если таблицы связаны через первичный и вторичный ключи.

Левое внешнее объединение

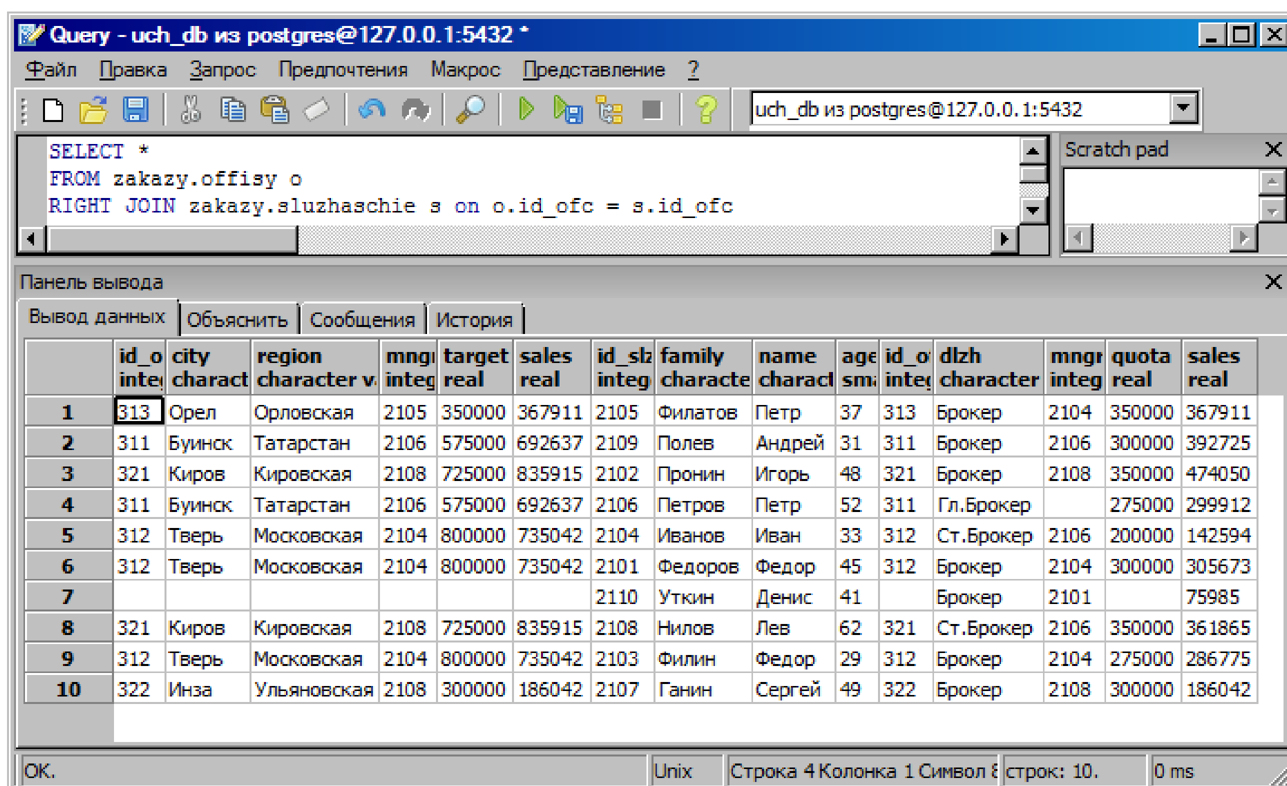
Левое внешнее объединение, результат выполнения которого приведен на Рис. 4.5. , получается при выполнении п.п. 1 и 2 из приведенного выше правила. Всегда содержит как минимум один экземпляр каждой записи из набора, указанного слева от ключевого слова JOIN. Отсутствующие поля из правого набора заполняются значениями NULL.

В приведенном примере столбец ID_OFC таблицы является внешним ключом таблицы offisy; он содержит номера офисов, в которых работают служащие и допускает наличие значений NULL, если для нового офиса еще не набраны служащие. В нашей таблице offisy такой офис есть – это офис, расположенный в г. Омск.

Правое внешнее объединение

Правое внешнее объединение получается при выполнении п.п. 1 и 3 из приведенного выше правила. Всегда содержит как минимум один экземпляр каждой записи из набора, указанного справа от ключевого слова `JOIN`. Допускает наличие значения `NULL`. Если новому служащему еще не был назначен офис. В нашей таблице `sluzhaschie` такой служащий есть – это `Уткин Денис`. Запрос для выполнения правого внешнего объединения, целью которого является отображение сведений о служащих, которым еще не назначены офисы, имеет вид, приведенный на Рис. 4.6. .

```
SELECT *
FROM zakazy.offisy o
RIGHT JOIN zakazy.sluzhaschie s on o.id_ofc = s.id_ofc
```



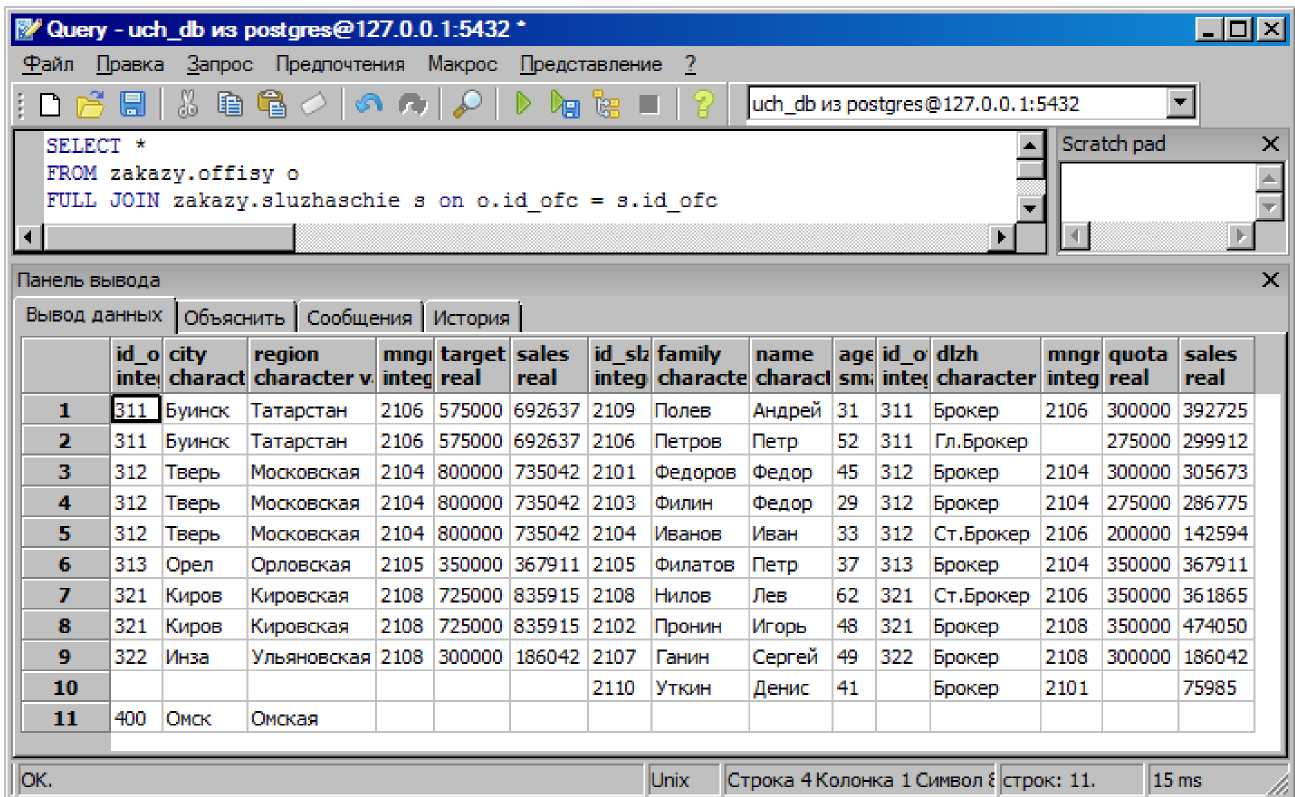
	id_o inte	city charact	region character v	mngr integ	target real	sales real	id_sl integ	family characte	name charact	age sm	id_o integ	dlzh character	mngr integ	quota real	sales real
1	313	Орел	Орловская	2105	350000	367911	2105	Филатов	Петр	37	313	Брокер	2104	350000	367911
2	311	Буинск	Татарстан	2106	575000	692637	2109	Полев	Андрей	31	311	Брокер	2106	300000	392725
3	321	Киров	Кировская	2108	725000	835915	2102	Пронин	Игорь	48	321	Брокер	2108	350000	474050
4	311	Буинск	Татарстан	2106	575000	692637	2106	Петров	Петр	52	311	Гл.Брокер		275000	299912
5	312	Тверь	Московская	2104	800000	735042	2104	Иванов	Иван	33	312	Ст.Брокер	2106	200000	142594
6	312	Тверь	Московская	2104	800000	735042	2101	Федоров	Федор	45	312	Брокер	2104	300000	305673
7							2110	Уткин	Денис	41		Брокер	2101		75985
8	321	Киров	Кировская	2108	725000	835915	2108	Нилов	Лев	62	321	Ст.Брокер	2106	350000	361865
9	312	Тверь	Московская	2104	800000	735042	2103	Филин	Федор	29	312	Брокер	2104	275000	286775
10	322	Инза	Ульяновская	2108	300000	186042	2107	Ганин	Сергей	49	322	Брокер	2108	300000	186042

Рис. 4.6. Результат выполнения запроса правого внешнего объединения

Полное внешнее объединение

Пример запроса, выполняющего полное внешнее объединение таблиц `offisy` и `sluzhaschie`, приведен ниже, а результат его выполнения – на Рис. 4.7. . Полное внешнее объединение всегда содержит как минимум один экземпляр каждой записи каждого объединяемого набора. Отсутствующие поля в записях нового набора заполняются значениями `NULL`.

```
SELECT *
FROM zakazy.offisy o
FULL JOIN zakazy.sluzhaschie s on o.id_ofc = s.id_ofc
```



	id_o inte	city charact	region character v	mngr integ	target real	sales real	id_sl integ	family characte	name charact	age sm	id_o integ	dlzh character	mngr integ	quota real	sales real
1	311	Буинск	Татарстан	2106	575000	692637	2109	Полев	Андрей	31	311	Брокер	2106	300000	392725
2	311	Буинск	Татарстан	2106	575000	692637	2106	Петров	Петр	52	311	Гл.Брокер		275000	299912
3	312	Тверь	Московская	2104	800000	735042	2101	Федоров	Федор	45	312	Брокер	2104	300000	305673
4	312	Тверь	Московская	2104	800000	735042	2103	Филин	Федор	29	312	Брокер	2104	275000	286775
5	312	Тверь	Московская	2104	800000	735042	2104	Иванов	Иван	33	312	Ст.Брокер	2106	200000	142594
6	313	Орел	Орловская	2105	350000	367911	2105	Филатов	Петр	37	313	Брокер	2104	350000	367911
7	321	Киров	Кировская	2108	725000	835915	2108	Нилов	Лев	62	321	Ст.Брокер	2106	350000	361865
8	321	Киров	Кировская	2108	725000	835915	2102	Пронин	Игорь	48	321	Брокер	2108	350000	474050
9	322	Инза	Ульяновская	2108	300000	186042	2107	Ганин	Сергей	49	322	Брокер	2108	300000	186042
10							2110	Уткин	Денис	41		Брокер	2101		75985
11	400	Омск	Омская												

Рис. 4.7. Результат выполнения запроса полного внешнего объединения

4.6. ОПЕРАТОРЫ ОБНОВЛЕНИЯ ДАННЫХ

4.6.1. ОПЕРАТОР INSERT

Добавление строки в реляционную таблицу происходит тогда, когда во «внешнем мире» появляется новый объект, представляемый этой строкой. На примере учебной базы данных это выглядит следующим образом:

– если вы принимаете на работу нового сотрудника, в таблицу СОТРУДНИКИ необходимо добавить новую строку с информацией о нем;

– если служащий заключает договор с новым клиентом, в таблицу CLIENTY должна быть добавлена новая строка, представляющая этого клиента;

– если клиент делает заказ, в таблицу ZAKAZY требуется добавить новую строку, содержащую сведения об этом заказе.

Во всех приведенных примерах новая строка добавляется для того, чтобы база данных оставалась точной копией реального мира. Наименьшей единицей информации, которую можно добавлять в реляционную базу данных, является одна строка.

Для добавления строки в реляционную таблицу применяется инструкция INSERT. В качестве объекта может выступить таблица базы данных или просмотр (VIEW), созданный оператором CREATE VIEW. В последнем случае записи могут добавляться сразу в несколько таблиц. Формат оператора INSERT

```
INSERT INTO <объект> [(столбец1 [, столбец2 ...])]
{VALUES (<значение1> [, <значение2> ...])|<оператор SELECT>}
```

Список столбцов указывает столбцы, которым будут присвоены значения в добавляемых записях. Список столбцов может быть опущен. В этом случае подразумеваются все столбцы объекта, причем в том порядке, в котором они определены в данном объекте.

Поставить в соответствие столбцам списки значений можно двумя способами. Первый состоит в явном указании значений после слова VALUES, второй – в формировании значений при помощи оператора SELECT.

Однострочная инструкция INSERT

Однострочная инструкция INSERT применяется для добавления одной записи в таблицу и имеет формат

```
INSERT INTO <объект> [(столбец1 [, столбец2 ...])]
VALUES (<значение1> [, <значение2> ...])
```

В предложении INTO указывается целевая таблица, в которую вставляется новая строка, а в предложении VALUES содержатся значения данных для новой строки. Значения присваиваются столбцам по порядку следования в операторе: первому по порядку столбцу присваивается первое значение, второму столбцу – второе значение и т. д.

Пример: *добавить информацию о новом служащем в таблицу* SLUZHASCHIE:

```
    INSERT INTO SLUZHASCHIE (ID_SLUZH, FAMILY, NAME, AGE, MNGR, ID_OFC,
QUOTA)
    VALUES (211, Аршавин, Андрей, 25, 106, 11, $3 000)
```

В приведенной инструкции данные о продажах нового служащего не вводятся, так как их у него еще нет. Поэтому при определении таблицы инструкцией CREATE TABLE, либо нужно определить для этого поля значение по умолчанию (ключевое слово DEFAULT), либо нужно допустить значение NULL (т. е. в определении этого поля не должно быть ключевого слова NOT NULL).

Если же столбцы таблицы SLUZHASCHIE указаны в полном составе и именно в том порядке, в котором перечислены при создании таблицы инструкцией CREATE TABLE, оператор можно упростить:

```
INSERT INTO SLUZHASCHIE
VALUES (211, Аршавин, Андрей, 25, 106, 11, $3 000, $0.00)
```

Для установки уникального значения поля первичного ключа ID_SLZH можно воспользоваться генератором:

```
INSERT INTO SLUZHASCHIE
VALUES (GEN_ID(ID_SLUZH, 2), Аршавин, Андрей, 25, 106, 11, $3 000, $0.00)
```

Многострочная инструкция INSERT

Многострочная инструкция INSERT добавляет в целевую таблицу несколько строк и имеет следующий формат:

```
INSERT INTO <объект> [(столбец1 [, столбец2 ...])]
<оператор SELECT>}
```

При этом значениями, которые присваиваются столбцам, являются значения, возвращаемые оператором SELECT. Порядок их назначения столбцам аналогичен предыдущей форме оператора INSERT: значение первого по порядку столбца результирующего набора присваивается первому столбцу оператора INSERT, второй – второму и т. д. Следует обратить внимание на важную особенность: поскольку оператор SELECT в общем случае возвращает множество записей, то и оператор INSERT в данной форме приведет к добавлению аналогичного числа новых записей.

Пример. *Скопировать старые заказы в таблицу* OLDORDERS:

```
INSERT INTO OLDORDERS (ID_ORDER, ORDER_DATE, PRICE)
   SELECT ID_ORDER, ORDER_DATE, PRICE
   FROM ZAKAZY
   WHERE ORDER_DATE < '01-JAN-2002'
```

4.6.2. Оператор UPDATE

Операция обновления данных UPDATE требуется тогда, когда происходят изменения во внешнем мире и их надо адекватно отразить в базе данных.

Инструкция обновления UPDATE имеет формат

```
UPDATE имя_табл
SET    имя_столбца = новое_значение
[WHERE условие_отбора]
```

В инструкции указывается целевая таблица, которая должна быть модифицирована, при этом пользователь должен иметь разрешение на обновление таблицы и каждого конкретного столбца. Предложение WHERE отбирает строки таблицы, подлежащие обновлению. В предложении SET указывается, какие столбцы должны быть обновлены, и для них задаются новые значения.

Пример. *Перевести всех служащих из Инзенского офиса (идентификатор* 22*) в Тверской офис (идентификатор* 12*) и понизить их личные планы (поле* QUOTA*) на* 10 *процентов*.

```
UPDATE SLUZHASCHIE
SET    ID_OFC = 12, QUOTA = 0.9*QUOTA
WHERE  ID_OFC = 22
```

Предложение WHERE в инструкции UPDATE является необязательным. Если оно опущено, то обновляются все строки целевой таблицы

Пример. *Увеличить планы для всех сотрудников на пять процентов*.

```
UPDATE SLUZHASCHIE
SET    QUOTA = 1.05*QUOTA
```

В инструкции UPDATE можно использовать подчиненные запросы, поскольку они дают возможность отбирать строки для обновления, опираясь на информацию из других таблиц.

Пример. *Увеличить на* $5 000 *лимит кредита для тех клиентов, которые сделали заказ на сумму более* $5 000.

```
UPDATE SLUZHASCHIE
```

```
SET    QUOTA = QUOTA + 5000.00
WHERE  ID_SLZH IN (SELECT DISTINCT ID_CLN
                   FROM ZAKAZY
                   WHERE PRICE > 25000)
```

4.6.3. Оператор DELETE

Оператор удаления данных позволяет удалить одну или несколько строк из таблицы в соответствии с условиями, которые задаются для удаленных строк.

Синтаксис оператора DELETE следующий

```
DELETE
FROM  имя_табл
[WHERE условия_отбора]
```

В предложении FROM указывается таблица, содержащая строки, которые требуется удалить. В предложении WHERE указывается критерий отбора строк, которые должны быть удалены. Если условия отбора не задаются, то из таблицы удаляются все строки, однако это не означает, что удаляется вся таблица. Исходная таблица остается, но она остается пустой, незаполненной. Условия отбора в части WHERE имеют тот же вид, что и условия фильтрации в операторе SELECT.

Предположим, что некий сотрудник решил уволиться из компании. Вот инструкция DELETE, удаляющая, относящуюся к данному сотруднику строку из таблицы СОТРУДНИКИ

```
DELETE
FROM  SLUZHASCHIE
WHERE FAMILY = ’Быков’ AND NAME = ‘Игорь’
```

В предложении WHERE может находиться встроенный запрос. Например, если нам надо удалить все заказы принятые уволенным сотрудником необходимо использовать условие отбора с встроенным запросом.

```
DELETE
FROM  ZAKAZY
WHERE ID_SLZH = (SELECT ID_SLUZH
                 FROM   SLUZHASCHIE
                 WHERE  FAMILY = ’Быков’ AND NAME = ‘Игорь’)
```

Встроенный запрос находит идентификатор сотрудника с фамилией Быков и именем Игорь, а затем предложение WHERE отбирает заказы с данным идентификатором. Как видно из этого примера, встроенные запросы в инструкции DELETE играют важную роль, поскольку они позволяют удалять строки, основываясь на информации, содержащейся в других таблицах.

ГЛАВА 5. DDS – СРЕДСТВА АДМИНИСТРИРОВАНИЯ БАЗ ДАННЫХ

Поддержание целостности данных предполагает их защиту от случайного или намеренного искажения. Поэтому доступ к отдельным данным должен быть ограничен по соображениям их конфиденциальности. Все это требует ограничения прав доступа к данным различных групп пользователей. предоставление соответствующих прав и их изъятие реализуется следующими командами.

GRANT – предоставляет права доступа к специфицированным объектам данных со стороны указанных пользователей или других объектов базы;

REVOKE – ликвидирует права доступа к специфицированным объектам данных со стороны указанных пользователей или других объектов базы.

5.1.НАЗНАЧЕНИЕ И ЛИКВИДАЦИЯ ПРАВ

5.1.1. Команда GRANT

Создание пользователя само по себе не дает ему никаких прав на доступ к объектам базы данных. Права доступа предоставляются командой GRANT.GRANT устанавливает права на объекты базы данных пользователям, ролям или другим объектам базы данных. Когда объект создается право на него имеет только его создатель и только он может выдавать права другим пользователям или объектам.

Для доступа к таблице пользователь или объект нуждается в правах на SELECT, INSERT, UPDATE, DELETE или REFERENCES. Для вызова процедуры в приложении пользователь должен иметь права на EXECUTE. Права могут быть даны всем пользователям опцией PUBLIC на месте списка пользователей.

Перечень прав приведен в нижеследующей таблице.

Права	Позволяет пользователям
ALL	SELECT, INSERT, UPDATE, DELETE, EXECUTE, REFERENCES
SELECT	Дает право выбирать строки из таблицы или просмотра
INSERT	Дает право добавлять строки в таблицы или просмотры
UPDATE	Дает право изменять строки в таблице или просмотре
DELETE	Дает право удалять строки из таблицы или просмотра
EXECUTE	Дает право выполнять хранимую процедуру
REFERENCES	Дает право ссылаться при работе с FK на специфицированные столбцы

Чтобы показать, насколько гибкой является система безопасности сервера БД, приведем синтаксис команды предоставления прав:

```
GRANT{
  <список прав>
  ON [TABLE] {имяТаблицы | имяПросмотра}
  TO {<объект> | <списокПользователей>} | <роль>
  TO {PUBLIC | <списокРолей>}};
```

Здесь

```
<права> = {ALL [PRIVILEGES] | <списокПрав>} где
<списокПрав> = {
   SELECT |
   INSERT |
   UPDATE [(столбец [,столбец ...])] |
   REFERENCES [(столбец [,столбец ...])]
[,<списокПрав> ...]}
```

Это права на выполнение либо отдельных операций SELECT, DELETE, INSERT, UPDATE, REFERENCES, либо их комбинаций, либо всех – ALL. предоставление права REFERENCES означает возможность создавать ссылочную целостность таблиц (ограничения внешнего ключа FOREIGN KEY).

Далее в команде GRANT под <объект> понимают:

```
<объект> =
   PROCEDURE имяПроцедуры |
   TRIGGER имяТриггера |
   VIEW имяПросмотра |
   PUBLIC |
 [,<объект> ...]
```

<списокПользователей> - это пользователи, которым предоставлены права на выполнение хранимых процедур, триггеров и просмотров.

<роль> – в системе безопасности InterBase-сервера пользователей можно объединять в группы. Например, на фирме есть служба, которая использует базы данных, но не может их удалять. То есть ее сотрудникам достаточно прав на выполнение операций SELECT, INSERT, UPDATE. В этом случае создаем роль SEL_INS_UP:

```
CREATE ROLE SEL_INS_UP
```

Предоставляем этой роли право на просмотр, вставку и изменение данных таблицы, например, Owner:

```
GRANT SELECT, INSERT, UPDATE ON Owner TO SEL_INS_UP
```

Назначаем эту роль каждому из сотрудников данной службы:

```
GRANT SEL_INS_UP TO <имяПользователя>
```

Использование роли позволяет, например, исключить возможность ошибки в случае зачисления в штат данной службы нового сотрудника. Как видите, для предоставления пользователю права доступа к таблице необходимо указать как минимум вид права доступа, имя таблицы и пользователя. Но не только адми-

нистратор имеет полномочия назначать права. Если вы создаете таблицу или хранимую процедуру, то имеете к ней права доступа наравне с администратором. Подробнее роли будут рассмотрены в разделе 5.3.

При назначении прав доступа на операцию можно уточнить, значения каких столбцов может изменять пользователь. Допустим, что пользователь Petrov имеет право изменять цену на предоставление услуги. Предположим, что цена задается в столбце COST таблицы SERVICE. Тогда операция назначения прав пользователю Petrov выглядит следующим образом:

```
GRANT SELECT, UPDATE (COST) ON SERVICE TO Petrov
```

5.1.2. Команда REVOKE

Ликвидация ранее выданных прав осуществляется командой REVOKE. Ее формат:

```
REVOKE <права> ON [TABLE] {имяТаблицы | имяПросмотра}
FROM {<объект> | <списокПользователей>}
```

Пример, лишить пользователя Petrov права вставки данных в таблицу Owner:

```
REVOKE INSERT ON Owner FROM Petrov
```

5.2.НАЗНАЧЕНИЕ ПРАВ ИСПОЛНЕНИЯ ХРАНИМЫХ ПРОЦЕДУР

Система безопасности InterBase-сервера позволяет решать проблему разрешения манипуляции данными не только на уровне прав пользователей, но и на уровне прав вызова хранимой процедуры. Другими словами, если пользователь не имеет прав, например, на вставку данных в таблицу, то такая операция все же будет произведена, если этим правом обладает исполняемая в данный момент хранимая процедура.

Синтаксис команды предоставления прав на уровне хранимой процедуры:

```
GRANT EXECUTE ON PROCRDURE ИмяПроцедуры TO
      {<объект> | <списокПользователей>}
```

Здесь под <объект> понимают:

```
PROCEDURE имяПроцедуры |
TRIGGER имяТриггера |
VIEW имяПросмотра |
PUBLIC
[, <объект> . . . ]
```

а <списокПользователей> – это

```
{[USER] имяПользователя | имяРоли}
[, <списокПользователей> . . .] [их права]
```

Например, следующая команда предоставляет право вызова процедуры MAX_VALUE пользователю Petrov и процедуре CALC_BALANCE:

```
GRANT EXECUTE ON PROCRDURE MAX_VALUE TO Petrov,
                 PROCRDURE CALC_BALANCE
```

5.3.СОЗДАНИЕ ГРУППЫ УПРАВЛЕНИЯ ПРАВАМИ – РОЛИ

Прежде всего, разберемся с понятием роли. Рассмотрим некоторое множество команд предоставления прав (GRANT) и дадим ему имя. В этом случае вместо перечисления команд GRANT можно, указав имя, сослаться на это множество. Такой подход позволяет одной командой передать пользователю права сразу на несколько объектов. Если пользователей много, то действия по отслеживанию их прав становятся достаточно громоздкими и работа с такими множествами существенно ее облегчает. Особенно удобно то, что вместо предоставления прав на какой-либо новый объект или ограничения ранее предоставленных прав каждому из пользователей, можно провести такое изменение в нашем поименованном множестве, которым и является роль.

Процедура работы с ролями включает несколько этапов:

– создание роли, т. е. объявление ее в базе (создание имени и пустого множества);

– формирование списка прав, связанных с ролью (включение элементов прав в множество);

– формирование прав пользователей на основе ролей;

– связывание пользователей с ролями, т. е. передача им множества прав, описанных в роли.

5.3.1. Команда CREATE ROLE

Команда CREATE ROLE реализует первый этап действий при работе с ролями: создает (объявляет) роль в базе данных. Синтаксис команды:

```
CREATE ROLE ИмяРоли;
```

Следующая команда создает роль, называемую «bibrole».

```
CREATE ROLE bibrole;
```

5.3.2. Команда DROP ROLE

Команда DROP ROLE выполняет действия, обратные CREATE ROLE – удаляет роль из базы данных. Синтаксис команды:

```
DROP ROLE ИмяРоли;
```

Роль может быть удалена либо ее создателем, либо пользователем SYSDBA, либо другим пользователем с аналогичными правами.

Следующая команда удаляет роль, называемую «bibrole».

```
DROP ROLE bibrole;
```

5.3.3. Формирование списка прав, связанных с ролью

Ролям, созданным командой CREATE ROLE, могут быть предоставлены права так же, как и пользователям. Предоставление прав ролям осуществляется командой GRANT.

Рассмотрим пример предоставления прав (привилегий) роли «bibrole» на выполнение процедуры PAUTHOR.

```
CREATE ROLE bibrole;
GRANT EXECUTE ON PROCEDURE PAUTHOR TO bibrole;
```

5.3.4. Формирование прав пользователей на основе ролей

Сами по себе роли носят вспомогательный характер. С базой данных работают пользователи, а не роли, следовательно, именно пользователям и должны передаваться права на работу с объектами базы данных. Передача прав пользователям на объекты базы данных, объявленных для ролей, осуществляется командой GRANT.

Рассмотрим пример, в котором пользователю передаются права, присвоенные ролям.

```
GRANT bibrole TO misha, masha, Ivan_Ivanovitch;
```

5.3.5. Связывание пользователей с ролями

В InterBase с пользователем во время его сеанса работы с базой может быть связана только одна роль. В то же время команд GRANT на передачу прав ролей может быть несколько. Такой механизм позволяет динамически связывать набор прав пользователя при его конкретном соединении. Это имеет смысл в тех случаях, когда один и тот же человек выступает в различном качестве. Например, сегодня он работает как кассир, а завтра как приемщик товаров.

Таким образом, связь между ролью и пользователем осуществляется не при выдаче команды GRANT, а при соединении пользователя с базой. Реализация такой связи осуществляется командой CONNECT. Базовый синтаксис команды CONNECT для нашего случая выглядит следующим образом:

```
CONNECT USER ‘username’
PASSWORD ‘password’
ROLE ‘rolename’;
```

Таким образом, один и тот же пользователь при входе в систему может получить различные наборы прав.

ГЛАВА 6. ИНФОРМАЦИОННЫЕ СИСТЕМЫ С АКТИВНЫМ СЕРВЕРОМ БАЗ ДАННЫХ

В архитектуре современных систем обработки данных и неуклонно возрастает роль серверных СУБД. Дореляционные СУБД отвечали главным образом за доступ к данным и их хранение, предоставляя приложениям возможность перемещаться по базе данных как им угодно, а также сортировать, отбирать и обрабатывать информацию. Однако с появлением реляционных СУБД и SQL ситуация коренным образом изменилась. Операции поиска и сортировки стали командами языка SQL, выполняемыми исключительно самой СУБД, и ею же производится вычисление итоговых данных. Теперь явная навигация по базе данных больше не нужна. Последующие расширения SQL, такие как первичные и внешние ключи, ограничения на значения, продолжают эту тенденцию, вытесняя функции проверки данных и обеспечения целостности базы данных из клиентских приложений. А конечная цель этой тенденции проста: чем больше ответственности на себя берет серверная СУБД, тем более эффективной и надежной становится система в целом благодаря централизованному управлению базой данных и снижению вероятности разрушения данных из-за ошибок в клиентских приложениях.

Общую тенденцию к расширению функций СУБД продолжают еще две важные возможности, которыми обладают практически все современные реляционные корпоративные СУБД: поддержка хранимых процедур и триггеров. *Хранимые процедуры* позволяют переносить часть прикладных функций, связанных с обработкой данных, в саму базу данных. Например, хранимая процедура может управлять приемом заказа или переводом денег с одного банковского счета на другой.

Триггеры служат для автоматического выполнения хранимых процедур при возникновении в базе данных определенных условий. Например, триггер может автоматически переводить деньги со сберегательного счета на чековый, когда остаток последнего исчерпывается.

В своей исходной форме SQL не является полноценным языком программирования. Он был задуман и создан как язык, предназначенный для выполнения операций над базами данных − создания их структуры, ввода и обновления данных − и особенно для выполнения запросов. SQL может использоваться как интерактивный командный язык: пользователь по очереди вводит инструкции

SQL с клавиатуры, а СУБД их выполняет. В этом случае последовательность операций над базой данных определяется пользователем. Инструкции SQL могут встраиваться в программы, написанные на других языках программирования, и тогда последовательность операций над базой данных определяется приложением.

С появлением хранимых процедур язык SQL обогатился рядом дополнительных базовых возможностей, обеспечиваемых практически всеми языками программирования, что позволило писать на «расширенном SQL» настоящие программы и процедуры.

6.1. ХРАНИМЫЕ ПРОЦЕДУРЫ ИЛИ ФУНКЦИИ

Хранимые процедуры (в некоторых СУБД функции) – это скомпилированный набор SQL-предложений, сохраненный в базе данных как именованный объект и выполняющийся как единый фрагмент кода. Хранимые процедуры могут принимать и возвращать параметры. Когда пользователь создает хранимую процедуру, сервер компилирует ее и помещает в разделяемый кэш, после чего скомпилированный код может быть применен несколькими пользователями. Когда приложение использует хранимую процедуру, оно передает ей параметры, если таковые ей потребуются, и сервер выполняет процедуру без перекомпиляции. Хранимая процедура позволяет повысить производительность приложений.

Во-первых, по сравнению с обычными SQL-запросами, посылаемыми из клиентского приложения, они требуют меньше времени для подготовки к выполнению, поскольку они скомпилированы и сохранены.

Во-вторых, сетевой трафик в этом случае меньше, чем в случае передачи SQL-запроса, т.к. по сети передается меньшее количество данных.

Хранимые процедуры автоматически перекомпилируются, если с объектами, на которые они влияют, произведены какие-либо изменения; иными словами, они всегда актуальны. Хранимые процедуры обычно используются для поддержки ссылочной целостности данных и реализации бизнес-правил. В последнем случае достигается дополнительная гибкость, поскольку если бизнес-правила изменяются, можно изменить только текст хранимой процедуры, не изменяя клиентские приложения.

В данном учебном пособии в качестве примера средства написания хранимых процедур будем рассматривать язык PL/pgSQL, который используется в СУБД PostgreSQL. Язык PL/pgSQL позволяет группировать на сервере код SQL и программные команды, что приводит к снижению затрат сетевых и коммуникационных ресурсов, обусловленных частыми запросами данных со стороны клиентских приложений и выполнением логической обработки этих данных на удаленных хостах.

В программах PL/pgSQL могут использоваться все типы данных, операторы и функции PostgreSQL. SQL в название PL/pgSQL указывает на то, что программист может напрямую использовать команды языка SQL в своих программах. Использование SQL в коде PL/pgSQL расширяет возможности, а также повышает гибкость и быстродействие программ. Несколько команд SQL в программном

блоке PL/pgSQL выполняются за одну операцию вместо обычной обработки каждой команды.

6.1.1. Структура языка

Язык PL/pgSQL имеет относительно простую структуру, что объясняется в основном тем, что каждый логически обособленный фрагмент кода существует в виде функции. Хотя на первый взгляд PL/pgSQL мало похож на другие языки программирования (такие, как язык C), сходство все же существует: логические фрагменты создаются и выполняются в виде функций, все переменные обязательно объявляются перед использованием, функции получают аргументы при вызове и возвращают некоторое значение в конце своей работы.

Регистр символов в именах функций PL/pgSQL не учитывается. В ключевых словах и идентификаторах допускается использование произвольных комбинаций символов верхнего и нижнего регистров. Также обратите внимание на частое удвоение апострофов во многих местах этой главы – всюду, где обычно используются одиночные апострофы. Удвоение экранирует апострофы в определениях функций, поскольку определение функции в действительности представляет собой большую строковую константу в команде CREATE FUNCTION.

В этом разделе рассматривается блочная структура программ PL/pgSQL, комментарии, структура выражений PL/pgSQL и использование команд.

Блоки

Программы PL/pgSQL состоят из блоков. Такой метод организации программного кода обычно называется блочной структурой.

Команда CREATE FUNCTION. Функция (или хранимая процедура) вводится с помощью команды SQL CREATE FUNCTION, которая используются для определения функций PL/pgSQL в базах данных PostgreSQL. Команда CREATE FUNCTION определяет имя функции, типы ее аргументов и возвращаемого значения.

Блок DECLARE. Основной блок функции начинается с секции объявлений. Все переменные объявляются (а также могут инициализироваться значениями по умолчанию) в секции объявлений программного блока. В объявлении указывается имя и тип переменной. Секция объявлений обозначается ключевым словом DECLARE, а каждое объявление завершается символом точки с запятой (;).

Основной программный блок. После объявления переменных следует ключевое слово BEGIN, обозначающее начало основного программного блока. За ключевым словом BEGIN находятся команды, входящие в блок. Конец программного блока обозначается ключевым словом END.

Возвращаемое значение. Основной блок функции PL/pgSQL должен вернуть значение заданного типа, а все вложенные блоки (блоки, начинающиеся внутри других блоков) должны быть завершены до достижения ключевого слова END.

Структура программного блока PL/pgSQL приведена в листинге 6.1.

Листинг 6.1. Структура программного блока PL/pgSQL

```
CREATE FUNCTION идентификатор (аргументы) RETURNS тип AS '
DECLARE
объявление: [...]
```

```
BEGIN команда: [...]
    . . .
END;
' LANGUAGE 'plpgsql':
```

Программный блок PL/pgSQL может содержать неограниченное количество вложенных блоков, которые читаются и интерпретируются по тем же правилам, что и обычные блоки. В свою очередь, они могут содержать свои вложенные блоки.

Вложенные блоки упрощают структуру кода в больших функциях PL/pgSQL. Структура вложенных блоков не отличается от структуры обычных блоков: они также начинаются с ключевого слова DECLARE, за которым следует ключевое слово BEGIN и последовательность команд, а затем ключевое слово END.

Комментарии

В PL/pgSQL поддерживаются два вида комментариев, у которых имеются аналоги в других языках программирования: однострочные и блочные (многострочные) комментарии.

Однострочные комментарии начинаются с двух дефисов (--) и не имеют специального завершителя. Модуль лексического разбора интерпретирует все символы, следующие после двух дефисов, как часть комментария. Пример использования однострочных комментариев приведен в листинге 6.2.

Листинг 6.2. Однострочный комментарий

```
-- Это будет интерпретировано как однострочный комметарий.
```

Блочные комментарии знакомы каждому, кто когда-либо программировал на других языках. Блочный комментарий начинается с последовательности символов /* и завершается последовательностью */. Они могут распространяться на несколько строк, при этом весь текст между начальной и завершающей парой /* и */ считается комментарием. Пример блочного комментария приведен в листинге 6.3.

Листинг 6.3. Блочный комментарий

```
/*
* Здесь размещен
* блочный комментарий. */
```

ПРИМЕЧАНИЕ Хотя блочные комментарии могут содержать вложенные однострочные комментарии, вложение блочных комментариев в другие блочные комментарии не допускается.

6.1.2. Команды и выражения

Программы PL/pgSQL, как и в большинстве языков программирования, состоят из команд и выражений. Вероятно, вам довольно часто придется пользоваться выражениями, потому что они крайне важны для некоторых типов манипуляций с данными. Общие концепции команд и выражений одинаковы (или, по крайней мере, очень похожи) во всех языках. Если вы прежде работали с другими языками программирования, то наверняка знакомы с этими концепциями.

Команды

Команда выполняет некоторое действие в коде PL/pgSQL – например, присваивает значение переменной или выполняет запрос. Последовательность команд в программных блоках PL/pgSQL определяет порядок выполнения действий в этом блоке. Большая часть команд обычно размещается в основной части блока, находящейся между ключевыми словами BEGIN и END. Некоторые команды также могут присутствовать в секции объявлений (после ключевого слова DECLARE), но они всего лишь объявляют и/или инициализируют переменные, используемые в программном блоке.

Каждая команда завершается символом точки с запятой (;). В этом прослеживается сходство с языком SQL, в котором команды завершаются этим же символом. Почти вся оставшаяся часть этой главы посвящена типам команд, их использованию и основным задачам, решаемым при помощи команд в PL/pgSQL.

Выражения

Выражения представляют собой условную запись последовательности операций, результат которой принадлежит одному из базовых типов данных PostgreSQL. В листинге 6.4 приведена простая функция PL/pgSQL, возвращающая результат простого выражения, а в листинге 6.5 продемонстрирован результат вызова этой функции в psql.

Листинг 6.4. Использование выражений

```
CREATE FUNCTION a_function () RETURNS int4 AS '
DECLARE
  an_integer int4;
BEGIN
  an_integer := 10 * 10;
  return an_integer;
END;
' LANGUAGE 'plpgsql';
```

Листинг 6.5. Результат вызова функции a_function()

```
booktown=# SELECT a_function() AS output;
output
100
(1 row)
```

6.1.3. Переменные

Переменные используются в программах PL/pgSQL для хранения изменяемых данных заранее определенного типа.

Типы данных

Переменные PL/pgSQL могут относиться к любому из стандартных типов данных SQL (например, integer или char). Помимо типов данных SQL, в PL/pgSQL также предусмотрен дополнительный тип RECORD, предназначенный для хранения записей без указания полей – эта информация передается при сохранении данных в переменной. Дополнительная информация о типе данных RECORD приводится ни-

же. Типы данных SQL были описаны в разделе 3.1.5. Самые распространенные типы PL/pgSQL: Boolean, text, char, integer, double precision, date, time.

Объявление переменных

Все переменные программного блока должны быть предварительно объявлены с ключевым словом DECLARE. Если переменная не инициализируется при объявлении, по умолчанию ей присваивается псевдозначение SQL NULL.

ПРИМЕЧАНИЕ. Как будет показано в разделе «Передача управления», в одной из команд – цикле FOR – предусмотрена возможность инициализации управляющей переменной. Переменную цикла FOR не нужно заранее объявлять в секции DECLARE того блока, в котором находится цикл. Таким образом, переменные цикла FOR составляют единственное исключение из правила, согласно которому все переменные должны объявляться в начале соответствующего блока.

Переменные, объявленные в блоке, доступны во всех его вложенных блоках, но обратное неверно: переменные, объявленные во вложенном блоке, уничтожаются в конце этого блока и недоступны во внешнем блоке. Синтаксис объявления переменной приведен ниже.

```
имя_переменной тип_данных [ := значение ];
```

Таким образом, объявление состоит из имени и типа переменной (следующих именно в этом порядке) и завершается символом точки с запятой (;).

В листинге 6.6 приведены объявления переменных типов integer, varchar (число в круглых скобках обозначает максимальную длину строки в символах) и float.

Листинг 6.6. Объявление переменных

```
CREATE FUNCTION identifier (arguments) RETURNS type AS '
DECLARE
  -- Объявить числовую переменную типа integer.
  subject_id integer;
  -- Объявить строковую переменную переменной длины.
  book_title varchar(10);
  -- Объявить вещественную числовую переменную.
  book price float;
BEGIN
  команды;
END;
' LANGUAGE 'plpgsql';
```

Объявление переменной также может содержать дополнительные модификаторы. Ключевое слово CONSTANT указывает на то, что вместо переменной определяется константа. Пример определения константы рассматриваются в листинге 6.7 в этом разделе.

Ключевые слова NOT NULL означают, что переменной не может присваиваться псевдозначение NULL. Если переменной, объявленной с модификатором NOT NULL, в программном блоке присваивается псевдозначение NULL, происходит ошибка времени выполнения. Поскольку при объявлении без инициализации всем переменным автоматически присваивается псевдозиачение NULL, переменные с модификатором NOT NULL обязательно должны инициализироваться.

Ключевое слово DEFAULT определяет значение по умолчанию для переменной. Вместо него можно воспользоваться оператором (:=), эффект будет тем же.

Ниже приведен расширенный синтаксис объявления переменной:

```
имя_переменной [ CONSTANT ] тип_данных [ NOT NULL ]
[ { DEFAULT | := } значение ];
```

В листинге 6.7 приведены примеры объявлений целочисленной константы, равной 5, переменной со значением 10, которой не может быть присвоено псевдозначение NULL, и символьной переменной, содержащей символ «a».

Листинг 6.7. Объявления переменных

```
CREATE FUNCTION example_function O RETURNS texi AS '
DECLARE
  -- Объявление целочисленной константы,
  -- инициализированной значением 5.
  five CONSTANT integer := 5;
  -- Объявление целочисленной переменной,
  -- инициализированной значением 10.
  -- Переменной не может присваиваться NULL.
  ten integer NOT NULL := 10;
  -- Объявление символьной переменной,
  -- инициализированной значением "a".
  letter char DEFAULT ''a'';
BEGIN
  -- Функция возвращает символ и прекращает работу.
  return letter;
END;
' LANGUAGE 'plpgsql':
```

Присваивание

Присваивание в PL/pgSQL выполняется оператором присваивания (:=) в форме левая_ переменная := правая_переменная.

Команда присваивает левой переменной значение правой переменной. Также допускается запись вида левая_переменная := выражение.

В этом случае левой переменной присваивается результат выражения, расположенного справа от оператора присваивания.

Значения по умолчанию также могут присваиваться переменным в секции объявлений программных блоков PL/pgSQL. Инициализация переменной производится оператором присваивания (:=) в одной строке с объявлением переменной. Эта тема подробно рассматривается ниже, а в листинге 6.8 приведен небольшой пример.

Листинг 6.8. Инициализация переменной

```
CREATE FUNCTION идентификатор (аргументы) RETURNS тип AS '
DECLARE
  a_integer int4 := 10;
BEGIN
  команда;
END;
' LANGUAGE 'plpgsql';
```

Возможен и другой вариант – присваивание переменной результата запроса командой SELECT INTO. Не путайте этот вариант использования команды SELECT INTO с командой SQL SELECT INTO, которая заносит результаты запроса в новую таблицу.

Команда SELECT INTO в основном требуется для сохранения данных записей в переменных, объявленных с типами ROWTYPE и RECORD. Чтобы команда SELECT INTO могла использоваться с обычной переменной, тип этой переменной должен соответствовать типу поля, упоминаемому в команде SQL SELECT.

Синтаксис команды SELECT INTO:

```
CREATE FUNCTION идентификатор (аргументы) RETURNS тип AS '
DECLARE
  команда;
BEGIN
  SELECT INTO переменная [. ...] поле [, ...] секции_select;
END;
' LANGUAGE 'plpgsql':
```

В этом описании переменная – имя переменной, участвующей в присваивании, а секции select – любые поддерживаемые секции команды SQL SELECT, обычно следующие за списком целевых полей в команде SELECT.

В листинге 6.9 приведена простая функция, в которой используется команда SELECT INTO. Ключевое слово ALIAS описано в подразделе 0 этого раздела. Примеры выполнения команды SELECT INTO для переменных типа RECORD и ROWTYPE приведены в разделе 6.1.7.

Листинг 6.9. Использование команды SELECT INTO

```
CREATE FUNCTION get_customer_id (text.text) RETURNS integer AS '
DECLARE
  -- Объявление псевдонимов для аргументов.
  l_name ALIAS FOR $1;
  f_name ALIAS FOR $2;
  -- Объявление переменной для хранения кода клиента.
  customer_id integer;
BEGIN
  -- Получение кода клиента, имя и фамилия которого
  -- совпадают с переданными значениями.
  SELECT INTO customer_id id FROM customers
  WHERE last_name = l_name AND first_name = f_name;
  -- Вернуть код.
  RETURN customer_id;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.10 показан результат вызова функции get_customer_id() с аргументами Jackson и Annie. Возвращенное число равно коду клиента «Annie Jackson» в таблице customers.

Листинг 6.10. Вызов функции get_customer_id()

```
booktown=# SELECT get_customer_id ('Jackson','Annie');
get_customer_id
107
(1 row)
```

Если требуется присвоить несколько значений нескольким переменным, в команду включаются две группы, разделенные запятыми и отделенные друг от друга пробелом. В первой группе перечисляются имена переменных, а во второй – имена полей.

Функция, приведенная в листинге 6.11, решает обратную задачу по сравнению с функцией `get_customer_id()` из листинга 6.9 – она возвращает имя и фамилию клиента по заданному коду.

Листинг 6.11. *Использование команды SELECT INTO с несколькими полями*

```
CREATE FUNCTION get_customer_name (integer) RETURNS text AS '
DECLARE
  -- Объявление псевдонимов для аргументов,
  customer_id ALIAS FOR $1;
  -- Объявление переменных для хранения компонентов
  -- полного имени клиента.
  customer_fname text;
  customer_lname text;
BEGIN
  -- Получение имени и фамилии клиента, код которого
  -- совпадает с переданным значением.
  SELECT INTO customer_fname, customer_lname
  first_name, last_name
  FROM customers WHERE id = customer_id;
  -- Вернуть полное имя.
  RETURN customer_fname | '' '' | customer_lname;
END;
' LANGUAGE 'plpgsql1;
```

В листинге 6.12 показан результат вызова функции `get_customer_name()` с аргументом `107`.

Листинг 6.12. Вызов функции get_customer_name()

```
booktown=# SELECT get_customer_name(107);
get_customer_name
Annie Jackson
(1 row)
```

Чтобы узнать, успешно ли были присвоены значения переменным командой `SELECT INTO`, воспользуйтесь специальной логической переменной `FOUND`. Кроме того, можно проверить значение заданной переменной ключевыми словами `ISNULL` или `IS NULL` (в большинстве случаев положительный результат означает, что команда `SELECT INTO` завершилась неудачно!).

Ключевые слова `FOUND`, `IS NULL` и `ISNULL` следует использовать в условных командах (`IF`/`THEN`). Условные команды `PL/pgSQL` описаны в разделе 6.1.7.

В листинге 6.13 приведен простейший пример использования логической переменной `FOUND` в функции `get_customer_id()`.

Листинг 6.13. Использование логической переменной FOUND в функции get_customer_id()

```
[...]
  SELECT INTO customer_id id FROM customers
```

```
  WHERE last_name = l_name AND first_name = f_name;
  -- Если совпадение не найдено, вернуть -1.
  -- Другая функция, в которой вызывается
  get_customer_id();
  -- может интерпретировать -1 как признак ошибки.
  IF NOT FOUND THEN
     return -1;
  END IF;
[...]
```

В листинге 6.14 показано, что теперь функция `get_customer_id()` при передаче имени несуществующего клиента возвращает `-1`.

Листинг 6.14. Вызов нового варианта функции get_customer_id()

```
booktown=# SELECT get_customer_id('Schmoe','Joe');
get_customer_id
-1
(1 row)
```

Аргументы

При вызове функции `PL/pgSQL` могут получать аргументы различных типов. В аргументах пользователь передает исходные данные, необходимые для работы функции. Аргументы делают функции `PL/pgSQL` более универсальными и значительно расширяют область их возможного применения. Список аргументов приводится после имени функции в круглых скобках и разделяется запятыми.

Количество и типы аргументов должны соответствовать первоначальному определению функции.

В листинге 6.15 приведены примеры двух вызовов функции из клиента `psql`.

Листинг 6.15. Примеры вызовов функций

```
booktown=# SELECT get_author('John');
get_author
John Worsley
(1 row)

booktown=# SELECT
get_author(1111);
get_author
Ariel Denham
(1 row)
```

ПРИМЕЧАНИЕ. Функции get_author(text) и get_author(integer) будут рассмотрены позднее в этой главе.

Аргументы, полученные функцией, поочередно присваиваются идентификаторам, состоящим из знака доллара (`$`) и порядкового номера. Первому аргументу соответствует идентификатор `$1`, второму -- `$2` и т. д.

Максимальное количество аргументов равно `16`, поэтому идентификаторы аргументов лежат в интервале от `$1` до `$16`. В листинге 6.16 приведен пример функции, которая удваивает свой целочисленный аргумент.

Листинг 6.16. Непосредственное использование аргументов в переменных

```
CREATE FUNCTION double_price (float) RETURNS float AS '
DECLARE
```

```
BEGIN
  -- Вернуть значение аргумента, умноженное на 2.
  return $1 * 2;
END;
' LANGUAGE 'plpgsql';
```

Если функция имеет большое количество аргументов, в обозначениях вида «$+номер» легко запутаться. Чтобы программисту было проще отличить одни аргумент от другого (или если он хочет присвоить переменной аргумента более содержательное имя), в PL/pgSQL предусмотрена возможность определения псевдонимов переменных.

Псевдоним создается при помощи ключевого слова ALIAS и представляет собой альтернативный идентификатор для ссылки на аргумент. Перед использованием все псевдонимы (как и обычные переменные) должны быть объявлены в секции объявлений блока. В листинге 6.17 показан синтаксис применения ключевого слова ALIAS.

Листинг 6.17. Синтаксис использования ключевого слова ALIAS

```
CREATE FUNCTION функция (аргументы) RETURNS тип AS '
DECLARE
  идентификатор ALIAS FOR $1;
  идентификатор ALIAS FOR $2;
BEGIN
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.18 приведен простой пример, демонстрирующий применение псевдонимов в функциях PL/pgSQL. Функция triple_price получает вещественное число, умножает его на три и возвращает результат.

Листинг 6.18. Псевдонимы PL/pgSQL

```
CREATE FUNCTION triple_price (float) RETURNS float AS '
DECLARE
  -- Переменная input_price объявляется как псевдоним
  -- для переменной аргумента, обычно обозначаемой
  -- идентификатором $1.
  input_price ALIAS FOR $1;
BEGIN
  -- Вернуть аргумент, умноженный на три.
  RETURN input_price * 3;
END;
' LANGUAGE 'plpgsql';
```

Если теперь вызвать функцию triple_price при выполнении команды SQL SELECT в клиенте psql, будет получен результат, показанный в листинге 6.19.

Листинг 6.19. Результат вызова функции triple_price()

```
booktown=# SELECT triple_price(12.50);
triple_price
37.5
(1 row)
```

6.1.4. Возвращение переменных

Тип величины, возвращаемой функцией PL/pgSQL, должен соответствовать типу возвращаемого значения, указанному при создании функции командой CREATE FUNCTION. Значение возвращается командой RETURN. Команда RETURN находится в конце функции, но она также часто встречается в командах IF или других командах, осуществляющих передачу управления в программе. Даже если команда RETURN вызывается в одной из этих команд, функция вес равно должна заканчиваться командой RETURN (даже если управление никогда не будет передано этой завершающей команде).

Синтаксис команды RETURN приведен в листинге 6.20.

Листинг 6.20. Синтаксис команды RETURN

```
CREATE FUNCTION функция (аргументы) RETURNS тип AS '
DECLARE
  объявление;
  [...]
BEGIN
  команда;
  [...]
  RETURN { переменная \ значение }
END;
' LANGUAGE 'plpgsql' ;
```

Пример использования команды RETURN можно найти в любой функции PL/pgSQL, встречающейся в этой главе.

6.1.5. Атрибуты

Для упрощения работы с объектами базы данных в PL/pgSQL существуют атрибуты переменных – %TYPE и %R0WTYPE. Атрибуты требуются для объявления переменной, тип которой совпадает с типом объекта базы данных (атрибут %TYPE) или структурой записи (атрибут %ROWTYPE). Переменные объявляются с атрибутами в том случае, если они будут использоваться в программном блоке для хранения значений, полученных от объекта базы данных. Таким образом, при объявлении переменной с атрибутом знать тип объекта базы данных не обязательно. Если в будущем тип изменится, то переменная также автоматически переключится на новый тип данных, причем это не потребует дополнительных усилий со стороны программиста.

Атрибут %TYPE

Атрибут %TYPE используется при объявлении переменных с типом данных, совпадающих с типом некоторого объекта базы данных (чаще всего поля). Синтаксис объявления переменной с атрибутом %TYPE приведен в листинге 6.21.

Листинг 6.21. Объявление переменной с атрибутом %TYPE

```
переменная таблица.поле%TYPE
```

В листинге 6.22 приведена функция, использующая атрибут %TYPE для хранения фамилии автора. В ней задействован оператор конкатенации (||), описанный ниже. Команда SELECT INTO рассматривалась ранее в этой главе.

В листинге 6.22 следует обратить особое внимание на атрибут %TYPE. Фактически мы объявляем переменную, тип которой совпадает с типом поля таблицы authors. Затем команда SELECT находит запись, у которой поле first_name совпадает с аргументом, переданным при вызове функции. Команда SELECT читает значение поля last_name этой записи и сохраняет его в переменной l_name. Пример вызова функции с передачей аргумента приведен ниже, в листинге 6.23. Кроме того, передача аргумента пользователем встречается во многих примерах этой главы.

Листинг 6.22. Использование атрибута %TYPE

```
CREATE FUNCTION get_author (text) RETURNS text AS '
DECLARE
  -- Объявление псевдонима для аргумента функции.
  -- в котором должно передаваться имя автора,
  f_name ALIAS FOR $1;
  -- Объявление переменной, тип которой совпадает
  -- с типом поля last_name таблицы authors.
  l_name authors.last_name l TYPE;
BEGIN
  -- Найти в таблице authors фамилию автора.
  -- имя которого совпадает с переданным аргументом.
  -- и присвоить ее переменной l_name.
  SELECT INTO l_name last_name
  FROM authors
  WHERE first_name = f_name;
  - Вернуть имя и фамилию, разделенные пробелом,
  return f_name || '' '' || l_name;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.23 приведен пример вызова функции get_author().

Листинг 6.23. Результат вызова функции get_author()

```
booktown=# SELECT get_author('Andrew');
get_author
Andrew Brook Ins
(1 row)
```

Атрибут %ROWTYPE

Атрибут %ROWTYPE используется в PL/pgSQL для переменной-записи, имеющей одинаковую структуру с записями заданной таблицы. Не путайте атрибут %ROWTYPE с типом данных RECORD – переменная с атрибутом ROWTYPE точно воспроизводит структуру записи конкретной таблицы, а переменная RECORD не структурирована и ей можно присвоить запись любой таблицы.

В листинге 6.24 приведена перегруженная версия функции get_author() (см. листинг 6.22). Она делает то же, что и прототип, но получает аргумент типа integer вместо text и ищет автора, сравнивая код с переданным аргументом.

Обратите внимание: в реализации функции используется переменная, объявленная с атрибутом %ROWTYPE. Возможно, в данном случае применение %ROW-

TYPE только напрасно усложняет очень простую задачу, но по мере изучения PL/pgSQL важность атрибута %ROWTYPE становится все более очевидной.

Точка (.) после имени переменной found_author в листинге 6.24 используется для ссылки на имя поля, входящего в структуру found_author.

Листинг 6.24. Использование атрибута %ROWTYPE

```
CREATE FUNCTION get_author (integer) RETURNS text AS '
DECLARE
  -- Объявление псевдонима для аргумента функции.
  -- в котором должен передаваться код автора,
  author_id ALIAS FOR $1;
  -- Объявление переменной, структура которой
  -- совпадает со структурой таблицы authors,
  found author authors ROWTYPE;
BEGIN
  -- Найти в таблице authors фамилию автора,
  -- код которого совпадает с переданным аргументом.
  SELECT INTO found_author * FROM authors WHERE id = author_id;
  -- Вернуть имя и фамилию, разделенные пробелом.
  RETURN found_author.first_name || " " || found_author.last_name;
END;
' LANGUAGE 'plpgsql':
```

Обратите внимание на звездочку (*) в списке полей команды SELECT. Поскольку переменная found_author объявлялась с атрибутом %ROWTYPE для таблицы authors, она имеет такую же структуру, как и записи таблицы authors. Таким образом, конструкция SELECT * заполняет переменную found_author значениями полей найденной записи. Пример вызова новой версии get_author() приведен в листинге 6.25.

Листинг 6.25. Вызов новой версии функции get_author()

```
booktown=# SELECT get_author(1212);
get_author
John Worsley
(1 row)
```

6.1.6. Конкатенация

Конкатенацией называется процесс построения новой строки посредством объединения двух (и более) строк. Конкатенация принадлежит к числу стандартных операций PostgreSQL и поэтому может напрямую использоваться с переменными в функциях PL/pgSQL. Это незаменимый инструмент форматирования при работе с несколькими переменными, содержащими символьные данные.

Конкатенация используется только со строками. Оператор конкатенации (|) ставится между объединяемыми компонентами (литералами или строковыми переменными).

В листинге 6.26 операция конкатенации создает строку, возвращаемую функцией.

Листинг 6.26. Возвращение результата конкатенации

```
CREATE FUNCTION compound_word (text, text) RETURNS text AS '
DECLARE
  -- Объявление псевдонимов для аргументов функций.
  word1 ALIAS FOR $1;
  word2 ALIAS FOR $2;
BEGIN
  -- Вернуть объединение двух слов.
  RETURN word1 || word2;
END;
' LANGUAGE 'plpgsql' ;
```

Если передать функции аргументы «break» и «fast», функция вернет объединенную строку «breakfast»:

```
booktown=# SELECT compound_word ('break', 'fast');
compound_word
breakfast
(1 row)
```

6.1.7. Передача управления

Команды передачи управления существуют практически во всех современных языках программирования, и PL/pgSQL не является исключением. С технической точки зрения сам вызов функции можно рассматривать как передачу управления последовательности команд PL/pgSQL. Тем не менее существуют и другие, более совершенные средства, определяющие последовательность выполнения команд PL/pgSQL. Речь идет об условных командах IF/THEN и циклах.

Условные команды

Условная команда указывает на то, что некоторое действие (или последовательность действий) выполняется в зависимости от результатов проверки заданного логического условия. Определение выглядит запутанно, но на самом деле условные команды весьма просты. В неформальной формулировке условная команда означает следующее: «если условие истинно, выполнить такое-то действие».

Команда IF/THEN

В соответствии с данным определением команда IF/THEN задает команду (или блок команд), выполняемых в случае истинности некоторого условия. Синтаксис команды IF/THEN показан в листинге 6.27.

Листинг 6.27. Синтаксис команды IF/THEN

```
CREATE FUNCTION функция (аргументы) RETURNS тип AS '
DECLARE
  объявления
BEGIN
  IF условие THEN
     команда;
     [...]
  END IF;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.28 приведена функция, которая проверяет количество экземпляров книги на складе по коду и номеру издания. Код книги используется во внутрениих операциях базы данных и присутствует в нескольких таблицах.

Следовательно, функция `stock_amount()` предназначена для вызова из других функций, а не прямого вызова из клиентской программы, поскольку большинству пользователей коды книг неизвестны.

Сначала мы читаем код `ISBN` командой `SELECT INTO`. Если команда `SELECT INTO` не смогла найти код `ISBN` по заданному коду книги и номеру издания, функция `stock_amount()` возвращает `-1`.

Функция, вызвавшая `stock_amount()`, интерпретирует это значение как признак ошибки. Если код `ISBN` найден, то другая команда `SELECT INTO` получает количество экземпляров книги на складе и возвращает полученную величину. На этом работа функции завершается.

Листинг 6.28. Использование команды IF/THEN

```
CREATE FUNCTION stock_amount (integer, integer) RETURNS integer AS '
DECLARE
  -- Объявление псевдонимов для аргументов функции.
  b_id ALIAS FOR $1;
  b_edition ALIAS FOR $2;
  -- Объявление переменной для кода ISBN.
  b_isbn text;
  -- Объявление переменной для количества экземпляров.
  stock_amount integer;
BEGIN
  -- Команда SELECT INTO находит в таблице editions запись,
  -- у которой код книги и номер издания совпадают с аргументами
  -- функции. Код ISBN найденной записи присваивается переменной.
  SELECT INTO b_isbn isbn FROM editions WHERE
  book_id = b_id AND edition = b_edition;
  -- Проверить, не был ли полученный код ISBN равен NULL.
  -- Значение NULL говорит о том, что в базе данных
  -- не существует записи книги с кодом и номером издания,
  -- переданными в аргументах функции. Если запись не существует,
  -- функция возвращает -1 и завершает работу.
  IF b_isbn IS NULL THEN
     RETURN -1;
  END IF;
  -- Получить из таблицы stock количество экземпляров книги
  -- на складе и присвоить его переменной stock_amount.
  SELECT INTO stock_amount stock FROM stock WHERE isbn = b_isbn;
  -- Вернуть количество экземпляров на складе.
  RETURN stock_amount;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.29 показан результат вызова функции `stock_amount()` для кода книги `7808` и издания `1`.

Листинг 6.29. Результаты вызова функции stock_amount()

```
booktowrHf SELECT stock_amount(7808,1):
```

```
stock amount
22
(1 row)
```

Команда IF/THEN/ELSE

В команде IF/THEN/ELSE задаются два блока команд. Первый блок выполняется в том случае, если условие истинно, а второй – если оно ложно. Синтаксис команды IF/THEN/ELSE приведен в листинге 6.30.

Листинг 6.30. Синтаксис команды IF/THEN/ELSE

```
CREATE FUNCTION функция (аргументы) RETURNS тип AS '
DECLARE
  объявления
BEGIN
  IF условие THEN
     команда;
     [...]
  ELSE
     команда;
     [...]
  END IF;
END;
' LANGUAGE 'plpgsql';
```

Функция в листинге 6.31 делает практически то же, что и функция в листинге 6.28: она также определяет код ISBN по коду книги и номеру издания, сохраняет его в переменной и получает количество экземпляров книги на складе.

Затем команда IF/THEN/ELSE проверяет, является ли количество книг на складе положительной величиной. Если число положительно, функция возвращает TRUE – признак наличия книг на складе. В противном случае функция возвращает FALSE. Стоит напомнить, что функция in_stock() предназначена для вызова из других функций, а возвращаемое значение должно интерпретироваться той функцией, из которой она была вызвана.

Листинг 6.31. Команда IF/THEN/ELSE

```
CREATE FUNCTION in_stock (integer, integer) RETURNS boolean AS '
DECLARE
-- Объявление псевдонимов для аргументов функции.
b_id ALIAS FOR $1;
b_edition ALIAS FOR $2;
-- Объявление текстовой переменной для найденного кода ISBN.
b_isbn text;
-- Объявление целочисленной переменной для количества экземпляров.
stock_amount integer;

BEGIN
-- Команда SELECT INTO находит в таблице editions запись.
-- у которой код книги и номер издания совпадают с аргументами
-- функции. Код ISBN найденной записи присваивается переменной.
SELECT INTO b_isbn isbn FROM editions WHERE
book_id = b_id AND edition = b_edition;
-- Проверить, не был ли полученный код ISBN равен NULL.
```

```
-- Значение NULL говорит о тон. что в базе данных
-- не существует записи книги с кодом и номером издания.
-- переданными в аргументах функции. Если запись не существует.
-- функция возвращает FALSE и завершает работу.
IF b_isbn IS NULL THEN
RETURN FALSE;
END IF;
-- Получить из таблицы stock количество экземпляров книги
-- на складе и присвоить его переменной stock_amount.
SELECT INTO stock_amount stock FROM stock
WHERE isbn = b_isbn;
-- Проверить, является ли количество книг на складе
-- положительной величиной. Если количество положительно,
-- функция возвращает TRUE, а если отрицательно
-- или равно нулю - FALSE.
IF stock_amount <= 0 THEN
   RETURN FALSE;
ELSE
   RETURN TRUE;
END IF;
END;
' LANGUAGE 'plpgsql':
```

В листинге 6.32 показан результат вызова in_stock() для кода книги 4513 и издания 2.

Листинг 6.32. Результат вызова функции in_stock()

```
booktown=# SELECT in_stock(4513,2);
in stock
t
(1 row)
```

Функция вернула значение TRUE – признак наличия книги на складе.

Команда IF/THEN/ELSE/IF

Команда IF/THEN/ELSE/IF предназначена для последовательной проверки нескольких условий. Сначала проверяется первое условие; если оно окажется равным FALSE, проверяется следующее условие и т. д. Последняя секция ELSE содержит команды, выполняемые в том случае, если пи одно из проверенных условий не было истинным. Синтаксис команды IF/THEN/ELSE/IF:

```
CREATE FUNCTION функция (аргументы) RETURNS тип AS '
DECLARE
объявление BEGIN
IF условие THEN
   команда;
   [...]
ELSE IF условие
   команда;
   [...]
END IF;
END;
' LANGUAGE 'plpgsql' :
```

В листинге 6.33 приведен практический пример применения функции с командой IF/ THEN/ELSE/IF. Функция books_by_subject() сначала использует переданный аргумент (тему книги) для выборки кода темы. Затем первая команда IF проверяет, не содержит ли переданный аргумент строку all.

Если при вызове был передан аргумент all, команда IF/THEN вызывает функцию extract_all_titles() и присваивает полученный список книг и тем (возвращаемый в виде текстовой переменной) переменной found_text.

Если аргумент отличен от all, следующая команда ELSE IF проверяет, является ли код темы нулем или положительным числом. Если значение sub_id больше либо равно нулю, выполняются команды, содержащиеся в теле конструкции ELSE IF – сначала вызывается функция extract_title(), которая возвращает список всех существующих книг по заданной теме, после чего тема возвращается вместе с полученным списком.

Затем другая команда ELSE IF сравнивает код темы с псевдозначением NULL. Если значение sub_id равно NULL, значит, переданная при вызове функции тема не встречается в базе данных booktown, а выполненная в самом начале команда SELECT INTO завершилась неудачей. В этом случае функция возвращает строку «subject not found».

ПРИМЕЧАНИЕ Функции extract_all_titles() и extract_title(), используемые в листинге 11.40, будут рассмотрены ниже, когда речь пойдет о циклах.

Листинг 6.33. Команда IF/THEN/ELSE/IF

```
CREATE FUNCTION books_by_subject (text) RETURNS text AS '
DECLARE
  -- Объявление псевдонима для аргумента, содержащего либо
  -- строку all, либо тему.
  sub_title ALIAS FOR $1;
  -- Объявление целочисленной переменной для хранения кода темы
  -- и текстовой переменной для хранения списка найденных книг.
  -- Текстовая переменная инициализируется пустой строкой.
  sub_id integer;
  found_text text;
BEGIN
  -- Получить код темы, описание которой передано в аргументе.
  SELECT INTO sub_id id FROM subjects WHERE subject = sub_title;
  -- Проверить, запросил ли пользователь информацию обо всех темах
  -- (строка all). В этом случае вызвать функцию extract_all_titles() и вернуть полученную текстовую переменную.
  IF sub_title = ''all''
  THEN
     found_text extract_all_titles();
     RETURN found text;
  -- Если в аргументе НЕ БЫЛА передана строка "all", проверить,
  -- входит ли код темы в интервал допустимых значений.
  -- Если это так, вызвать функцию extract_title() с кодом темы
  -- и присвоить результат переменной found_text.
  ELSE IF sub_id >= 0
  THEN
     found_text := extract_title(sub_id);
     RETURN "\n" || sub_title || ":\n" | found_text;
```

```
  -- Если код темы равен NULL, вернуть сообщение о том. что
  -- заданная тема не найдена.
  ELSE IF sub_id IS NULL
  THEN
     RETURN "Subject not found.";
  END IF;
  END IF;
  END IF;
  RETURN "An error occurred.";
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.34 сначала приведен результат вызова функции `books_by_subject()` с аргументом `all` (признак того, что пользователь хочет получить список книг по всем темам). Затем следуют результаты, полученные при вызове функции с аргументом `Computers` (получение списка книг, посвященных компьютерам).

Листинг 6.34. Результаты вызова функции books_by_subject()

```
booktown=# SELECT books_by_subject('all');
books_by_subject

Arts:
Dynamic Anatomy

Business:

Children's Books:
The Cat in the Hat
Bartholomew and the Oobleck
Franklin in the Dark
Goodnight Moon
[...]

Science:

Science Fiction:
Dune
2001: A Space Odyssey
(1 row)
booktown=# SELECT books_by_subject('Computers');
books by_subject

Computers:
Learning Python
Perl Cookbook
Practical PostgreSQL
Programming Python
(1 row)
```

Циклы

Другую категорию команд, передающих управление внутри функций, составляют циклы. В циклах используются разные виды итераций, предназначенные для решения разных задач. Итеративные вычисления значительно расширяют возможности функций `PL/pgSQL`.

В `PL/pgSQL` реализованы три типа циклов: простейший (безусловный) цикл, цикл `WHILE` и цикл `FOR`. Вероятно, из этих трех циклов чаще всего применяется

цикл FOR, подходящий для широкого круга задач программирования, хотя и другие циклы также достаточно часто встречаются на практике.

Безусловный цикл

Ключевое слово LOOP начинает простейший безусловный цикл. Команды безусловного цикла выполняются до тех пор, пока не будет достигнуто ключевое слово EXIT. За ключевым словом EXIT может следовать секция WHEN с выражением, определяющим условие выхода. Выражение должно относиться к логическому типу. Например, оно может проверять, достигла ли переменная некоторой величины. Ниже приведен синтаксис безусловного цикла (без ключевого слова LOOP):

```
LOOP
команда;
[...]
END LOOP;
```

Команда EXIT завершает работу безусловного цикла и может дополнительно содержать метку и/или условие завершения.

Метка представляет собой произвольный идентификатор, заключенный между «префиксом и суффиксом». Чтобы назначить метку циклу, следует расположить ее непосредственно перед началом цикла. Синтаксис определения цикла с меткой:

```
«метка»
LOOP
[...]
END LOOP;
```

Назначение метки циклу позволяет указать нужный цикл при выходе из нескольких вложенных циклов (команда EXIT с меткой работает лишь в том случае, если завершаемому циклу была присвоена соответствующая метка).

Если команда EXIT содержит условие, цикл прерывается только в том случае, если это условие истинно.

Синтаксис вызова EXIT в цикле LOOP:

```
[ «метка» ]
LOOP
statement;
[...]
EXIT [ метка ] [ WHEN условие ];
END LOOP;
```

В листинге 6.35 приведен пример безусловного цикла и команды EXIT, завершающей цикл при выполнении некоторого условия. Функция square_integer_loop() возводит целое число в квадрат (умножает его само на себя) до тех пор, пока его значение не превысит 10 000, после чего возвращает полученный результат.

Листинг 6.35. Использование безусловного цикла

```
CREATE FUNCTION square_integer_loop (integer) RETURNS integer AS '
DECLARE
  -- Объявление псевдонима для аргумента,
  numl ALIAS FOR $1;
```

```
  -- Объявление целочисленной переменной для хранения результа-
та,
  result integer;
BEGIN
  -- Исходное число присваивается переменной
  result, result := numl;
  LOOP
    result := result * result;
    EXIT WHEN result >= 10000;
  END LOOP;
  RETURN result;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.36 показан результат вызова `square_integer_loop()` с аргументом `3`.

Листинг 6.36. Результат вызова функции square_integer_loop()

```
booktown=# SELECT square_integer_loop(3);
squa re_i nteger_l oop
6561
(1 row)
```

Цикл WHILE

Цикл `WHILE` выполняет блок команд до тех пор, пока заданное условие не станет ложным. При каждой итерации цикла `WHILE` условие проверяется перед выполнением первой команды блока, и если условие равно `TRUE` – блок выполняется. Таким образом, если условие никогда не становится равным `FALSE`, блок выполняется в бесконечном цикле вплоть до принудительного завершения клиентского процесса. Синтаксис цикла `WHILE`:

```
[ «метка» ]
WHILE условие LOOP
  команда;
  [...]
END LOOP;
```

В листинге 6.37 циклы `WHILE` продемонстрированы на примере функции `add_two_loop()`. Функция увеличивает число на `1` до тех пор, пока не будет достигнуто некоторое пороговое значение. Начальное и конечное значения передаются функции в виде аргументов. Обозначение `!=`, встречающееся в листинге 6.37, является оператором неравенства. В данном примере условие означает следующее: цикл `WHILE` продолжает выполняться, пока переменная `result` не равна переменной `high_number`. Иначе говоря, цикл `WHILE` завершается в тот момент, когда переменная `result` становится равной `high_number`.

Листинг 6.37. Использование цикла WHILE

```
CREATE FUNCTION add_two_loop (integer, integer) RETURNS integer AS '
DECLARE
  -- Объявление псевдонимов для аргументов.
  low_number ALIAS FOR $1;
  high_number ALIAS FOR $2;
  -- Объявление переменной для хранения результата,
  result integer = 0;
BEGIN
```

```
  -- Увеличивать переменную result на 1 до тех пор,
  -- пока она не станет равна high_number.
  WHILE result != high_number LOOP
    result := result + 1;
  END LOOP;
  RETURN result;
END;
' LANGUAGE 'plpgsql';
```

Цикл FOR

Возможно, циклы FOR – самая важная разновидность циклов, реализованных в PL/pgSQL. Цикл FOR выполняет программный блок для целых чисел из заданного интервала. У циклов FOR в PL/pgSQL существуют аналоги в других процедурных языках программирования.

Заголовок цикла FOR начинается с объявления целочисленной переменной, управляющей выполнением цикла. Затем указывается интервал принимаемых ею значений, а далее следует блок команд. Управляющая переменная уничтожается сразу же после выхода из цикла, причем ее не нужно объявлять в секции объявлений блока.

Синтаксис цикла FOR:

```
[ «метка» ]
FOR переменная IN [ REVERSE ] выражение! . . выражение? LOOP
    команда;
    [...]
END LOOP;
```

Цикл FOR выполняет одну итерацию для каждого значения переменной переменная в интервале, границы которого определяются выражениями выражение! и выражение? (включительно). В начале цикла переменная инициализируется значением выражения выражение! и увеличивается на 1 после каждой итерации. Если в заголовке цикла присутствует ключевое слово REVERSE, то переменная не увеличивается, а уменьшается.

ПРИМЕЧАНИЕ Управляющую переменную цикла не обязательно объявлять вне блока FOR, если вы не собираетесь работать с ней после завершения цикла.

Циклы FOR также используются для перебора результатов запросов. Пример приведен в листинге 6.38, где цикл FOR работает с переменными RECORD и %ROWTYPE.

Синтаксис цикла FOR с перебором записей:

```
[ «метка» ]
FOR { переменная_record %переменная_rowtype } IN секция_select LOOP
    команда;
    [...]
END LOOP;
```

В листинге 6.38 функция extract_all_titles() получает из базы данных список всех названий книг, упорядоченных по теме. Если по какой-либо теме в базе данных не находится ни одной книги, выводится пустая строка. Список возвращается в виде текстовой переменной. Перебор тем по кодам в функции extract_all_titles() осуществляется в цикле FOR.

Внутри первого цикла FOR находится другой, вложенный цикл FOR. Он перебирает все книги в базе данных и отбирает те из них, у которых поле subject_id совпадает с управляющей переменной исходного цикла (текущим кодом темы). В листинге 6.38 управляющая переменная i инициализируется нулевым значением, поскольку нумерация кодов тем в таблице subjects начинается с 0.

Листинг 6.38. Пример использования цикла FOR

```
CREATE FUNCTION extract_all_titles2 () RETURNS text AS '
DECLARE
  -- Объявление переменной для кода темы.
  sub_id integer;
  -- Объявление переменной для хранения списка названий книг.
  text_output text;
  -- Объявление переменной для названия темы.
  sub_title text;
  -- Объявление переменной для хранения записей,
  -- полученных при выборке из таблицы books.
  row_data books %ROWTYPE;
BEGIN
  -- Внешний цикл FOR: тело цикла выполняется до тех пор,
  -- пока переменная 1 не станет равна 15. Перебор начинается с 0.
  -- Следовательно, тело цикла будет выполнено 16 раз
  -- (по одному пля каждой темы).
  FOR i IN 0..15 LOOP
    -- Получить из таблицы subjects название темы,
    -- код которой совпадает со значением переменной 1.
    SELECT INTO sub_title subject FROM subjects WHERE id = 1;
    -- Присоединить название темы, двоеточие и символ новой строки
    -- к переменной text_output.
    text_output = text_output || "\n" | sub_title | ":\n";
    -- Перебрать все записи таблицы books,
    -- у которых код темы совпадает со значением переменной i.
    FOR row_data IN SELECT * FROM books
    WHERE subject_id = i LOOP
      -- Присоединить к переменной text_output название книги
      -- и символ новой строки.
     text_output := text_output || row_data.title || "\n";
    END LOOP;
  END LOOP;
  -- Вернуть список.
  RETURN text_output;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.39 приведена другая функция, в которой цикл FOR используется для перебора результатов запроса SQL. При каждой итерации цикла FOR в листинге 6.39 содержимое одной из записей запроса к таблице books помещается в переменную row_data, после чего значение поля title присваивается переменной text_output.

Цикл продолжается до тех пор, пока не будет достигнута последняя запись в таблице books. В конце цикла переменная text_output содержит полный спи-

сок всех книг по теме, код которой был передан в аргументе функции. Работа функции завершается возвращением переменной text_output.

Листинг 6.39. Использование цикла FOR с атрибутом %ROWTYPE

```
CREATE FUNCTION extract_title (integer) RETURNS text AS '
DECLARE
  -- Объявление псевдонима для аргумента функции.
  sub_id ALIAS FOR $1;
  -- Объявление переменной для хранения названий книг.
  -- Переменная инициализируется символом новой строки,
  text output text : = ''\n'';
  -- Обьявление переменной для хранения записей
  -- таблицы books.
  row_data books %ROWTYPE;
BEGIN
  -- Перебор результатов запроса.
  FOR row_data IN SELECT * FROM books
  WHERE subject_id = sub_id ORDER BY title LOOP
    -- Присоединить название книги к переменной text_output.
    text_output := text_output || row_data.title || "\n";
  END LOOP;
  -- Вернуть список книг.
  RETURN text_output;
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.40 показан результат вызова функции extract_title() с аргументом 2. В таблице subjects этот код соответствует теме «Children's Books» (книги для детей).

Листинг 6.40. Результат выполнения функции extract_title()

```
booktown=# SELECT extract_title(2);
extract_title
Bartholomew and the Oobleck
Franklin in the Dark
Goodnight Moon
The Cat in the Hat
(1 row)
```

Переменная row_data объявляется с атрибутом %ROWTYPE по отношению к таблице books, поскольку она будет использоваться только для хранения записей из таблицы books. С таким же успехом можно было объявить row_data с типом RECORD:

```
row_data RECORD;
```

Но это следует делать только в том случае, если в переменной предполагается хранить записи из нескольких таблиц.

Функция extract_title() возвращает одинаковые результаты как при объявлении переменной с типом RECORD, так и с атрибутом %ROWTYPE.

6.1.8. Обработка ошибок и исключений

Команда RAISE предназначена для инициирования ошибок и исключений в функциях PL/pgSQL. Она передает заданную информацию механизму PostgreSQL

elog (стандартное средство ведения протокола ошибок – данные обычно направляются в файл /var/log/messages или $PGDATA/serverlog с одновременным выводом в поток stderr).

В команде RAISE также указывается уровень ошибки и строка, передаваемая PostgreSQL. Кроме того, в команду можно включить переменные и выражения, значения которых будут содержаться в выходных данных. Соответствующие позиции строки помечаются знаками процента (%). Синтаксис команды RAISE: RAISE уровень ''сообщение'' [. идентификатор [...]]:

В таблице 6.1 приведены три допустимые значения уровня ошибки с краткими описаниями.

Таблица 6.1

Допустимые значения уровня ошибки

Значение	Описание
DEBUG	Команда уровня DEBUG направляет заданный текст в виде сообщения DEBUG: в журнал PostgreSQL и клиентской программе, если клиент подключен к кластеру базы данных, работающему в отладочном режиме. Команды RAISE уровня DEBUG игнорируются базами данных, работающими в режиме реальной эксплуатации
NOTICE	Команда уровня NOTICE направляет заданный текст в виде сообщения NOTICE: в журнал PostgreSQL и клиентской программе. Сообщение передается в любом режиме работы PostgreSQL
EXCEPTION	Команда уровня EXCEPTION направляет заданный текст в виде сообщения ERROR: в журнал PostgreSQL и клиентской программе. Ошибка уровня EXCEPTION также вызывает откат текущей транзакции

В листинге 6.41 первая команда RAISE выводит отладочное сообщение, а вторая и третья команды выводят сообщение для пользователя. Обратите внимание на знак % в третьей команде – он отмечает позицию, в которой выводится значение an_integer. Наконец, четвертая команда RAISE выводит сообщение об ошибке и инициирует исключение, приводящее к отмене транзакции.

Листинг 6.41. Команда RAISE

```
CREATE FUNCTION raise_test () RETURNS integer AS '
DECLARE
  -- Объявление целочисленной переменной для тестового вывода.
  an_integer integer = 1;
BEGIN
  -- Вывести отладочное сообщение уровня
  DEBUG.RAISE DEBUG "The raise_test() function began.";
  an_integer = an_integer * 1;
  -- Вывести сообщение об изменении переменной an_integer,
  -- а затем вывести другое сообщение с ее новым значением.
  RAISE NOTICE "Variable an_integer was cnanged.";
  RAISE NOTICE "Variable an_integer's value is now £." an_integer;
  -- Инициировать исключение.
  RAISE  EXCEPTION  "Variable  %  cnanged.  Transaction  aborted.  ",
an_integer;
  RETURN 1;
```

```
END;
' LANGUAGE 'plpgsql';
```

В листинге 6.42 приведены результаты, полученные при вызове функции raise_test() из базы данных booktown. Отладочное сообщение DEBUG отсутствует, поскольку база данных работает не в отладочном режиме.

Листинг 6.42. Результаты вызова raise_test()

```
booktown=# SELECT raise_test();
NOTICE: Variable anjnteger was changed.
NOTICE: Variable anjnteger's value is now 2.
ERROR: Variable 2 changed. Aborting transaction.
```

6.1.9. Вызов функций

При вызове функции PL/pgSQL из кода PL/pgSQL имя функции обычно включается в команду SQL SELECT или в команду присваивания. Примеры:

```
SELECT функция (аргументы);
переменная := функция(аргументы);
```

Подобный способ вызова функций при выборке и присваивании стал стандартным, поскольку любая функция PostgreSQL должна возвращать значение некоторого типа. Ключевое слово PERFORM позволяет вызвать функцию и проигнорировать возвращаемое значение. Синтаксис вызова функции с ключевым словом PERFORM:

```
PERFORM функция (аргументы);
```

В листинге 6.43 приведены примеры вызова функции PL/pgSQL с ключевым словом PERFORM и вызова другой функции PL/pgSQL посредством присваивания (в команде SELECT INTO).

Функция ship_item() является удобной «оболочкой» для вызова функции add_shipment(). Она получает исходные данные, убеждается в существовании покупателя и книги, а затем передает данные add_shipment().

Листинг 6.43. Использование ключевого слова PERFORM

```
CREATE FUNCTION ship_item (text, text, text) RETURNS integer AS '
DECLARE
  -- Объявление псевдонимов для аргументов функции.
  l_name ALIAS FOR $1;
  f_name ALIAS FOR $2;
  book_isbn ALIAS FOR $3;
  -- Объявление переменной для хранения кода книги.
  -- Переменная используется для проверки переданного кода ISBN.
  book_id integer;
  -- Объявление переменной для хранения кода покупателя.
  -- Переменная используется для проверки данных покупателя.
  customer_id integer;
BEGIN
  -- Получить код покупателя при помощи ранее определенной функции.
  SELECT INTO customer_id get_customer_id(l_name, f_name);
  -- Если покупатель не найден, функция get_customerjd
  -- возвращает -1. В этом случае вернуть -1 и выйти из функции.
```

```
    IF customer_id = -1 THEN
       RETURN -1;
    END IF;
    -- Получить код книги с заданным кодом ISBN.
    SELECT INTO book_id FROM editions WHERE isbn = book_isbn;
      -- Если данные книги отсутствуют в базе, вернуть -1.
    IF NOT FOUND
    THEN
       RETURN -1;
    END IF;
    -- Если книга и покупатель существуют.
    -- сохранить информацию о поставке в базе.
    PERFORM add_shipment(customer_id, book_isbn);
    -- Вернуть 1 - признак успешного выполнения функции.
    RETURN 1;
  END;
  ' LANGUAGE 'plpgsql';
```

6.2. ТРИГГЕРЫ

Довольно часто перед некоторыми событиями SQL или после них должны выполняться определенные операции − например, проверка логической целостности данных, заносимых в базу, предварительное форматирование данных перед вставкой или модификация других таблиц, логически обусловленная удалением или модификацией записей. Традиционно такие операции выполнялись на программном уровне приложением, подключившимся к базе данных, а не самой СУБД.

В PostgreSQL поддерживаются нестандартные расширения, называемые триггерами (trigger) и упрощающие взаимодействие приложения с базой данных. Триггер определяет функцию, которая должна выполняться до или после некоторой операции с базой данных. Триггеры реализуются на языке C, PL/pgSQL или любом другом функциональном языке (кроме SQL), который может использоваться в PostgreSQL для определения функций.

ВНИМАНИЕ. Триггеры относятся к числу специфических расширений PostgreSQL, поэтому их не рекомендуется использовать в решениях, требующих высокой степени совместимости с другими РСУБД.

Триггеры срабатывают при выполнении с таблицей команды SQL INSERT, UPDATE или DELETE.

6.2.1. Создание триггера

Триггер создается на основе существующей функции. PostgreSQL позволяет создавать функции на разных языках программирования, в том числе на SQL, PL/pgSQL и C. В PostgreSQL 7.1.x триггеры могут вызывать функции, написанные на любом языке, но за одним исключением: функция не может быть полностью реализована на SQL.

В определении триггера указывается, должна ли заданная функция вызываться до или после выполнения некоторой операции с таблицей. Синтаксис определения триггера выглядит так:

```
    CREATE TRIGGER триггер { BEFORE | AFTER } { событие [ OR событие
...]}
    ON таблица
    FOR EACH { ROW STATEMENT }
    EXECUTE PROCEDURE функция ( аргументы )
```

Ниже приводятся краткие описания компонентов этого определения.

– `CREATE TRIGGER` триггер. В аргументе триггер указывается произвольное имя создаваемого триггера. Имя может совпадать с именем триггера, уже существующего в базе данных — при условии, что этот триггер установлен для другой таблицы. Кроме того, по аналогии с большинством других несистемных объектов баз данных, имя триггера (в сочетании с таблицей, для которой он устанавливается) должно быть уникальным лишь в контексте базы данных, в которой он создается.

– `{BEFORE AFTER}`. Ключевое слово `BEFORE` означает, что функция должна выполняться перед попыткой выполнения операции, включая все встроенные проверки ограничений данных, реализуемые при выполнении команд `INSERT` и `DELETE`. Ключевое слово `AFTER` означает, что функция вызывается после завершения операции, приводящей в действие триггер.

– `{событие [OR событие ... ]}`. События `SQL`, поддерживаемые в `PostgreSQL` При перечислении нескольких событий в качестве разделителя используется ключевое слово `OR`.

– `ON таблица`. Имя таблицы, модификация которой заданным событием приводит к срабатыванию триггера.

– `FOR EACH {ROW STATEMENT}`. Ключевое слово, следующее за конструкцией `FOR EACH` и определяющее количество вызовов функции при наступлении указанного события. Ключевое слово `ROW` означает, что функция вызывается для каждой модифицируемой записи. Если функция должна вызываться всего один раз для всей команды, используется ключевое слово `STATEMENT`.

– `EXECUTE PROCEDURE` функция (аргументы). Имя вызываемой функции с аргументами.

ПРИМЕЧАНИЕ. Создание триггеров разрешено только владельцу базы данных или суперпользователю.

Механизм ограничений `PostgreSQL` позволяет реализовать простое сравнение данных со статическими значениями, но иногда проверка входных данных должна производиться по более сложным критериям. Это типичный пример ситуации, в которой удобно воспользоваться триггером.

Проверка входных данных с применением триггеров может осуществляться перед вставкой данных в таблицу или перед их обновлением в таблице. Функция триггера может убедиться в том, что новые данные удовлетворяют сложной системе ограничений, и даже вернуть признак ошибки через систему регистрации ошибок `PostgreSQL`.

Предположим, вы написали на процедурном языке функцию, которая проверяет данные, переданные при вызове команды `INSERT` или `UPDATE` для таблицы `shipments`, и затем обновляет таблицу `stock`, снимая поставленный товар со складского учета. Такую функцию можно написать на любом языке, поддерживаемом `PostgreSQL` (кроме «чистого» `SQL`, о чем говорилось выше).

Прежде всего функция убеждается в том, что переданный код покупателя (customer_id) и код ISBN (isbn) присутствуют в таблицах customers и editions. Если хотя бы один из кодов отсутствует, функция возвращает признак ошибки. Если оба кода присутствуют в таблицах, команда SQL выполняется, и после успешного завершения количество товара на складе в таблице stock автоматически уменьшается в соответствии с объемом поставки.

Триггер, создаваемый в листинге 6.44, срабатывает непосредственно перед выполнением команды INSERT или UPDATE в таблице shipments. Триггер вызывает функцию check_shipment _addition() для каждой изменяемой записи.

Листинг 6.44. Создание триггера check_shipment

```
booktown=# CREATE TRIGGER check_shipment
booktown-# BEFORE INSERT OR UPDATE
booktown-# ON shipments FOR EACH ROW
booktown-# EXECUTE PROCEDURE check_shipment_addition();
CREATE
```

Триггер check_shipment настроен на выполнение функции check_shipment_addition() для команд INSERT и UPDATE, поэтому он достаточно надежно обеспечивает логическую целостность данных в полях customer_id и isbn. Ключевое слово ROW гарантирует, что каждая добавляемая или модифицируемая запись будет обработана функцией проверки check_argument_ addition().

Функция check_shipment_addition() вызывается без аргументов, поскольку для проверки записей в ней используются внутренние переменные PL/pgSQL.

6.2.2. Получение информации о триггерах

В PostgreSQL триггеры хранятся в системной таблице pg_trigger, что позволяет получить информацию о существующих триггерах на программном уровне. Структуру таблицы pg_trigger иллюстрирует таблице 6.2.

Таблица 6.2

Таблица pgjrigger

Поле	Тип
TGRELID	oid
TGNAME	name
TGFOID	oid
TGTYPE	smallint
TGENABLED	boolean
TGISCONSTRAINT	boolean
TGCONSTRNAME	name
TGCONSTRRELID	oid
TGDEFERRABLE	boolean

Окончание табл. 6.2

TGINLTDEF ERRED	boolean
TGNARGS	smallint
TGATTR	int2vector
TGARGS	bytea

Большинство полей, перечисленных в таблице 6.2, в прямых запросах не используется. Среди атрибутов триггеров в системной таблице pg_trigger центральное место занимают атрибуты tgrelid и tgname.

В поле tgrelid хранится идентификатор отношения, с которым связан данный триггер. Значение относится к типу oid и соответствует содержимому поля relfilenode системной таблицы pg_class. В поле tgname хранится имя триггера, указанное в команде CREATE TRIGGER при его создании.

6.2.3. Удаление триггера

Команда DROP TRIGGER удаляет триггер из базы данных. Удаление триггеров, как и их создание командой CREATE TRIGGER, может выполняться только владельцем триггера или суперпользователем.

Синтаксис удаления существующих триггеров:

```
DROP TRIGGER имя ON таблица
```

В листинге 6.45 приведен пример удаления триггера check_shipment, установленного для таблицы shipments.

Листинг 6.45. Удаление триггера

```
booktown=# DROP TRIGGER check_shipment ON shipments:
DROP
```

Сообщение DROP означает, что триггер успешно удален. Обратите внимание: при удалении указывается не только имя удаляемого триггера, но и имя таблицы.

Если вы не помните, в какой таблице был установлен удаляемый триггер, необходимую информацию можно получить из системных таблиц PostgreSQL. Например, можно провести объединение полей tgrelid системной таблицы pg_trigger и поля relfilenode системной таблицы pg_class и сравнить имя триггера с полем tgname. Запрос, приведенный в листинге 6.46, возвращает имя отношения (rel name), связанного с триггером check_shipment.

Листинг 6.46. Получение имени таблицы, связанной с триггером

```
booktown=# SELECT relname FROM pg_class
booktown-# INNER JOIN pg_trigger
booktown-# ON (tgrelid = relfilenode)
booktown-# WHERE tgname = 'check_shipment': .
relname
shipments
(1 row)
```

ВНИМАНИЕ. При удалении функции, вызываемой по срабатыванию триггера, триггер перестает работать, причем повторное определение функции с тем же именем не исправит проблему. После повторного создания функции триггер также приходится создавать заново.

6.2.4. PL/pgSQL и триггеры

Определения триггеров PostgreSQL могут содержать ссылки на триггерные функции (то есть функции, которые должны вызываться при срабатывании триггера), написанные на языке PL/pgSQL. Триггер определяет операцию, которая должна выполняться при наступлении некоторого события в базе данных.

Не путайте определение триггера с определением триггерной функции. Триггер определяется командой SQL CREATE TRIGGER, а триггерная функция определяется командой SQL CREATE FUNCTION.

Триггерная функция определяется без аргументов и возвращает значение специального типа данных opaque. Синтаксис определения трнггерпой функции PL/pgSQL командой CREATE FUNCTION приведен в листинге 6.47.

Листинг 6.47. Определение триггерной функции

```
CREATE FUNCTION функция () RETURNS opaque AS '
DECLARE
  объявления;
  [...]
BEGIN
  команды;
  [...]
END;
' LANGUAGE 'plpgsql':
```

В триггерных функциях используются специальные переменные, содержащие информацию о сработавшем триггере. С помощью этих переменных триггерная функция работает с данными таблиц. Специальные переменные триггерных функций перечислены в таблице 6.3.

Таблица 6.3

Специальные переменные в триггерных функциях

Имя	Тип данных	Описание
NEW	RECORD	Новая запись базы данных, созданная командой INSERT или UPDATE при срабатывании триггера уровня записи (ROW). Переменная используется для модификации новых записей
OLD	RECORD	Старая запись базы данных, оставшаяся после выполнения команды INSERT или UPDATE при срабатывании триггера уровня записи (ROW)
TGNAME	name	Имя сработавшего триггера
TG_WHEN	text	Строка BEFORE или AFTER в зависимости от момента срабатывания триггера, указанного в определении (до или после операции)
TG_LEVEL	text	Строка ROW или STATEMENT в зависимости от уровня триггера, указанного в определении
TG_OP	text	Строка INSERT, UPDATE или DELETE в зависимости

Имя	Тип данных	Описание
		от операции, вызвавшей срабатывание триггера
TG_RELID	old	Идентификатор объекта таблицы, в которой сработал триггер
TG_RELNAME	name	Имя таблицы, в которой сработал триггер
TG_NARGS	Integer	Количество аргументов триггерной функции, указанных в определении триггера
TG_ARGV[]	Массив text	Аргументы, указанные в команде CREATE TRIGGER. Индексация массива начинается с нуля

В листинге 6.48 приведен пример определения трнггерной функции PL/pgSDL, использующей некоторые из перечисленных переменных. Триггерная функция check_shipment_addition() вызывается после выполнения операции INSERT или UPDATE с таблицей shipments.

Функция check_shipment_addition() убеждается в том, что каждая новая запись содержит действительный код покупателя и код ISBN книги. Затем общее количество экземпляров в таблице stock уменьшается на 1, если триггер сработал по команде SQL INSERT (но не по команде UPDATE!)

Листинг 6.48. Триггерная функция check_shipment_addition()

```
CREATE FUNCTION check_shipment_addition () RETURNS opaque AS '
DECLARE
  -- Объявление переменной для хранения кода покупателя.
  id_number integer;
  -- Объявление переменной для хранения кода ISBN.
  book_isbn text;
BEGIN
  -- Если в таблице customers существует код, совпадающий с кодом
  -- покупателя в таблице new, присвоить его переменной id_number.
  SELECT   INTO   id_number   id   FROM   customers   WHERE   id   =
NEW.customer_id;
  -- Если совпадение не найдено, инициировать исключение.
  IF NOT FOUND THEN
     RAISE EXCEPTION "Invalid customer ID number.";
  END IF;
  -- Если в таблице editions существует код ISBN, совпадающий
  -- с кодом ISBN в таблице new, присвоить его переменной book_isbn.
  SELECT INTO book_isbn isbn FROM editions WHERE isbn = NEW.isbn;
  -- Если совпадение не найдено, инициировать исключение.
  IF NOT FOUND THEN
     RAISE EXCEPTION "Invalid ISBN.";
  END IF;
  -- Если обе предыдущие проверки завершились успешно,
  -- обновить количество экземпляров.
  IF TG_OP = "INSERT" THEN
     UPDATE stock SET stock = stock -1 WHERE isbn = NEW.isbn;
  END IF;
  RETURN NEW;
END;
' LANGUAGE 'plpgsql';
```

После создания функции check_shipment_addition() в таблице shipments устанавливается триггер для ее вызова. В листинге 6.49 приведен синтаксис команды, создающей триггер check_shipment в базе данных booktown (для клиента psql).

Листинг 6.49. Триггер check_shipment

```
booktown=# CREATE TRIGGER check_shipment
booktown-* BEFORE INSERT OR UPDATE
booktown-# ON shipments FOR EACH ROW
booktown-* EXECUTE PROCEDURE check_shipment_addition();
CREATE
```

Обратите внимание: функция check_shipment_addition() должна определяться в базе данных booktown до определения триггера, по которому она вызывается. Триггерные функции всегда определяются раньше триггеров.

Часть 3
SQL и XML как средства представления структурированных данных

Современные информационные системы гетерогенны и форматы хранения данных, определяемые реляционными СУБД, не являются единственными в своем роде. Например, данные о читателях библиотеки могут сопровождаться, используя электронные таблицы Excel, а отдел кадров может хранить информацию о студентах, которые одновременно являются читателями той же библиотеки в формате, определяемой реляционной СУБД, например, PostgreSQL. В случае зачисления студентов на первый курс сведения о студентах, введенные в базу данных отдела кадров, могут понадобиться библиотеке. Но в этом случае возникает проблема обмена данными. Дело в том, что электронные таблицы Excel не могут адекватно отобразить данные формата PostgreSQL.

Здесь на простом примере обозначена проблема совместимости форматов. Для решения этой проблемы предложено решение – хранение данных в формате XML. Так называется стандартный язык для представления и обмена *структурированными данными*. На сегодняшний день XML широко используется в Web-среде, выступая промежуточным универсальным форматом хранения данных.

XML представляет собой структуру и правила описания любой информации содержательным способом. Используя XML, вы можете создать свой язык разметки для представления информации любого вида. XML – это одна из самых важных новейших технологий, порожденных развитием Интернета.

XML – это самостоятельная технология, хотя и возникшая исторически как Internet-технология. Представляется важным рассмотреть технологию XML в отрыве от Internet в связи с технологиями баз данных. Поскольку и SQL, и XML является языками определения структурированных данных, можно предположить, что между этими двумя языками должна существовать взаимосвязь.

Вопрос заключается, в том как эти языки соотносятся друг к другу – находятся ли эти языки в конфликте или дополняют друг друга. Как связан язык XML с реляционным языком запросов SQL, можно ли на основе XML и построить модель данных, что собой представляет XML-ориентированная база данных, существуют

ли таковые на сегодняшний день – этим вопросам посвящена последняя часть учебного пособия. В седьмой главе описываются основы XML, а в восьмой главе, на основе имеющихся сведений об этих языках, анализируются развивающиеся взаимоотношения между XML и SQL и рассматривается вопрос интеграции XML в SQL-продукты.

ГЛАВА 7. XML КАК СПОСОБ ЛОГИЧЕСКОГО ПРЕДСТАВЛЕНИЯ ИНФОРМАЦИИ

В конце 80-х годов в стенах CERN (Европейский центр ядерных исследований, Женева) возникла идея, которая в короткие сроки воплотились в интенсивно развивающуюся глобальную, бесконечно масштабируемую и распределенную систему. Эта система получила название Всемирной паутины (World Wide Web – сокращенно Web) и предоставляла пользователям свободный доступ к большинству информационных ресурсов в любой момент времени.

Среда Web представляет собой очень привлекательную платформу для доступа к сведениям, содержащимся в базах данных, по всему миру, обеспечивая возможность глобального доступа к данным для пользователей и организаций. При этом:

– традиционная двухуровневая архитектура «клиент/сервер» требует применения «толстых» клиентов, недостаточно эффективно реализующих как функции интерфейса пользователя, так и прикладные алгоритмы. В отличие от этого, решения на основе технологии Web позволяют создать более естественную трехуровневую архитектуру «клиент/сервер приложений/ сервер баз данных», обеспечивающую масштабируемость системы. Поэтому все функциональные средства приложения можно разместить на отдельном Web-сервере, удаляя их из клиентской части приложения. Этим обеспечивается глобальный доступ к приложениям баз данных и экономия времени и затрат при развертывании приложений;

– Web-клиенты (или браузеры) обладают независимостью от платформы. Поскольку браузеры имеются практически для всех существующих вычислительных платформ, при условии поддержки ими стандартов HTML/Java пользователи могут получить доступ к приложению баз данных независимо от того, в какой вычислительной платформе они работают. Кроме этого, разработчикам не потребуется вносить в приложения изменения для того, чтобы они могли работать с разными операционными системами или различными интерфейсами.

Таким образом, при наличии Web-сервера приложения и Web-клиента доступ к функциям приложения легко осуществить с любого компьютера, расположенного в любой точке планеты. Но при этом появляется проблема согласованности данных, обрабатываемых в различных вычислительных платформах.

В случае использования традиционных баз данных для переноса приложений на другие платформы потребуется выполнить существенную модификацию (или даже полную реорганизацию) клиентской части каждого приложения.

В свое время базы данных развивались как способ интеграции систем хранения данных в рамках компании. Теперь, с появлением Всемирной паутины решается эта же проблема, но уже в глобальных масштабах. В этой части учебного пособия рассмотрим решение вопроса интеграции данных с целью обеспечения совместимости при передаче структурированных данных через Интернет.

7.1. ЯЗЫК HTML И ЕГО НЕДОСТАТКИ

Решающую роль в информационной революции, вызванной Web-технологиями, сыграли так называемые языки разметки, получившие свое развитие в печатном деле. При подготовке к печати, в соответствии с технологическими требованиями, документ представлялся в виде трех логических частей:

– *содержимого,* т. е. данных документа, состоящего из текста и графики;

– *структуры*, представленной заголовками, абзацами, подписями и т. д.;

– *форматирования,* т. е. визуального представления, определяемого шрифтами, отступами, оформлением страницы документа.

Таким образом, в описании документа вырисовывается следующая формула:

Документ = Данные + Структура + Форматирование

Для определения структуры документа редакторы использовали специальные символы, а для представления, т. е. форматирования, – разметки и пометки, которые включались прямо в содержимое документа. С появлением компьютерных издательских систем команды разметки, встроенные в содержимое документа, стали использоваться в издательских программах. При этом каждый тип издательского программного обеспечения или оборудования поддерживал свой набор команд разметки, что затрудняло переход от одной системы к другой. Поэтому для стандартизации разметки был разработан язык SGML (Standard Generalized Markup Language – стандартный обобщенный язык разметки), который со временем был принят как стандарт ISO.

После стандартизации общих элементов появилась возможность генерировать семейство языков разметки. Одним из таких языков стал HTML (HyperText Markup Language – язык гипертекстовой разметки), предназначенный для создания гипертекста, связывающего между собой отдельные документы.

Сегодня основная часть информации в среде Web хранится в документах, созданных на языке HTML, поэтому браузеры при отображении документов должны понимать и правильно интерпретировать дескрипторы этого языка. Но за несколько лет интенсивного развития потенциал качественного совершенствования технологий существующей версии Web оказался в значительной мере исчерпанным. Сдерживающее влияние на дальнейшую эволюцию приложений Web-технологий стали оказывать, прежде всего, слабые стороны языка HTML.

Основная проблема этого языка заключается в том, что изначально в этом языке данные, которые необходимо отобразить, и указание на то, как следует

отобразить эти данные, содержатся в одном файле. Из-за того, что данные не выделены в самостоятельный слой, изменение данных неизбежно приводит к изменению самого HTML-документа.

Этот недостаток приводит и к другой проблеме: при создании сайтов, работающих с базами данных, требуется наличие программистов, имеющих навыки как в области программирования, так и в области представления данных. Но представление данных, т. е. дизайн – это совершенно другая область знаний.

Главный недостаток HTML состоит в том, что он был разработан с прицелом на возможности человека, который в состоянии понять значение и назначение большинства документов, отображаемых в графическом виде. Машина, к сожалению, сделать этого не может. Теги в этом документе говорят браузеру, как отображать информацию, но не говорят о характере информации.

В языке HTML отсутствует поддержка метаданных, а это делает невозможной эффективную интеграцию информационных ресурсов, поддерживаемых в Web-среде и в других взаимодействующих с Web средах. Технически средства языка HTML позволяют интегрировать в среду Web ресурсы баз данных, большие архивы текстовых документов, различные мультимедийные ресурсы. Но эти инородные для гипертекста ресурсы, хотя и становятся доступными пользователю, с точки зрения их семантики остаются для среды Web «черным ящиком». Такая интеграция сводится по существу лишь к обеспечению доступа к «внешним» ресурсам посредством Web.

7.2. ЯЗЫК XML И ЕГО ОСНОВЫ

В последние годы консорциум W3C (WWW Consorcium) ведет активную деятельность, направленную на радикальный пересмотр основ Web-технологий. В результате был создан язык разметки XML (Extensible Markup Language – расширяемый язык разметки), служащий для описания и обработки информационных ресурсов Web.

Основная цель создания XML – обеспечение совместимости при передаче структурированных данных между разными системами обработки информации, особенно при передаче таких данных через Интернет. Этого удалось добиться путем возврата к разметке документов на логическом уровне, а не на уровне форматирования отдельных элементов.

У языка XML много общего с языком HTML, так как концепции обоих языков базируются на понятии разметки документа. Таким образом, XML и HTML являются родственным языками, происходящими от общего предка – языка SGML. На основе XML могут быть созданы другие специализированные языки разметки (например, MathML), иногда называемые *словарями*.

XML – это иерархическая структура, предназначенная для хранения любых данных. Визуально эта структура может быть представлена как дерево. Важнейшее синтаксическое требование заключается в том, что документ имеет только один *корневой элемент*. Это означает, что текст или другие данные всего документа должны быть расположены между единственным начальным корневым тегом и соответствующим ему конечным тегом.

В XML реализованы более жесткие правила определения структуры документа, чем в HTML, и большинство компонентов и возможностей данного языка ориентированы на представление логической структуры документа. Эту ориентацию еще больше усиливают сопутствующие стандарты, такие как XML Schema.

На Рис. 7.1. приведен фрагмент типичного XML-документа, содержащего описание второй части нашего учебного пособия. Продемонстрируем на основе этого примера основные концепции XML.

7.2.1. Объявление XML

Первая строка XML-документа называется *объявлением* XML. Это необязательная строка, указывающая версию стандарта XML (обычно это 1.0), также здесь может быть указана кодировка символов и внешние зависимости.

```
1   <?xml version="1.0" encoding="UTF-8"?>
```

Спецификация требует, чтобы процессоры XML обязательно поддерживали Юникод-кодировки UTF-8 и UTF-16 (UTF-32 не обязателен). Признаются допустимыми, поддерживаются и широко используются (но не обязательны) другие кодировки, основанные на стандарте ISO/IEC 8859. Также допустимы другие кодировки, например, русские Windows-1251, KOI-8.

Объявление типа документа решает три задачи:

– задает корневой элемент документа (для XML-документа корневым элементом является xml, а для HTML-документа – html);

– определяет элементы, атрибуты и сущности, используемые в документе.

Первая строка XML-документа на Рис. 7.1. идентифицирует его как документ XML 1.0. Остальные его части определяют структуру, атрибуты и содержимое элементов.

7.2.2. Элементы и теги

Остальная часть этого XML-документа состоит из вложенных *элементов*. Некоторые из этих элементов имеют *атрибуты* и *содержимое*. *Элемент* обычно состоит из открывающего и закрывающего тегов, обрамляющих текст и другие элементы. *Открывающий тег* состоит из имени элемента в угловых скобках. Например, в рассмотренном примере абзацы идентифицируются открывающим тегом <para>, а заголовки – открывающим тегом <header>.

Конец каждого элемента в XML-документе идентифицируется *закрывающим тегом*, содержащим символ косой черты '/' и имя типа элемента, заключенные между символами угловыми скобками. Так, на Рис. 7.1. абзацы заканчиваются тегом </para>, а заголовки – тегом </header>.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <book>
3      . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
4  <!-- Эта секция описывает содержимое 2-й части учебного пособия -->
5    <bookPart partNum="2" title="Язык  SQL. Унификация доступа к данным">
6      <para>
7        Логический подход к данным сделал также возможным создание языков запросов,
8        с ключевого слова (например, WHERE, FROM, INTO и т.д.).
9      </para>.
10     <!-- Эта секция описывает содержимое 3-й главы-->
11     <chapter chapNum="3">
12       <title> DDL – Язык определения данных реляционной модели </title >
13       <para>
14       Для того чтобы работать с данными, сначала необходимо их описать, т. е. указать
15       их структуру, . . .  информационных объектов.
16       </para>
17       <section>
18         <header hdrNum="3.1">Создание базы данных</header>
19         <header hdrNum="3.1.1">Общий формат оператора SELECT</header>
20         <para>
21           В стандарте SQL1 задается спецификация оператора описания ...
22           Пример оператора: CREATE DATABASE «D:\BD\Library».
23         </para>
24         <header hdrNum="3.1.2">Определение пароля</header>
25           . . . . . . . . . .
26         <figure figNum="3.1">Имена связей в инструкции CREATE TABLE</figure>
27         <para>Переменные ... широко используются.</para>
28           . . . . . . . . . .
29         <header hdrNum="3.1.4">Указание национальной кодировки символов</header>
30           . . . . . . . . . .
31       </section>
32       <section>
33         <header hdrLevel="3.2">Создание доменов</header>
34         <para> Если в таблице БД или ... столбцов имя домена.</para>
35           . . . . . . . . . .
36       </section>
37     </chapter>
38     <!—Эта секция описывает содержимое 4-й главы-->
39     <chapter chapNum="4">
40       <title>DML – Язык манипулирования данными  реляционной модели</title >
41       <para>С точки зрения ... отдельного рассмотрения.</para>.
42       <section>
43         <header hdrNum="4.1">Создание базы данных</header>
44         <header hdrNum="4.1.1">Общий формат оператора SELECT</header>
45           ... инструкции ...
46       </section>
47     </chapter>
48       . . . . . . . . . .
49   </bookPart>
50     . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
51 </book>
```

Рис. 7.1. `XML`*-документ, содержащий описание фрагмента второй части учебного пособия*

Содержимым элемента (англ. *content*) называется все, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы. Содержимое элемента может быть пустым, может быть некоторым значением или содержать экземпляры элементов других типов.

7.2.3. Атрибуты

Часто требуется связать некоторую информацию с блоком данных, а не просто включить эту информацию в качестве содержания этих данных. Поэтому кроме содержания у элемента могут быть *атрибуты* – пары «имя-значение», добавляемые в открывающий тег после названия элемента. Значения атрибутов всегда заключаются в кавычки. Атрибут ассоциируется с отдельным XML-элементом и описывает некоторые его характеристики. У каждого атрибута имеется имя и значение. На Рис. 7.1. элемент `<chapter>` содержит атрибут `ChapNum`, значением которого является номер главы. Элемент `<chapter>`, таким образом, связывает номер главы с ее содержимым. У элементов `<header>` тоже есть атрибут, названный `hdrNum`. Значением этого атрибута является номер раздела.

7.2.4. Иерархичность структуры XML-документа

Иерархическая структура XML является одним из его ключевых параметров. В приведенном примере показана иерархия элементов, типичная для большинства XML-документов. На верхнем уровне располагается элемент `<book>`. Его содержимым является не текст, а последовательность элементов `<chapter>`. Каждый элемент `<chapter>` содержит элемент `<title>`, за ним может следовать несколько вводных элементов `<para>` и затем последовательность элементов `<section>`.

Каждый элемент `<section>` содержит элемент `<header>` и один или несколько элементов `<para>`, которые могут перемежаться элементами `<figure>` и `<table>`. Содержимым элемента `<para>` является только текст.

Спецификация XML определяет правила, в соответствии с которыми необходимо оформлять любой XML-документ. В ней сказано, что элементы XML-документа должны быть строго вложены один в другой. Это означает, что закрывающий тег элемента нижнего уровня должен располагаться до тега закрывающего элемента более высокого уровня, его содержащего.

7.2.5. Комментарии

Как в HTML, так и в XML позволяет включать в документ *комментарии*, которые не интерпретируются ни как содержимое, ни как разметка. XML комментарии размещаются внутри пары тегов `<!--` и `-->`, и могут быть помещены в любом месте дерева. Комментарии полезны для создания заметок о структуре документа и изменениях, которые вы намерены внести в него в будущем.

Вот пример комментария

```
2   <!-- Эта секция описывает содержимое 3-й главы-->
```

7.3. XML СХЕМЫ И МЕТАДАННЫЕ

В реляционной модели обеспечивается жесткая поддержка типов и структур данных, реализованная в определениях таблиц. Кроме этого, системный каталог реляционной базы данных содержит метаданные или «данные о данных». При помощи запросов к системному каталогу можно узнать структуру базы данных, включая информацию о типах данных ее столбцов, наборе столбцов таблиц и отношениях между таблицами.

Что касается XML-документов, то они, напротив, сами по себе содержат очень мало метаданных. Единственными реальными данными о структуре, содержащимися в них, являются имена элементов и атрибутов, с указанием вложений одних элементов в другие, отражающие иерархические взаимосвязи.

При этом XML-документ может строго соответствовать стандартам и все же иметь довольно необычную структуру. Например, ничто не мешает такому XML-документу содержать именованный элемент с текстовыми данными в одном экземпляре и вложенными элементами в другом, или же содержать именованный атрибут с целочисленным значением для одного элемента и датой для другого. Очевидно, что XML-документы, несмотря на строгое соответствие стандарту, не представляет данные, которые легко переносятся в базу данных и из нее.

Поэтому при использовании XML для хранения программно обрабатываемых данных необходима более основательная поддержка типов данных и их структуры. Привнесение в среду Web метаданных, описывающих свойства поддерживаемых в ней информационных ресурсов, является одной из важнейших целей создания платформы XML. Речь идет, прежде всего, об описании структуры XML-документов и их смыслового содержания (семантики). Необходимость решения этой задачи аргументируется стремлением к получению возможностей автоматической проверки правильности структуры XML-документов. Имеется в виду, что при наличии явного описания структуры документов проверку их правильности может осуществлять браузер.

Однако чаще всего не учитывается еще одно важное назначение метаданных, описывающих информационные ресурсы Web. Метаданные необходимы для создания принципиально новых высокоуровневых приложений Web, обеспечивающих интеграцию неоднородных информационных ресурсов.

7.3.1. Структурирование данных и схема XML

Из вышеизложенного следует, что поддержка данных о структуре данных является важной целью языка XML, и практически любой его аспект направлен на то, чтобы упростить восприятие содержимого документа. Но как же решается задача определения метаданных в XML?

Любая задача структурирования связана с построением модели данных. В качестве модели XML-документа выбирается некий шаблон, определяющий тип используемой в документе информации и ее структуру. Такие шаблоны называют XML *схемами* и используют для описания класса данных. Задав класс с использованием схемы, вы получаете возможность создавать хорошо структурированные документы, которые можно проверить на допустимость.

В первое время для этой цели использовалась спецификация DTD (Document Type Definition), но фирма Microsoft разработала новый подход, который обладает значительными возможностями для описания типов данных. XML Schema задает структуру документа при помощи особого словаря; другими словами, сама схема документа представляет собой XML-документ, который включает в себя теги и атрибуты. Поэтому консорциум W3C рекомендовал описывать структуру документов XML на языке XSD.

Язык XSD различает простые и сложные элементы XML. *Простыми (simple)* элементами описываемого документа XML считаются элементы, не содержащие атрибутов и вложенных элементов. *Сложные (complex)* элементы содержат атрибуты и/или вложенные элементы. Схема XML описывает *простые типы* – типы простых элементов, и *сложные типы* – типы сложных элементов.

На **Ошибка! Источник ссылки не найден.** показана схема XML для документа-заказа, приведенного на Рис. 8.2. . Даже этот простой пример показывает основательную поддержку типизации данных в схемах XML; элементы и атрибуты имеют типы данных, очень похожие на типы данных SQL. Кроме того, схема, приведенная на **Ошибка! Источник ссылки не найден.**, сама является XML-документом, и поэтому человеку, знакомому с основами XML, прочитать ее легче, чем DTD-определение.

Возвращаясь к выполняемой XML функции разметки, следует еще раз подчеркнуть, что он (в отличие от HTML) не является полнофункциональным языком, который должен решать все задачи представления, поддержки и обработки

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema">
3      <element name="purchaseOrder" type="P0Type"/>
4      <complexType name="POType">
5          <sequence>
6              <element name="id_cln" type="integer"/>
7              <element name="id_order" type="integer"/>
8              <element name="date_order" type="date"/>
9              <element name="id_slzh" type="integer" length="3"/>
10             <element name="orderItem" min0ccurs="0" maxOccurs-"unbounded">
11                 <complexType>
12                     <sequence>
13                         <element name="id_mfr" type="string" length="3"/>
14                         <element name="id_prd" type="string"/>
15                         <element name="count" type="integer"/>
16                         <element name="price_all" type="decimal" fractionDigits="2"/>
17                     </sequence>
18                 </complexType>
19             </element>
20         </sequence>
21     </complexType>
22 </schema>
```

Рис. 7.2. XML-документ, содержащий расширенный заказ товаров

информационных ресурсов Web.

Если проводить аналогию с технологиями баз данных, то XML можно квалифицировать как язык определения данных. Специфика XML как языка определения данных заключается в том, что в нем сочетаются возможности описания свойств экземпляров элементов XML-документов, составляющих содержание данного конкретного документа, с возможностями определения свойств типа XML-документов (XML Schema) в терминах типов элементов этих документов.

1. Первая группа средств (теги разметки) используется по принципу самоописываемости, определяя некоторые свойства элементов конкретного документа с помощью встраиваемых в него тегов разметки.

2. Что касается XML Schema, то она описывает типовые свойства элементов документа и свойства типов документов в целом.

Роль XML Schema аналогична роли схемы базы данных. Экземпляр XML можно сравнить с хранением данных на физическом диске в СУБД, а XML-схему – со схемой языка описания SQL-данных. При этом XML Schema отчуждается от описываемых документов и хранится где-либо в Web. Конкретные XML-документы ссылаются на это определение, хотя они могут и включать его непосредственно в явном виде.

7.3.2. Типы данных в схеме XML

С точки зрения базы данных основательная поддержка типов и структур данных является одним из главных достоинств XML Схем. В спецификации XML Schema определено более 30 встроенных типов данных, очень похожих на типы данных SQL. Наиболее важные с точки зрения баз данных типы данных XML Схем перечислены в таблице 7.1.

Если сравнить типы данных XML Схемы с типами данных баз данных, приведенных в таблице 3.1, то можно сделать вывод, что все они могут быть взаимно обратно преобразованы друг в друга без потери информации. Именно это обстоятельство является важным качеством XML-формата.

Таблица 7.1

Типы данных в XML Схеме

Тип данных XML Схемы	**Описание**
Числовые данные	
Integer	Целое число
PositiveInteger	Положительное целое число
NegativeInteger	Отрицательное целое число
NonPositiveInteger	Нуль или отрицательное целое число
NonNegativeInteger	Нуль или положительное целое число
Int	32-битовое целое число со знаком
UnsignedInt	32-битовое целое число без знака
Long	64-битовое целое число со знаком
UnsignedLong	64-битовое целое число без знака
Short	16-битовое целое число со знаком
UnsignedShort	16-битовое целое число без знака

Тип данных XML Схемы	Описание
Decimal	Число, содержащее дробную часть
Float	Число с плавающей запятой стандартной точности
Double	Число с плавающей запятой двойной точности
Символьные данные	
String	Символьная строка переменной длины
NormalizedString	Строка, в которой символы новой строки, возврата каретки и табуляции заменены пробелами
Token	Строка, обработанная как NormalizedString, в которой удалены начальные и конечные пробелы и подряд идущие пробелы заменены одним пробелом
Дата и время	
Time	Время дня (часы/минуты/секунды/миллисекунды)
DateTime	День и время (эквивалент SQL-типа TIMESTAMP)
Duration	Длительность временного интервала (эквивалент SQL-типа DURATION)
Date	Год/месяц/день
Gmonth	Месяц по григорианскому календарю (от 1 до 12)
Gyear	Год по григорианскому календарю (от 0000 до 9999)
Gday	День месяца по григорианскому календарю (от 1 до 31)
GmonthDay	Месяц/день по григорианскому календарю
Другие данные	
Boolean	Значение TRUE/FALSE
Byte	Один байт данных со знаковым битом
UnsignedByte	Один байт данных без знакового бита
base64Binary	Двоичные данные по основанию 64
HexBinary	Двоичные данные по основанию 16
AnyURI	URI-адрес в Интернете, например, http://www.w3.org
Language	Допустимый язык XML (английский, французский ...)

Так же как стандарты `SQL2` и `SQL3`, `XML Schema` поддерживает пользовательские типы данных, производные от встроенных типов или других пользовательских типов данных. Пользовательский тип данных определяется как ограничение, накладываемое на другой тип данных `XML`.

Ниже приведено определение производного типа данных `repNumType`, ограничивающего допустимые коды служащих диапазоном значений от `101` до `199`.

```
2  <simpleType name="repNumType">
3    <restriction base ="integer">
4      <minInclusive value="101" />
5      <maxExclusive value="200" />
6    </restriction>
7  </simpleType>
```

С таким определением типа данных можно объявлять сущности и атрибуты схемы как относящиеся к типу `repNumType`, и для них будет автоматически применяться заданное вами ограничение.

Спецификация XML Schema предоставляет богатый набор характеристик типов данных (называемых *аспектами),* которые можно использовать в ограничениях. Это длина данных (для строк и двоичных данных), включающие и исключающие диапазоны значений, количество цифр целой и дробной части (для числовых данных) и явные перечни допустимых значений. Имеется даже встроенная возможность проверки на соответствие шаблону, позволяющая ограничить набор допустимых значений данных при помощи специального синтаксиса.

Еще XML Schema позволяет определять сложные типы данных, то есть пользовательские структуры. Вот, например, определение сложного типа данных custAddrType, составленного из вложенных элементов стандартных типов:

```
2  <complexType name="custAddrType">
3    <sequence>
4      <element name="city" type="string"/>
5      <element name-"street" type="string"/>
6      <element name="state" type="string"/>
7      <element name-"postcode" type="integer"/>
8    </sequence>
9  </complexType>
```

Можно также создать пользовательский тип данных, представляющий собой список элементов данных другого типа. Вот, в частности, определение сложного типа repListType, представляющего собой список кодов служащих:

```
2  <simpleType name="repListType">
3    <list itemType="repNumType"/>
4  </simpleType>
```

Обычно в определении любого языка содержится описание типов и структур данных. Информация о типе указывает характер размещения данных в двоичной памяти, выделяемой ОС в ходе выполнения этой программы. Язык программирования воспринимает данные через собственный набор типов данных, которые, по сути, являются абстракцией двоичного кода, используемого для физического хранения и манипулирования битами и байтами.

Поэтому программные средства используют только собственный, уникальный механизм для формирования, передачи и хранения данных, как будто не существует других языков программирования или словно эта программная система решает многолетнюю проблему абстрагирования данных раз и навсегда.

XML делает следующий шаг, который заключается в том, что программы должны уметь отображать свои форматы данных лишь в XML и обратно. Этот шаг представляет собой значительный сдвиг в вопросе восприятия данных приложениями, особенно это касается общих данных, совместно используемых программами и приложениями различных типов.

XML хранит все данные в виде текста, как и положено языку разметки. Программы, обращающиеся к XML, отображают эти данные в свое представление и обратно в текстовое, используя сведения о связи типов. Поскольку такие сведения хранятся отдельно, допускается множественный доступ с возможностью независимого изменения этих данных.

Таким образом, XML-процессоры осуществляют преобразование данных приложений в данные XML и обратно, практически так же, как браузеры производят HTML-разметку текста. Различие заключается в том, что в данном случае целью преобразования является не графический интерфейс пользователя, а файл данных или программа.

XML-процессоры находят соответствие имен элементов в файле схемы с именами элементов в файле данных и применяют сведения о типе и структуре. XML-процессоры также должны «понимать» специальные элементы схемы, касающиеся интерпретации данных, по таким вопросам, как упорядочивание данных для передачи по HTTP (сериализация) или порядок отображения SOAP-сообщения на определенный метод объекта.

Отображение любых значений в текстовую форму и обратно является неэффективным действием в плане как использования пространства памяти, так и в отношении скорости обработки. Но нередко производительность является «наименьшим злом» по сравнению с нереализованными возможностями. И в этом случае, поскольку язык XML предлагает выход для важнейшей, ранее неразрешимой проблемы, производительность отступает на второй план.

7.3.3. Элементы и атрибуты в XML Схеме

Помимо богатых возможностей для определения типов данных XML Schema включает богатый словарь для определения структуры документа и разрешенных элементов и атрибутов.

– Простое содержимое. Элемент содержит только текст (хотя, как говорилось в предыдущем параграфе, текст можно ограничить данными отдельного типа, такими как дата или числовое значение). Содержимое этого типа определяется при помощи элемента simpleContent.

– Только элементы. Элемент содержит только вложенные элементы. Содержимое этого типа определяется при помощи элемента complexType.

– Смешанное содержимое. Элемент может содержать и текстовое содержимое, и вложенные элементы. XML Schema требует, чтобы последовательность элементов и текстового содержимого была строго определена, и допустимые документы должны соответствовать этой последовательности.

– Пустое содержимое. Элемент содержит только атрибуты и никакого текстового содержимого. XML Schema интерпретирует такие элементы как особый случай содержимого типа «только элементы» без объявленных элементов.

– Любое содержимое. Элемент может быть пустым, содержать вложенные элементы и/или текст. Содержимое этого типа определяется при помощи элемента anyType.

Эти базовые типы элементов могут задаваться в объявлениях элементов схемы. Кроме того, можно указать, что элемент может встречаться в документе несколько раз, и задать минимальное и максимальное количество вхождений. Подобно SQL, XML Schema поддерживает значение элементов NULL, указывающее, что содержимое элемента неизвестно. В терминологии XML это значение называется nil, но смысл его тот же самый. Поддержка этого значения упрощает

перенос данных между XML и столбцами баз данных, которые могут содержать значения NULL.

Схема XML позволяет определить логическую группу элементов, которые, как правило, используются совместно, и задать для этой группы собственное имя. Данную группу можно включать в последующие объявления элементов как единое целое. Группировка элементов делает их структуру еще более гибкой. Группа может определять *последовательность* элементов, которые должны обязательно присутствовать в документе в заданном порядке. Или же она может определять набор элементов, из которого в документе обязательно должен присутствовать только один элемент.

Аналогичные возможности имеются и для управления атрибутами. Отдельный атрибут можно определить как обязательный или необязательный. Можно задать значение атрибута по умолчанию, которое будет использоваться в том случае, если значение этого атрибута не задано. Можно задать фиксированное значение атрибута, то есть неизменное значение для этого атрибута. Можно определить имя группы атрибутов, которые всегда используются вместе, после этого для определения группы атрибутов для очередного элемента схемы достаточно задать только имя этой группы.

7.3.4. Пространство имен

Поскольку возможно существование множества схем, поэтому во избежание совпадения имен тэгов в XML вводятся *пространства имен* тэгов, используемых для хранения *словарей* XML – наборов определений типов данных и структур, используемых для различных целей. В большой организации может быть полезно определить стандартизированное XML-представление для основных дел объектов, таких как адреса, номера товаров, коды клиентов и т. п., и держать их в общем хранилище. Полезны и определения более высокого уровня, описывающие такие документы, как заказы, заявления на отпуск и т. п., для совместного использования их обычно объединяют в группы.

Пространство имени указывается перед именем тэга: `<namespace:tag/>`. Пространство имен может быть подключено в любом тэге документа XML, напр.: `<ntb:notebook xmlns:ntb = «http://some.firm.com/2003/ntbml»>`. В качестве имени пространства имен рекомендовано указывать некоторый URL. Но это вовсе не означает, что описание пространства имен находится по данному адресу. *Данный сайт может вообще не существовать, никаких обращений по этому адресу не будет.* Использование URL рекомендовано лишь для того, чтобы обеспечить уникальность именования пространств имен. Поэтому в отношении пространств имен важно иметь в виду, что:

– строка в определении пространства имен является только строкой. *Да, эти строки выглядят как URL, но ими не являются*. Вы можете определить `xmlns:addr="mike"`, и это также будет работать;

– только одно важно в отношении строки пространства имен: она должна быть уникальной; вот почему большинство пространств имен выглядят как URL;

– XML-парсер не обращается к http://www.zyx.com/books/, чтобы найти схему, он просто использует этот текст как строку. Это несколько сбивает с толку, но именно так работают пространства имен.

Если пространство имен явно не указывается для тэга, то используется пространство имен по умолчанию. Это пространство имен можно переопределить:

```
2  <ntb:notebook xmlns:ntb = "http://some.firm.com/2003/ntbml" xmlns = "http://another.firm.com/xmlns">
3
```

Если XML-схема содержит определения, состоящие из более чем одного пространства имен XML, существует потенциальная возможность конфликтов имен. В двух пространствах имен одно и то же имя может представлять две совершенно разные структуры XML или два разных типа данных. Во избежание неоднозначности на типы данных и структуры XML можно ссылаться при помощи *уточненных* имен, используя технологию, похожую на уточнение имен столбцов в SQL.

Каждому пространству имен, заданному в заголовке схемы, может быть присвоен *префикс,* который далее в документе используется для уточнения ссылок на элементы из этого пространства имен. В примерах этой главы, чтобы не загромождать их лишней информацией, мы не использовали префиксов. Вот заголовок схемы и фрагмент тела схемы, в котором основное пространство имен XML-схемы, поддерживаемое W3C, и пространство имен компании идентифицируются префиксами:

```
2  <schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3      xmlns:corp=:"http://www.mycompany.com/schemas/purchasing">
4      <complexType name="purchaseOrderType">
5        <element name="orderDate" type="xsd:date" />
6        <element name="billAddr" type="corp:custAddrType" />
7        <element name="shipAddr" type="corp:custAddrType" />
8        <element name="repNums" type="corp:repListType" nillable="true" />
9
```

В этом примере пространство имен компании идентифицируется префиксом corp, а основное пространство имен XML – префиксом xsd. Все ссылки на типы данных уточняются одним из этих префиксов, благодаря чему они абсолютно однозначны.

7.4. СТИЛИ И ФОРМАТИРОВАНИЕ ДАННЫХ XML

Создатели языка XML стремились отделить содержимое документа не только от его структуры, но и от форматирования, т. е. от данных, определяющих его внешний вид. Поскольку информация XML-документа не указывает на то, как она будет отображена на экране, то дополнительно определяется таблица стилей, с помощью которой документу придается желаемый внешний вид. *Таблицей стилей* называется специальный документ, содержащий список стилей, которые применяются к информации XML-документа, а процесс описания того, как будет визуально представлено его содержимое называется *стилизацией* XML-документа.

В качестве примера рассмотрим телевизионный прогноз погоды. На экране появляется разноцветная карта, на которую нанесены изображения дождя, вет-

ра, солнца и т. д. Такая карта основана на наборах цифр, характеризующих состояние погоды в различных географических точках. Специальная программа обрабатывает эти цифры и создает изображение. Другими словами, цифры, или данные, стилизуются так, чтобы их легко было воспринимать. Представьте себе, что данные о погоде хранятся в XML-документе, тогда назначение таблицы стилей – определять внешний вид карты.

Итак, таблица стилей переводит информацию XML-документа в некоторую форму, которая может быть визуально представлена. Если, к примеру, документ XML просматривается с помощью Web-браузера, то таблица стилей должна создать подходящий HTML-документ.

7.4.1. Основы XSL

Технология таблиц стилей, которая преобразует или трансформирует документы XML в другие форматы, представлена спецификацией XSL. Основная идея этой технологии состоит в том, что трансформированный документ можно открыть для просмотра в определенной программе, например, в Web-браузере. Следует иметь в виду, что XSL не ограничивается преобразованием XML-документов в формат HTML, а может перевести документ из XML в любой другой язык разметки, поскольку обладает высокой степенью обобщенности.

Спецификация XSL предназначена для преобразования XML-документа. Схема выполнения такого преобразования показана на Рис. 7.3. . Преобразованием управляет таблица стилей, в которой указано, какие элементы входного XML-документа необходимо преобразовать и как они должны объединяться с другими элементами для получения выходного XML-документа. Одним из часто используемых применений XSL является преобразование одной общей версии Web-страницы в различные формы, предназначенные для вывода на экранах разных типов.

Как и документы HTML, любая таблица стилей XSL полезна лишь тогда, когда ее содержимое может быть обработано. Обработка таблицы стилей может выполняться целым рядом программ, предназначенных для работы с XML, например, Web-браузерами. Сейчас нам необходимо разобраться в том, как таблица стилей и XML-документ связаны друг с другом и как осуществляется преобразование документа с использованием таблицы стилей.

XML-документ представляет собой древовидную структуру, растущую сверху вниз. Вершиной дерева является корневой элемент. Все элементы, расположенные ниже корня, называются ветвями дерева. Важность корневого элемента заключается в том, что он определяет стартовую точку для XSL-процессора. *XSL-процессор* – это приложение, обрабатывающее таблицу стилей XSL и использующее ее для трансформации данных XML, например, в HTML-документ. Как правило, вам придется иметь дело с XSL-процессорами, встроенными в Web-браузеры.

Входной XML-документ

Zakaz.xml *

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl"
3  href="file:///F:/Zakaz.xslt"?>
4  <purchaseOrder
5  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6  xsi:noNamespaceSchemaLocation="F:\Zakaz.xsd">
7      <id_cln>12117</id_cln>
8      <id_order>312961</id_order>
9      <date_order>1989-12-17</date_order>
10     <id_slzh>2106</id_slzh>
11     <terms ship="ground" bill="Net30"/>
12     <orderItem>
13         <id_mfr>УАЗ</id_mfr>
14         <id_prd>2A44L</id_prd>
15         <count>7</count>
16         <price_all>31500.00</price_all>
17     </orderItem>
18     <orderItem>
19         <id_mfr>УПЗ</id_mfr>
20         <id_prd>41003</id_prd>
21         <count>1</count>
22         <price_all>652.00</price_all>
23     </orderItem>
24 </purchaseOrder>
25
```

Text | Grid | Schema | WSDL | XBRL | Authentic

Таблица стилей XSL

Zakaz.xslt *

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xsl:stylesheet version="2.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:fn="http://www.w3.org/2005/xpath-functions"
5   xmlns:xdt="http://www.w3.org/2005/xpath-datatypes"
6   xmlns:xs="http://www.w3.org/2001/XMLSchema"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8    xmlns:altova="http://www.altova.com">
9       <xsl:output version="4.0" method="html" indent="no"
10      encoding="UTF-8"
11      doctype-public="-//W3C//DTD HTML 4.0 Transitional//EN"
12      doctype-system="http://www.w3.org/TR/html4/loose.dtd"/>
13      <xsl:param name="SV_OutputFormat" select="'HTML'"/>
14      <xsl:variable name="XML" select="/"/>
15      <xsl:decimal-format name="format1"
16      grouping-separator=" " decimal-separator=","/>
17      <xsl:import-schema schema-location="F:\Zakaz.xsd"/>
18      <xsl:template match="/">
19          <html>
20              <head>
21                  <title/>
22              </head>
23              <body>
24                  <xsl:for-each select="$XML">
25                      <br/>
26                      <table border="0" width="500">
27                          <tbody>
28                              <tr>
29                                  <td align="center">
30                                      <span style="font-weight:bold; ">
31                                          <xsl:text>Накладная на отпуск

202     <xsl:template name="DateToMonthNum">
203         <xsl:param name="sDate"/>
204         <xsl:value-of select="number(substring($sDate, 6, 2))"/>
205     </xsl:template>
206 </xsl:stylesheet>
207
```

Text | Grid | Schema | WSDL | XBRL | Authentic | Browser

Программа обработки XSL (XSL-процессор), встроенная в Web-браузер и преобразующая XML-файл в HTML-файл для отображения

F:\SPS1.html - Microsoft Internet Explorer

Файл Правка Вид Избранное Сервис Справ

Назад Поиск

Адрес: F:\SPS1.html Переход Ссылки

Накладная на отпуск товаров

Номер клиента: 12117
Номер заказа: 312961
Дата заказа: 17 декабря 1989
Номер служащего: 2106

Отпускаемые товары:

Производитель	Товар	Число	Стоимость
УАЗ	2A44L	7	31500.00
УПЗ	41003	1	652.00

Мой компьютер

Выходной документ

Рис. 7.3. Преобразование документа XML-файла в оформленный документ при помощи таблицы стилей XSL

Обрабатывая таблицу стилей, XSL-процессор ищет шаблоны, описывающие определенные последовательности XML-документа. Фрагменты информации, подлежащие преобразованию, выделяются XSL-процессором на основе проверки того, соответствует ли информация некоторой последовательности. Последовательностью, к примеру, может являться имя элемента. Каждый раз, когда в процессе обработки XSL-процессор встречает это имя, он применяет соответст-

вующий шаблон, трансформируя тем самым данные. Проверка на соответствие последовательности начинается с корневого элемента и постепенно распространяется на весь документ.

Когда все последовательности найдены и к ним применены соответствующие шаблоны, вы получаете полностью трансформированный XML-документ. Если вы преобразовали данные в формат HTML, то содержимое документа теперь доступно для просмотра в Web-браузере. При открытии документа XML в браузере его таблица стилей автоматически обрабатывается, в результате чего в окне браузера появляется сгенерированный HTML-документ. Весь описанный процесс происходит совершенно незаметно для пользователя.

7.4.2. Структура таблицы стилей XSL

Общая структура таблицы стилей довольно проста: в ее состав входят последовательности и шаблоны. Здесь мы увидим, каким образом эти компоненты используются для визуализации XML-документов.

Таблица стилей XSL содержит обязательный корневой элемент с названием stylesheet. Этот элемент наряду с набором других элементов и свойств XSL является частью словаря XSLT. Чтобы использовать содержимое XSLT, необходимо сначала объявить пространство имен, в которое оно будет помещено. Ниже показано, как с помощью элемента stylesheet объявляется пространство имен XSL:

```
2  <xsl:stylesheet ...
3      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4      ...
5  />
```

Данный код делает доступными все элементы и атрибуты пространства имен XSL и назначает им префикс xsl. Это действие стандартно для всех таблиц стилей; его смысл поясняется в следующих подразделах, где мы рассмотрим кодирование последовательностей и шаблонов.

Последовательности

При обработке таблицы стилей XSL поиск данных для трансформации осуществляется с помощью последовательностей. Говоря точнее, последовательность идентифицирует элемент или атрибут XML-документа, являющийся ветвью его древовидной структуры.

Последовательности задаются весьма просто. Их можно сравнить путями файловой системы: если пути указывают на папки и файлы вашего жесткого диска, то последовательности определяют элементы и атрибуты документа. Так, к примеру, элемент head, вложенный в элемент html любого HTML-документа, идентифицируется как html/head.

Назначение последовательностей – выделять фрагменты XML-документов, подлежащие трансформации. Когда XSL-процессор обнаруживает некоторые данные, соответствующие последовательности он передает их для выполнения трансформации шаблону.

Шаблоны

Шаблоны – часть таблицы стилей, предназначенная для трансформации данных. Когда XSL-процессор обнаруживает последовательность в документе, он пропускает ее через шаблон и трансформирует. Количество шаблонов таблицы стилей не ограничено. Если в таблице стилей имеется несколько шаблонов, то каждый выполняет трансформацию определенного фрагмента XML-документа.

Поскольку XSL представляет собой XSL-словарь, таблицы стилей закодированы с помощью XML. Шаблон задается элементом xsl:template, а последовательность, связанная с ним, – атрибутом match элемента xsl:template.

Следующий пример демонстрирует использование шаблона для корневого элемента документа:

```
2  <xsl:template match="/">
```

Этот шаблон будет осуществлять трансформацию всего документа целиком, начиная с корневого элемента. Как правило, вам необходимо будет создавать шаблоны для элементов, находящихся ниже корневого.

Например, шаблон для элемента title XML-документа, приведенного на Рис. 7.1. , будет выглядеть так:

```
2  <xsl:template match="book/bookPart/chapter/title">
3  </xsl:template>
```

Как видим, элемент title задан с перечислением всех его родительских элементов. Обратите внимание на закрывающий тег </xsl:template>, обязательный для всех шаблонов. Чтобы задать преобразование для данных находящихся внутри элемента title, следует поместить тег <xsl:value-of/> внутри шаблона, как показывает следующий код.

```
2  <xsl:template match="book/bookPart/chapter/title">
3      <b>Глава</b>
4      <xsl:value-of/>
5  </xsl:template>
```

Здесь преобразование заключается в том, что перед данными элемента title будет помещен текст **Глава**, выделенный полужирным шрифтом. Элемент <xsl:value-of/> представляет содержимое элемента title. Кроме <xsl:value-of/> существует еще несколько элементов, которые часто используются при создании шаблонов и входят в стандартное пространство имен XSL:

xsl:value-of – вставляет содержимое элемента (атрибута) XML;
xsl:if – задает условное соответствие для шаблонов;
xsl:for-each – создает цикл для элементов документа XML;
xsl:apply-templates – применяет шаблон к XML-документу.

ГЛАВА 8. SQL И XML

Тесная связь Web-технологий с технологиями баз данных сложилась еще на ранних этапах развития сети Интернет. Она сводилась к обеспечению теледоступа к системам баз данных через среду Web. В настоящее время создано и функционирует огромное количество приложений такого рода в самых различных областях деятельности.

Однако до появления технологии XML не удавалось обеспечить реальную интеграцию информационных ресурсов Web и баз данных. Система базы данных выступали здесь по отношению к Web как «черный ящик». Только с развитием технологии XML стали проявляться более глубокие связи между этими двумя направлениями информационных технологий.

Стремление к обеспечению в Web полноценных возможностей управления данными, поддерживаемыми в этой среде в рамках XML-технологий, объективно привело к необходимости использования подходов и принципов, аналогичных тем, которые на протяжении десятилетий прошли испытание временем в технологиях баз данных.

В результате использования этих подходов и принципов в Web-технологиях в лексиконе спецификаций стандартов платформы XML появились такие ключевые термины технологий баз данных, как модель данных, схема, ограничение целостности, язык запросов.

Со временем эта тенденция привела к тому, что было создано несколько коммерческих компаний, занявшихся разработкой баз данных на основе XML. Данные в этих базах данных хранились в виде XML-документов либо непосредственно в текстовом виде.

Производители баз данных формата XML высказывают в пользу своих продуктов те же аргументы, которые в свое время приводили производители объектно-ориентированных баз данных.

1. Поскольку огромное количество внешних данных представлено в формате XML, в базах данных удобнее всего использовать этот же формат и соответствующую модель данных.

2. Так как все большее количество пользователей осваивает HTML и XML, базы данных XML-формата также доступны для пользователей, как и реляционные базы данных SQL-типа.

На сегодняшний день базы данных XML-формата являются пока новым направлением рынка СУБД, и время покажет, будут ли они иметь успех. Однако история развития строго объектно-ориентированных баз данных показала, что производители реляционных СУБД способны достаточно быстро расширять свои продукты, включая в них важнейшие элементы новых моделей данных, благодаря чему их продукты сохраняют доминирующую роль в области обработки данных.

Если сервер СУБД выполняет множество дополнительных функций по ведению базы данных (поддерживает транзакции, блокирует таблицу или запись от конфликтных изменений, сохраняет ее целостность, выполняет различные действия по оптимизации запросов), то работа с XML-файлами таких возможностей не дает. При этом надо учитывать, что полная открытость XML-файлов делает их беззащитными от внешнего просмотра, поэтому вряд ли разумно хранить в них конфиденциальную информацию.

Основной недостаток использования XML-файлов в качестве базы данных заключается в том, что организовать корректную работу множества пользователей с одним файлом практически невозможно. Как только одна из клиентских программ начинает модифицировать такой файл-базу, все остальные пользователи будут либо ждать окончания этого процесса, либо пытаться одновременно внести в файл противоречивые данные, модифицированные разными пользователями.

Поэтому лучше всего задействовать XML-файлы в интеграционных приложениях, когда данные из одних баз и систем передаются во временное хранилище. При этом интеграция реляционных СУБД с XML будет возрастать и реляционные продукты будут включать все больше XML-ориентированных функций.

8.1. XML КАК СРЕДСТВО ПРЕДСТАВЛЕНИЯ СТРУКТУРИРОВАННЫХ ДАННЫХ

8.1.1. Представление структурированных данных в XML

Хотя XML разрабатывался, прежде всего, для представления и обработки документов, он может быть полезен для представления структурированных данных, которые обычно представляются в базах данных. На **Ошибка! Источник ссылки не найден.** показан типичный XML-документ из сферы обработки данных – очень простой заказ товаров. По содержанию он сильно отличается от документа, показанного на Рис. 7.1. , но содержит те же основные компоненты. Вместо `<book>` элементом верхнего уровня является `<purchaseOrder>`. Он содержит, подобно элементу `<book>`, вложенные элементы – `<customerNum>`, `<orderNum>`, `<orderDate>` и `<orderItem>`. Элемент `<orderItem>` тоже содержит вложенные элементы. На **Ошибка! Источник ссылки не найден.** с заказом связано несколько условий приобретения товара, представленных атрибутами элемента `<terms>`. Атрибут `ship` определяет, как будет доставлен заказ, а атрибут `bill` – условия его кредитования.

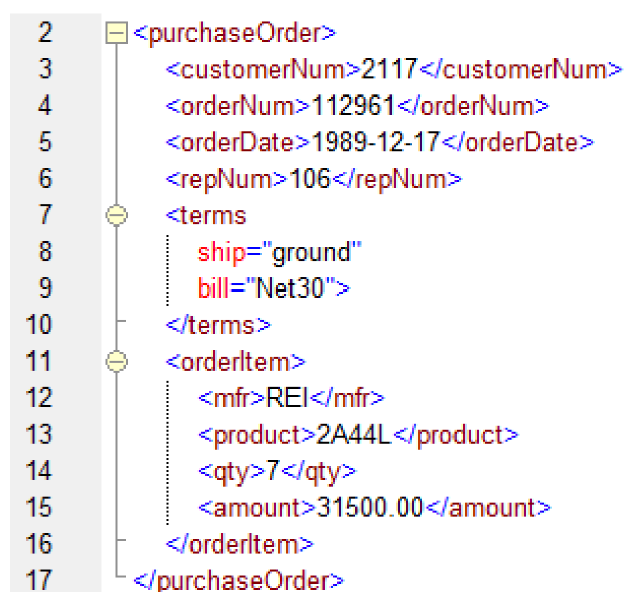


Рис. 8.1. XML-документ, содержащий расширенный заказ товаров

Очевидно, что этот документ-заказ тесно связан с таблицей ZAKAZY из нашей базы данных. Вы можете сравнить его со структурой таблицы ZAKAZY, приведенной в Приложении (см. Рисунок П.1 и Таблицу ZAKAZY). Элементы нижнего уровня за исключением элемента <terms> соответствуют столбцам таблицы ZAKAZY. Элемент верхнего уровня представляет строку таблицы.

Преобразование документов описанного типа в наборы строк таблицы ZAKAZY представляет собой механическую процедуру, поэтому его может выполнять простая компьютерная программа.

В отличие от таблицы ZAKAZY приведенный XML-документ содержит еще один, средний уровень иерархии, группирующий информацию о заказанном товаре – код производителя (mfr), код товара (product), количество товара (qty) и сумму заказа (amount). В реальном заказе, содержащем несколько позиций, эта группа элементов данных может повторяться несколько раз. XML-документ легко расширить для поддержки такой структуры данных, добавив еще один или несколько элементов <orderItem>.

Базу данных нельзя так легко расширить. Для поддержки многострочных заказов таблицу ZAKAZY, скорее всего, придется разбить на две: в одной будет храниться заголовок заказа (номер заказа (orderNum), дата заказа (orderDate), код клиента (customerNum) и т. п.), а в другой – соответствующие строки заказа: код производителя (mfr), код товара (product), количество товара (qty) и сумму заказа (amount).

8.1.2. Сравнение XML и SQL

Сходства XML и SQL. Поскольку язык XML произошел от SGML, он обладает рядом полезных характеристик, сближающих его с языком SQL.

– *Описательный подход*. В XML принят такой подход к определению структуры документов, при котором задается структура и содержимое каждого элемента до-

кумента, а не то, как он должен обрабатываться. Этот же подход используется и в SQL, ориентированном на определение запрашиваемых данных, а не на то, как они должны извлекаться.

– *Строительные блоки*. XML-документ состоит из небольшого количества базовых строительных блоков, в число которых входят такие фундаментальные компоненты, как *элементы* и *атрибуты*. В SQL этим понятиям соответствуют таблицы и столбцы.

– *Типы документов*. В XML каждый документ воспринимается не сам по себе, а в качестве представителя определенного типа, соответствующего документам реального мира, например заказ, ответ на деловое письмо или резюме кандидата на должность. И здесь тоже очевидна параллель с SQL, поскольку таблицы также представляют типы сущностей реального мира.

Различия XML и SQL. Однако несмотря на очевидные параллели между XML и SQL, между ними есть и значительные различия.

– *Ориентация на документы или на данные*. Базовые концепции XML сформированы на основе структуры типовых документов. Это язык с ориентацией на текст, и в нем строго различается содержимое (элементы документа) и характеристики этого содержимого (атрибуты). В то же время базовые концепции SQL сформированы на основе структур, типичных для обработки данных. SQL ориентирован на данные, поддерживает широкий диапазон типов данных (в двоичном представлении), и его структуры (таблицы и столбцы) ориентированы на содержимое (данные). Это расхождение между фундаментальными моделями SQL и XML приводит к некоторым сложностям их совместного использования.

– *Иерархическая или табличная структура*. Естественные структуры XML имеют иерархическую природу и отражают иерархию элементов большинства распространенных типов документов (например, книга содержит главы, главы включают параграфы, параграфы содержат заголовки, абзацы и рисунки). Эти структуры не являются жесткими. Например, один параграф содержит пять абзацев и один рисунок, а в следующем параграфе будет три абзаца и два рисунка и т. д. В противоположность этому структуры SQL имеют табличную, а не иерархическую организацию. Более того, это жесткие структуры – все строки таблицы SQL содержат одинаковый набор столбцов в одинаковом порядке. Эти отличия также затрудняют совместное использование SQL и XML.

– *Объекты или операции*. Основной задачей языка XML является *представление* объектов. Если выделить осмысленный фрагмент текста XML и спросить, что он представляет, выяснится, что представляет он какой-нибудь объект: абзац, заказ товаров, адрес клиента и т. п. У языка SQL более широкие задачи, но в первую очередь он ориентирован на *обработку объектов.* Если выделить осмысленный фрагмент текста SQL и спросить, что он представляет, выяснится, что он представляет *операцию* над объектом: создание объекта, удаление объекта, поиск одного и/или более объектов, либо обновление содержимого объекта. Эти отличия делают назначение и использование языков XML и SQL взаимодополняющими.

8.2. ИСПОЛЬЗОВАНИЕ XML С БАЗАМИ ДАННЫХ

Как уже было отмечено, стремительный рост популярности XML привел к тому, что производители баз данных стали включать его поддержку в свои продукты. Формы поддержки XML различаются, но все их можно условно разделить на пять следующих категорий.

– *Хранение данных в формате XML*. Реляционные базы данных могут принимать XML-документ как символьную строку переменой длины (VARCHAR) или данные большого символьного объекта (CLOB). Вэтом случае XML-документ является содержимым одного столбца одной строки базы данных. При усиленной поддержке XML, по сравнению с этим элементарным уровнем СУБД может позволять явно объявлять столбцы как относящиеся к типу данных XML.

– *Вывод в формате XML*. Данные одной или более строк результата запроса легко представить в виде XML-документа. Поддержка выходных данных в формате XML означает, что в ответ на SQL-запрос СУБД вместо обычного набора строк и столбцов может генерировать XML-документ.

– *Ввод в формате XML*. XML-документ может содержать данные, предназначенные для вставки в одну или более новых строк таблицы базы данных, или же в нем могут содержаться данные, предназначенные для обновления строки таблицы, либо данные, идентифицирующие удаляемую строку. Поддержка входных данных в формате XML означает, что вместо SQL-запросов СУБД может принимать в качестве входных данных XML-документы.

– *Обмен данными в формате XML*. XML представляет собой очень удобный и естественный способ выражения данных для обмена данными между разными СУБД или серверами баз данных. Данные исходной базы данных преобразуются в XML-документ и направляются в принимающую базу данных, где они вновь преобразуются в формат базы данных.

– *Интеграция данных XML*. Это более высокий уровень поддержки интегрированного хранения данных в формате XML, суть которого состоит в том, что СУБД может выполнить синтаксический анализ XML-документа, разделить его на составляющие, и сохранить отдельные элементы в отдельных столбцах. После этого для поиска данных в полученной таблице может использоваться обычный SQL – таким образом реализуется поддержка поиска элементов и XML-документе. В ответ на запрос СУБД может снова собрать XML-документ из хранящихся в таблице составляющих элементов.

8.2.1. Хранение данных в формате XML

Ввод, вывод и обмен данными в формате XML открывают очень эффективный путь интеграции существующих реляционных баз данных с расширяющимся миром XML. Формат XML используется во внешнем по отношению к базам данных мире для представления структурированных данных, но данные в самой базе данных сохраняют табличную структуру, состоящую из строк и столбцов. Очевидно, что следующим шагом в развитии этой интеграции является хранение XML-документов прямо в базе данных.

Если СУБД на базе SQL поддерживает большие объекты, это означает, что она уже содержит элементарные средства поддержки, хранения и извлечения XML-Документов. Некоторые коммерческие базы данных хранят и извлекают большие текстовые документы при помощи двух типов данных: больших символьных объектов (CLOB) и больших двоичных объектов (BLOB). Во многих коммерческих продуктах поддерживаются значения типа BLOB или CLOB объемом до 4 Гбайт, что достаточно для хранения подавляющего большинства XML-документов.

Для хранения XML-документа в базе данных с использованием этой технологии нужно определить таблицу с одним столбцом типа BLOB или CLOB для хранения текста документа и несколькими вспомогательными столбцами (стандартных типов данных) для хранения атрибутов, идентифицирующих данный документ. Например, если в таблице должны храниться документы с заказами товаров, в дополнение к столбцу типа CLOB для хранения XML-документов можно определить вспомогательные столбцы для хранения номера заказчика, даты заказа и номера заказа, используя тип данных INTEGER, VARCHAR или DATE. Тогда можно будет выполнять поиск документов в таблице заказов по номеру документа, дате заказа или номеру клиента, и для извлечения или хранения XML-документа использовать технологии обработки CLOB-данных.

Преимуществом этого подхода является то, что его относительно просто реализовать. Он поддерживает четкое разделение между операциями SQL (такими, как обработка запросов) и операциями XML. Недостатком же его является очень низкий уровень интеграции между XML и СУБД. В простейшей реализации XML-документ совершенно прозрачен для СУБД. Последняя ничего не знает о его содержимом. Его нельзя искать по значению одного из его атрибутов или элементов, если только этот атрибут или элемент не извлечен из документа хранится в отдельном столбце таблицы. Впрочем, если вы заранее можете предположить, какие типы поиска наиболее вероятны, это ограничение становится не таким уж значительным.

Используемые для обмена данными между приложениями XML-документы, хранящиеся в файлах или базах данных в столбцах типа CLOB, всегда имеют текстовый формат. Этот формат обеспечивает максимальную переносимость содержимого, но компьютерным программам работать с ним крайне неудобно. Синтаксический анализатор XML представляет собой элемент компьютерного программного обеспечения, выполняющий преобразование XML-документа из текстового формата в более подходящее для программной обработки внутреннее представление. Любая СУБД на основе SQL, поддерживающая XML, должна включать синтаксический анализатор для обработки XML-документов. Если СУБД поддерживает тип данных CLOB, для усиления интеграции с XML она позволяет синтаксическому анализатору XML работать прямо с содержимым CLOB-столбцов.

Существует два популярных типа XML-анализаторов, поддерживающих два стиля обработки XML-документов.

– Document Object model (DOM). DOM-анализаторы преобразуют XML-документ в иерархическую древовидную структуру. После этого при помощи API DOM про-

грамма может перемещаться по дереву вверх и вниз, следуя иерархии документа. Интерфейс API DOM облегчает программистам доступ к структуре документа и его элементам.

– Simple XPI for XML (SAX). SAX-анализаторы преобразуют XML-документ в последовательность обратных вызовов программы, которые информируют программу о каждой встреченной анализатором части документа. В ответ программа может выполнять определенные действия, например, реагировать на начало каждого раздела документа или на конкретный атрибут. Интерфейс API SAX предлагает использующим его программам более последовательный стиль обработки документа, лучше соответствующий программной структуре приложений, управляемых событиями.

Любой синтаксический анализатор XML проверит правильность оформления документа и, кроме того, сверит XML-документ со схемой.

DOM-анализатор удобен в тех случаях, когда размер XML-документа относительно мал; этот анализатор генерирует в оперативной памяти древообразное представление документа, занимающее вдвое больше места, чем исходный документ.

Анализатор типа SAX позволяет обрабатывать большие документы небольшими фрагментами. Но поскольку документ не находится в памяти целиком, программе приходится выполнять по нему несколько проходов, если она обрабатывает его фрагменты не по порядку.

8.2.2. Вывод в формате XML

Одним из самых естественных способов объединения технологий баз данных и XML является использование XML в качестве формата выходных данных SQL-запросов. Результаты запроса имеют структурированный табличный формат, который легко преобразовать в XML-представление. Рассмотрим простой запрос из учебной базы данных:

```
SELECT ID_ORDER, ID_MFR, ID_PRD, COUNT, PRICE_ALL
FROM ZAKAZY
WHERE ID_CLN = 12103;
```

ID_ORDER	ID_MFR	ID_PRD	COUNT	PRICE_ALL
312963	ВАЗ	41004	28	$3,276.00
312983	ВАЗ	41004	5	$702.00
313027	ВАЗ	41002	54	$4,104.00
312987	ВАЗ	4100Y	11	$27,500.00

Если СУБД получила команду вывести результаты запроса в формате XML, те же выходные данные могут быть представлены так:

```
SELECT ID_ORDER, ID_MFR, ID_PRD, COUNT, PRICE_ALL
FROM ZAKAZY
WHERE ID_CLN = 12103;
```

```
2   <queryResults>
3     <row>
4       <id_order>312963</id_order>
5       <id_mfr>ВАЗ< id_mfr>
6       <id_prd>41004</id_prd>
7       <count>28</count>
8       <price_all>3276.00</price_all>
9     </row>
10    <row>
11      <id_order>3112983</id_order>
12      <id_mfr> ВАЗ </id_mfr>
13      <id_prd>41004</id_prd>
14      <count>6</count>
15      <price_all>702.00</price_all>
16    </row>
17    <row>
18      <id_order>3113027</id_order>
19      <id_mfr>ВАЗ</id_mfr>
20      <id_prd>41002</id_prd>
21      <count>54</count>
22      <price_all>4104.00</price_all>
23    </row>
24    <row>
25      <id_order>3112987</id_order>
26      <id_mfr>ВАЗ</id_mfr>
27      <id_prd>4100Y</id_prd>
28      <count>11</count>
29      <price_all>27500.00</price_all>
30    </row>
31  </queryResults>
```

Это типичный вид выходной информации, которую можно получить от популярных СУБД, поддерживающих вывод данных в XML-формате. Результаты запроса представляют собой сформированный по всем правилам самодостаточный XML-документ. Если подать его на вход синтаксического анализатора XML, анализатор правильно его интерпретирует и выделит в нем:

– один корневой элемент <queryResults>;

– четыре вложенных элемента <row>;

– для каждого элемента <row> пять вложенных элементов, расположенных в одном и том же порядке.

Возможность получать результаты запросов в XML-формате бывает очень полезной по следующим причинам.

1. Такие данные можно непосредственно подать на вход программы выполняющей их дальнейшую обработку и принимающей информацию в XML-формате.

2. Их можно переслать по сети другой системе, и благодаря XML-формату, содержащему описания элементов, любая получающая система или приложение интерпретируют результаты запроса одинаково – как четыре строки по пять элементов в каждой.

3. А поскольку выходные данные имеют текстовый формат, то исключается их неправильная интерпретация из-за различий двоичного представления данных в системе-отправителе и системе-получателе.

4. Наконец, если XML-документ передается через HTTP с использованием стандартного протокола Simple Object Access Protocol (SOAP), он может проходить через корпоративные брандмауэры и связывать приложение-отправитель в одной компании с приложением-получателем в другой.

Однако у выходного формата XML имеется и несколько недостатков. Один их них связан с размером строки данных. В XML-документе строка содержит вчетверо больше символов, чем та же строка в табличном формате. Соответственно для записи такого документа на диск потребуется вчетверо больше места, а если пересылать этот документ по сети, процесс займет вчетверо больше времени.

Для нашего примера с маленьким количеством данных это не страшно, но для результатов запросов, содержащих тысячи или десятки тысяч строк, умноженных на сотни приложений, выполняющихся на крупном предприятии, это может быть очень существенно.

Кроме того, в данном простейшем XML-формате теряется некоторая информация о данных. Исчез символ денежной единицы, присутствовавший в табличном представлении данных, из-за чего на основании только содержимого XML-документа нельзя определить, что данные имеют денежный тип, и узнать, какая денежная единица используется. Для сохранения этой информации может использоваться XML схема.

8.2.3. Ввод в формате XML

Формат XML может использоваться не только для представления выходных данных запросов, но и для представления входных данных, в частности строк данных, добавляемых в таблицы базы данных. Для обработки XML-данных СУБД должна идентифицировать отдельные составляющие данных (представленные элементами или атрибутами), а после этого сопоставить имена элементов или атрибутов со столбцами целевой таблицы (используя имена столбцов или схему, специфическую для конкретной СУБД). Например, следующую простую инструкцию INSERT

```
INSERT INTO OFFICY (ID_OFC, CITY, REGION, SALES)
VALUES (321, 'Киров', 'Кировская', 835915.00)
```

легко преобразовать в эквивалентную гибридную SQL/XML-инструкцию:

```
INSERT INTO OFFICY (ID_OFC, CITY, REGION, SALES)
VALUES (<?xml version="1.0"?>
```

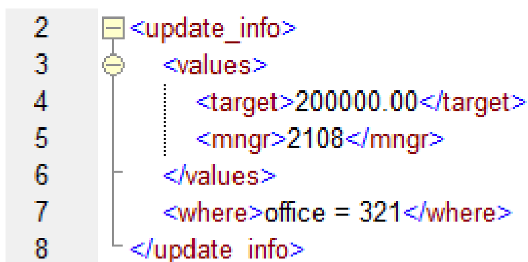
```
2  <row>
3      <id_ofc>321</id_ofc>
4      <city>Киров</city>
5      <region>Кировская</region>
6      <sales>835915.00</sales>
7  </row>
```

Аналогичным образом выполняется обновление базы данных. Например, приведенную ниже инструкцию UPDATE

```
UPDATE OFFICY
SET TARGET = 200000.00, MNGR = 2108
WHERE OFFICE = 23
```

можно преобразовать в такую гибридную SQL/XML инструкцию:

```
UPDATE OFFICY
WHERE ID_OFC = 321
```



В инструкции DELETE необходимо указать только предложение WHERE, для задания которого используются те же соглашения.

Уже несколько ведущих производителей СУБД добавили в свои продукты возможности обработки XML-операций вставки, обновления и удаления строк, но в каждом из них используется свой способ представления имен таблиц и столбцов, а также значений данных в тексте XML и сопоставления их с соответствующими структурами базы данных. Стандарты для этого гибридного синтаксиса SQL/XML пока еще не разработаны.

Хотя в небольшом XML-документе значения для вставки или обновления представляются очень просто, и здесь есть некоторые проблемы. Например, список столбцов в SQL-инструкции INSERT будет излишним, если в XML-документе наряду со значениями данных, которые необходимо вставить, также содержатся имена столбцов базы данных, заданные с помощью имен элементов или атрибутов. Возникает вопрос, почему бы не убрать из запроса список столбцов, чтобы нужная информация извлекалась из XML-документа.

Для программного использования SQL проблема заключается в том, что XML-документ и содержащиеся в нем значения данных определяются и передаются СУБД во время выполнения программы. И если имена столбцов или даже имя таблицы указаны только в XML-документе, до момента выполнения программы СУБД не знает, какие таблицы и столбцы участвуют в запросе. В такой ситуации СУБД должна использовать динамический SQL, со всем связанным с этим ущербом для производительности.

8.2.4. Обмен данными в формате XML

СУБД может поддерживать обмен данными в формате XML в очень простой форме – поддерживая вывод результатов запросов и ввод данных для инструкции INSERT в формате XML. Однако это требует от пользователя или программиста тщательной проработки формата генерируемых результатов запроса, чтобы он в точности соответствовал формату инструкции INSERT в принимающей базе

данных. Обмен данными XML может быть по-настоящему полезен в том случае, если он более явно поддерживается СУБД.

В настоящее время несколько коммерческих продуктов предлагают возможность пакетного экспорта таблиц (или результатов запроса) во внешний файл, форматированный как XML-документ. Кроме того, они предлагают аналогичную возможность пакетного импорта данных из файла того же типа в таблицу СУБД. Эта схема делает XML стандартным форматом представления содержимого таблиц для обмена данными.

Обратите внимание, что использование предлагаемых СУБД возможностей импорта/экспорта данных таблиц в формате XML не ограничивает их использование для обмена между базами данных.

8.2.5. Интеграция данных в формате XML

Хранение XML-документов в базе данных в виде больших объектов очень удобно для определенных типов интеграции SQL/XML. Если XML-документы, например, представляют собой деловые документы текстового типа или же являются текстовыми компонентами Web-страниц, СУБД не обязательно понимать их внутреннюю структуру. Каждый документ может идентифицироваться одним или несколькими ключевыми словами или атрибутами, которые можно извлекать из документа и хранить в обыкновенных столбцах, используемых для поиска данных.

А вот если XML-документ содержит данные в форме набора записей, предназначенных для обработки, возможностей больших объектов явно недостаточно. Вам, скорее всего, потребуется доступ к отдельным элементам документа и осуществление поиска по их содержимому и атрибутам. СУБД предоставляет такие возможности для своих обычных данных в форме строк и столбцов.

Так почему бы ей не выполнить автоматическую декомпозицию входящего XML-документа и не преобразовать содержимое его элементов и значения его атрибутов в соответствующий набор строк и столбцов, удобный для обработки стандартными средствами СУБД? Мы с вами уже видели, как этот подход используется для преобразования результатов запросов в XML-документе. Та же технология может применяться и для повторного формирования XML-документа из таблицы базы данных, если он снова потребуется в исходной текстовой форме.

Проблема возникает при преобразовании XML-документов (которые замечательно подходят для внешнего представления данных) во внутреннее представление данных (более удобное для программной обработки), так как внутреннее представление не является уникальным для различных систем управления базами данных.

Та же проблема существует и в языке Java, когда XML-документ преобразуется в набор экземпляров классов Java для внутренней обработки.

Процесс декомпозиции XML-документа на составляющие элементы и атрибуты называется демаршалингом. А процесс повторной сборки текста XML-документа из составляющих элементов и атрибутов носит название маршалинга.

Для очень простого XML-документа процесс маршалинга и демаршалинга несложен, и коммерческие СУБД развиваются в направлении его поддержки.

Давайте еще раз рассмотрим простой документ, содержащий заказ товаров, показанный на **Ошибка! Источник ссылки не найден.**. Его элементы можно поставить в соответствие (один к одному) отдельным столбцам таблицы ZAKAZY. В простейшем случае имена элементов или атрибутов будут идентичны именам соответствующих столбцов.

СУБД может получить XML-документ, подобный приведенному на этом рисунке, автоматически преобразовать его элементы (или, в зависимости от стиля документа его атрибуты) в значения столбцов, используя имена элементов (или атрибутов). Восстановление такого XML-документа выполняется так же просто. Если имена элементов в XML-документе не соответствуют именам столбцов, СУБД придется проделать немного больше работы. В этом случае необходима дополнительная информация о соответствии между столбцами и элементами или атрибутами. Такую информацию проще всего поместить в системный каталог СУБД.

Однако многие реальные XML-документы имеют несколько более сложную структуру и их нельзя преобразовать в отдельные строки таблицы. На Рис. 8.2. показан такой же заказ, как на **Ошибка! Источник ссылки не найден.**, но с несколько расширенной структурой: этот заказ содержит не одну, а несколько позиций заказываемых товаров. Как же выполнить демаршалинг этого документа для помещения его в нашу учебную базу данных?

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <purchaseOrder>
3      <id_cln>12117</id_cln>
4      <id_order>312961</id_order>
5      <date_order>1989-12-17</date_order>
6      <id_slzh>2106</id_slzh>
7      <terms
8          ship="ground"
9          bill="Net30">
10     </terms>
11     <orderItem>
12         <id_mfr>УАЗ</id_mfr>
13         <id_prd>2A44L</id_prd>
14         <count>7</count>
15         <price_all>31500.00</price_all>
16     </orderItem>
17     <orderItem>
18         <id_mfr>УПЗ</id_mfr>
19         <id_prd>41003</id_prd>
20         <count>1</count>
21         <price_all>652.00</price_all>
22     </orderItem>
23 </purchaseOrder>
```

Рис. 8.2. XML-документ, содержащий расширенный заказ товаров

Можно поместить каждую строку заказа в отдельную строку таблицы ZAKAZY. (Для этого примера мы проигнорируем требования уникальности номеров-заказов в таблице ZAKAZY, связанное с тем, что номер заказа является пер-

вичным ключом этой таблицы.) В результате некоторые данные таблицы будут повторяться, в нескольких строках будет стоять один и тот же номер заказ, его дата, номер клиента и номер продавца.

Кроме того, это усложнит маршалинг данных для восстановления документа – СУБД должна будет знать, что в строки с одним номером заказа нужно собрать в один многострочный XML-документ. Очевидно, что для маршалинга и демаршалинга даже такого просто документа требуется достаточно сложная информация о соответствии между компонентами документа, таблицами и столбцами базы данных.

Приведенный пример с многострочным заказом затрагивает лишь самые поверхностные проблемы, связанные с маршалингом и демаршалингом документов. Более общая ситуация показана на Рис. 8.3. , где СУБД должна выполнить демаршалинг XML-документа, преобразовав его в несколько строк нескольких взаимосвязанных таблиц. Для маршалинга этого документа СУБД должна проанализировать связи между таблицами, чтобы найти связанные строки и воспроизвести иерархию XML. Причиной всех этих сложностей является несоответствие между естественной иерархической структурой XML и плоской, нормализованной структурой реляционных баз данных, составленных из строк и столбцов.

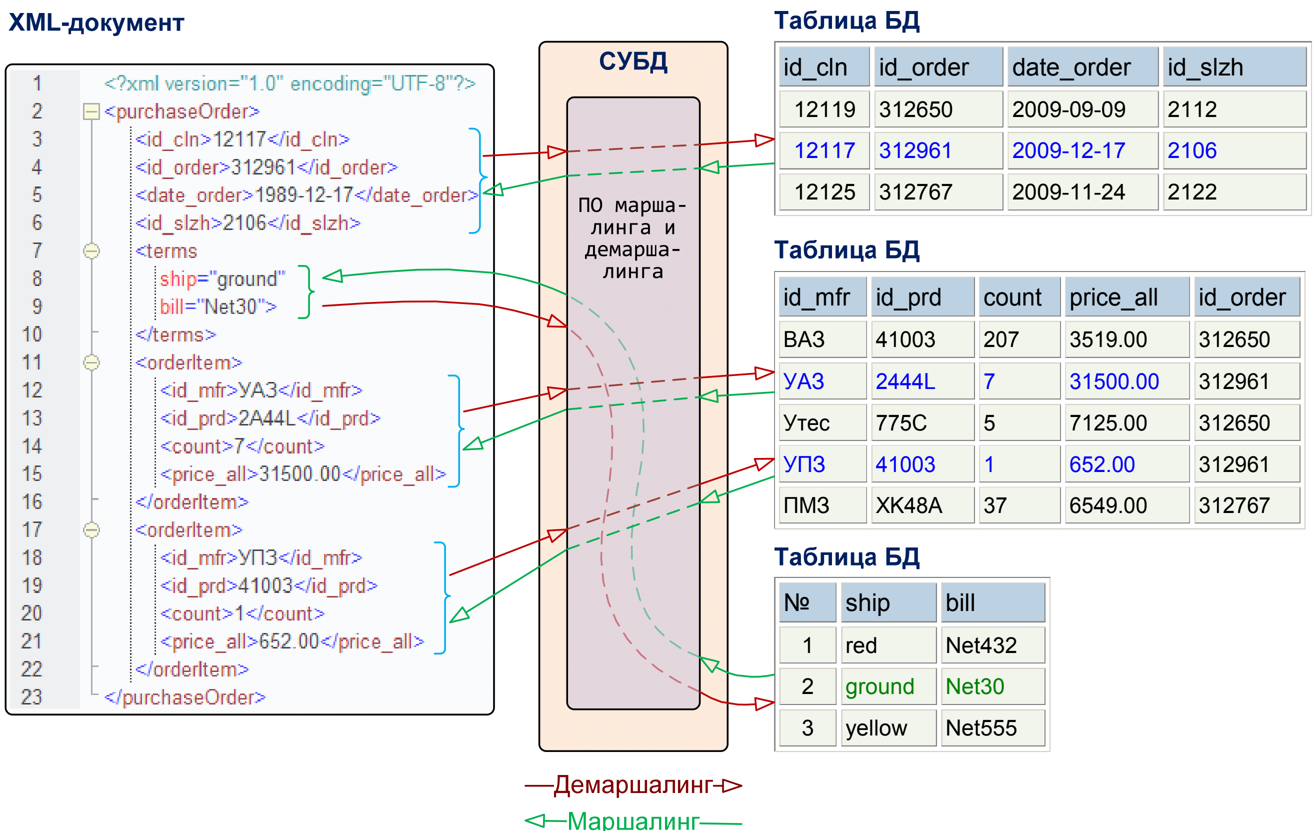


Рис. 8.3. Маршалинг и демаршалинг XML-документа

Если СУБД поддерживает объектно-реляционные расширения, такие как структурированные типы данных, маршалинг и демаршалинг одновременно и усложняются, и упрощаются. Упрощение достигается за счет того, что отдельные столбцы таблицы могут иметь иерархическую структуру. В этом случае

XML-элемент более высокого уровня (такой как адрес, состоящий из элементов номера дома, улицы, города, области, страны и индекса) может сопоставляться одному столбцу абстрактного типа ADDRESS с собственной внутренней иерархией. Однако при этом усложняется структура базы данных, и для упрощения преобразования между базой данных и XML приходится поступиться гибкостью структуры строки/столбцы.

Средства маршалинга/демаршалинга уже включены в несколько коммерческих баз данных, а в других эти возможности анонсированы на ближайшее будущее. Данные операции существенно сказываются на производительности, и будущее покажет, насколько популярными они окажутся на практике. Тем не менее, если приложение обрабатывает внешние данные в форме XML, в какой-то момент оно должно будет выполнять преобразование между данными XML и SQL, и самым эффективным решением является выполнение этого преобразования средами СУБД.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Энциклопедия SQL / Дж. Грофф, П. Вайнберг. – 3-е изд. – СПб. : Питер, 2004. – 896 с.

2. Голицына, О. Л. Системы управления базами данных : учебное пособие / О. Л. Голицына, Т. Л. Партыка, И. И. Попов. − М. : Форум: Инфра-М, 2006.

3. Голицына, О. Л. Информационные системы : учебное пособие / О. Л. Голицына, Н. В. Максимов, И. И. Попов. − М. : Форум: Инфра-М, 2007. – 496 с.

4. Шпак, Ю. А. SQL. Просто как дважды два / Ю. А. Шпак. – М. : Эксмо, 2007. – 304 с.

5. Мартин, Дж. Организация баз данных в вычислительных системах / Дж. Мартин. – М. : Мир, 1980.

6. Базы данных: модели, разработка, реализация / Т. С. Карпова. – СПб. : Питер, 2001. – 304 с.

7. Брукшир, Дж. Информатика и вычислительная техника / Дж. Брукшир. – 7-е изд. – СПб. : Питер, 2004. – 620 с.

ПРИЛОЖЕНИЕ

УЧЕБНАЯ БАЗА ДАННЫХ

В настоящем приложении описана учебная база данных, на которой основано большинство примеров, приведенных в данном курсе лекций. Учебная база данных состоит из пяти таблиц:

– CLIENTY (клиенты) – содержит по одной строке для каждого из клиентов компании;

– SLUZHASCHIE (служащие) – содержит по одной строке для каждого служащего компании;

– OFFISY (офисы) – содержит по одной строке для каждого из пяти офисов компании;

– TOVARY (товары) – содержит по одной строке для каждого наименования товара, продаваемого компанией;

– ZAKAZY (заказы) – содержит по одной строке для каждого из заказов, сделанных клиентом. Для простоты считается, что один заказ может содержать только один товар.

На рис. П.1 приведена логическая схема учебной базы данных.

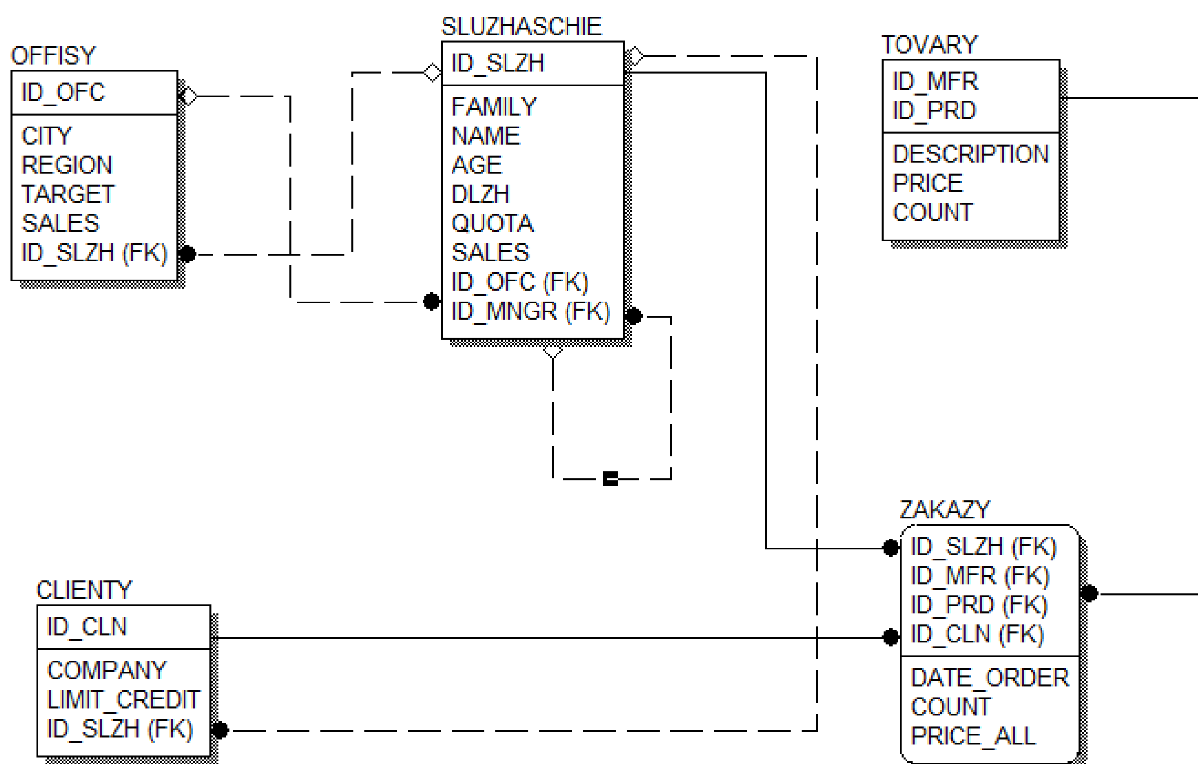


Рис. П.1. Структура учебной базы данных

Таблицы учебной базы данных были созданы под управлением СУБД PostgreSQL с помощью инструкций CREATE TABLE, приведенных ниже.

```
-- Table: zakazy.clienty

-- DROP TABLE zakazy.clienty;

CREATE TABLE zakazy.clienty
(
  id_cln serial NOT NULL,
  company character varying(50) NOT NULL,
  id_slzh integer NOT NULL,
  limit_credit zakazy.d_price NOT NULL,
  CONSTRAINT pkclienty PRIMARY KEY (id_cln),
  CONSTRAINT fk_clienty_sluzhaschie FOREIGN KEY (id_slzh)
      REFERENCES zakazy.sluzhaschie (id_slzh) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT
)
WITH (OIDS=FALSE);
ALTER TABLE zakazy.clienty OWNER TO postgres;
```

Рис. П.2. Инструкция CREATE TABLE ZAKAZY.CLIENTY

```
-- Table: zakazy.sluzhaschie

-- DROP TABLE zakazy.sluzhaschie;

CREATE TABLE zakazy.sluzhaschie
(
  id_slzh serial NOT NULL,
  "family" zakazy.d_fio NOT NULL,
  "name" zakazy.d_fio NOT NULL,
  age zakazy.d_age NOT NULL,
  id_ofc integer,
  dlzh character varying(50) NOT NULL,
  mngr integer,
  quota zakazy.d_price,
  sales zakazy.d_price NOT NULL,
  CONSTRAINT pksluzhaschie PRIMARY KEY (id_slzh),
  CONSTRAINT fk_sluzhaschie_offisy FOREIGN KEY (id_ofc)
      REFERENCES zakazy.offisy (id_ofc) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
WITH (OIDS=FALSE);
ALTER TABLE zakazy.sluzhaschie OWNER TO postgres;
```

Рис. П.3. Инструкция CREATE TABLE ZAKAZY.SLUZHASCHIE

```
-- Table: zakazy.offisy

-- DROP TABLE zakazy.offisy;

CREATE TABLE zakazy.offisy
(
  id_ofc serial NOT NULL,
  city character varying(70) NOT NULL,
  region character varying(70) NOT NULL,
  mngr integer,
  target zakazy.d_price,
  sales zakazy.d_price,
  CONSTRAINT pkoffisy PRIMARY KEY (id_ofc)
)
WITH (OIDS=FALSE);
ALTER TABLE zakazy.offisy OWNER TO postgres;
```

Рис. П.4. Инструкция CREATE TABLE ZAKAZY.OFFISY

```
-- Table: zakazy.tovary

-- DROP TABLE zakazy.tovary;

CREATE TABLE zakazy.tovary
(
  id_mvr character varying(20) NOT NULL,
  id_prd character varying(10) NOT NULL,
  description character varying(100) NOT NULL,
  price zakazy.d_price NOT NULL,
  count_tvr zakazy.d_count NOT NULL,
  CONSTRAINT pktovary PRIMARY KEY (id_mvr, id_prd)
)
WITH (OIDS=FALSE);
ALTER TABLE zakazy.tovary OWNER TO postgres;

-- Index: zakazy.index_descript

-- DROP INDEX zakazy.index_descript;

CREATE INDEX index_descript
  ON zakazy.tovary
  USING btree
  (description);
```

Рис. П.5. Инструкция CREATE TABLE ZAKAZY.TOVARY

```
-- Table: zakazy.zakazy

-- DROP TABLE zakazy.zakazy;

CREATE TABLE zakazy.zakazy
(
  id_order serial NOT NULL,
  date_order date NOT NULL,
  id_cln integer NOT NULL,
  id_slzh integer NOT NULL,
  id_mvr character varying(20) NOT NULL,
  id_prd character varying(10) NOT NULL,
  count_ord zakazy.d_count NOT NULL,
  price_all zakazy.d_price NOT NULL,
  CONSTRAINT pkzakazy PRIMARY KEY (id_order),
  CONSTRAINT fk_zakazy_clienty FOREIGN KEY (id_cln)
      REFERENCES zakazy.clienty (id_cln) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE,
  CONSTRAINT fk_zakazy_sluzhaschie FOREIGN KEY (id_slzh)
      REFERENCES zakazy.sluzhaschie (id_slzh) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT,
  CONSTRAINT fk_zakazy_tovary FOREIGN KEY (id_mvr, id_prd)
      REFERENCES zakazy.tovary (id_mvr, id_prd) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE RESTRICT
)
WITH (OIDS=FALSE);
ALTER TABLE zakazy.zakazy OWNER TO postgres;
```

Рис. П.6. Инструкция CREATE TABLE ZAKAZY.ZAKAZY

Ниже приведено содержимое пяти таблиц учебной базы данных после выполнения соответствующих команд `INSERT`.

Таблица CLIENTY

ID_CLN	COMPANY	ID_SLZH	LIMIT_CREDIT
12111	«Заря»	2103	$50000.00
12102	«Гранит»	2101	$65000.00
12103	«Базальт»	2105	$50000.00
12123	«Марс»	2102	$40000.00
12107	«Комета»	2110	$35000.00
12115	«Сатурн»	2101	$20000.00
12101	«Метеор»	2106	$65000.00
12112	«Юпитер»	2108	$50000.00
12121	«Агат»	2103	$45000.00
12114	«Аметист»	2102	$20000.00
12124	«Меркурий»	2107	$40000.00
12108	«Алмаз»	2109	$55000.00
12117	«Цезарь»	2106	$35000.00
12122	«Центурион»	2105	$30000.00
12120	«Взлет»	2102	$50000.00
12106	«Атлант»	2102	$65000.00
12119	«Аргонавт»	2109	$25000.00
12118	«Стрела»	2108	$60000.00
12113	«Кристалл»	2104	$20000.00
12109	«Самолет»	2103	$25000.00
12105	«Энергия»	2101	$45000.00

Таблица SLUZHASCHIE

ID_SLZH	FAMILY	NAME	AGE	ID_OFC	DLZH	MNGR	QUOTA	SALES
2105	Филатов	Петр	37	313	Брокер	2104	$350000.00	$367911.00
2109	Полев	Андрей	31	311	Брокер	2106	$300000.00	$392725.00
2102	Пронин	Игорь	48	321	Брокер	2108	$350000.00	$474050.00
2106	Петров	Петр	52	311	Гл.Брокер	NULL	$275000.00	$299912.00
2104	Иванов	Иван	33	312	Ст.Брокер	2106	$200000.00	$142594.00
2101	Федоров	Федор	45	312	Брокер	2104	$300000.00	$305673.00
2110	Уткин	Денис	41	NULL	Брокер	2101	NULL	$75985.00
2108	Нилов	Лев	62	321	Ст.Брокер	2106	$350000.00	$361865.00
2103	Филин	Федор	29	312	Брокер	2104	$275000.00	$286775.00
2107	Ганин	Сергей	49	322	Брокер	2108	$300000.00	$186042.00

Таблица OFFISY

ID_OFC	CITY	REGION	MNGR	TARGET	SALES
322	Инза	Ульяновская	2108	$300000.00	$186042.00
311	Буинск	Татарстан	2106	$575000.00	$692637.00
312	Тверь	Московская	2104	$800000.00	$735042.00
313	Орел	Орловская	2105	$350000.00	$367911.00
321	Киров	Кировская	2108	$725000.00	$835915.00
400	Омск	Омская	NULL	NULL	NULL

Таблица TOVARY

ID_MFR	ID_PRD	DESCRIPTION	PRICE	COUNT
УАЗ	2A45C	Деталь кузова	$79.00	210
ВАЗ	4100Y	Деталь двигателя	$2750.00	25
ПМЗ	XK47	Сопло	$355.00	38
УПЗ	41672	Плата	$180.00	0
Утес	779C	Пылесос А300	$1875.00	9
ВАЗ	41003	Дверь 1	$107.00	207
ВАЗ	41004	Дверь 2	$117.00	139
УПЗ	41003	Корпус	$652.00	3
Утес	887P	Пылесос ручной	$250.00	24
ПМЗ	XK48	Ось	$134.00	203
УАЗ	2A44L	Левая дверь	$4500.00	12
УМЗ	112	Цепь	$148.00	115
Утес	887H	Шланг пылесоса	$54.00	223
УПЗ	41089	Крышка	$225.00	78
ВАЗ	41001	Ручка	$55.00	277
Утес	775C	Пылесос В200	$1425.00	5
ВАЗ	4100Z	Карбюратор	$2500.00	28
ПМЗ	XK48A	Редуктор	$177.00	37
ВАЗ	41002	Педаль	$76.00	167
УАЗ	2A44R	Правая дверь	$4500.00	12
Утес	773C	Пылесос Д100	$975.00	28
ВАЗ	4100X	Дворник	$25.00	37
УМЗ	114	Поршень	$243.00	15
Утес	887X	Brace Retainer	$475.00	32
УАЗ	2A44G	Ручной тормоз	$350.00	14

Таблица ZAKAZY

ID_ORDER	DATE_ORDER	ID_CLN	ID_SLZH	ID_MFR	ID_PRD	COUNT	PRICE_ALL
312961	12/17/89	12117	2106	УАЗ	2A44L	7	$31500.00
313012	01/11/90	12111	2105	ВАЗ	41003	35	$3745.00
312989	01/03/90	12101	2106	УМЗ	114	6	$1458.00
313051	02/10/90	12118	2108	ПМЗ	XK47	4	$1420.00
312968	10/12/89	12102	2101	ВАЗ	41004	34	$3978.00
313036	01/30/90	12107	2110	ВАЗ	4100Z	9	$22500.00
313045	02/02/90	12112	2108	УАЗ	2A44R	10	$45000.00
312963	12/17/89	12103	2105	ВАЗ	41004	28	$3276.00
313013	01/14/90	12118	2108	УПЗ	41003	1	$652.00
313058	02/23/90	12108	2109	УМЗ	112	10	$1480.00
312997	01/08/90	12124	2107	УПЗ	41003	1	$652.00
312983	12/27/89	12103	2105	ВАЗ	41004	6	$702.00
313024	01/20/90	12114	2108	ПМЗ	XK47	20	$7100.00
313062	02/24/90	12124	2107	УМЗ	114	10	$2430.00
312979	10/12/89	12114	2102	ВАЗ	4100Z	6	$15000.00
313027	01/22/90	12103	2105	ВАЗ	41002	54	$4104.00
313007	01/08/90	12112	2108	УТЕС	773C	3	$2925.00
313069	03/02/90	12109	2107	УТЕС	775C	22	$31350.00
313034	01/29/90	12107	2110	УАЗ	2A45C	8	$632.00
312992	11/04/89	12118	2108	ВАЗ	41002	10	$760.00
312975	10/12/89	12111	2103	УАЗ	2A44G	6	$2100.00
313055	02/15/90	12108	2101	ВАЗ	4100X	6	$150.00
313048	02/10/90	12120	2102	УТЕС	779C	2	$3750.00
312993	01/04/89	12106	2102	УАЗ	2A45C	24	$1896.00
313065	02/27/90	12106	2102	ПМЗ	XK47	6	$2130.00
313003	01/25/90	12108	2109	УТЕС	779C	3	$5625.00
313049	02/10/90	12118	2108	ПМЗ	XK47	2	$776.00
312987	12/31/89	12103	2105	ВАЗ	4100Y	11	$27500.00
313057	02/18/90	12111	2103	ВАЗ	4100X	24	$600.00
313042	02/02/90	12113	2101	УАЗ	2A44R	5	$22500.00

Учебное издание

ТОКМАКОВ Геннадий Петрович

Базы данных. Концепция баз данных,
реляционная модель данных,
языки SQL и XML

Учебное пособие

Редактор М. В. Штаева

ЛР №020640 от 22.10.97.
Подписано в печать 15.04.2011. Формат 60×84/16.
Усл. печ. л. 11,39. Тираж 100 экз. Заказ 414.

Ульяновский государственный технический университет
432027, г. Ульяновск, ул. Сев. Венец, 32.

Типография УлГТУ, 432027, г.Ульяновск, ул. Сев. Венец, 32.