

Университетский учебник

С. Д. Кузнецов

# БАЗЫ ДАННЫХ



Прикладная математика  
и информатика

УНИВЕРСИТЕТСКИЙ УЧЕБНИК

---

Серия «Прикладная математика и информатика»

С. Д. КУЗНЕЦОВ

# БАЗЫ ДАННЫХ

*Рекомендовано  
учебно-методическим объединением  
по классическому университетскому образованию  
в качестве учебника для студентов  
высших учебных заведений, обучающихся  
по направлению подготовки «Прикладная  
математика и информатика»*



Москва  
Издательский центр «Академия»  
2012

УДК 621.38(075.8)  
ББК 3281я733  
К891

Рецензент —

канд. техн. наук, ст. науч. сотр., доц. *М.Р.Козаловский*  
(зав. лабораторией систем баз данных Института проблем рынка РАН)

## **Кузнецов С. Д.**

К891

Базы данных : учебник для студ. учреждений высшего проф. образования / С.Д.Кузнецов. — М. : Издательский центр «Академия», 2012. — 496 с. — (Университетский учебник. Сер. Прикладная математика и информатика).

ISBN 978-5-7695-8430-5

Учебник создан в соответствии с Федеральным государственным образовательным стандартом по направлению подготовки «Прикладная математика и информатика» (квалификация «бакалавр»).

В учебнике обсуждаются потребности разработчиков информационных систем в технологии баз данных, рассматриваются основные функции и типовая архитектура СУБД, а также приводится краткая характеристика нескольких популярных моделей данных. Подробно описывается реляционная модель данных, проектирование реляционных баз данных с использованием принципов нормализации и на основе семантических диаграммных моделей данных. В учебнике представлены также основные методы и алгоритмы, используемые в SQL-ориентированных СУБД; наиболее важные черты языка SQL как отдельной модели данных.

Для студентов учреждений высшего профессионального образования. Может быть использован студентами, обучающимися по направлениям подготовки «Информатика и вычислительная техника» и «Прикладная математика».

УДК 621.38(075.8)  
ББК 3281я733

*Оригинал-макет данного издания является собственностью  
Издательского центра «Академия», и его воспроизведение  
любым способом без согласия правообладателя запрещается*

© Кузнецов С.Д., 2012  
© Образовательно-издательский центр «Академия», 2012  
© Оформление. Издательский центр «Академия», 2012

ISBN 978-5-7695-8430-5

Хорошее знание и понимание технологии баз данных (БД) требуется любому специалисту в области программного обеспечения. Фундаментальное образование в этой области нельзя заменить тренингом. Как показывает практический опыт, люди, которые прошли тренинг-курсы, направленные на подготовку разработчиков приложений, проектировщиков, администраторов баз данных в среде конкретных систем управления базами данных (СУБД), но не получили предварительной базовой подготовки, обычно не оказываются способными достичь высоких профессиональных результатов. И наоборот, при наличии такой базовой подготовки люди зачастую быстро и качественно осваивают специфику конкретных систем без посторонней помощи. Поэтому нельзя переоценить важность обязательных курсов по технологии БД, читаемых в университетах и других высших учебных заведениях.

Данный учебник основан на материалах обязательного учебного курса, читаемого с 1994 г. автором на программистском потоке факультета вычислительной математики и кибернетики (ВМиК) Московского государственного университета им. М. В. Ломоносова.

За время своего существования курс неоднократно модифицировался при сохранении в целом его программы. Вначале основное внимание уделялось вопросам алгоритмов и методов построения СУБД, а модельно-языковые аспекты построения баз данных стояли на втором плане. Однако со временем стало ясно, что, во-первых, правильно понять и усвоить программистскую специфику организации СУБД можно только при наличии хорошего понимания модели данных, на которой основываются соответствующие базы данных, и языка, поддерживаемого СУБД для взаимодействия с этими базами данных. Во-вторых, очевидно, что среди выпускников ВМиК МГУ им. М. В. Ломоносова (как и других университетов и вузов) значительно больше специалистов, проектирующих базы данных и разрабатывающих их приложения, чем разработчиков СУБД.

Поэтому в последние годы модели данных и языковые средства заняли в курсе основное место, хотя по-прежнему значительная часть курса посвящается архитектуре СУБД и алгоритмам построения их наиболее важных подсистем. Данный учебник достаточно точно соответствует современному состоянию курса. Основная часть материала обычно укладывается в 68 ч (третий курс, первый семестр), а в спецкурсе (второй, весенний, семестр) рассматриваются более сложные аспекты моделей данных, описываемые, например, в [38]. От студентов, сдающих экзамен по спецкурсу, требуется полное знание материала основного курса.

Учебник состоит из 14 глав, сгруппированных в пять частей. В ч. I, вводной, содержится две главы. В гл. 1 обсуждается, каким образом потребности разработчиков информационных систем приводят к потребностям в системах управления базами данных, а также описываются типовая организация СУБД и ее основные подсистемы; гл. 2 содержит обзор семи моделей данных — трех ранних моделей, на которых основывались дореляционные СУБД (иерархическая и сетевая модели данных и модель инвертированных таблиц); классическая реляционная модель, введенная Эдгаром Коддом; и три современные модели данных (объектно-ориентированная модель данных, модель данных SQL и «истинно» реляционная модель, определенная Кристофером Дейтом и Хью Дарвенем).

Часть II также состоит из двух глав и посвящается развернутому описанию реляционной модели данных в авторской трактовке. В гл. 3 определяются основные понятия и термины реляционной модели, характеризуются три ее части и описываются структурная и целостная части; гл. 4 посвящена манипуляционной части реляционной модели данных. В ней обсуждаются два варианта реляционной алгебры — алгебра Кодда и современная, изящная, алгебра, введенная К. Дейтом и Х. Дарвенем, а также две разновидности реляционного исчисления — исчисление кортежей и доменное исчисление.

Часть III учебника, затрагивающую проблемы проектирования реляционных (и SQL-ориентированных) баз данных, составляют три главы. В гл. 5 и 6 описывается классический подход к проектированию реляционных баз данных на основе принципов нормализации. В гл. 5 излагаются основные понятия теории реляционных баз данных, связанные с функциональными зависимостями, и определяются вторая, третья нормальные формы и нормальная форма Бойса—Кодда. В гл. 6 вводится понятие многозначной зависимости, формулируются и дока-

зываются требуемые утверждения и определяется четвертая нормальная форма отношения. После этого определяются зависимость проекции/соединения и заключительная пятая нормальная форма, свойства которой невозможно улучшить на основе нормализации. В гл. 7 излагается материал по проектированию SQL-ориентированных баз данных на основе использования семантических диаграммных моделей — диаграмм «сущность-связь» и диаграмм классов языка UML.

В ч. IV книги, также состоящей из трех глав, описываются структуры данных, методы и алгоритмы, используемые в системах управления базами данных. В гл. 8 обсуждаются наиболее распространенные методы физической организации реляционных баз данных во внешней памяти, описываются применяемые для ускорения работы СУБД индексные структуры. В гл. 9 разобраны методы управления транзакциями — разновидностями двухфазного протокола синхронизационных блокировок; методы временных меток; методы, основанные на поддержке нескольких версий объектов базы данных. В гл. 10 описываются методы восстановления баз данных после различных сбоев на основе журнализации изменений базы данных.

Наконец, ч. V, последняя, состоящая из четырех глав, целиком посвящена языку SQL. Цель этой части состоит не в том, чтобы читатели могли выучить синтаксис языка и научиться писать простые запросы, а в том, чтобы показать, что язык SQL определяет полную и законченную модель данных, похожую на реляционную модель, но разительно от нее отличающуюся во многих важных аспектах. В гл. 11 обсуждаются система типов языка SQL, а также языковые средства определения базовых таблиц и ограничений целостности баз данных. В гл. 12 рассмотрен оператор выборки языка SQL; описаны синтаксис и семантика этого оператора, а также основные виды предикатов, которые можно использовать в условиях выборки. В гл. 13 изложены «поисковые» варианты операторов вставки, удаления и модификации таблиц в SQL-ориентированных базах данных, а также механизм триггеров. Наконец, в гл. 14, заключительной, описаны языковые средства управления транзакциями, сессиями и подключениями. В этой книге модель данных SQL представлена в сокращенной и неполной форме, позволяющей изложить соответствующий материал на лекциях. Более полное описание можно найти, например, в [38, 39].

*Автор*

### Глава 1

#### **НАЗНАЧЕНИЕ ТЕХНОЛОГИИ БАЗ ДАННЫХ. ФУНКЦИИ И ОСНОВНЫЕ КОМПОНЕНТЫ СИСТЕМ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ**

В данной главе определяется понятие информационной системы, обсуждаются предпосылки появления в компьютерах устройств внешней памяти, а также обосновывается принципиальная важность для организации информационных систем дисковых устройств с подвижными магнитными головками. Далее рассматриваются особенности организации и основное функциональное назначение одного из ключевых компонентов современных операционных систем — систем управления файлами. В третьем подразделе главы демонстрируется, что возможности файловых систем оказываются недостаточными для создания информационных программных систем и то, что естественные требования информационных систем к средствам управления данными во внешней памяти приводят к необходимости наличия систем управления базами данных (СУБД). В ходе этого анализа определяются основные функции, которые должны поддерживаться СУБД, и основные компоненты, которые содержит типичная СУБД.

#### **1.1. Информационные системы и устройства внешней памяти**

В общем смысле информационная система представляет собой программный комплекс, функции которого состоят в поддержке надежного долговременного хранения информации в памяти компьютера, выполнении требуемых для данного приложения преобразований информации и/или вычислений, предостав-

лении пользователям системы удобного и легко осваиваемого интерфейса. Обычно объемы данных, с которыми приходится иметь дело таким системам, достаточно велики, а сами данные обладают достаточно сложной структурой. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т. д.

Надежное и долговременное хранение информации можно обеспечить только при наличии запоминающих устройств, сохраняющих информацию после выключения электропитания. Оперативная (основная) память этим свойством обычно не обладает. В первые десятилетия развития вычислительной техники использовались два вида устройств внешней памяти: магнитные ленты и магнитные барабаны. Емкость магнитных лент была достаточно велика, но по своей природе они обеспечивали только последовательный доступ к данным. Емкость магнитной ленты пропорциональна ее длине. Чтобы получить доступ к требуемой порции данных, нужно в среднем перемотать половину ее длины. Но чисто механическую операцию перемотки нельзя выполнить очень быстро. Поэтому быстрый произвольный доступ к данным на магнитной ленте, очевидно, невозможен.

Магнитный барабан представлял собой массивный металлический цилиндр с намагниченной внешней поверхностью и неподвижным пакетом магнитных головок. Такие устройства обеспечивали возможность достаточно быстрого произвольного доступа к данным, но позволяли сохранять сравнительно небольшой объем данных. Быстрый произвольный доступ осуществлялся благодаря высокой скорости вращения барабана и наличию отдельной головки на каждую дорожку магнитной поверхности; ограниченность объема была обусловлена наличием всего одной магнитной поверхности.

Указанные ограничения не очень существенны для систем численных расчетов. Обсудим более подробно, какие реальные потребности возникают у разработчиков таких систем. Прежде всего, для получения требуемых результатов серьезные вычислительные программы должны проработать достаточно долгое время (недели, месяцы и даже, может быть, годы). Наличие гарантий надежности со стороны производителей аппаратных компьютерных средств не избавляет программистов от необходимости использования программного сохранения частичных результатов вычислений, чтобы при возникновении непредвиденных сбоев аппаратуры можно было продолжить выполнение расчетов с некоторой контрольной точки. Для сохранения проме-



жуточных результатов идеально подходят магнитные ленты: при выполнении процедуры установки контрольной точки данные последовательно сбрасываются на ленту, а при необходимости перезапуска от сохраненной контрольной точки данные также последовательно с ленты считываются.

Вторая традиционная потребность программистов вычислительных приложений — максимально большой объем оперативной памяти. Большая оперативная память требуется, во-первых, для того, чтобы обеспечить программе быстрый доступ к большому количеству обрабатываемых данных. Во-вторых, сложные вычислительные программы сами могут иметь большой объем. Поскольку объем реально доступной в ЭВМ оперативной памяти всегда являлся недостаточным для удовлетворения текущих потребностей вычислений, требовалась быстрая внешняя память для организации оверлеев и/или виртуальной памяти. Здесь не будем вдаваться в детали организации этих механизмов программного расширения оперативной памяти, но заметим, что для этого идеально подходили магнитные барабаны. Они обеспечивали быстрый доступ к внешней памяти, а для расширения оперативной памяти одной программы (сложные вычислительные программы, как правило, выполняются на компьютере в одиночку) большой объем внешней памяти не требуется.

Далее заметим, что, даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти, чтобы программа работала как можно быстрее. Развитая поддержка работы с внешней памятью со стороны общесистемных программных средств не обязательна, а иногда и вредна, поскольку приводит к дополнительным накладным расходам аппаратных ресурсов.

Однако для информационных систем, в которых объем постоянно хранимых данных определяется спецификой бизнес-приложения, а потребность в текущих данных — пользователем приложения, одних только магнитных барабанов и лент недостаточно. Емкость магнитного барабана просто не позволяет долговременно хранить данные большого объема. Что же касается лент, то представьте себе состояние человека у билетной кассы, который должен дожидаться полной перемотки магнитной ленты. Естественным требованием к таким системам является обеспечение высокой средней скорости выполнения операций при наличии больших объемов данных.

Именно требования к устройствам внешней памяти со стороны бизнес-приложений вызвали появление устройств внешней

памяти со съёмными пакетами магнитных дисков и подвижными головками чтения/записи, что явилось революцией в истории вычислительной техники. Эти устройства памяти обладали существенно большей емкостью, чем магнитные барабаны (за счет наличия нескольких магнитных поверхностей), обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь архив данных практически неограниченного объема.

Магнитные диски представляют собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге движется пакет магнитных головок (рис. 1.1). Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр пакета магнитных дисков. На каждой поверхности цилиндр «высекает» дорожку, так что каждая поверхность содержит число дорожек, равное числу цилиндров. При разметке магнитного диска (специальном действии, предшествующем использованию диска) каждая дорожка размечается на одно и то же количество блоков; таким образом, предельная емкость каждого блока составляет одно и то же число байтов. Для произведения обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока.

При выполнении обмена с диском аппаратура выполняет три основных действия:

- подвод головок к нужному цилиндру (обозначим время выполнения этого действия как  $t_{\text{ит}}$ );
- поиск на дорожке нужного блока (время выполнения —  $t_{\text{иб}}$ );
- собственно обмен с этим блоком (время выполнения —  $t_{\text{об}}$ ).

Тогда, как правило,  $t_{\text{ит}} \gg t_{\text{иб}} \gg t_{\text{об}}$ , потому что подвод головок — это механическое действие, причем в среднем нужно переместить головки на расстояние, равное половине радиуса

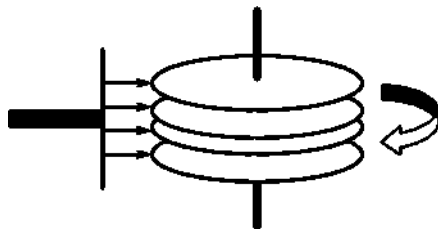


Рис. 1.1. Грубая схема дискового устройства памяти с подвижными головками

поверхности, а скорость передвижения головок не может быть слишком большой по физическим соображениям. Поиск блока на дорожке требует прокручивания пакета магнитных дисков в среднем на половину длины внешней окружности; скорость вращения диска может быть существенно больше скорости движения головок, но она тоже ограничена законами физики. Для выполнения же обмена нужно прокрутить пакет дисков всего лишь на угловое расстояние, соответствующее размеру блока. Таким образом, из всех этих действий в среднем наибольшее время занимает первое, и поэтому существенный выигрыш в суммарном времени обмена при считывании или записи только части блока получить практически невозможно.

С появлением магнитных дисков началась история систем управления данными во внешней памяти. До этого каждая прикладная программа, которой требовалось хранить данные во внешней памяти, сама определяла расположение каждой порции данных на магнитной ленте или барабане и выполняла обмены между оперативной и внешней памятью с помощью программно-аппаратных средств низкого уровня (машинных команд или вызовов соответствующих программ операционной системы). Такой режим работы не позволял или очень затруднял поддержание на одном внешнем носителе нескольких архивов долговременно хранимой информации. Кроме того, каждой прикладной программе приходилось решать проблемы именования частей данных и структуризации данных во внешней памяти.

## **1.2. Файловые системы**

Историческим шагом явилось появление систем управления файлами. С точки зрения программиста приложений — это именованная область внешней памяти, в которую можно записывать и из которой можно считывать данные. Правила именования файлов, способ доступа к данным, хранящимся в файле, и структура этих данных зависят от конкретной системы управления файлами и, возможно, от типа файла. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в соответствующие адреса внешней памяти и обеспечение доступа к данным.

В этом подразделе будут обсуждаться история файловых систем, их основные черты и области разумного применения. Однако сначала уместно сделать два замечания. Во-первых, заметим, что в области управления файлами исторически

существует некоторая терминологическая путаница. Термин *файловая система* (*file system*) используется для обозначения как программной системы, управляющей файлами, так и архива файлов, хранящегося во внешней памяти. Было бы лучше в первом случае использовать термин *система управления файлами*, оставив за термином *файловая система* только второе значение. Однако принятая практика вынуждает использовать в этой книге термин *файловая система* в обоих смыслах. Точный смысл термина должен быть понятен из контекста. (Аналогичная путаница возникает при некорректном использовании терминов *база данных* и *система управления базами данных*. В данной книге эти термины строго различаются.) Во-вторых, для целей этой главы достаточно ограничиться описанием основных свойств так называемых традиционных файловых систем, не затрагивая особенности современных систем с повышенной надежностью.

Первая развитая файловая система была разработана специалистами IBM в середине 1960-х гг. для выпускавшейся компанией серии компьютеров System/360. В этой системе поддерживались как чисто последовательные, так и индексно-последовательные файлы (а также файлы с прямым доступом к записям), а реализация во многом опиралась на возможности только появившихся к этому времени контроллеров управления дисковыми устройствами. Контроллеры обеспечивали возможность обмена с дисковыми устройствами порциями данных произвольного размера, а также индексный доступ к записям файлов, и эти функции контроллеров активно использовались в файловой системе OS/360.

Файловая система OS/360 обеспечила будущих разработчиков уникальным опытом использования дисковых устройств с подвижными головками, который отражается во всех современных файловых системах.

### 1.2.1. Структуры файлов

Практически во всех современных компьютерах основными устройствами внешней памяти являются магнитные диски с подвижными головками, и именно они служат для хранения файлов. Как отмечалось ранее, аппаратура магнитных дисков допускает выполнение обмена с дисками порциями данных произвольного размера. Однако возможность обмениваться с магнитными дисками порциями, размеры которых меньше полного объема блока, в настоящее время в файловых системах не используется. Это связано с двумя обстоятельствами.

Во-первых, считывание или запись только части блока не приводит к существенному выигрышу в суммарном времени обмена, поскольку, как указывалось в подразд. 1.1, основная часть этого времени тратится на перемещение магнитных головок. Во-вторых, для работы с частями блоков файловая система должна обеспечить буферы оперативной памяти соответствующего размера, что существенно усложняет распределение основной памяти. Алгоритмы распределения памяти порциями произвольного размера плохи тем, что любой из них рано или поздно приводит к *внешней фрагментации* памяти. В памяти образуется большое число мелких свободных фрагментов. Их совокупный размер может быть больше размера любого требуемого буфера, но его можно выделить, только если произвести сжатие памяти, т.е. подвижку всех занятых фрагментов таким образом, чтобы они располагались вплотную один к другому. Во время выполнения операции сжатия памяти нужно приостановить выполнение обменов, а сама эта операция занимает много времени.

Поэтому во всех современных файловых системах явно или неявно выделяется уровень, обеспечивающий работу с *базовыми файлами*, которые представляют собой наборы блоков, последовательно нумеруемых в адресном пространстве файла и отображаемых на физические блоки диска (рис. 1.2). Размер логического блока файла совпадает с размером физического блока диска или кратен ему; обычно размер логического блока выбирается равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В некоторых файловых системах базовый уровень был доступен пользователю, но чаще он прикрывался некоторым более

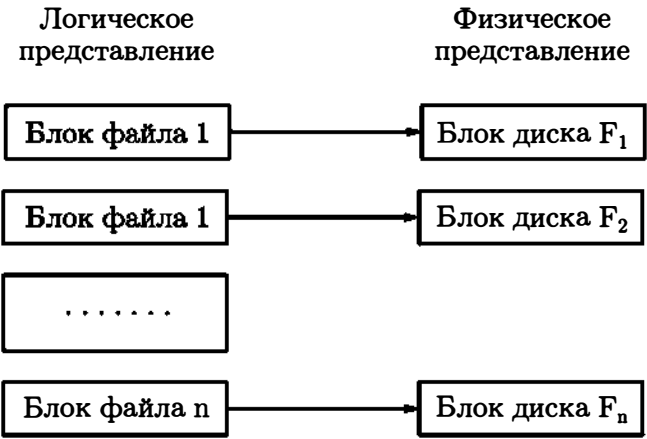


Рис. 1.2. Схематичное изображение базового файла

высоким уровнем, стандартным для пользователей. Исторически существует два основных подхода. При первом подходе, собственном, например, файловой системе операционной системы компании Hewlett-Packard OpenVMS, пользователи представляют файл как последовательность записей. Каждая запись — это последовательность байтов, имеющая постоянный или переменный размер. Можно читать или писать записи последовательно либо позиционировать файл на запись с указанным номером.

В некоторых файловых системах допускается структуризация записей на поля и объявление указываемых полей ключами записи. В таких файловых системах можно потребовать выборку записи из файла по ее заданному ключу. Естественно, в этом случае файловая система поддерживает в том же (или другом, служебном) базовом файле дополнительные, невидимые пользователю, служебные структуры данных — *индексы*. Распространенные способы организации индексов ключевых файлов основаны на технике *хэширования* и *В-деревьев* (подробно эта техника обсуждается в гл. 8 в контексте физической организации хранения баз данных). Существуют и многоключевые способы организации файлов (у одного файла объявляется несколько ключей, и можно выбирать записи по значению каждого ключа).

Второй подход, получивший распространение вместе с операционной системой UNIX, состоит в том, что любой файл представляется как непрерывная последовательность байтов. Из файла можно прочесть указанное число байтов, либо начиная с его начала, либо предварительно выполнив его позиционирование на байт с указанным номером. Аналогично можно записать указанное число байтов либо в конец файла, либо предварительно выполнив позиционирование файла. Тем не менее заметим, что скрытым от пользователя, но существующим во всех разновидностях файловых систем ОС UNIX является базовое блочное представление файла.

Конечно, в обоих случаях можно обеспечить набор преобразующих функций, приводящих представление файла к другому виду. Примером тому может служить поддержка стандартной файловой среды UNIX в среде операционной системы OpenVMS.

### **1.2.2. Логическая структура файловых систем и именование файлов**

Во всех современных файловых системах обеспечивается многоуровневое именование файлов за счет наличия во внешней памяти каталогов — дополнительных файлов со специальной

структурой. Каждый каталог содержит имена каталогов и/или файлов, хранящихся в данном каталоге. Таким образом, полное имя файла состоит из списка имен каталогов плюс имя файла в каталоге, непосредственно содержащем данный файл.

Поддержка многоуровневой схемы именования файлов обеспечивает несколько преимуществ, основным из которых является простая и удобная схема логической классификации файлов и генерации их имен. Можно сопоставить каталог или цепочку каталогов с пользователем, подразделением, проектом и затем образовывать в этом каталоге файлы или каталоги, не опасаясь коллизий с именами других файлов или каталогов.

Разница между способами именования файлов в разных файловых системах состоит в том, с чего начинается эта цепочка имен. В любом случае первое имя должно соответствовать корневому каталогу файловой системы. Вопрос заключается в том, как сопоставить этому имени корневой каталог — где его искать? В связи с этим имеются два радикально различных подхода.

Во многих системах управления файлами требуется, чтобы каждый архив файлов (полное дерево каталогов) целиком располагался на одном дисковом пакете или логическом диске — разделе физического дискового пакета, логически представляемом в виде отдельного диска с помощью средств операционной системы. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Такой способ именования использовался в файловых системах компаний IBM и DEC; очень близки к этому и файловые системы, реализованные в операционных системах семейства Windows компании Microsoft. Можно назвать такую организацию поддержкой изолированных файловых систем.

Другой крайний вариант был реализован в файловых системах операционной системы Multics. Эта система заслуживает отдельного разговора, в ней был реализован целый ряд оригинальных идей, но мы остановимся только на особенностях организации архива файлов. В файловой системе Multics пользователям обеспечивалась возможность представлять всю совокупность каталогов и файлов в виде единого дерева. Полное имя файла начиналось с имени корневого каталога, и пользователь не обязан был заботиться об установке на дисковое устройство каких-либо конкретных дисков. Сама система, выполняя поиск файла по его имени, запрашивала у оператора установку необходимых дисков. Такую файловую систему можно назвать полностью централизованной.

Конечно, во многом централизованные файловые системы удобнее изолированных: система управления файлами выполняет больше рутинной работы. В частности, администратор файловой системы автоматически оповещается о потребности установки требуемых дисковых пакетов; система обеспечивает равномерное распределение памяти на известных ей дисковых томах; возможна организация автоматического перемещения редко используемых файлов на более медленные носители внешней памяти; облегчается рутинная работа, связанная с резервным копированием.

Но в таких системах возникают существенные проблемы, если требуется перенести поддерево файловой системы на другую вычислительную установку. Поскольку файлы и каталоги любого логического поддерева могут быть физически разбросаны по разным дисковым пакетам и даже магнитным лентам, для такого переноса требуется специальная утилита, собирающая все объекты требуемого поддерева на одном внешнем носителе, не входящем в состав штатных устройств централизованной файловой системы. Конечно, даже при наличии такой утилиты выполнение процедуры физической сборки требует существенного времени.

Компромиссное решение применяется в файловых системах ОС UNIX. На базовом уровне в этих файловых системах поддерживаются изолированные архивы файлов. Один из таких архивов объявляется корневой файловой системой. Это делается на этапе генерации операционной системы, и после запуска операционная система «знает», на каком дисковом устройстве (физическом или логическом) располагается корневая файловая система. После запуска системы можно «смонтировать» корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему. Технически это осуществляется посредством создания в корневой файловой системе специальных пустых каталогов (точек монтирования).

Специальный системный вызов *mount* ОС UNIX позволяет подключить к одному из пустых каталогов корневой каталог указанного архива файлов. Выполнение такого действия приводит к «наложению» корневого каталога монтируемой файловой системы на каталог точки монтирования; корневой каталог приобретает имя каталога точки монтирования. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Если учесть, что обычно монтирование файловой системы производится при раскрутке системы (при



выполнении стартового командного файла), пользователи ОС UNIX, как правило, и не задумываются о происхождении общей файловой системы.

Кроме того, поддерживается системный вызов `unmount`, «отторгающий» ранее смонтированную файловую систему от общей иерархии. Конечно, все это заметно облегчает перенос частей файловой системы на другие установки.

### 1.2.3. Авторизация доступа к файлам

Поскольку файловая система является общим хранилищем файлов, принадлежащих, вообще говоря, разным пользователям, системы управления файлами должны обеспечивать *авторизацию* доступа к файлам. В общем виде подход состоит в том, что для каждого зарегистрированного пользователя и для каждого существующего файла файловой системы указываются действия, выполнение которых над данным файлом разрешено или запрещено данному пользователю (так называемый мандатный способ защиты — у каждого пользователя имеется или не имеется отдельный мандат для работы с каждым файлом). Применение мандатного способа авторизации доступа влечет за собой существенные накладные расходы, связанные с потребностью хранения избыточной информации и использованием этой информации для проверки правомочности доступа.

Поэтому в большинстве современных систем управления файлами применяется упрощенный подход к *авторизации* доступа к файлам, традиционно поддерживаемый, например в ОС UNIX (так называемый *дискреционный* подход). При использовании этого подхода с каждым зарегистрированным пользователем связывается пара целочисленных идентификаторов: идентификатор группы, к которой относится пользователь, и его собственный идентификатор. Этими же идентификаторами снабжается каждый процесс, запущенный от имени данного пользователя и имеющий возможность обращаться к системным вызовам файловой системы. Соответственно, при каждом файле хранится полный идентификатор пользователя (собственный идентификатор плюс идентификатор группы), который создал этот файл, и помечается, какие действия с файлом может производить он сам, какие действия с файлом доступны для остальных пользователей той же группы и что могут делать с файлом пользователи других групп. Для каждого файла контролируется возможность выполнения трех действий: чтение, запись и выполнение. Хранимая информация очень компактна (два целых

числа для представления идентификаторов и шкала из 9 бит для характеристики возможных действий), при проверке требуется небольшое количество действий, и этот способ контроля доступа в большинстве случаев удовлетворителен.

#### 1.2.4. Синхронизация многопользовательского доступа

Если операционная система поддерживает многопользовательский режим, может возникнуть ситуация, когда два или более пользователей (процессов) одновременно пытаются работать с одним и тем же файлом. Если все эти пользователи собираются только читать файл, ничего страшного не произойдет. Но если хотя бы один из них будет изменять файл, для корректной работы этой группы требуется взаимная синхронизация.

В файловых системах обычно применяется следующий подход. В операции открытия файла (первой и обязательной операции, с которой должен начинаться сеанс работы с файлом) помимо прочих параметров указывается режим работы (чтение или изменение). Если к моменту выполнения этой операции от имени некоторого процесса *A* файл уже открыт некоторым другим процессом *B*, причем существующий режим открытия несовместим с требуемым режимом (совместимы только режимы чтения), то в зависимости от особенностей системы либо процессу *A* сообщается о невозможности открытия файла в нужном режиме, либо процесс *A* блокируется до тех пор, пока процесс *B* не выполнит операцию закрытия файла.

#### 1.2.5. Области разумного применения файлов

После краткого обзора технологии файловых систем обсудим возможные области их применения. Чаще всего файлы используются для хранения текстовых данных: документов, текстов программ и т. д. Такие файлы обычно создаются и модифицируются с помощью различных текстовых редакторов. Эти редакторы могут быть очень простыми, такими, как *ed* в мире UNIX или утилиты редактирования (*Far Manager*, *WordPad* и т. д.) в среде Windows. Они могут быть сложными и многофункциональными, синтаксически ориентированными, как, например, *GNU Emacs*. Но обычно структура текстовых файлов очень проста (с точки зрения файловой системы): это либо последовательность записей, содержащих строки текста, либо последовательность байтов, среди которых встречаются специальные символы (например, символы конца строки). Конечно же, сложность логической

структуры текстового файла определяется текстовым редактором, но в любом случае файловой системе она не видна.

Файлы, содержащие тексты программ, используются как входные файлы компиляторов (чтобы правильно воспринять текст программы, компилятор должен понимать логическую структуру текстового файла), которые, в свою очередь, формируют файлы, содержащие объектные модули. С точки зрения файловой системы объектные файлы также обладают очень простой структурой — последовательность записей или байтов. Система программирования накладывает на такую структуру более сложную и специфичную для этой системы логическую структуру объектного модуля. Подчеркнем, что логическая структура объектного модуля файловой системе неизвестна; эта структура поддерживается инструментами системы программирования.

Аналогично обстоит дело с файлами, формируемыми редакторами связей (редактор связей должен понимать логическую структуру файлов объектных модулей) и содержащими образы выполняемых программ. Логическая структура таких файлов остается известной только редактору связей и загрузчику — программе операционной системы. Общая схема взаимодействия программных компонентов при построении программы показана на рис. 1.3. Мы кратко обозначили способы использования файлов в процессе разработки программ, но можно сказать, что ситуация аналогична и в других случаях: например, при создании и использовании файлов, содержащих графическую, аудио- и видеоинформацию.

Одним словом, файловые системы обычно обеспечивают хранение данных с очень «аморфной» стандартной структурой, оставляя дальнейшую структуризацию прикладным программам. В перечисленных ранее случаях использования файлов это

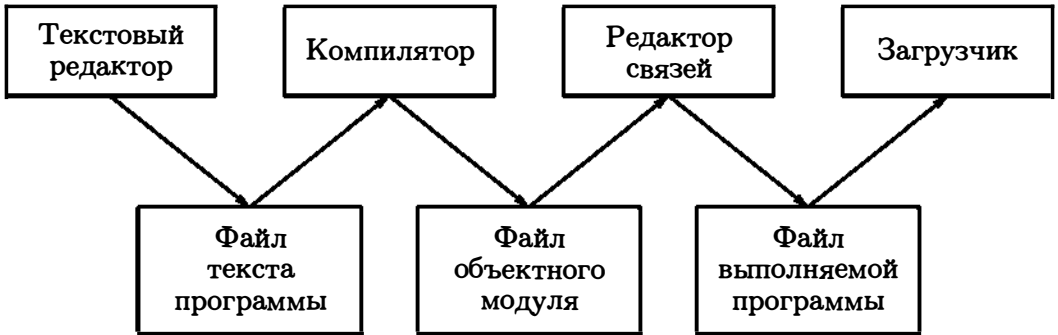


Рис. 1.3. Связи между программными компонентами по пониманию логической структуры файлов

даже хорошо, поскольку при разработке любой новой прикладной системы, опираясь на простые, стандартные и сравнительно дешевые средства файловой системы, можно реализовать те структуры хранения, которые наиболее точно соответствуют специфике данной прикладной области.

### 1.3. Потребности информационных систем

Удовлетворяют ли рассмотренные ранее базовые возможности файловых систем потребности информационных систем? Типовая информационная система, главным образом, ориентирована на хранение, выбор и модификацию данных соответствующей прикладной области. Структура таких данных зачастую очень сложна, и, хотя структуры данных различны в разных информационных системах, между ними часто бывает много общего.

На начальном этапе использования вычислительной техники для построения информационных систем проблемы структуризации данных решались индивидуально в каждой информационной системе. Производились необходимые надстройки над файловыми системами (библиотеки программ), подобно тому, как это делается в компиляторах, редакторах и т. д. (рис. 1.4).

Но поскольку для функционирования информационных систем требуются сложные структуры данных, эти дополнительные индивидуальные средства управления данными являлись существенной частью информационных систем и практически повторялись от одной системы к другой. Стремление выделить общую часть информационных систем, ответственную за управление сложно структурированными данными, явилось, на мой взгляд, первой побудительной причиной создания СУБД. Очень скоро стало понятно, что невозможно обойтись общей библиотекой программ (рис. 1.5), реализующей над стандартной базовой файловой системой более сложные методы хранения данных.

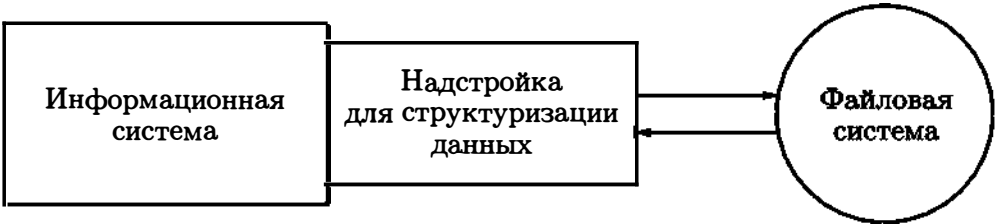


Рис. 1.4. Примитивная схема структуризации данных в информационной системе

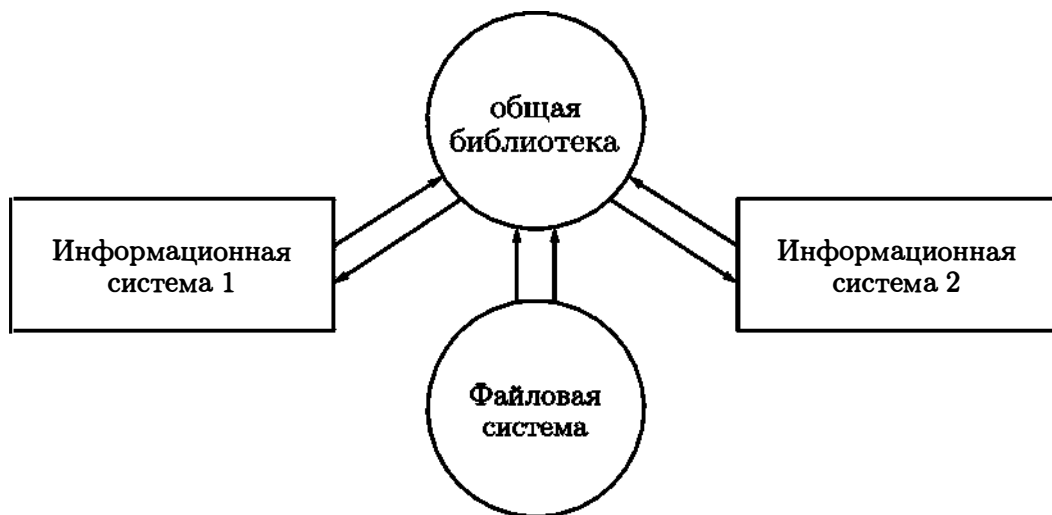


Рис. 1.5. Две информационные системы с общей библиотекой

Поясним это на примере. Предположим, что требуется реализовать простую информационную систему, поддерживающую учет служащих некоторой организации. 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.

Кроме того, для каждого отдела должна поддерживаться возможность получения:

- имени руководителя отдела;
- общей численности отдела;
- общей суммы заработной платы служащих отдела, среднего размера заработной платы 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 заработной платы.

### 1.3.1. Структуры данных

Предположим, что мы решили основывать эту информационную систему на файловой системе и пользоваться одним файлом

**СЛУЖАЩИЕ**, расширив базовые возможности файловой системы за счет специальной библиотеки функций. Поскольку минимальной информационной единицей в нашем случае является служащий, в этом файле должна содержаться одна запись для каждого служащего. Чтобы можно было удовлетворить указанные ранее требования, запись о служащем должна иметь следующие поля:

- полное имя служащего (**СЛУ\_ИМЯ**);
- номер его удостоверения (**СЛУ\_НОМЕР**);
- данные о соответствии служащего занимаемой должности (**СЛУ\_СТАТ**; для простоты «да» или «нет», соответствует или не соответствует должности);
- размер заработной платы (**СЛУ\_ЗАРП**);
- номер отдела (**СЛУ\_ОТД\_НОМЕР**).

Поскольку мы решили ограничиться одним файлом **СЛУЖАЩИЕ**, та же запись должна содержать имя руководителя отдела (**СЛУ\_ОТД\_РУК**). Иначе было бы невозможно, например, получить имя руководителя отдела с известным номером.

Чтобы информационная система могла эффективно выполнять свои базовые функции, необходимо обеспечить многоключевой доступ к файлу **СЛУЖАЩИЕ** по уникальным ключам (ключ называется уникальным, если его значения гарантированно различны во всех записях файла) **СЛУ\_ИМЯ** и **СЛУ\_НОМЕР**. Очевидно, что в противном случае для выполнения наиболее часто используемых операций получения данных о конкретном служащем понадобится последовательный просмотр в среднем половины записей файла.

Кроме того, должна обеспечиваться возможность эффективного выбора всех записей с общим значением **СЛУ\_ОТД\_НОМЕР**, т.е. доступ по неуникальному ключу. Если не поддерживать специальный механизм доступа, то для получения данных об отделе в целом в общем случае потребуется полный просмотр файла. Требуемая общая структура файла **СЛУЖАЩИЕ** показана на рис. 1.6. Но даже в этом случае, чтобы получить численность отдела или общий размер заработной платы, 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:

- 1) требуется создание достаточно сложной надстройки для многоключевого доступа к файлам;
- 2) возникает существенная избыточность данных (для каждого служащего повторяется имя руководителя его отдела);

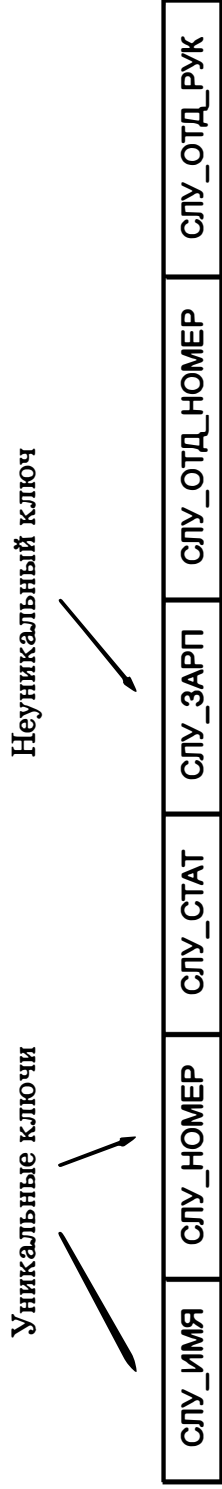


Рис. 1.6. Структура файла СЛУЖАЩИЕ на уровне приложения (случай одного файла)

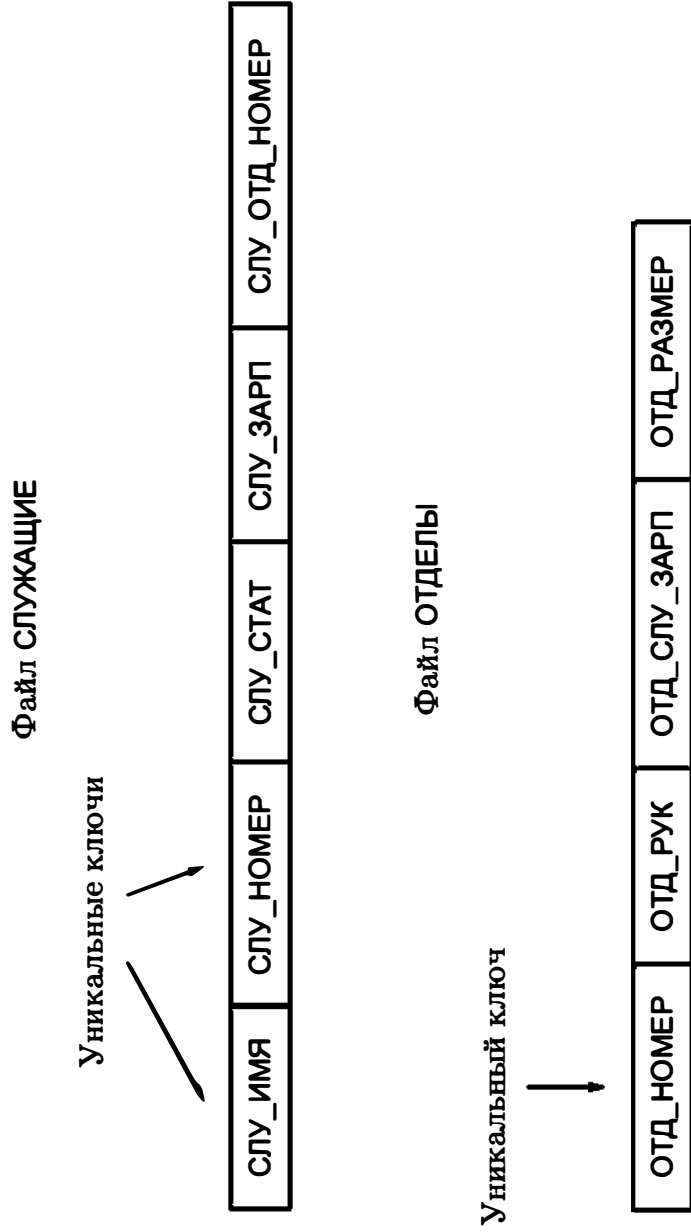


Рис. 1.7. Структура файлов СЛУЖАЩИЕ и ОТДЕЛЫ на уровне приложения (случай двух файлов)

3) 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 заработную плату, то либо придется при выполнении каждой такой операции полностью просматривать файл, либо нужно будет реструктурировать файл **СЛУЖАЩИЕ**, объявляя ключевым и поле **СЛУ\_ЗАРП**.

Для улучшения ситуации можно было бы поддерживать два многоключевых файла: **СЛУЖАЩИЕ** и **ОТДЕЛЫ**. Первый файл должен был бы содержать поля **СЛУ\_ИМЯ**, **СЛУ\_НОМЕР**, **СЛУ\_СТАТ**, **СЛУ\_ЗАРП** и **СЛУ\_ОТД\_НОМЕР**, а второй — **ОТД\_НОМЕР**, **ОТД\_РУК** (номер удостоверения служащего, являющегося руководителем отдела), **ОТД\_СЛУ\_ЗАРП** (общий размер заработной платы служащих данного отдела) и **ОТД\_РАЗМЕР** (общее число служащих в отделе). Структура этих файлов показана на рис. 1.7.

Введение этих двух файлов позволило бы преодолеть большинство неудобств, перечисленных в предыдущем абзаце. Каждый из файлов содержал бы только не дублируемую информацию, не возникала бы необходимость в динамических вычислениях суммарной информации по отделам. Но заметим, что при таком переходе наша информационная система должна обладать некоторыми новыми особенностями, сближающими ее с СУБД.

### 1.3.2. Целостность данных

Теперь система должна «знать», что она работает с двумя информационно связанными файлами (это шаг в сторону схемы базы данных), и иметь информацию о структуре и смысле каждого поля. Например, системе должно быть известно, что у полей **СЛУ\_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\_СЛУ\_ЗАРП** и **ОТД\_РАЗМЕР** 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 **СЛУ\_ОТД\_НОМЕР**, равным  $n$ , то и в файле **ОТДЕЛЫ** долж-



на содержаться запись со значением поля ОТД\_НОМЕР, также равным  $n$ ;

(2) если в файле ОТДЕЛЫ содержится запись со значением поля ОТД\_РУК, равным  $m$ , то и в файле СЛУЖАЩИЕ должна содержаться запись со значением поля СПУ\_НОМЕР, также равным  $m$ ; далее в книге мы увидим, что правила (1) и (2) являются частными случаями общего правила *ссылочной целостности*: поле СПУ\_ОТД\_НОМЕР содержит «ссылки» на записи таблицы ОТДЕЛЫ, и поле ОТД\_РУК содержит «ссылки» на записи таблицы СЛУЖАЩИЕ;

(3) при любом корректном состоянии информационной системы значение поля ОТД\_СПУ\_ЗАРП любой записи  $отд\_k$  файла ОТДЕЛЫ должно быть равно сумме значений поля СПУ\_ЗАРП всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СПУ\_ОТД\_НОМЕР совпадает со значением поля ОТД\_НОМЕР записи  $отд\_k$ ;

(4) при любом корректном состоянии информационной системы значение поля ОТД\_РАЗМЕР любой записи  $отд\_k$  файла ОТДЕЛЫ должно быть равно числу всех тех записей файла СЛУЖАЩИЕ, в которых значение поля СПУ\_ОТД\_НОМЕР совпадает со значением поля ОТД\_НОМЕР записи  $отд\_k$ ; далее будет показано, 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 в информационной системе (даже такой простой, как в рассматриваемом примере) поддерживается согласованное хранение данных в нескольких файлах, можно говорить о том, что в ней поддерживается база данных. Если же некоторая вспомогательная система управления данными позволяет работать с несколькими файлами, обеспечивая их согласованность, можно назвать ее системой управления базами данных.

Уже только требование поддержки согласованности данных в нескольких файлах не позволяет при построении информационной системы обойтись библиотекой функций: такая система должна обладать некоторыми собственными данными (их принято называть метаданными), определяющими структуру и целостность данных. В рассматриваемом 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).

### 1.3.3. Языки запросов

Обеспечение целостности данных — это далеко не все, что обычно требуется от СУБД. Начнем с того, что даже в рассматриваемом примере пользователю информационной системы будет не слишком просто получить, например, общую численность отдела, в котором работает Петр Иванович Сидоров. Придется сначала узнать номер отдела, в котором работает указанный служащий, а затем установить численность этого отдела. Было бы гораздо проще, если бы СУБД позволяла сформулировать такой запрос на языке, более близком пользователям. Такие языки называются *языками запросов к базам данных*. Например, на языке запросов SQL запрос можно было бы выразить в следующей форме (*запрос 1*):

```
SELECT ОТД_РАЗМЕР
FROM СЛУЖАЩИЕ, ОТДЕЛЫ
WHERE СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ' AND
      СЛУ_ОТД_НОМЕР = ОТД_НОМЕР;
```

Это пример запроса на языке SQL с «*полусоединением*»: с одной стороны, запрос адресуется к двум файлам — СЛУЖАЩИЕ и ОТДЕЛЫ, но с другой стороны, данные выбираются только из файла ОТДЕЛЫ. Условие СЛУ\_ОТД\_НОМЕР = ОТД\_НОМЕР всего лишь «ограничивает» интересующий нас набор записей об отделах до одной записи, если Петр Иванович Сидоров действительно работает на данном предприятии. Если же Петр Иванович Сидоров не работает на предприятии, то условие СЛУ\_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ' не будет удовлетворяться ни для одной записи файла СЛУЖАЩИЕ, и поэтому запрос выдаст пустой результат.

Возможна и другая формулировка того же запроса (*запрос 2*):

```
SELECT ОТД_РАЗМЕР
FROM ОТДЕЛЫ
WHERE ОТД_НОМЕР =
      (SELECT СЛУ_ОТД_НОМЕР
       FROM СЛУЖАЩИЕ
       WHERE СЛУ_ИМЯ = 'ПЕТР ИВАНОВИЧ СИДОРОВ');
```

Это пример запроса на языке SQL с *вложенным подзапросом*. Во вложенном подзапросе выбирается значение поля СЛУ\_ОТД\_НОМЕР из записи файла СЛУЖАЩИЕ, в которой значение поля СЛУ\_ИМЯ равняется строковой константе 'ПЕТР ИВАНОВИЧ СИДОРОВ'. Если такая запись существует, то она единственная,

поскольку поле **СЛУ\_ИМЯ** является уникальным ключом файла **СЛУЖАЩИЕ**. Тогда результатом выполнения подзапроса будет единственное значение — номер отдела, в котором работает Петр Иванович Сидоров. 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 Сидоров 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 БД будет содержаться информация о том, что поле **СЛУ\_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 **СЛУ\_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. Можно формально доказать, что формулировки *запроса 1* и *запроса 2* эквивалентны, т.е. вне зависимости от состояния данных всегда производят один и тот же результат. Наиболее вероятным способом выполнения запроса в обеих формулировках будет выборка записи из файла **СЛУЖАЩИЕ** со значением поля **СЛУ\_ИМЯ**, равным строке 'ПЕТР ИВАНОВИЧ СИДОРОВ', взятие из этой записи значения поля **СЛУ\_ОТД\_НОМЕР** и выборка из таблицы **ОТДЕЛЫ** записи с таким же значением поля **ОТД\_НОМ**.

При возникновении потребности в получении списка сотрудников, не соответствующих занимаемой должности, достаточно обратиться к системе с запросом (*запрос 3*):

```
SELECT СЛУ_ИМЯ, СЛУ_НОМЕР  
FROM СЛУЖАЩИЕ  
WHERE СЛУ_СТАТ = 'НЕТ';
```

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

#### **1.3.4. Транзакции, журнализация и многопользовательский режим**

Представим себе, что в первоначальной реализации информационной системы, основанной на использовании библиотек

расширенных методов доступа к файлам, обрабатывается операция принятия на работу нового служащего. Следуя требованиям согласованного изменения файлов, информационная система вставляет новую запись в файл **СЛУЖАЩИЕ** и собирается модифицировать соответствующую запись файла **ОТДЕЛЫ** (или вставлять в этот файл новую запись, если служащий является первым в своем отделе), но именно в этот момент происходит (например) аварийное выключение питания компьютера.

Очевидно, что после перезапуска системы ее база данных будет находиться в рассогласованном состоянии (точно будут нарушены правила (3) и (4), а может быть, и правила (1) и (2), сформулированные в подразд. 1.3.2). Потребуется выяснить это (а для этого нужно явно проверить соответствие данных в файлах **СЛУЖАЩИЕ** и **ОТДЕЛЫ**) и привести данные в согласованное состояние. Проверку и коррекцию можно выполнить, например, следующим образом. Сгруппировать записи файла **СЛУЖАЩИЕ** по значениям поля **СЛУ\_ОТД\_НОМЕР**. Для каждой группы (а) проверить, существует ли в файле **ОТДЕЛЫ** запись, значение поля **ОТД\_НОМ** которой равняется значению поля **СЛУ\_ОТД\_НОМЕР** записей данной группы; если такой записи в файле **ОТДЕЛЫ** нет, то (b) исключить группу из файла **СЛУЖАЩИЕ** и перейти к обработке следующей группы; иначе (c) посчитать число записей в группе и вычислить суммарное значение заработной платы; (d) обновить полученными значениями поля **ОТД\_РАЗМЕР** и **ОТД\_СЛУ\_ЗАРП** соответствующей записи файла **ОТДЕЛЫ** и перейти к обработке следующей группы.

Настоящие СУБД берут такую работу на себя, поддерживая транзакционное управление и журнализацию изменений базы данных (см. подразд. 1.4). Прикладная система не обязана заботиться о поддержке корректности состояния БД, хотя и должна «знать», какие цепочки операций изменения данных являются допустимыми.

Представим теперь, что в информационной системе требуется обеспечить параллельную (например, многотерминальную) работу с базой данных служащих и отделов. Если опираться только на использование файлов, то для обеспечения корректности на все время модификации любого из двух файлов доступ других пользователей к этому файлу будет заблокирован (вспомните возможности файловых систем в отношении синхронизации параллельного доступа, упоминавшиеся в подразд. 1.2.5). Таким образом, зачисление на работу Петра Ивановича Сидорова существенно затормозит получение информации о служащем Иване Сидоровиче Петрове, даже если они работают в разных

отделах. Настоящие СУБД обеспечивают гораздо более тонкую синхронизацию параллельного доступа к данным.

## 1.4. Основные функции и компоненты СУБД

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

### 1.4.1. СУБД как независимый системный компонент

В начале подраздела необходимо уточнить некоторые положения. Во-первых, очевидно, что СУБД должна поддерживать достаточно развитую функциональность. Повторять эту функциональность в каждой информационной системе неразумно. Во-вторых, неясно, каким образом можно обеспечить готовый к использованию компонент СУБД, который можно было бы встраивать в информационные системы\*. В-третьих, уже должно быть понятно, что набор файлов можно назвать базой данных только при наличии метаданных. На рис. 1.8 метаданные являются принадлежностью информационной системы, и поэтому, например, файлы **СЛУЖАЩИЕ** и **ОТДЕЛЫ** можно эффективно использовать только через нашу гипотетическую систему регистрации служащих.

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

*Вариант 1.* Теоретически можно «внедрить» бухгалтерскую систему в состав системы регистрации сотрудников. Но ведь, как правило, бухгалтерские системы покупаются в виде готовых и отдельных продуктов, не приспособленных к подобному «внедрению».

*Вариант 2.* Можно скопировать метаданные системы регистрации служащих в бухгалтерскую систему. Но метаданные (как и данные) не обязательно являются статичными. Структура

---

\* Это замечание применимо, по крайней мере, к СУБД *общего назначения*, поддерживающей все возможности, рассмотренные в подразд. 1.3. Встраиваемые *специализированные* СУБД достаточно распространены.



Рис. 1.8. СУБД в составе информационной системы

базы данных может со временем изменяться, могут исчезать одни правила целостности и появляться другие. Как согласовывать копии метаданных, поддерживаемые независимыми информационными системами? Так мы приходим к организации системы, показанной на рис. 1.9, где можно видеть 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 рис. 1.9

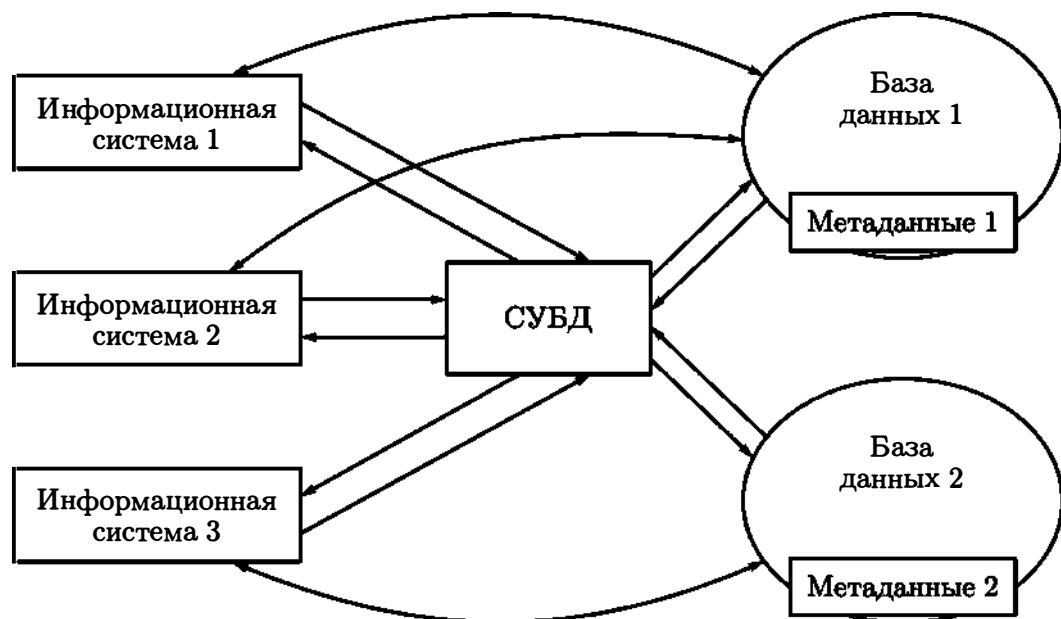


Рис. 1.9. Отдельная СУБД и базы данных с метаданными

вплотную приближает нас к наиболее распространенной в последние десятилетия архитектуре «клиент-сервер». СУБД играет роль «сервера», обсуживающего нескольких «клиентов» — прикладных информационных систем.

### 1.4.2. Функции СУБД

В подразд. 1.3 было выявлено несколько потребностей информационных систем, которые не покрываются возможностями систем управления файлами: поддержка логически согласованного набора файлов; восстановление согласованного состояния данных после разного рода сбоев; обеспечение высокого уровня параллелизма работы нескольких пользователей; поддержка языка манипулирования данными. Эти и другие функции традиционно поддерживаются СУБД. В этом подразделе обсудим функции СУБД более строго.

**Структуризация данных во внешней памяти.** Эта функция обеспечивает *поддержку необходимых структур данных во внешней памяти* как для хранения данных и матаданных, непосредственно входящих в БД, так и для служебных целей, например, для ускорения доступа к данным в некоторых случаях (обычно для этого используются индексы). В некоторых реализациях СУБД активно используются возможности существующих файловых систем, в других работа производится на уровне устройств внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД файловую систему, и, если использует, то как организованы файлы. В частности, в СУБД обычно поддерживается собственная система именования объектов БД.

**Управление буферами оперативной памяти.** СУБД обычно работают с БД значительного размера; по крайней мере, этот размер обычно существенно больше объема доступной оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является *буферизация данных в оперативной памяти*. Даже если операционная система производит общесистемную буферизацию данных (как, например, в случае ОС UNIX), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части БД. Поэтому в развитых СУБД поддерживается собственный набор буферов

оперативной памяти с использованием собственной дисциплины замены буферов\*.

**Управление транзакциями.** *Транзакция* — это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно завершается и СУБД фиксирует изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример информационной системы с файлами **СЛУЖАЩИЕ** и **ОТДЕЛЫ**, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового служащего является объединение элементарных операций над файлами **СЛУЖАЩИЕ** и **ОТДЕЛЫ** в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД. Но понятие транзакции гораздо более важно в многопользовательских СУБД.

То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к БД. При соответствующем управлении посредством СУБД параллельно выполняемыми транзакциями каждый пользователь может в принципе ощущать себя единственным пользователем СУБД (на самом деле, это несколько идеализированное представление, поскольку в некоторых случаях пользователи многопользовательских СУБД могут ощутить присутствие своих коллег).

С управлением транзакциями в многопользовательской СУБД связаны важные понятия *сериализации транзакций*. Под сериализацией параллельно выполняемых транзакций понимается такой порядок планирования выполнения операций, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Понятно, что если удастся добиться действительно сериализованного выполнения

---

\* Следует оговориться, что существует класс СУБД, ориентированный на работу с базами данных, постоянно и полностью размещаемыми в основной памяти (Main-Memory DBMS). В этих системах, естественно, отсутствует компонент управления буферами и вообще многое устроено по-другому. Однако, по крайней мере, пока эти системы играют лишь вспомогательную роль при управлении данными, и в этой книге их особенности не обсуждаются.



смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. Наиболее распространены алгоритмы, основанные на синхронизационных блокировках объектов БД. При использовании любого алгоритма сериализации возможны конфликты между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержки сериализации необходимо выполнить *откат* (ликвидировать все изменения, произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

**Журнализация.** Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризующиеся потерей информации на носителях внешней памяти. Примерами программных сбоев могут быть аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной. Первую ситуацию можно рассматривать как особый вид мягкого аппаратного сбоя; при возникновении последней требуется ликвидировать последствия только одной транзакции.

Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, для обеспечения надежного хранения данных в БД требуется хранение избыточных данных, причем та часть данных, которая используется для восстановления БД, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение *журнала изменений* БД.

Журнал — это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью (например,

можно поддерживать две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. В разных СУБД изменения БД журналируются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения БД (например, операции удаления строки из таблицы реляционной БД), иногда — минимальной внутренней операции модификации страницы внешней памяти; в некоторых системах одновременно используются оба подхода.

Во всех случаях принято придерживаться стратегии «упреждающей» записи в журнал (так называемого протокола Write Ahead Log — WAL). Грубо говоря, эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части БД. Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

**Поддержка языков БД.** Для работы с базами данных используются специальные языки, в целом называемые *языками баз данных*. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка — *язык определения данных* (*DDL — Data Definition Language*) и *язык манипулирования данными* (*DML — Data Manipulation Language*). При этом язык DDL служил, главным образом, для определения логической структуры БД, т. е. той структуры БД, какой она представляется пользователям; язык DML содержал набор операторов манипулирования данными, т. е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Более подробно языки ранних СУБД будут рассмотрены в гл. 2.

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (*Standard Query Language*). В нескольких главах этой книги язык SQL будет рассматриваться достаточно подробно, а пока мы перечислим основные функции реляционной СУБД, поддерживаемые на «языковом» уровне (т. е. функции, поддерживаемые при реализации интерфейса SQL).

Прежде всего, язык SQL сочетает средства языков DDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД — в основном, именование таблиц и их столбцов) поддерживается на языковом уровне в том смысле, что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых служебных таблиц-каталогов. Внутренняя часть СУБД (ядро) вообще не работает с именами таблиц и их столбцов.

Язык SQL содержит специальные средства определения ограничений целостности БД. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при первичной обработке операторов модификации БД компилятор SQL на основании имеющихся в БД ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые представления БД, фактически являющиеся хранимыми в БД запросами (результатом любого запроса к реляционной БД является таблица) с именованными столбцами. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в БД, но с помощью представлений можно ограничить или, наоборот, расширить видение БД для конкретного пользователя. Поддержание представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам БД производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу БД, обладает полным набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

### **1.4.3. Типовая организация современной СУБД**

Организация типичной СУБД и состав ее компонентов соответствуют рассмотренному ранее набору функций. Напомним, что мы выделили следующие основные функции СУБД:

- управление данными во внешней памяти;
- управление буферами оперативной памяти;
- управление транзакциями;
- журнализация и восстановление БД после сбоев;
- поддержка языков БД.

Логически в современной реляционной СУБД можно выделить внутреннюю часть — ядро СУБД (часто его называют Data Base Engine), компилятор языка БД (например, SQL), подсистему поддержки времени выполнения, набор утилит. В некоторых системах эти части выделяются явно, в других — нет, но логически такое разделение можно провести во всех СУБД.

Ядро СУБД отвечает за управление данными во внешней памяти, управление буферами оперативной памяти, управление транзакциями и журнализацию. Соответственно, можно выделить такие компоненты ядра (по крайней мере, логически, хотя в некоторых системах эти компоненты выделяются явно), как менеджер данных, менеджер буферов, менеджер транзакций и менеджер журнала. Как можно понять из предшествующего материала этой главы, функции этих компонентов взаимосвязаны, и для обеспечения корректной работы СУБД все эти компоненты должны взаимодействовать по тщательно продуманным и проверенным протоколам. Ядро СУБД обладает собственным интерфейсом, обычно недоступным пользователям напрямую и используемым в программах, производимых компилятором SQL (или в подсистеме поддержки выполнения таких программ) и утилитах БД. Ядро СУБД является основной резидентной частью СУБД. При использовании архитектуры «клиент-сервер» ядро является базовой составляющей серверной части системы.

Главной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Существенной проблемой реляционных СУБД является то, что языки этих систем (а это, как правило, SQL) являются процедурными, т. е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия. Поэтому компилятор должен решить, каким образом выполнять оператор языка, прежде чем произвести программу. Применяются достаточно сложные методы оптимизации операторов. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах, но более часто в выполняемом внутреннем машинно-независимом коде. В последнем случае реальное вы-

полнение оператора производится с привлечением подсистемы поддержки времени выполнения, представляющей собой, по сути дела, интерпретатор этого внутреннего языка.

Наконец, в отдельные утилиты БД обычно выделяют такие процедуры, которые слишком накладно выполнять с использованием языка БД, например, загрузка и выгрузка БД, сбор статистики, глобальная проверка целостности БД и т.д. Утилиты программируются с использованием интерфейса ядра СУБД, а иногда даже с проникновением внутрь ядра.

\* \* \*

Развитие аппаратных и программных средств управления внешней памятью диктовалось потребностями информационных систем, для построения которых требовалась возможность надежного долговременного хранения больших объемов данных, а также обеспечение достаточно быстрого доступа к этим данным.

Системы управления файлами во внешней памяти обеспечивают минимальные потребности информационных систем, предоставляя средства распределения и структуризации дисковой памяти, именования файлов, авторизации доступа и поддержки многопользовательского режима на уровне файлов. По мере развития технологии информационных систем их потребности возрастали, выходя за пределы возможностей, обеспечиваемых файловыми системами.

На примере тривиальной информационной системы были показаны ситуации, в которых возможности файловых систем явно недостаточны. Более того, попытки расширения возможностей файловой системы путем включения в приложение дополнительных программных компонентов во многих случаях не приводят к успеху. В пределах такие попытки могут привести к появлению самостоятельного программного продукта, обладающего некоторыми чертами СУБД. Однако настоящие СУБД являются настолько большими и сложными программными системами, что вероятность успешного создания «самодельной» СУБД ничтожно мала.

При выборе технологии построения информационной системы нужно тщательно оценивать и прогнозировать ее потенциальные потребности в средствах управления данными. Конечно, любую информационную систему можно основывать на использовании промышленной, большой и мощной СУБД. Но вполне может

оказаться так, что в действительности приложение будет использовать доли процентов общих возможностей СУБД. Накладные расходы (затраты на дополнительную аппаратуру, лицензирование дорогостоящего программного продукта, увеличение общего времени выполнения операций) могут оказаться неоправданными.

СУБД решают множество проблем, которые затруднительно или вообще невозможно решить при использовании файловых систем. При этом существуют приложения, для которых вполне достаточно файлов; приложения, для которых необходимо решать, какой уровень работы с данными во внешней памяти для них требуется, и приложения, для которых, безусловно, нужны базы данных.

# ПОНЯТИЕ МОДЕЛИ ДАННЫХ. ОБЗОР РАЗНОВИДНОСТЕЙ МОДЕЛЕЙ ДАННЫХ

Историю технологии БД принято отсчитывать с начала 1960-х гг., когда были предприняты первые попытки создания специальных программных средств управления базами данных. За прошедшие десятилетия возникали и использовались различные подходы к организации баз данных. Для описания и сравнения некоторых из них воспользуемся понятием *модели данных*, предложенным в 1969 г. Э.Коддом [1] для описания конкретного *реляционного подхода* к организации БД. Соответственно, он говорил о *реляционной модели данных*, различным теоретическим и реализационным аспектам которой в основном посвящена эта книга. Однако понятие модели данных оказалось удобным не только для описания реляционного подхода и сравнения реализаций реляционных СУБД, но и для реализационно-независимого представления и сопоставления других подходов к организации баз данных.

## 2.1. Модель данных

В модели данных описывается некоторый набор родовых понятий и признаков, которыми должны обладать все конкретные СУБД и управляемые ими базы данных, если они основываются на этой модели. Наличие модели данных позволяет сравнивать конкретные реализации, используя один общий язык.

Несмотря на то что понятие модели данных было введено Коддом, наиболее распространенная трактовка модели данных, по-видимому, принадлежит Кристоферу Дейту, который воспроизводит ее (с различными уточнениями) применительно к реляционным БД практически во всех своих книгах (см., например, [2]). Согласно Дейту реляционная модель состоит из трех частей, описывающих разные аспекты реляционного подхода, — структурной, манипуляционной и целостной.

В структурной части модели данных фиксируются виды основных логических структур данных, которые могут при-

меняться на уровне пользователя при организации БД, соответствующих данной модели. Например, в модели данных SQL основным видом структур базы данных являются таблицы, а в объектной модели данных — объекты ранее определенных типов. В структурной части модели данных может также специфицироваться некоторый язык определения логической структуры (структурной схемы) баз данных, соответствующих данной модели (язык определения данных, DDL).

В манипуляционной части модели данных специфицируется один или нескольких языков, предназначенных для написания запросов к БД (языков манипулирования данными, DML). Эти языки могут быть абстрактными, не обладающими точно проработанным синтаксисом (что свойственно языками реляционной алгебры и реляционного исчисления, используемым в реляционной модели данных), или законченными производственными языками (как в случае модели данных SQL). Основное назначение манипуляционной части модели данных — обеспечить эталонный «модельный» операционный язык БД, уровень выразительности которого должен поддерживаться в реализациях СУБД, соответствующих данной модели.

Наконец, в целостной части модели данных (которая явно выделяется не во всех известных моделях) специфицируются механизмы ограничений целостности, которые обязательно должны поддерживаться во всех реализациях СУБД, соответствующих данной модели. Например, в целостной части реляционной модели данных категорически требуется поддержка ограничения первичного ключа в любой переменной отношения, а аналогичное требование к таблицам в модели данных SQL отсутствует.

В этой главе применим понятие модели данных для обзора подходов, как предшествовавших появлению реляционных баз данных, так возникших позже, не касаясь особенностей каких-либо конкретных систем — это привело бы к изложению многих технических деталей, которые, хотя и интересны, но находятся несколько в стороне от основной цели данной книги. Детали можно найти в рекомендуемой дополнительной литературе.

## **2.2. Ранние модели данных**

Начнем с рассмотрения общих подходов к организации трех типов ранних систем, а именно систем, основанных на инвертированных списках, иерархических и сетевых систем управления



базами данных. В целом ранние системы можно охарактеризовать следующим образом\*:

- эти системы активно использовались в течение многих лет, задолго до появления работоспособных реляционных СУБД. На самом деле некоторые из ранних систем используются даже в наше время, накоплены громадные базы данных, и одной из актуальных проблем информационных систем является использование этих систем совместно с современными;
- все ранние системы не основывались на каких-либо абстрактных моделях. Как мы упоминали, понятие модели данных фактически вошло в обиход специалистов в области БД только вместе с реляционным подходом. Абстрактные представления ранних систем появились позже на основе анализа и выявления общих признаков у различных конкретных систем;
- В ранних системах доступ к БД производился на уровне записей. Пользователи этих систем осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный доступ к БД поддерживался только путем создания соответствующих прикладных программ с собственным интерфейсом;
- можно считать, что уровень средств ранних СУБД соотносится с уровнем файловых систем примерно так же, как уровень языка Cobol соотносится с уровнем языков ассемблера. Заметим, что при таком взгляде уровень реляционных систем соответствует уровню языков Ада или APL;
- навигационная природа ранних систем и доступ к данным на уровне записей заставляли пользователей самих производить всю оптимизацию доступа к БД, без какой-либо поддержки системы;
- после появления реляционных систем большинство ранних систем было оснащено «реляционными» интерфейсами.

---

\* Заметим, что перечисляемые далее характеристики в полной мере относятся и к другим не реляционным подходам к организации баз данных, которые возникли до появления реляционного подхода или почти одновременно с ним. В частности, подобными свойствами обладают системы, основанные на подходах MUMPS (наиболее известной в России является реализация этого подхода в СУБД Cache компании Intersystems) и Pick (этот подход реализован во многих СУБД, в частности, в СУБД UniVerse и UniData семейства U2 компании IBM).

Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме.

Следует заметить, что приводимые далее описания ранних моделей данных являются упрощенными и неполными. Целью является лишь общее ознакомление читателей с их основными понятиями и характеристиками. Более подробное описание этих моделей данных можно найти, например, в [3].

### **2.2.1. Модель данных инвертированных таблиц**

К числу наиболее известных и типичных представителей систем, в основе которых лежит эта модель данных, относятся СУБД Datacom/DB, выведенная на рынок в конце 1960-х гг. компанией Applied Data Research, Inc. (ADR) и принадлежащая в настоящее время компании Computer Associates, и Adabas (ADApable DAtabase System), которая была разработана компанией Software AG в 1971 г. и до сих пор является ее основным продуктом.

Организация доступа к данным на основе инвертированных таблиц используется практически во всех современных реляционных СУБД, но в этих системах пользователи не имеют непосредственного доступа к инвертированным таблицам (индексам). Кстати, когда мы будем рассматривать внутренние интерфейсы реляционных СУБД, можно будет увидеть, что они очень близки к пользовательским интерфейсам систем, основанных на инвертированных таблицах.

**Структуры данных.** База данных в модели инвертированных таблиц похожа на БД в модели SQL, но с тем отличием, что пользователям видны и хранимые таблицы, и пути доступа к ним. При этом:

- строки таблиц упорядочиваются системой в некоторой физической, видимой пользователям последовательности;
- физическая упорядоченность строк всех таблиц может определяться и для всей БД (так делается, например, в Datacom/DB);
- для каждой таблицы можно определить произвольное число ключей поиска, для которых строятся индексы. Эти индексы автоматически поддерживаются системой, но явно видны пользователям.

**Манипулирование данными.** Поддерживаются два класса операций:

1. Операции, устанавливающие адрес записи и разбиваемые на два подкласса:

- прямые поисковые операторы (например, установить адрес первой записи таблицы по некоторому пути доступа);
- операторы, устанавливающие адрес записи при указании относительной позиции от предыдущей записи по некоторому пути доступа.

2. Операции над адресуемыми записями.

Вот типичный набор операций:

- **LOCATE FIRST** — найти первую запись таблицы  $T$  в физическом порядке; возвращается адрес записи;
- **LOCATE FIRST WITH SEARCH KEY EQUAL** — найти первую запись таблицы  $T$  с заданным значением ключа поиска  $k$ ; возвращается адрес записи;
- **LOCATE NEXT** — найти первую запись, следующую за записью с заданным адресом в заданном пути доступа; возвращается адрес записи;
- **LOCATE NEXT WITH SEARCH KEY EQUAL** — найти следующую запись таблицы  $T$  в порядке пути поиска с заданным значением  $k$ ; должно быть соответствие между используемым способом сканирования и ключом  $k$ ; возвращается адрес записи;
- **LOCATE FIRST WITH SEARCH KEY GREATER** — найти первую запись таблицы  $T$  в порядке ключа поиска  $k$  со значением ключевого поля, большим заданного значения  $k$ ; возвращается адрес записи;
- **RETRIVE** — выбрать запись с указанным адресом;
- **UPDATE** — обновить запись с указанным адресом;
- **DELETE** — удалить запись с указанным адресом;
- **STORE** — включить запись в указанную таблицу; операция генерирует и возвращает адрес записи.

**Ограничения целостности.** Общие правила определения целостности БД отсутствуют. В некоторых системах поддерживаются ограничения уникальности значений некоторых полей, но в основном вся поддержка целостности данных возлагается на прикладную программу.

### 2.2.2. Иерархическая модель данных

Типичным представителем (наиболее известным и распространенным) является СУБД IMS (Information Management System) компании IBM. Первая версия системы появилась в 1968 г.

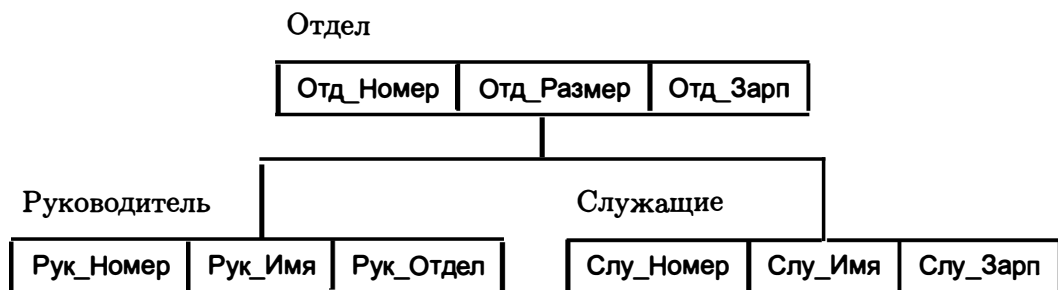


Рис. 2.1. Пример типа дерева

**Иерархические структуры данных.** Иерархическая БД состоит из упорядоченного набора деревьев; более точно, из упорядоченного набора нескольких экземпляров одного типа дерева. Тип дерева состоит из одного «корневого» типа записи и упорядоченного набора из нуля или более типов поддеревьев (каждое из которых является некоторым типом дерева). Тип дерева в целом представляет собой иерархически организованный набор типов записей («сегментов» в терминологии IMS).

На рис. 2.1 показан пример типа дерева (схемы иерархической БД). Здесь тип записи **Отдел** является предком для типов записи **Руководитель** и **Служащие**, а **Руководитель** и **Служащие** — потомки типа записи **Отдел**. Смысл полей типов записей в основном должен быть понятен по их именам. Поле **Рук\_Отдел** типа записи **Руководитель** содержит номер отдела, в котором работает служащий, являющийся данным руководителем (предполагается, что он работает не обязательно в том же отделе, которым руководит). Между типами записи поддерживаются связи (правильнее сказать, *типы связей*, поскольку реальные связи появляются в экземплярах типа дерева).

База данных с такой схемой могла бы выглядеть так, как показано на рис. 2.2 (здесь показан один экземпляр дерева).

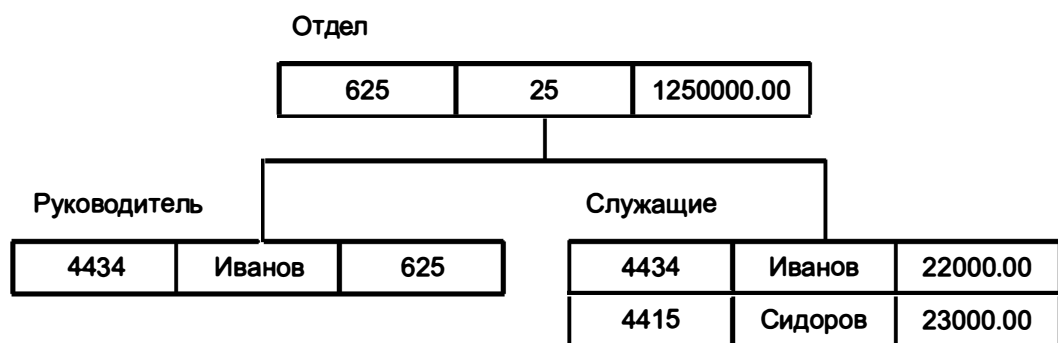


Рис. 2.2. Пример иерархической базы данных

Все экземпляры данного типа потомка с общим экземпляром типа предка называются близнецами. Для иерархической базы данных определяется полный порядок обхода дерева: сверху-вниз, слева-направо. Заметим, что в терминологии IMS вместо термина *запись* использовался термин *сегмент*, а под *записью* базы данных понималось все дерево сегментов.

**Манипулирование данными.** Примерами типичных операций манипулирования иерархически организованными данными могут быть следующие:

- найти указанный экземпляр типа дерева БД (например, отдел 310);
- перейти от одного экземпляра типа дерева к другому;
- перейти от экземпляра одного типа записи к экземпляру другого типа записи внутри дерева (например, перейти от отдела к первому сотруднику);
- перейти от одной записи к другой в порядке обхода иерархии;
- вставить новую запись в указанную позицию;
- удалить текущую запись.

**Ограничения целостности.** В иерархической модели данных автоматически поддерживается целостность ссылок между предками и потомками. Основное правило: *никакой потомок не может существовать без своего родителя*. Заметим, что аналогичная поддержка целостности по ссылкам между записями без связи «предок — потомок» не обеспечивается. Примером такой «внешней» ссылки является содержимое поля Рук\_Отдел в экземпляре типа записи Руководитель.

### 2.2.3. Сетевая модель данных

Типичным представителем систем, основанных на сетевой модели данных, является СУБД IDMS (Integrated Database Management System), разработанная компанией Cullinet Software, Inc. и изначально ориентированная на использование на мейнфреймах компании IBM. Архитектура системы основана на предложениях Data Base Task Group (DBTG) организации CODASYL (COncference on DAta SYstems Languages), которая разрабатывала спецификацию языка программирования COBOL. Отчет DBTG был опубликован в 1969 г., и вскоре после этого появилось несколько систем, поддерживающих архитектуру CODASYL, среди которых присутствовала и СУБД IDMS. В настоящее время IDMS принадлежит компании Computer Associates.

**Сетевые структуры данных.** Сетевой подход к организации данных является расширением иерархического подхода. В иерархических структурах запись-потомок должна иметь в точности одного предка; в сетевой структуре данных у потомка может иметься любое число предков.

Сетевая БД состоит из набора записей и набора связей между этими записями, а если говорить более точно, из набора экземпляров каждого типа из заданного в схеме БД набора типов записи и набора экземпляров каждого типа из заданного набора типов связи.

Тип связи определяется для двух типов записи: предка и потомка. Экземпляр типа связи состоит из одного экземпляра типа записи предка и упорядоченного набора экземпляров типа записи потомка. Для данного типа связи  $L$  с типом записи предка  $P$  и типом записи потомка  $C$  должны выполняться следующие два условия:

- каждый экземпляр типа записи  $P$  является предком только в одном экземпляре типа связи  $L$ ;
- каждый экземпляр типа записи  $C$  является потомком не более чем в одном экземпляре типа связи  $L$ .

На формирование типов связи не накладываются особые ограничения; возможны, например, следующие ситуации:

- тип записи потомка в одном типе связи  $L1$  может быть типом записи предка в другом типе связи  $L2$  (как в иерархии);
- данный тип записи  $P$  может быть типом записи предка в любом числе типов связи;
- данный тип записи  $P$  может быть типом записи потомка в любом числе типов связи;
- может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка; и если  $L1$  и  $L2$  — два типа связи с одним и тем же типом записи предка  $P$  и одним и тем же типом записи потомка  $C$ , то правила, по которым образуется родство, в разных связях могут различаться;
- типы записи  $X$  и  $Y$  могут быть предком и потомком в одной связи и потомком и предком — в другой;
- предок и потомок могут быть записями одного типа.

На рис. 2.3 показан простой пример схемы сетевой БД. На этом рисунке показаны три типа записи: Отдел, Служащие и Руководитель и три типа связи: Состоит из служащих, Имеет руководителя и Является служащим. В типе связи Состоит из служащих типом записи-предком является Отдел, а типом записи-

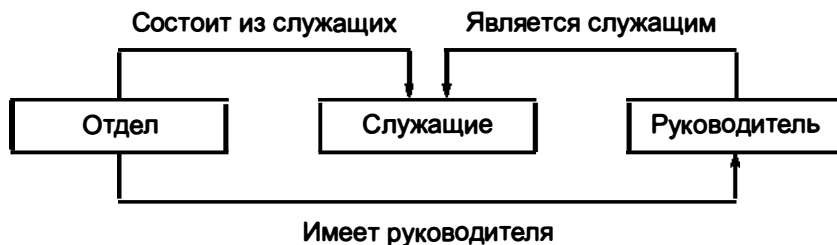


Рис. 2.3. Пример схемы сетевой базы данных

потомком — **Служащие** (экземпляр этого типа связи связывает экземпляр типа записи **Отдел** со многими экземплярами типа записи **Служащие**, соответствующими всем служащим данного отдела). В типе связи **Имеет руководителя** типом записи-предком является **Отдел**, а типом записи-потомком — **Руководитель** (экземпляр этого типа связи связывает экземпляр типа записи **Отдел** с одним экземпляром типа записи **Руководитель**, соответствующим руководителю данного отдела). Наконец, в типе связи **Является служащим** типом записи-предком является **Руководитель**, а типом записи-потомком — **Служащие** (экземпляр этого типа связи связывает экземпляр типа записи **Руководитель** с одним экземпляром типа записи **Служащие**, соответствующим тому служащему, которым является данный руководитель).

**Манипулирование данными.** Вот примерный набор операций манипулирования данными:

- найти конкретную запись в наборе однотипных записей (например, служащего с именем Иванов);
- перейти от предка к первому потомку по некоторой связи (например, к первому служащему отдела 625);
- перейти к следующему потомку в некоторой связи (например, от Иванова к Сидорову);
- перейти от потомка к предку по некоторой связи (например, найти отдел, в котором работает Сидоров);
- создать новую запись;
- уничтожить запись;
- модифицировать запись;
- включить в связь;
- исключить из связи;
- переставить в другую связь и т. д.

**Ограничения целостности.** Имеется (необязательная) возможность потребовать для конкретного типа связи отсутствие потомков, не участвующих ни в одном экземпляре этого типа связи (как в иерархической модели).

## 2.3. Неформальное введение в реляционную модель данных

Как отмечалось в начале этой главы, основные идеи реляционной модели данных были предложены Эдгаром Коддом в 1969 г. [1]. Следует заметить, что, несмотря на общепризнанную значимость этой и последующих работ Кодда, посвященных реляционной модели данных, эти работы писались на идейном уровне, не были (по теперешним меркам) глубоко технически проработанными, во многих важных местах допускали неоднозначное толкование, и поэтому эти работы невозможно было использовать как непосредственное руководство для реализации СУБД, поддерживающей реляционную модель.

За прошедшие десятилетия реляционная модель развивалась в двух направлениях. Первое направление заложил знаменитый экспериментальный проект компании IBM System R (см. ч. III). В этом проекте возник язык SQL, изначально основанный на идеях Кодда (который также работал в IBM), но нарушающий некоторые принципиальные предписания реляционной модели. К настоящему времени в действующем стандарте языка SQL, по сути, специфицирована некоторая собственная, законченная модель данных, обзор которой мы приведем в следующем разделе этой главы, а более подробно обсудим далее (см. ч. IV).

Второе направление, начиная с 1990-х гг., возглавляет Кристофер Дейт, к которому позже примкнул Хью Дарвен. Оба этих ученых также работали в компании IBM и до 1990-х гг. внесли большой вклад в развитие языка SQL. Однако в 1990-е гг. Дейт и Дарвен пришли к выводу, что искажения реляционной модели данных, свойственные языку SQL, достигли настолько высокого уровня, что пришло время предложить альтернативу, опирающуюся на неискаженные идеи Эдгара Кодда и обеспечивающую все возможности как SQL, так и объектно-ориентированного подхода к организации баз данных и СУБД (обзор объектно-ориентированной модели данных приводится в следующем разделе).

Новые идеи Дейта и Дарвена были впервые изложены в их *Третьем манифесте* [4], а позже на основе этих идей была специфицирована модель данных [5]. Авторы считают, что в [5] приводится всего лишь современная и полная интерпретация идей Кодда. С этим можно соглашаться или спорить, но бесспорен один факт — Кодд не участвовал в написании этих материалов и никогда не писал ничего подобного. В следующих главах этой книги, тем не менее, при обсуждении реляционной



модели будем использовать в основном интерпретацию Дейта и Дарвена.

В этом подразделе приведем краткий и неформальный обзор основных идей реляционной модели в том виде, в котором она была предложена Коддом, а в подразд. 2.4 обсудим предложения Дейта и Дарвена.

Более строгое и формальное описание реляционной модели данных приводится в гл. 3 и 4.

### 2.3.1. Реляционные структуры данных

Основная идея Кодда состояла в том, чтобы выбрать в качестве родовой логической структуры хранения данных структуру, которая, с одной стороны, была бы достаточно удобной для большинства приложений и, с другой стороны, допускала бы возможность выполнения над базой данных ненавигационных операций. Иерархические и, в особенности, сетевые структуры данных являются навигационными по своей природе. Ненавигационному использованию таблиц мешает упорядоченность их столбцов и, в особенности, строк.

По сути, Кодд предложил использовать в качестве родовой структуры БД «таблицы», в которых и столбцы, и строки не являются упорядоченными. Легко видеть, что такая «таблица» со множеством столбцов  $\{A_1, A_2, \dots, A_n\}$ , в которой каждый столбец  $A_i$  может содержать значения из множества  $T_i = \{v_{i1}, v_{i2}, \dots, v_{im}\}$  (все множества конечны), в математическом смысле представляет собой отношение над множествами  $\{T_1, T_2, \dots, T_n\}$ . Напомню, что в математике отношением над множествами  $\{T_1, T_2, \dots, T_n\}$  называется подмножество декартова произведения этих множеств, т. е. некоторое множество кортежей  $\{v_1, v_2, \dots, v_n\}$ , где  $v_i \in T_i^*$ . Поэтому для обозначения родовой структуры Кодд стал использовать термин *отношение* (*relation*), а для обозначения элементов отношения — термин *кортеж*. Соответственно, модель данных получила название *реляционной модели*.

Схема БД в реляционной модели данных — это набор именованных *заголовков отношений* вида  $H_i = \langle A_i^1, T_i^1 \rangle, \langle A_i^2, T_i^2 \rangle, \dots, \langle A_i^{n_i}, T_i^{n_i} \rangle$ .  $T_i$  называется *доменом* атрибута  $A_i$ . По Кодду, каждый домен  $T_i$  является подмножеством значений некоторого базового типа данных  $T_i^*$ , а значит, к его элементам применимы все операции этого базового типа (в конце 1960-х гг. базовыми типами

---

\* Обозначение  $s \in S$  означает, что элемент  $s$  принадлежит множеству  $S$ .

данных считались типы данных распространенных тогда языков программирования; в IBM наиболее популярными языками были PL1 и COBOL).

Реляционная база данных в каждый момент времени представляет собой набор именованных отношений, каждое из которых обладает заголовком, таким как он определен в схеме БД, и телом. Имя отношения  $R_i$  совпадает с именем заголовка этого отношения  $H_{R_i}$ . Тело отношения  $B_{R_i}$  — это множество кортежей вида  $\{ \langle A_i^1, T_i^1, v_i^1 \rangle, \langle A_i^2, T_i^2, v_i^2 \rangle, \dots, \langle A_i^{n_i}, T_i^{n_i}, v_i^{n_i} \rangle \}$ , где  $t_i^j \in T_i^j$ . Во время жизни БД тела отношений могут изменяться, но все содержащиеся в них кортежи должны соответствовать заголовкам соответствующих отношений.

### 2.3.2. Манипулирование реляционными данными

Поскольку в реляционной модели данных заголовки и тело любого отношения представляют собой множества, к отношениям, вообще говоря, применимы обычные теоретико-множественные операции: объединение, пересечение, вычитание, взятие декартова произведения. Напомним, что для двух множеств  $S_1 \{s_1\}$  и  $S_2 \{s_2\}$  результатом операции объединения этих двух множеств  $S_1 \text{ UNION } S_2$  является множество  $S\{s\}$  такое, что  $s \in S_1$  или  $s \in S_2$ . Результатом операции пересечения  $S_1 \text{ INTERSECT } S_2$  является множество  $S\{s\}$  такое, что  $s \in S_1$  и  $s \in S_2$ . Результатом операции вычитания  $S_1 \text{ MINUS } S_2$  является множество  $S\{s\}$  такое, что  $s \in S_1$  и  $s \notin S_2^*$ . На рис. 2.4 эти операции проиллюстрированы в интуитивной графической форме. Про операцию взятия декартова произведения уже говорилось ранее.

Понятно, что эти операции применимы к любым телам отношений, но результатом не будет являться отношение, если у отношений-операндов не совпадают заголовки. Кодд предложил в качестве средства манипулирования реляционными базами данных специальный набор операций, которые гарантированно производят отношения. Этот набор операций принято называть *реляционной алгеброй Кодда*, хотя он и не является *алгеброй* в математическом смысле этого термина, поскольку некоторые бинарные операции этого набора применимы не к произвольным парам отношений.

В алгебре Кодда имеется 10 операций: объединение (UNION), пересечение (INTERSECT), вычитание (MINUS), взятие расширен-

---

\* Обозначение  $s \notin S$  означает, что элемент  $s$  не принадлежит множеству  $S$ .

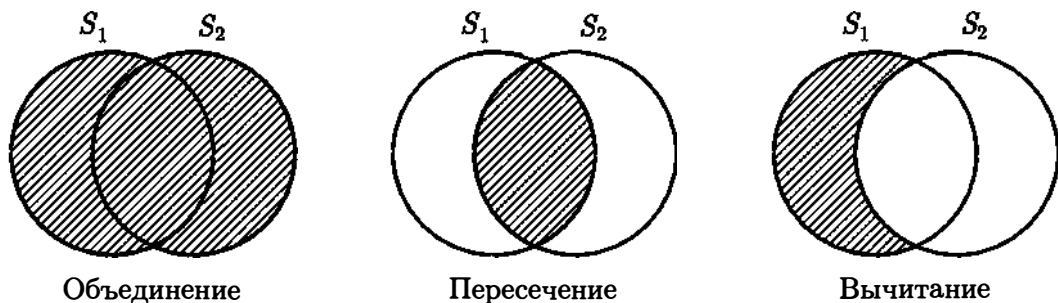


Рис. 2.4. Иллюстрация результатов теоретико-множественных операций

ного декартова произведения (**TIMES**), переименование атрибутов (**RENAME**), проекция (**PROJECT**), ограничение (**WHERE**), соединение ( **$\Theta$ -JOIN**), деление (**DIVIDE BY**) и присваивание. Если не вдаваться в некоторые тонкости, которые рассмотрены в гл. 4, то почти все операции предложенного ранее набора обладают очевидной и простой интерпретацией.

- При выполнении операции *объединения* (**UNION**) двух отношений с одинаковыми заголовками производится отношение, включающее все кортежи, входящие хотя бы в одно из отношений-операндов.
- Операция *пересечения* (**INTERSECT**) двух отношений с одинаковыми заголовками производит отношение, включающее все кортежи, входящие в оба отношения-операнда.
- Отношение, являющееся *разностью* (**MINUS**) двух отношений с одинаковыми заголовками, включает все кортежи, входящие в отношение-первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом.
- При выполнении *декартова произведения* (**TIMES**) двух отношений, пересечение заголовков которых пусто, производится отношение, кортежи которого производятся путем объединения кортежей первого и второго операндов.
- Операция *переименования* (**RENAME**) производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены; эта операция позволяет выполнять первые три операции над отношениями с «почти» совпадающими заголовками (совпадающими во всем, кроме имен атрибутов) и выполнять операцию **TIMES** над отношениями, пересечение заголовков которых не является пустым.
- Результатом *ограничения* (**WHERE**) отношения по некоторому условию является отношение, включающее кортежи отношения-операнда, удовлетворяющие этому условию.

- При выполнении *проекции* (PROJECT) отношения на заданное подмножество множества его атрибутов производится отношение, кортежи которого являются соответствующими подмножествами кортежей отношения-операнда.
- При *Θ-соединении* (Θ-JOIN) двух отношений по некоторому условию (Θ) образуется результирующее отношение, кортежи которого производятся путем объединения кортежей первого и второго отношений и удовлетворяют этому условию.
- У операции *реляционного деления* (DIVIDE BY) два операнда — бинарное и унарное отношения. Результирующее отношение состоит из унарных кортежей, включающих значения первого атрибута кортежей первого операнда, таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) включает множество значений второго операнда.
- Операция *присваивания* (:=) позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД.

### 2.3.3. Целостность в реляционной модели данных

Кодд предложил два декларативных механизма поддержки целостности реляционных баз данных, которые закреплены в реляционной модели данных и должны поддерживаться в любой реализующей ее СУБД: *ограничение целостности сущности*, или *ограничение первичного ключа* и *ограничение ссылочной целостности*, или *ограничение внешнего ключа*. Приведем изложение общих идей (подробнее см. гл. 3).

*Ограничение целостности сущности* формулируется следующим образом: для заголовка любого отношения базы данных должен быть явно или неявно определен *первичный ключ*, являющийся таким минимальным подмножеством заголовка отношения, что в любом теле этого отношения, которое может появиться в базе данных, значение первичного ключа в любом кортеже этого тела является уникальным, т.е. отличается от значения первичного ключа в любом другом кортеже. Под минимальностью первичного ключа понимается то, что если из множества атрибутов первичного ключа удалить хотя бы один атрибут, то ограничение целостности изменится, т.е. в БД смогут появляться тела отношений, которые не допускались исходным первичным ключом.

Если первичный ключ не объявляется явно, то в качестве первичного ключа отношения принимается весь его заголовок.

Понятно, что поскольку по определению любое тело отношения с заданным заголовком является множеством, следовательно, в нем отсутствуют дубликаты, и первичный ключ, совпадающий с заголовком отношения, всегда обладает свойством уникальности. Должно быть понятно, что в этом случае определение первичного ключа не задает никакого ограничения целостности.

Чтобы пояснить смысл *ограничения ссылочной целостности*, нужно сначала ввести понятие *внешнего ключа*. В принципе при использовании реляционной модели данных можно хранить все данные, соответствующие предметной области в одной таблице. Пример такой базы данных демонстрировался в гл. 1 (см. рис. 1.6), где в одном файле (интуитивном аналоге отношения) хранилась информация и о служащих, и об отделах, в которых они работают. Как было показано в гл. 1, такой подход приводит к избыточности хранения (данные об отделе повторяются в каждой записи о служащем этого отдела) и усложняет выполнение некоторых операций.

На рис. 1.7 для хранения информации о служащих и отделах использовалось два файла, в одном из которых хранились данные, индивидуальные для каждого служащего, а во втором — данные об отделах. Для возможности получения полной информации о служащих и отделах, в которых они работают, в файле **СЛУЖАЩИЕ** содержалось поле **СПУ\_ОТД\_НОМЕР**, содержащее для каждого служащего его уникальный номер отдела. В то же время, в файле **ОТДЕЛЫ** имелось поле **ОТД\_НОМЕР**, являющееся уникальным ключом этого файла. На самом деле, введя файлы **СЛУЖАЩИЕ** и **ОТДЕЛЫ**, а также обеспечив связь между ними с помощью полей **СПУ\_ОТД\_НОМЕР** и **ОТД\_НОМЕР**, мы смогли обеспечить табличное представление иерархии **ОТДЕЛ-СЛУЖАЩИЕ**. Если говорить в терминах реляционной модели данных, то в отношении **ОТДЕЛЫ** поле **ОТД\_НОМЕР** является *первичным ключом*, а в отношении **СЛУЖАЩИЕ** поле **СПУ\_ОТД\_НОМЕР** является *внешним ключом*, ссылающимся на отношение **ОТДЕЛЫ**.

Более точно, *внешним ключом отношения  $R_1$ , ссылающимся на отношение  $R_2$* , называется подмножество заголовка  $H_{R_1}$ , которое совпадает с первичным ключом отношения  $R_2$  (с точностью до имен атрибутов). Тогда *ограничение ссылочной целостности* реляционной модели данных можно сформулировать следующим образом: в любом теле отношения  $R_1$ , которое может появиться в базе данных, для «не пустого» \* значения внешнего ключа, ссы-

---

\* Понятие «пустого», или *неопределенного*, значения будет уточнено в гл. 3.

лающего на отношение  $R_2$ , в любом кортеже этого тела должен найтись кортеж в теле отношения  $R_2$ , которое содержится в базе данных, с совпадающим значением первичного ключа. Легко заметить, что это почти то же самое ограничение, о котором говорилось в подразд. 2.2.2: *никакой потомок не может существовать без своего родителя*, но немного уточненное — *ссылки на родителя должны быть корректными*.

## 2.4. Современные модели данных

Предположительно, история современных моделей данных началась с 1989 г., когда группа известных специалистов в области языков программирования баз данных опубликовала статью под названием «Манифест систем объектно-ориентированных баз данных» [6]. К этому времени уже существовало несколько реализаций объектно-ориентированных СУБД (ООСУБД), но каждая из них опиралась на некоторое расширение объектной модели какого-либо объектно-ориентированного языка программирования (Smalltalk, Object Lisp, C++), отсутствовали какие-либо общие подходы.

В Манифесте не предлагалась единая объектно-ориентированная модель данных, но выделялся набор требований к ООСУБД. Базовыми требованиями являлось преодоление несоответствия между типами данных, используемыми в языках программирования, и типами данных, поддерживаемыми в набравших к тому времени силу реляционных (вернее, SQL-ориентированных) СУБД, а также придание СУБД возможностей хранить в БД данные произвольно сложной структуры. Эти требования сопровождались утверждениями об ограниченности реляционной модели данных и языка SQL и потребности использовать более развитые модели данных.

Под влиянием Манифеста в 1991 г. возник консорциум ODMG (Object Database Management Group), задачей которого была разработка стандарта объектно-ориентированной модели данных. В течение времени своего существования ODMG опубликовал три базовых версии стандарта, последняя из которых называется ODMG 3.0 [7]. На этот документ мы и будем опираться в дальнейшем изложении.

В ответ на публикацию [6] группа исследователей, близких к индустрии баз данных, в 1990 г. опубликовала документ «Манифест систем баз данных третьего поколения» [8], который во многом направлен на защиту инвестиций крупных компаний-

производителей программного обеспечения SQL-ориентированных СУБД. Соглашаясь с авторами [6] относительно потребности обеспечения развитой системы типов данных в СУБД, авторы [8] утверждали, что можно добиться аналогичных результатов, не производя революцию в области технологии баз данных, а эволюционно развивая технологию SQL-ориентированных СУБД.

За публикацией [8] последовало появление *объектно-реляционных* продуктов ведущих компаний-поставщиков SQL-ориентированных СУБД (Informix Universal Server, Oracle8, IBM DB2 Universal Database). В 1999 г. был принят стандарт языка SQL (SQL:1999), в котором был зафиксирован ряд новых черт языка, придающих ему черты полноценной самостоятельной модели данных. В последнем ко времени написания этой книги стандарте SQL:2003 эта модель уточнена и расширена. Далее под *моделью данных SQL* понимается модель данных, соответствующая этим стандартам языка. В ч. 4 мы достаточно подробно обсудим стандарт SQL, а в этом подразделе остановимся лишь на некоторых особенностях модели данных SQL, отличающих ее от реляционной модели данных.

Итак, в конце 1980-х — начале 1990-х гг. были провозглашены два манифеста, каждый из которых претендовал на роль программы будущего развития технологии баз данных. В первом манифесте реляционная модель данных отвергалась полностью, а во втором заменялась еще незрелой к тому времени моделью данных SQL, которая уже тогда была далека от реляционной модели. На защиту реляционной модели данных в ее первоначальном виде встали К. Дейт и Х. Дарвен, опубликовавшие в 1995 г. статью под названием «Третий манифест» [4].

«Третий манифест» являлся одновременно наиболее консервативным и наиболее радикальным. Консервативность *Третьего манифеста* заключается в том, что его авторы всеми силами утверждают необходимость и достаточность использования в системах баз данных следующего поколения классической реляционной модели данных. Радикальность состоит в том, что, во-первых, авторы полностью отрицают подходы, предлагаемые в первых двух манифестах, расценивая их как необоснованные, плохо проработанные, избыточные и даже вредные (за исключением одной общей идеи о потребности обеспечения развитой системы типов); во-вторых, фактически, авторы полностью отбрасывают технологию, созданную индустрией баз данных за последние 25 лет, и предлагают вернуться к истокам реляционной модели данных, т.е. начальным статьям Э. Кодда [1].

Позже Дейт и Дарвен написали книгу, первое издание которой вышло в 1998 г. под названием «Foundation for Object/Relational Databases: The Third Manifesto» [9], второе — в 2000 г. под названием «Foundation for Future Database Systems: The Third Manifesto» [10] (издан перевод второго издания на русский язык [4]) и третье — под названием «Databases, Types and the Relational Model: The Third Manifesto» в 2006 г. [11]. В этих книгах очень подробно изложен подход авторов к построению СУБД на основе, как они утверждают, истинных идей Эдгара Кодда, изложенных им в его первых статьях, посвященных реляционной модели данных. Некоторые более поздние идеи Кодда относительно той же реляционной модели авторами отвергаются. В любом случае, Кодд и Дарвен предлагают некоторый современный вариант реляционной модели данных (далее для определенности мы будем называть ее *истинной* реляционной моделью), который, безусловно, заслуживает внимания и изучения. В данной книге ограничимся только кратким очерком основных черт этой модели.

#### 2.4.1. Объектно-ориентированная модель данных

Если не обращать внимания на особенности объектно-ориентированной терминологии (предполагается, что читатели этой книги в общих чертах знакомы с ней), то объектно-ориентированная модель данных ODMG, специфицированная в [7], отличается от других двух моделей, описываемых в этом разделе, прежде всего, в одном принципиальном аспекте. В модели данных SQL и истинной реляционной модели данных БД представляет собой набор именованных контейнеров данных одного родового типа: таблиц или отношений соответственно. В объектно-ориентированной модели данных база данных — это набор объектов (контейнеров данных) произвольного типа.

**Типы и структуры данных объектной модели.** В объектной модели данных вводятся две разновидности типов: *литеральные* и *объектные* типы. Литеральные типы данных — это обычные типы данных, принятые в традиционных языках программирования. Они подразделяются на базовые скалярные числовые типы, символьные и булевские типы (*атомарные литералы*), конструируемые типы записей (*структур*) и *коллекций*.

Литеральный тип записи — это традиционный, определяемый пользователем структурный тип, подобный структурному типу языка C или типу записи языка Pascal. Отличие состоит лишь



в том, что в объектной модели атрибут типа записи может определяться не только на литеральном, но и на объектном типе, т. е. значение литерального типа записи может в качестве компонентов включать объекты. На первый взгляд, это звучит странно и страшно, но здесь все странности проистекают из особенностей объектно-ориентированной терминологии. У любого существующего объекта имеется одно и только одно местоположение, характеризующееся его идентификатором (OID). Когда в модели говорится, что некоторое структурное значение в качестве компонента имеет некоторый объект, то, конечно, имеется в виду OID этого объекта, являющийся всего лишь аналогом указательного значения в традиционных языках программирования.

Имеются четыре вида типов коллекций: типы множеств, мультимножеств (неупорядоченные наборы элементов, возможно, содержащие дубликаты), списков (упорядоченные наборы элементов, возможно, содержащие дубликаты) и словарей (множества пар <ключ, значение>, причем все ключи в этих парах должны быть различными). Типом элемента любой коллекции может являться любой скалярный или объектный тип, за исключением того же типа коллекции.

Объектные типы в объектной модели данных по смыслу ближе всего к понятию *класса* в объектно-ориентированных языках программирования. У каждого объектного типа имеется операция создания и инициализации нового объекта этого типа. Эта операция возвращает значение OID нового объекта, который можно хранить в любом месте, где допускается хранение объектов данного типа, и использовать для обращения к *операциям* объекта, определенным в его объектном типе.

Имеются два вида объектных типов. Первый из них называется атомарным объектным типом. Нестрого говоря, при определении атомарного объектного типа указывается его внутренняя структура (набор свойств — атрибутов и связей) и набор операций, которые можно применять к объектам этого типа. Для определения атомарного объектного типа можно использовать механизм наследования, расширяя набор свойств и/или переопределяя существующие и добавляя новые операции.

*Атрибутами* называются свойства объекта, значение которых можно получить по OID объекта. Значениями атрибутов могут быть и литералы, и объекты (т. е. OID), но только тогда, когда не требуется обратная ссылка. *Связи* — это инверсные свойства. В этом случае значением свойства может быть только объект. Связи определяются между атомарными объектными

типами. В объектной модели ODMG поддерживаются только бинарные связи, т.е. связи между двумя типами. Связи могут быть разновидностей «один-к-одному», «один-ко-многим» и «многие-ко-многим» в зависимости от того, сколько экземпляров соответствующего объектного типа может участвовать в связи.

Связи явно определяются путем указания *путей обхода*. Пути обхода указываются парами, по одному пути для каждого направления обхода связи. Например, в базе данных **СЛУЖАЩИЕ-ОТДЕЛЫ** служащий работает (**works**) в одном отделе, а отдел состоит из (**consists of**) множества служащих. Тогда путь обхода **consists\_of** должен быть определен в объектном типе **ОТДЕЛ**, а путь обхода **works** — в типе **СЛУЖАЩИЙ**. Тот факт, что пути обхода относятся к одной связи, указывается в разделе **inverse** обоих объявлений пути обхода. Это связь «один-ко-многим». Путь обхода **consists\_of** ассоциирует объект типа **ОТДЕЛ** с литеральным множеством объектов типа **СЛУЖАЩИЙ**, а путь обхода **works** ассоциирует объект типа **СЛУЖАЩИЙ** с объектом типа **ОТДЕЛ**. Пути обхода, ведущие к коллекциям объектов, могут быть упорядоченными или неупорядоченными в зависимости от вида коллекции, указанного в объявлении пути обхода.

Заметим, что хотя связь является модельным понятием, другие понятия модели наталкивают на мысль, что единственным способом реализации связей является хранение в объекте **OID** или коллекции **OID** связанных объектов в зависимости от вида связи. Это можно сделать и с использованием должным образом типизированных атрибутов. Однако явное определение связи обеспечивает системе дополнительную информацию, которая используется в объектной модели как ограничение целостности (см. далее).

Второй вид — это объектные типы коллекций. Как и в случае использования литеральных типов коллекций, можно определять объектные типы множеств, мультимножеств, списков и словарей. Типом элемента объектного типа коллекции может быть любой литеральный или объектный тип, за исключением того же типа коллекции. У объектных типов коллекций имеются предопределенные наборы операций. В отличие от литеральных типов коллекций, которые, как и все литеральные типы являются множествами значений, объектные типы коллекций обладают операцией создания объекта, имеющего, как и все объекты, собственный **OID**.

Интересен и важен один специальный случай неявного использования объектов типа множества. При определении ато-

марного объектного типа можно в качестве одного из дополнительных свойств этого типа указать, что для него должен быть создан объект типа множества, элементами которого являются объекты данного атомарного типа (*экстен*т объектного структурного типа). Поскольку такой объект создается неявно, его OID неизвестен, но зато у него имеется имя, явно задающееся в определении и совпадающее с именем атомарного объектного типа. Наличие этой возможности позволяет создавать объектные базы данных, состоящие из именованных контейнеров одно-типных объектов, причем в действительности эти контейнеры содержат OID соответствующих объектов.

**Манипулирование данными в объектной модели.** В стандарте ODMG в качестве базового средства манипулирования объектными базами данных предлагается язык OQL (Object Query Language). Это небольшой, но достаточно сложный язык запросов. Разработчики в целом характеризуют его следующим образом:

- OQL опирается на объектную модель ODMG (имеется в виду, что в нем поддерживаются средства доступа ко всем возможным структурам данных, допускаемых в структурной части модели).
- OQL очень близок к SQL/92. Расширения относятся к объектно-ориентированным понятиям, таким как сложные объекты, объектные идентификаторы, путевые выражения, полиморфизм, вызов операций и отложенное связывание.
- В OQL обеспечиваются высокоуровневые примитивы для работы с множествами объектов, но, кроме того, имеются настолько же эффективные примитивы для работы со структурами, списками и массивами.
- OQL является функциональным языком, допускающим неограниченную композицию операций, если операнды не выходят за пределы системы типов. Это является следствием того факта, что результат любого запроса обладает типом, принадлежащим к модели типов ODMG, и поэтому к результату запроса может быть применен новый запрос.
- OQL не является вычислительно полным языком. Он представляет собой простой язык запросов.
- Операторы языка OQL могут вызываться из любого языка программирования, для которого в стандарте ODMG определены правила связывания. И, наоборот, в запросах OQL могут присутствовать вызовы операций, запрограммированных на этих языках.

- В OQL не определяются явные операции обновления, а используются вызовы операций, определенных в объектах для целей обновления.
- В OQL обеспечивается декларативный доступ к объектам. По этой причине OQL-запросы могут хорошо оптимизироваться.
- Можно легко определить формальную семантику OQL.

Объем этой главы не позволяет привести развернутое описание языка OQL. Приведем лишь один характерный пример.

Получить номера руководителей отделов и тех служащих их отделов, заработная плата которых превышает 20 000 руб.

```
SELECT DISTINCT STRUCT ( ОТД_РУК: D.ОТД_РУК,
                        СЛУ: ( SELECT E
                              FROM D.CONSISTS_OF AS E
                              WHERE E.СЛУ_ЗАРП > 20000.00 ) )
FROM ОТДЕЛЫ D
```

Здесь предполагается, что для атомарного объектного типа **ОТДЕЛ** определен экстенс типа множества с именем **ОТДЕЛЫ**. В запросе перебираются все существующие объекты типа **ОТДЕЛ**, и для каждого такого объекта происходит переход по связи к литеральному множеству объектов типа **СЛУЖАЩИЙ**, соответствующих служащим, которые работают в данном отделе. На основе этого множества формируется «усеченное» множество объектов типа **СЛУЖАЩИЙ**, в котором остаются только объекты-служащие с заработной платой, большей 20000.00. Результатом запроса является литеральное значение-множество, элементами которого являются значения-структуры с двумя литеральными значениями, первое из которых есть атомарное литеральное значение типа **INTEGER**, а второе — литеральное значение-множество с элементами-объектами типа **СЛУЖАЩИЙ**. Более точно, результат запроса имеет тип `set < struct { integer ОТД_РУК; bag < СЛУЖАЩИЙ > СЛУ } >`.

В совокупности результатом допустимых в OQL выражений запросов могут являться:

- коллекция объектов;
- индивидуальный объект;
- коллекция литеральных значений;
- индивидуальное литеральное значение.

**Ограничения целостности в объектной модели.** В соответствии с общей идеологией объектно-ориентированного подхода в модели ODMG два объекта считаются совпадающими в том и только том случае, когда являются одним и тем же объектом,

т.е. имеют один и тот же OID. Объекты одного объектного типа с разными OID считаются разными, даже если обладают полностью совпадающими состояниями. Поэтому в объектной модели отсутствует аналог ограничения целостности сущности реляционной модели данных. Интересно, что при определении атомарного объектного типа можно объявить ключ — набор свойств объектного класса, однозначно идентифицирующий состояние каждого объекта, входящего в экстенст этого класса. Для класса может быть объявлено несколько ключей, а может не быть объявлено ни одного ключа даже при наличии определения экстенста. Но при этом определение ключа не трактуется в модели как ограничение целостности; утверждается, что объявление ключа способствует повышению эффективности выполнения запросов.

Что же касается ссылочной целостности, то она поддерживается, если между двумя атомарными объектными типами определяется связь вида «один-ко-многим». В этом случае объекты на стороне связи «один» рассматриваются как предки, а объекты на стороне связи «многие» — как потомки, и ООСУБД обязана следить за тем, чтобы не образовывались потомки без предков.

## 2.4.2. Модель данных SQL

Как отмечалось в начале подраздела, модель данных SQL в относительно законченном виде сложилась к 1999 г., когда был принят и опубликован стандарт SQL:1999. В приводимом в этом подразделе очерке этой модели данных мы затронем только наиболее важные, с точки зрения автора, ее черты, опуская многие менее существенные моменты.

**Типы и структуры данных SQL.** SQL-ориентированная база данных представляет собой набор таблиц, каждая из которых в любой момент времени содержит некоторое множество строк, соответствующих заголовку таблицы. В этом состоит первое и наиболее важное отличие модели данных SQL от реляционной модели данных. Вторым существенным отличием является то, что для таблицы поддерживается порядок столбцов, соответствующий порядку их определения. Другими словами, таблица — это вовсе не отношение, хотя во многом они похожи.

Имеется две основных разновидности таблиц, хранимых в базе данных: традиционная таблица и типизированная таблица. Традиционная таблица определяется как множество столбцов с указанными типами данных. В SQL поддерживаются

следующие категории типов данных: точные числовые типы; приближенные числовые типы; типы символьных строк; типы битовых строк; типы даты и времени; типы временных интервалов; булевский тип; типы коллекций; анонимные строчные типы; типы, определяемые пользователем; ссылочные типы. Подробно система типов SQL описывается в гл. 11, здесь ограничимся только пояснениями наименее очевидных случаев.

Булевский тип в SQL содержит три значения — *true*, *false* и *unknown*. Это связано с интенсивным использованием в SQL так называемого неопределенного значения (NULL), которое разрешается использовать вместо значения любого типа данных. Как отмечалось ранее, в этой главе мы не будем более подробно затрагивать запутанную тему неопределенных значений и оставим подробности на последующие главы.

В модели данных SQL допускается объявление двух видов типов коллекций: типы массива и типы мультимножества. Элементы типа коллекции могут быть любого типа данных, определенного к моменту определения данного типа коллекции. При объявлении типа мультимножества можно явно запретить наличие в его значениях элементов-дубликатов, что фактически приводит к объявлению типа множества.

Анонимный строчный тип — это безымянный структурный тип, значения которого являются строками, состоящими из элементов ранее определенных типов.

Поддерживается два вида типов данных, определяемых пользователями: индивидуальные и структурные типы. Индивидуальный тип — это именованный тип данных, основанный на единственном предопределенном типе. Индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Чтобы выполнить некоторую операцию базового типа над значениями определенного над ним индивидуального типа, требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями базового типа. Имеется также возможность явного определения методов, функций и процедур, связанных с данным индивидуальным типом.

Структурный тип данных — это именованный тип данных, включающий один или более атрибутов любого из допустимых в SQL типа данных, в том числе другого структурного типа, типа коллекций, анонимного строчного типа и т.д. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа. При определении структурного типа можно использовать механизм наследования от ранее определенного структурного типа.

При определении типизированной таблицы указывается ранее определенный структурный тип, и если в нем содержится  $n$  атрибутов, то в таблице образуется  $n+1$  столбец, из которых последние  $n$  столбцов с именами и типами данных, совпадающими именам и типам атрибутов структурного типа. Первый же столбец, имя которого явно задается, называется «самоссылающимся» и содержит типизированные уникальные идентификаторы строк, которые могут генерироваться системой при вставке строк в типизированную таблицу, явно указываться пользователями или состоять из комбинации значений других столбцов. Типом «самоссылающегося» столбца является ссылочный тип, ассоциированный со структурным типом типизированной таблицы. Способ генерации значений ссылочного типа указывается при определении соответствующего структурного типа и подтверждается при определении типизированной таблицы.

При определении типизированных таблиц можно использовать механизм наследования. Можно определить подтаблицу типизированной супертаблицы, если структурный тип подтаблицы является непосредственным подтипом структурного типа супертаблицы. Подтаблица наследует у супертаблицы способ генерации значений ссылочного типа и все ограничения целостности, которые были специфицированы в определении супертаблицы. Дополнительно можно определить ограничения, затрагивающие новые столбцы.

С типизированной таблицей можно обращаться, как с традиционной таблицей, считая, что у нее имеются неявно определенные столбцы, а можно относиться к строкам типизированной таблицы, как к объектам структурного типа, OID которых содержатся в «самоссылающемся» столбце. Ссылочный тип можно использовать для типизации столбцов традиционных таблиц и атрибутов структурных типов, на которых потом определяются типизированные таблицы. В последнем случае можно считать, что значениями атрибутов соответствующих объектов являются объекты структурного типа, с которыми ассоциирован данный ссылочный тип.

**Манипулирование данными в SQL.** Средства манипулирования данными составляют значительную часть языка SQL и сравнительно подробно обсуждаются в гл. 12. Здесь же мы ограничимся общей характеристикой оператора SQL **SELECT**, предназначенного для выборки данных и имеющего следующий синтаксис:

```
SELECT [ ALL | DISTINCT ] select_item_commalist  
FROM table_reference_commalist
```

```
[ WHERE conditional_expression ]  
  [ GROUP BY column_name_commlist ]  
    [ HAVING conditional_expression ]  
      [ ORDER BY order_item_commlist ]
```

Выборка данных производится из одной или нескольких таблиц, указываемых в разделе **FROM** запроса. В последнем случае на первом этапе выполнения оператора **SELECT** образуется одна общая таблица, получаемая из исходных таблиц с помощью операции расширенного декартова умножения. Таблицы могут быть как базовыми, реально хранимыми в базе данных (традиционными или типизированными), так и порожденными, т.е. задаваемыми в виде некоторого оператора **SELECT**. Это допускается, поскольку результатом выполнения оператора **SELECT** в его базовой форме является традиционная таблица. Кроме того, в разделе **FROM** можно указывать выражения соединения базовых и/или порожденных таблиц, результатами которых опять же являются традиционные таблицы.

На следующем шаге общая таблица, полученная после выполнения раздела, подвергается фильтрации путем вычисления для каждой ее строки логического выражения, заданного в разделе **WHERE** запроса. В отфильтрованной таблице остаются только те строки общей таблицы, для которых значением логического выражения является *true*.

Если в операторе отсутствует раздел **GROUP BY**, то после этого происходит формирование результирующей таблицы запроса путем вычисления выражений, заданных в списке выборки оператора **SELECT**. В этом случае список выборки вычисляется для каждой строки отфильтрованной таблицы, и в результирующей таблице появится ровно столько же строк.

При наличии раздела **GROUP BY** из отфильтрованной таблицы получается сгруппированная таблица, в которой каждая группа состоит из кортежей отфильтрованной таблицы с одинаковыми значениями столбцов группировки, задаваемых в разделе **GROUP BY**. Если в запросе отсутствует раздел **HAVING**, то результирующая таблица строится прямо на основе сгруппированной таблицы. Иначе образуется отфильтрованная сгруппированная таблица, содержащая только те группы, для которых значением логического выражения, заданного в разделе **HAVING**, является *true*.

Результирующая таблица на основе сгруппированной или отфильтрованной сгруппированной таблицы строится путем вычисления списка выборки для каждой группы. Тем самым, в результирующей таблице появится ровно столько строк, сколь-



ко групп содержалось в сгруппированной или отфильтрованной сгруппированной таблице.

Если в запросе присутствует ключевое слово **DISTINCT**, то из результирующей таблицы удаляются строки-дубликаты, т.е. запрос вырабатывает не мультимножество, а множество строк.

Наконец, в запросе может присутствовать еще и раздел **ORDER BY**. В этом случае результирующая таблица сортируется в порядке возрастания или убывания в соответствии со значениями ее столбцов, указанных в разделе **ORDER BY**. Результатом такого запроса является не таблица, а отсортированный список, который нельзя сохранить в базе данных. Сам же запрос, содержащий раздел **ORDER BY**, нельзя использовать в разделе **FROM** других запросов.

Приведенная характеристика средств манипулирования данными языка SQL является не вполне точной и полной. Кроме того, она отражает *семантику* оператора SQL, а не то, как он обычно исполняется в SQL-ориентированных СУБД.

**Ограничения целостности в модели SQL.** Как отмечалось в начале подраздела, наиболее важным отличием модели данных SQL от реляционной модели данных является то, что таблицы SQL могут содержать мультимножества строк. Из этого, в частности, следует, что в модели SQL отсутствует обязательное предписание об ограничении целостности сущности. В базе данных могут существовать таблицы, для которых не определен первичный ключ. С другой стороны, если для таблицы определен первичный ключ, то для нее ограничение целостности сущности поддерживается точно так же, как это требуется в реляционной модели данных.

Ссылочная целостность в модели данных SQL поддерживается в обязательном порядке, но в трех разных вариантах, лишь один из которых полностью соответствует реляционной модели. Это связано с уже упоминавшимся в этом разделе интенсивным использованием в SQL неопределенных значений. Подробнее особенности ограничений ссылочной целостности в SQL рассматриваются в гл. 11.

Кроме того, в SQL имеются развитые возможности явного определения ограничений целостности на уровне столбцов таблиц, на уровне таблиц целиком и на уровне базы данных.

### 2.4.3. Истинная реляционная модель

Дейт и Дарвен очень подробно и тщательно разработали предлагаемый ими вариант реляционной модели данных в по-

следнем издании книги [11], изданном в крупном формате (около 600 с.), причем это очень насыщенный текст. Поэтому в кратком очерке истинной реляционной модели, предлагаемом в данном подразделе, можно описать только ее самые общие и внешние черты (подробнее см. [4]).

**Типы и структуры данных истинной реляционной модели.** К. Дейт и Х. Дарвен поставили перед собой трудную задачу: показать, что на основе идей Э. Кодда можно реализовать СУБД, обеспечивающие возможности представления и хранения данных сложной структуры, не меньшие тех, которые обеспечивают объектные и SQL-ориентированные СУБД. Этому мешал, прежде всего, тезис Кодда о нормализации отношений: в реляционной базе данных должны содержаться только отношения, атрибуты которых определены на «доменах, элементы которых являются атомарными (не составными) значениями» [1]. В [12] Дейт пишет: «Я согласен с Коддом, что желательно оставаться в рамках логики первого порядка, если это возможно. В то же время я отвергаю идею “атомарных значений”, по крайней мере, в смысле абсолютной атомарности. В Третьем манифесте мы допускаем наличие доменов, содержащих значения произвольной сложности. (Они могут быть даже отношениями.) Тем не менее, мы остаемся в рамках логики первого порядка.» Если учесть, что [1] является первой официальной публикацией Кодда по поводу реляционной модели данных, то трудно сказать, что Дейт очень уж строго следует всем его заветам. Те постулаты Кодда, которые вредят достижению цели Третьего манифеста, просто отвергаются.

В истинно реляционной модели очень большое внимание уделяется типам данных. Предлагаются три категории типов данных: скалярные типы, кортежные типы и типы отношений. *Скалярный тип* данных — это привычный инкапсулированный тип, реальная внутренняя структура которого скрыта от пользователей. Предлагаются механизмы определения новых скалярных типов и операций над ними. Типом атрибута определяемого скалярного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Некоторые базовые скалярные типы данных должны быть предопределены в системе. В число этих типов должен входить тип **truth value** (так Дейт и Дарвен называют *булевский тип*) ровно с двумя значениями *true* и *false*.

*Кортежный тип* — это безымянный тип данных, определяемый с помощью генератора типа **TUPLE** с указанием множества пар <имя\_атрибута, тип\_атрибута> (заголовок кортежа). Типом

атрибута кортежного типа может являться любой определенный к этому моменту скалярный тип, любой кортежный тип и тип отношения. Значением кортежного типа является кортеж, представляющий собой множество триплетов *<имя\_атрибута, тип\_атрибута, значение\_атрибута>*, которое соответствует заголовку кортежа этого кортежного типа.

*Тип отношения* — это безымянный тип данных, определяемый с помощью генератора типа **RELATION** с указанием некоторого заголовка кортежа. Значением типа отношения является заголовок отношения, совпадающий с заголовком кортежа этого типа отношения, и тело отношения, представляющее собой множество кортежей, соответствующих этому заголовку. Кортежные типы и типы отношений не являются инкапсулированными: имеется возможность прямого доступа к атрибутам.

Для всех разновидностей типов данных разработана модель множественного наследования, позволяющая определять новые типы данных на основе уже определенных типов. Модель наследования по Дейту и Дарвену не является частью истинной реляционной модели данных.

Понятно, что при таких определениях значениями атрибутов отношения могут быть не только значения произвольно сложных скалярных типов, типами атрибутов которых могут быть, в частности, отношения, но и просто отношения. Тем не менее, в [8] Дейт и Дарвен говорят: «Каждый кортеж в [отношении] *R* содержит в точности одно значение *v* для каждого атрибута *A* в [заголовке отношения] *H*. Иными словами, *R* находится в *первой нормальной форме*, 1NF.» Это понятное определение первой нормальной формы, но трудно сказать, согласился ли бы с ним Кодд.

База данных в истинной реляционной модели — это набор долговременно хранимых именованных *переменных отношений*, каждая из которых определена на некотором типе отношений. В каждый момент времени каждая переменная отношения базы данных содержит некоторое значение отношения соответствующего типа.

**Манипулирование данными в истинной реляционной модели.** Вообще говоря, в качестве эталонного средства манипулирования данными в истинной реляционной модели можно использовать реляционную алгебру Кодда (см. подразд. 2.3.2). Однако в [9—11] Дейт и Дарвен предложили новую реляционную алгебру, названную ими Алгеброй *A*, которая основана на реляционных аналогах булевских операций *конъюнкции*, *дизъюнкции* и *отрицания*. В гл. 4 будет показано, что через операции этой алгебры выражаются все операции алгебры Кодда.

**Ограничения целостности в истинной реляционной модели.** В число обязательных требований истинной реляционной модели входит требование определения хотя бы одного возможного ключа для каждой переменной отношения (возможный ключ — это одно из подмножеств заголовка переменной отношения, обладающее упоминавшимися в подразд. 2.3.3 свойствами первичного ключа). Кроме того, говорится, что «любое условное выражение, которое является (или логически эквивалентно) замкнутой правильно построенной формулой (WFF) реляционного исчисления, должно быть допустимо в качестве спецификации ограничения целостности» [4].

Средства поддержки декларативной ссылочной целостности фигурируют только в разделе рекомендуемых возможностей: «В **D** [конкретную реализацию истинной реляционной модели] следует включить некоторую декларативную сокращенную форму для выражения ссылочных ограничений (называемых также ограничениями внешнего ключа)».

\* \* \*

В этой главе было введено важнейшее в технологии баз данных понятие модели данных, кратко рассмотрены особенности трех ранних моделей данных: модели инвертированных таблиц, иерархической модели и сетевой модели данных. В отдельном подразделе представлена исходная реляционная модель данных, определенная Э. Коддом. Описаны основные черты трех современных моделей данных, системы типов данных которых позволяют сохранять в базе данных и обрабатывать данные произвольно сложной структуры: объектно-ориентированная модель данных, модель данных SQL и истинно реляционная модель данных.

### Глава 3

#### **РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ. ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ. ОСНОВНЫЕ СВОЙСТВА ОТНОШЕНИЙ. ЦЕЛОСТНОСТЬ СУЩНОСТИ И ССЫЛОК**

В данной и нескольких последующих главах обсуждаются различные аспекты реляционных баз данных. Принято считать, что реляционный подход к организации баз данных был заложен в конце 1960-х годов Э. Коддом. В последние десятилетия этот подход является наиболее распространенным (с той оговоркой, что в называемых в обиходе реляционными системами баз данных, основанных на языке SQL, в действительности нарушаются некоторые важные принципы классического реляционного подхода). Признанными достоинствами реляционного подхода считают следующие свойства:

- реляционный подход основан на небольшом числе интуитивно понятных абстракций, на основе которых возможно простое моделирование наиболее распространенных предметных областей; эти абстракции могут быть точно и формально определены;
- теоретическим базисом реляционного подхода к организации баз данных служит простой и мощный математический аппарат теории множеств и математической логики;
- реляционный подход обеспечивает возможность ненавигационного манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

Компьютерный мир далеко не сразу признал реляционные системы. В 1970-е годы, когда уже были получены почти все основные теоретические результаты и даже существовали первые прототипы реляционных СУБД, многие авторитетные специалисты отрицали возможность добиться эффективной реализации

таких систем. Однако преимущества реляционного подхода и развитие методов и алгоритмов организации и управления реляционными базами данных привели к тому, что к концу 1980-х годов реляционные (вернее, SQL-ориентированные) системы заняли на мировом рынке СУБД доминирующее положение.

В данной главе на сравнительно неформальном уровне вводятся основные понятия реляционных баз данных, а также определяется существо реляционной модели данных. В своем изложении будем опираться на некоторую смесь классической реляционной модели Кодда [1] и истинной реляционной модели Дейта и Дарвена [4]. Основной целью главы является демонстрация простоты и возможности интуитивной интерпретации реляционной модели. В последующих главах будут приводиться более формальные определения, на которых основывается теория реляционных баз данных.

### 3.1. Базовые понятия реляционных баз данных

Выделим следующие основные понятия реляционных баз данных: *тип данных*, *домен*, *атрибут*, *кортеж*, *отношение*, *первичный ключ*. Вначале покажем смысл этих понятий на примере отношения **СЛУЖАЩИЕ**, содержащего информацию о служащих некоторого предприятия (рис. 3.1).

#### 3.1.1. Тип данных

Значения данных, хранимые в реляционной базе данных, являются типизированными, т.е. известен тип каждого хранимого значения. Понятие *типа данных* в реляционной модели данных полностью соответствует понятию типа данных в языках программирования. Напомним, что традиционное (нестрогое) определение типа данных состоит из трех основных компонентов: определение множества значений данного типа; определение набора операций, применимых к значениям типа; определение способа внешнего представления значений типа (литералов).

Обычно в современных реляционных базах данных допускается хранение символьных, числовых данных (точных и приближительных), специализированных числовых данных (таких как «деньги»), а также специальных «темпоральных» данных (дата, время, временной интервал). Кроме того, в реляционных системах поддерживается возможность определения пользователями собственных типов данных (более подробно см. гл. 11).

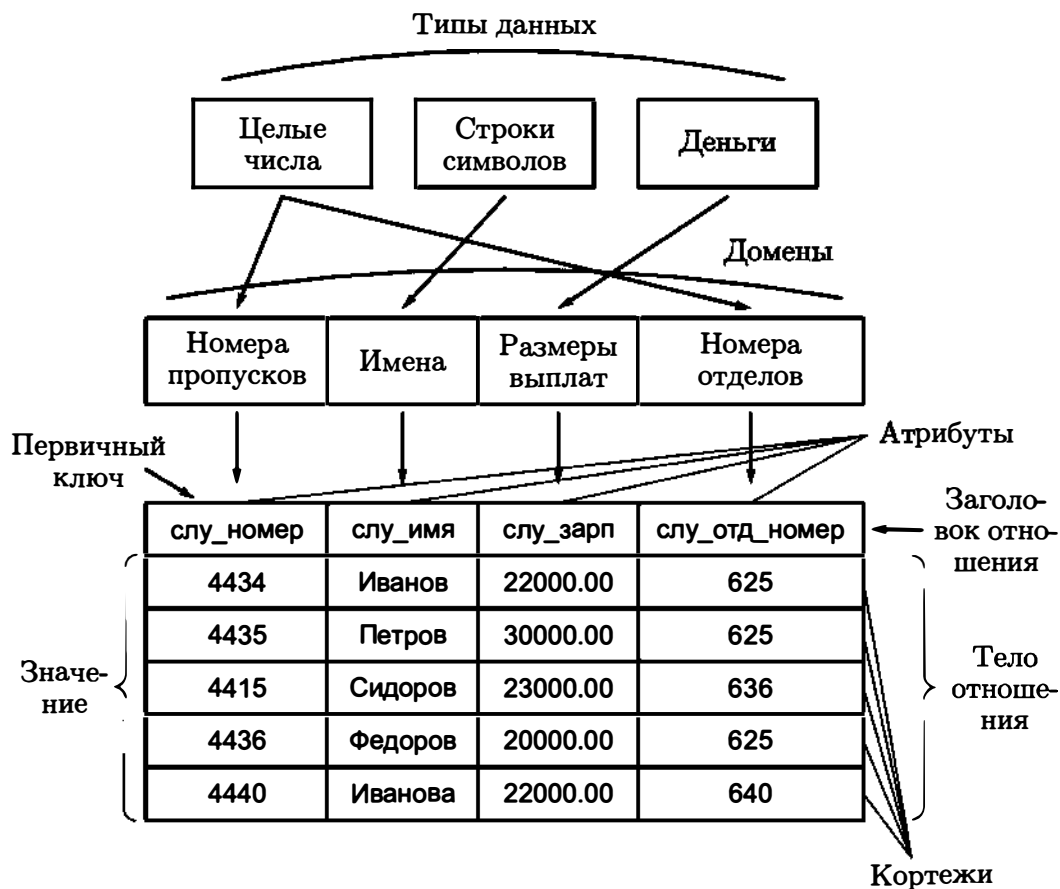


Рис. 3.1. Соотношение основных понятий реляционного подхода

В примере, приведенном на рис. 3.1, мы имеем дело с данными трех типов: строки символов, целые числа и «деньги».

### 3.1.2. Домен

Понятие *домена* более специфично для баз данных, хотя и имеются аналогии с подтипами в некоторых языках программирования (более того, в своем «Третьем манифесте» [4] К. Дейт и Х. Дарвен вообще ликвидируют различие между понятиями домена и типа данных). В общем виде домен определяется путем задания некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу этого типа данных (*ограничения домена*). Элемент данных является элементом домена в том и только том случае, если вычисление этого логического выражения дает результат *истина* (для логических значений мы будем попеременно использовать обозначения *истина* и *ложь* или *true* и *false* в зависимости от удобства). С каждым доменом

связывается имя, уникальное среди имен всех доменов соответствующей базы данных.

Наиболее правильной интуитивной трактовкой понятия домена является его восприятие как допустимого потенциального, ограниченного подмножества значений данного типа. Например, домен **ИМЕНА** в примере на рис. 3.1 определен на базовом типе символьных строк, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, для возможности представления русских имен такие строки не могут начинаться с мягкого или твердого знака и не могут быть длиннее, например, 20 символов). Если некоторый атрибут отношения определяется на некотором домене (как, например, на рис. 3.1 атрибут **СЛУ\_ИМЯ** определяется на домене **ИМЕНА**), то в дальнейшем ограничение домена играет роль ограничения целостности, накладываемого на значения этого атрибута.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов **НОМЕРА ПРОПУСКОВ** и **НОМЕРА ОТДЕЛОВ** относятся к типу целых чисел, но не являются сравнимыми (допускать их сравнение было бы бессмысленно).

### **3.1.3. Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения**

Понятие отношения является наиболее фундаментальным в реляционном подходе к организации баз данных, поскольку  $n$ -арное отношение является единственной родовой структурой данных, хранящихся в реляционной базе данных. Это отражено и в общем названии подхода — термин *реляционный* (*relational*) происходит от *relation* (*отношение*). Однако сам термин *отношение* является исключительно неточным, поскольку, говоря про любые сохраняемые данные, мы должны иметь в виду *тип* этих данных, *значения* этого типа и *переменные*, в которых сохраняются значения. Соответственно, для уточнения термина *отношение* выделяются понятия *заголовка отношения*, *значения отношения* и *переменной отношения*. Кроме того, нам потребуется вспомогательное понятие *кортежа*.

Итак, *заголовком* (или *схемой*) *отношения*  $R (H_R)$  называется конечное множество упорядоченных пар вида  $\langle A, T \rangle$ , где  $A$  называется именем *атрибута*, а  $T$  — имя некоторого базового типа или ранее определенного домена. По определению требуется, чтобы все имена атрибутов в заголовке отношения



были различны. В примере на рис. 3.1 заголовком отношения СЛУЖАЩИЕ является множество пар {<СЛУ\_НОМЕР, НОМЕРА\_ПРОПУСКОВ>, <СЛУ\_ИМЯ, ИМЕНА>, <СЛУ\_ЗАРП, РАЗМЕРЫ\_ВЫПЛАТ>, <СЛУ\_ОТД\_НОМЕР, НОМЕРА\_ОТДЕЛОВ>}

Если все атрибуты заголовка отношения определены на разных доменах, то, чтобы не плодить лишних имен, разумно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута).

Кортежем  $t_R$ , соответствующим заголовку  $H_R$ , называется множество упорядоченных триплетов вида  $\langle A, T, v \rangle$ , по одному такому триплету для каждого атрибута в  $H_R$ . Третий элемент —  $v$  — триплета  $\langle A, T, v \rangle$  должен являться допустимым значением типа данных или домена  $T$ . Заголовку отношения СЛУЖАЩИЕ соответствуют, например, следующие кортежи: {<СЛУ\_НОМЕР, НОМЕРА\_ПРОПУСКОВ, 4434>, <СЛУ\_ИМЯ, ИМЕНА, Иванов>, <СЛУ\_ЗАРП, РАЗМЕРЫ\_ВЫПЛАТ, 22.000>, <СЛУ\_ОТД\_НОМЕР, НОМЕРА\_ОТДЕЛОВ, 625>}, {<СЛУ\_НОМЕР, НОМЕРА\_ПРОПУСКОВ, 4455>, <СЛУ\_ИМЯ, ИМЕНА, Кузнецов>, <СЛУ\_ЗАРП, РАЗМЕРЫ\_ВЫПЛАТ, 35.000>, <СЛУ\_ОТД\_НОМЕР, НОМЕРА\_ОТДЕЛОВ, 320>}

Телом  $B_R$  отношения  $R$  называется произвольное множество кортежей  $t_R$ . Одно из возможных тел отношения СЛУЖАЩИЕ показано на рис. 3.1. Заметим, что в общем случае, как это демонстрируют, в частности, рис. 3.1 и пример предыдущего абзаца, могут существовать такие кортежи  $t_R$ , которые соответствуют  $H_R$ , но не входят в  $B_R$ .

Значением  $V_R$  отношения  $R$  называется пара множеств  $H_R$  и  $B_R$ . Одно из допустимых значений отношения СЛУЖАЩИЕ показано на рис. 3.1.

В изменчивой реляционной базе данных хранятся отношения, значения которых изменяются во времени. Переменной  $VAR_R$  называется типизированный именованный контейнер, который может содержать любое допустимое значение  $V_R$ . Как и для переменных в языках программирования, переменным отношения можно присваивать значения, а также выбирать из переменных ранее присвоенные им значения. Естественно, что при определении любой  $VAR_R$  требуется указывать соответствующий заголовок отношения  $H_R$ .

Здесь стоит подчеркнуть, что любая принятая на практике операция обновления базы данных — INSERT (вставка кортежа в переменную отношения), DELETE (удаление кортежа из значения-отношения переменной отношения) и UPDATE (модификация

кортежа значения-отношения переменной отношения) — с модельной точки зрения является операцией присваивания переменной отношения некоторого нового значения-отношения. Это совсем не означает, что перечисленные операции должны выполняться именно таким образом в СУБД: главное, чтобы результат операций соответствовал этой модельной семантике.

Заметим, что в дальнейшем в тех случаях, когда точный смысл термина понятен по контексту, будем использовать неуточненный термин *отношение*, как в смысле *значение отношения*, так и в смысле *переменная отношения*.

По определению *степенью*, или «*арностью*» заголовка отношения, кортежа, соответствующего этому заголовку, тела отношения, значения отношения и переменной отношения является мощность заголовка отношения. Например, степень отношения СПУЖАЩИЕ равна четырем, т.е. оно является 4-арным (*квартарным*).

При приведенных определениях разумно считать *схемой реляционной базы данных* набор пар  $\langle \text{имя\_VAR}_R, H_R \rangle$ , включающий имена и заголовки всех переменных отношения, которые определены в базе данных. *Реляционная база данных* — это набор  $\text{VAR}_R$  (конечно, каждая переменная отношения в любой момент времени содержит некоторое значение-отношение, в частности, пустое).

Заметим, что в классических реляционных базах данных после определения схемы базы данных могли изменяться только значения переменных отношений. Однако теперь в большинстве реализаций допускается и изменение схемы базы данных: определение новых и изменение заголовков существующих переменных отношений. Это принято называть *эволюцией схемы базы данных*.

### 3.1.4. Первичный ключ и интуитивная интерпретация реляционных понятий

По определению, первичным ключом переменной отношения является подмножество\*  $S$  множества атрибутов ее заголовка такое, что в любой момент времени значение  $S$  (составное, если в состав  $S$  входит более одного атрибута) в любом кортеже тела

---

\* В вырожденном случае, когда заголовок переменной отношения является пустым множеством, первичный ключ этой переменной отношения состоит из пустого подмножества заголовка. Легко проверить, что этот случай не противоречит общему определению.

отношения, являющегося значением данной переменной отношения, отличается от значения  $S$  в любом другом кортеже тела этого отношения, а никакое собственное подмножество\*  $S$  этим свойством не обладает. В следующем разделе мы покажем, что существование первичного ключа у любого значения отношения является следствием одного из фундаментальных свойств отношений, а именно, того свойства, что тело отношения является множеством кортежей.

Обычное представление отношения — *таблица*, заголовком которой является схема отношения, а *строками* — кортежи отношения-экземпляра; в этом случае имена атрибутов именуют *столбцы* этой таблицы. Поэтому иногда говорят про «столбцы таблицы», имея в виду «атрибуты отношения».

Конечно, это достаточно грубая терминология, поскольку у обычных таблиц и строки, и столбцы упорядочены, тогда как атрибуты и кортежи отношений являются элементами неупорядоченных множеств. Тем не менее при рассмотрении практических вопросов организации реляционных баз данных и средств управления ими будем использовать эту житейскую терминологию. Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД. Иногда также используются термины: *файл* как аналог таблицы, *запись* как аналог строки и *поле* как аналог столбца. Как вы помните, эта терминология была использована в гл. 1, хотя и не совсем в контексте реляционных баз данных.

## 3.2. Фундаментальные свойства отношений

Остановимся на некоторых важных свойствах отношений, которые следуют из приведенных ранее определений.

### 3.2.1. Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений

То свойство, что тело любого отношения никогда не содержит кортежей-дубликатов, следует из определения тела отношения как множества кортежей. В классической теории множеств по определению любое множество состоит из различных эле-

---

\* Напомним, что  $S'$  является *собственным подмножеством* множества  $S$  в том и только том случае, когда  $S'$  входит в  $S$ , но не совпадает с  $S$  (это обозначается как  $S' \subset S$ ).

ментов. Именно из этого свойства вытекает наличие у каждого значения отношения *первичного ключа* — минимального подмножества заголовка данного отношения, составное значение которого уникально определяет кортеж отношения. Действительно, поскольку в любое время все кортежи тела любого отношения различны, у любого значения отношения свойством уникальности обладает, по крайней мере, полный набор его атрибутов. Однако в формальном определении первичного ключа требуется обеспечение его «минимальности», т.е. в набор атрибутов первичного ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства — однозначного определения кортежа. Немного позже будет показано, почему свойство минимальности первичного ключа является критически важным. Понятно, что если у любого отношения существует набор атрибутов, обладающий свойством уникальности, то существует и минимальный набор атрибутов, обладающий свойством уникальности.

Конечно, могут существовать значения отношения с несколькими несовпадающими минимальными наборами атрибутов, обладающими свойствами уникальности. Например, если вернуться к предположениям гл. 1 об уникальности значений атрибутов `СЛУ_НОМЕР` и `СЛУ_ИМЯ` отношения `СЛУЖАЩИЕ`, то для каждого значения этого отношения мы имеем два множества атрибутов, претендующих на звание первичного ключа — `{СЛУ_НОМЕР}` и `{СЛУ_ИМЯ}`. В этом случае проектировщик базы данных должен решить, какое из альтернативных множеств атрибутов назвать первичным ключом, а остальные минимальные наборы атрибутов, обладающие свойством уникальности, называются *возможными ключами*<sup>\*</sup>.

Понятие первичного ключа является исключительно важным в связи с понятием целостности баз данных. Заметим, что хотя формально существование первичного ключа значения отношения является следствием того, что тело отношения — это множество, на практике первичные (и возможные) ключи *переменных отношений* появляются в результате явных указаний проектировщика отношения. Определяя переменную отношения, проектировщик моделирует часть предметной области, данные из которой будут в дальнейшем содержать база данных. И конечно, проектировщик должен знать природу этих данных. Например, ему должно быть известно, что никакие два служа-

---

<sup>\*</sup> В гл. 11 будут рассмотрены различия между первичными и возможными ключами в языке SQL.

щих ни в какой времени не могут иметь удостоверение с одним и тем же номером. Поэтому он может (и даже должен, как будет показано далее) явно объявить {СЛУ\_НОМЕР} возможным ключом. Если на предприятии установлено, что у всех сотрудников должны быть разные полные имена, то проектировщик может (и опять же должен) объявить возможным ключом и {СЛУ\_ИМЯ}. Потом проектировщик должен оценить, какой из возможных ключей является более надежным (свойство его уникальности никогда не будет отменено), и выбрать наиболее надежный возможный ключ в качестве первичного (в нашем случае естественным выбором был бы ключ {СЛУ\_НОМЕР}, потому что решение об уникальности полных имен сотрудников выглядит искусственным и странным и может быть легко отменено руководством предприятия).

Теперь поясним, почему проектировщику следует явно объявлять первичный и возможные ключи переменных отношений\*. Дело в том, в результате этого объявления СУБД получает информацию, которая в дальнейшем будет использоваться как ограничение целостности\*\*. СУБД никогда не допустит появления в переменной отношения значения-отношения, содержащего два кортежа с одинаковым значением атрибута СЛУ\_НОМЕР (определение первичного ключа для данной переменной отношения нельзя отменить). Появление двух кортежей с одинаковым значением атрибута СЛУ\_ИМЯ будет также невозможно до тех пор, пока остается в силе определение {СЛУ\_ИМЯ} как возможного ключа. Тем самым, объявления первичного и возможных ключей дают СУБД возможность поддерживать целостность базы данных при попытках занесения в нее некорректных данных.

Наконец, вернемся к свойству минимальности первичного (и возможных) ключей. Как было отмечено ранее, это свойство является критически важным, и важность проявляется именно при трактовке первичного и возможных ключей как ограничений целостности. В рассматриваемом примере с отношением

---

\* Если он является сторонником классического реляционного подхода; в языке SQL допускается определение таблиц без первичного и возможных ключей.

\*\* Кстати, заметим, что в классической реляционной модели, если при определении переменной отношения явно не указывается ее первичный ключ, то по умолчанию первичным ключом считается полный набор атрибутов заголовка отношения. Конечно, в этом случае такая переменная отношения может принимать любое значение-отношение, соответствующее заголовку, и первичный ключ не играет роли ограничения.

**СЛУЖАЩИЕ** свойством уникальности будет обладать не только множество атрибутов {СЛУ\_НОМЕР}, но и, например, множество {СЛУ\_НОМЕР, СЛУ\_ОТД\_НОМЕР}. Но если бы мы выставили в качестве ограничения целостности требование уникальности {СЛУ\_НОМЕР, СЛУ\_ОТД\_НОМЕР}, то СУБД гарантировала бы отсутствие кортежей с одинаковым значением атрибута СЛУ\_НОМЕР не во всем значении отношения **СЛУЖАЩИЕ**, а только в группах кортежей с одним и тем же значением атрибута СЛУ\_ОТД\_НОМЕР. Понятно, что это не соответствует смыслу моделируемой предметной области.

Забегаая вперед, заметим, что во многих практических реализациях реляционных СУБД допускается нарушение свойства уникальности кортежей для промежуточных отношений, порождаемых неявно при выполнении запросов. Такие отношения являются не множествами, а мультимножествами, что в ряде случаев позволяет добиться определенных преимуществ, но часто приводит к серьезным проблемам. Мы остановимся на этом подробнее при обсуждении языка SQL.

### 3.2.2. Отсутствие упорядоченности кортежей

Конечно, формально свойство отсутствия упорядоченности кортежей в значении отношения также является следствием определения тела отношения как множества кортежей. Однако на это свойство можно взглянуть и с другой стороны. Да, то свойство, что тело отношения является множеством кортежей, облегчает построение полного механизма реляционной модели данных, включая базовые средства манипулирования данными — реляционные алгебру и исчисление. Но, по мнению автора, основная причина не в этом.

Достаточно часто у пользователей реляционных СУБД и разработчиков информационных систем вызывает раздражение тот факт, что они не могут хранить кортежи отношений на физическом уровне в нужном им порядке. И ссылки на требования реляционной теории здесь не очень уместны. Можно было бы разработать другую теорию, в которой допускались бы упорядоченные «отношения». Однако хранить упорядоченные списки кортежей в условиях интенсивно обновляемой базы данных гораздо сложнее технически, а поддержка упорядоченности вызывает серьезные накладные расходы.

Отсутствие требования к поддержанию порядка на множестве кортежей отношения придает СУБД дополнительную гибкость при хранении баз данных во внешней памяти и при

выполнении запросов к базе данных. Это не противоречит тому, что при формулировании запроса к БД, например, на языке SQL можно потребовать сортировку результирующей таблицы в соответствии со значениями некоторых столбцов. Такой результат, вообще говоря, является не отношением, а некоторым упорядоченным списком кортежей, и он может быть только окончательным результатом, к которому уже нельзя адресовать запросы.

### 3.2.3. Отсутствие упорядоченности атрибутов

Атрибуты отношений не упорядочены, поскольку по определению заголовков отношения есть множество пар `<имя_атрибута, имя_типа_или_домена>`. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута.

Легко заметить явную аналогию между заголовками отношений и структурными типами в языках программирования. Так вот, даже в языке программирования C с его практически неограниченными возможностями работы с указателями настоятельно рекомендуется обращаться к полям структур только по их именам.

Если, например, на языке C определена структурная переменная

```
STRUCT {integer a; char b; integer c} d;
```

то в стандарте языка решительно *не* рекомендуется использовать для доступа к символьному полю *b* конструкцию `*(&d + sizeof(integer))` (взять адрес структурной переменной *d*, прибавить к нему число байтов в целом числе и взять значение байта по полученному адресу). Это объясняется тем, что при реальном расположении в памяти полей этой структурной переменной в том порядке, как они определены, во многих компьютерах потребуется выровнять поле *c* по байту с четным адресом. Поэтому один байт просто пропадет. При расположении структурной переменной в памяти экономный компилятор (вернее, оптимизатор) переставит местами поля *b* и *c*, и указанная ранее конструкция не обеспечит доступа к полю *b*.

Для корректного обращения к полю *b* переменной *d* нужно использовать конструкции `d.b` или `&d → b`, т. е. явно указывать имя поля.

Аналогичными практическими соображениями оправдывается и отсутствие упорядоченности атрибутов в заголовке отношения. В этом случае СУБД сама принимает решение

о том, в каком физическом порядке следует хранить значения атрибутов кортежей (хотя обычно один и тот же физический порядок поддерживается для всех кортежей каждого отношения). Кроме того, это свойство облегчает выполнение операции модификации схем существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих атрибутов.

Снова забегая вперед, заметим, что в языке SQL в некоторых случаях допускается индексное указание атрибутов, причем в качестве неявного порядка атрибутов используется их порядок в линейной форме определения схемы отношения (это одна из осуждаемых особенностей языка SQL).

### **3.2.4. Атомарность значений атрибутов, первая нормальная форма отношения**

Как отмечалось в гл. 2, за годы, прошедшие после появления исходной реляционной модели данных, некоторые ее требования, которые некогда считались несомненными, подверглись существенному пересмотру. К их числу относятся требования атомарности атрибутов и наличия первой нормальной формы отношений.

Эдгар Кодд в [1] справедливо утверждал, что для моделирования большинства предметных областей можно обойтись отношениями, атрибуты которых определены на простых доменах, элементы которых являются атомарными, не декомпозируемыми.

Он, в частности, говорил следующее: «Отношение, все домены которого являются простыми, может быть представлено двухмерным массивом ... с однородными столбцами. Для отношения с одним или более непростыми доменами требуются несколько более сложные структуры данных.»

Более того, он предлагал простую процедуру нормализации, приводящую отношение, значениями одного из атрибутов которых являются отношения, к нескольким отношениям над простыми доменами.

Однако, как было показано в гл.2, и в истинной реляционной модели данных, и в модели данных SQL теперь можно определять отношения (таблицы), значениями атрибутов (столбцов) которых являются отношения (мультимножества строк). Скорее всего, это произошло по двум причинам. Во-первых, в течение многих лет реляционную модель данных (и модель



SQL) многие люди критиковали за сложность представления иерархически организованных данных. Понятно, что при наличии механизма вложенных отношений иерархические данные представляются вполне естественно. Во-вторых, оказалось, что при тщательном определении системы типов, при введении типа отношения, как в истинной реляционной модели данных, или типа мультимножества строк, как в модели SQL, наличие атрибутов (столбцов) со значениями-отношениями никак не влияет на методы, разработанные в теории реляционных баз данных, которая исходно опиралась на требование первой нормальной формы по Кодду.

Поэтому, по всей видимости, теперь требование первой нормальной формы звучит так: отношение  $R$  находится в *первой нормальной форме* (1NF), если каждый кортеж в отношении  $R$  содержит в точности одно значение  $v$  для каждого атрибута  $A$  из заголовка отношения  $H_R$ . Понятно, что это требование теперь является тавтологией, поскольку следует из определения отношения.

Заметим, тем не менее, что доводы Кодда, которые привели к появлению исходной первой нормальной формы отношений, совсем не утратили смысла.

На рис. 3.2 показан пример бинарного отношения, в котором значениями атрибута ОТДЕЛЫ являются отношения. Заметим,

| НОМЕР ОТДЕЛА | ОТДЕЛ     |         |          |
|--------------|-----------|---------|----------|
|              | СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП |
| 625          | 4434      | Иванов  | 22000.00 |
|              | 4435      | Петров  | 30000.00 |
|              | 4436      | Федоров | 20000.00 |
| 636          | 4415      | Сидоров | 23000.00 |
| 640          | 4440      | Иванова | 22000.00 |

Рис. 3.2. Ненормализованное отношение ОТДЕЛЫ-СЛУЖАЩИЕ

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22000.00 | 625           |
| 4435      | Петров  | 30000.00 | 625           |
| 4415      | Сидоров | 23000.00 | 636           |
| 4436      | Федоров | 20000.00 | 625           |
| 4440      | Иванова | 22000.00 | 640           |

Рис. 3.3. Отношение СЛУЖАЩИЕ: нормализованный вариант отношения ОТДЕЛЫ-СЛУЖАЩИЕ

что исходное отношение СЛУЖАЩИЕ является нормализованным вариантом отношения ОТДЕЛЫ-СЛУЖАЩИЕ. Нормализованный (по Кодду) вариант показан на рис. 3.3.

Нормализованные по Кодду отношения составляют основу классического реляционного подхода к организации баз данных. Они обладают некоторыми ограничениями (не любую информацию удобно представлять в виде плоских таблиц), но существенно упрощают манипулирование данными. Рассмотрим, например, два идентичных оператора занесения кортежа:

- Зачислить служащего Кузнецова (пропуск номер 3000, заработная плата 115,000) в отдел номер 320;
- Зачислить служащего Кузнецова (пропуск номер 3000, заработная плата 115,000) в отдел номер 310.

Если информация о сотрудниках представлена в виде отношения СЛУЖАЩИЕ, оба оператора будут выполняться одинаково (вставить кортеж в отношение СЛУЖАЩИЕ). Если же работать с ненормализованным отношением ОТДЕЛЫ-СЛУЖАЩИЕ, то первый оператор выразится в занесение кортежа, а второй — в добавление кортежа с первичным ключом 310 в значение-отношение атрибута ОТДЕЛ.

При работе с ненормализованными отношениями аналогичные затруднения возникают при выполнении операций удаления и модификации кортежей.

### 3.3. Реляционная модель данных

Как отмечалось ранее, реляционная модель данных состоит из трех частей, описывающих разные аспекты реляционного подхода: структурной, манипуляционной и целостной частей (см. гл. 2).

### 3.3.1. Общая характеристика

В *структурной части* реляционной модели данных фиксируется, что единственной родовой\* структурой данных, используемой в реляционных БД, является нормализованное  $n$ -арное отношение. Определяются понятия доменов, атрибутов, кортежей, заголовка, тела и переменной отношения. По сути дела, в предыдущих двух подразделах этой главы были рассмотрены именно понятия и свойства структурной составляющей реляционной модели.

В *манипуляционной части* реляционной модели определяются два фундаментальных механизма манипулирования реляционными БД — реляционная алгебра и реляционное исчисление. Первый механизм базируется в основном на классической теории множеств (с некоторыми уточнениями и добавлениями), а второй — на классическом логическом аппарате исчисления предикатов первого порядка.

Мы кратко рассмотрели алгебру Кодда в гл. 2 и обсудим все эти механизмы более подробно в следующей главе. Пока лишь заметим, что основной функцией манипуляционной части реляционной модели является обеспечение меры реляционности любого конкретного языка реляционных БД (язык называется реляционно-полным, если он обладает не меньшей выразительностью и мощностью, чем реляционная алгебра или реляционное исчисление).

---

\* В этой книге уже несколько раз утверждалось, что нормализованное  $n$ -арное отношение является единственной родовой структурой данных, используемой в реляционных БД. Пришло время пояснить, что имеется в виду под термином *родовая структура*. В языках программирования с развитыми системами типов обычно имеются конструкции, называемые *родовыми типами*, *параметризуемыми типами*, *конструкторами типов*, *генераторами типов*, позволяющие породить конкретный тип данных на основе его абстрактной (обычно, предопределенной) спецификации. Особенность таких типов состоит в том, что и основные операции конкретного типа определяются на уровне этой абстрактной спецификации. Одним из наиболее известных примеров является *тип множества*, например, в языке Pascal. В случае реляционной модели данных в этой книге мы явно не говорим, что отношение является родовым типом, но, по существу, это именно так. Операции реляционной алгебры определяются на уровне абстрактного отношения и применимы к любым значениям-отношениям с конкретными заголовками.

### 3.3.2. Целостность сущности и ссылок

Наконец, в *целостной части* реляционной модели данных фиксируются два базовых требования целостности, которые должны поддерживаться в любой реляционной СУБД. Первое требование называется *требованием целостности сущности* (*entity integrity*). Объекту или сущности реального мира в реляционных БД соответствуют кортежи отношений. Конкретно, требование состоит в том, что любой кортеж любого значения-отношения любой переменной отношения должен быть отличим от любого другого кортежа этого значения отношения по составным значениям заранее определенного множества атрибутов переменной отношения, т. е. любая переменная отношения должна обладать первичным ключом. Как мы видели в предыдущем подразделе, это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений.

На самом деле, требование целостности сущности полностью звучит следующим образом: *у любой переменной отношения должен существовать первичный ключ, и никакое значение первичного ключа в кортежах значения-отношения переменной отношения не должно содержать неопределенных значений*. Чтобы эта формулировка была полностью понятна, мы должны хотя бы кратко обсудить понятие *неопределенного значения* (NULL)\*. Конечно, теоретически любой кортеж, заносимый в сохраняемое отношение, должен содержать все характеристики моделируемой им сущности реального мира, которые мы хотим сохранить в базе данных. Однако на практике не все эти характеристики могут быть известны к тому моменту, когда требуется

---

\* Здесь необходимо заметить, что поддержка в реляционной модели данных *неопределенного значения* является еще одним аспектом, в котором классическая реляционная модель Кодда отличается от истинной реляционной модели Дейта и Дарвена. Понятие NULL в реляционную модель ввел сам Кодд для того, чтобы обеспечить хотя бы какое-то средство для представления в базах данных отсутствующей информации [13]. Неопределенное значение порождает много проблем, которые мы будем отмечать в этой и следующих главах. Дейт и Дарвен категорически выступают против поддержки NULL и в [4] предлагают использовать взамен «специальные значения». Мы не будем в этой книге обсуждать эту часть истинной реляционной модели, но заметим, что в действительности «специальные значения» Дейта и Дарвена проблему не решают. Учитывая также то, что неопределенное значение давно и глубоко проникло в модель SQL, в этой книге будем считать, что в реляционной модели данных неопределенное значение поддерживается.

зафиксировать сущность в базе данных. Простым примером может быть процедура принятия на работу человека, размер заработной платы которого еще не определен. В этом случае работник отдела кадров, который заносит в отношении **СЛУЖАЩИЕ** кортеж, описывающий нового служащего, просто не может обеспечить значение атрибута **СЛУ\_ЗАРП** (любое значение домена **РАЗМЕРЫ\_ВЫПЛАТ** будет неверно характеризовать заработную плату нового сотрудника). Э. Кодд [13] предложил использовать в таких случаях неопределенные значения. Неопределенное значение не принадлежит никакому типу данных и может присутствовать среди значений любого атрибута, определенного на любом типе данных (если это явно не запрещено при определении атрибута). Если *a* — значение некоторого типа данных или **NULL**, *op* — любая двуместная «арифметическая» операция этого типа данных (например, «+»), а *lop* — операция сравнения значений этого типа (например, «=»), то по определению:

```
a op NULL = NULL
NULL op a = NULL
a lop NULL = unknown
NULL lop a = unknown
```

Здесь *unknown* — третье значение логического, или булевского, типа, обладающее следующими свойствами:

```
NOT unknown = unknown
true AND unknown = unknown
true OR unknown = true
false AND unknown = false
false OR unknown = unknown
```

(напомним, что операции **AND** и **OR** являются коммутативными)\*. В этой главе достаточно приведенного краткого введения в неопределенные значения, но в последующих главах будем неоднократно возвращаться к этому вопросу.

\* Как показывает опыт автора, не всегда и не все студенты помнят базовые логические операции. Для гарантии приведем полные таблицы истинности операций **AND** («&» конъюнкция), **OR** («V» дизъюнкция) и **NOT** («¬» отрицание):

|       |       |       |       |      |       |     |       |       |
|-------|-------|-------|-------|------|-------|-----|-------|-------|
| AND   | true  | false | OR    | true | false | NOT | true  | false |
| true  | true  | false | true  | true | true  |     | false | true  |
| false | false | false | false | true | false |     |       |       |

Так вот, требование целостности сущности означает, что первичный ключ должен полностью идентифицировать каждую сущность, а поэтому в составе любого значения первичного ключа не допускается наличие неопределенных значений. (В классической реляционной модели это требование распространяется и на возможные ключи (в SQL-ориентированных СУБД такое требование для возможных ключей не поддерживается, см. гл. 11).

Второе требование называется *требованием ссылочной целостности (referential integrity)* и является несколько более сложным. Очевидно, что при соблюдении нормализованности по Кодду отношений сложные сущности реального мира представляются в реляционной БД в виде нескольких кортежей нескольких отношений. Например, представим, что требуется представить в реляционной базе данных сущность ОТДЕЛ с атрибутами ОТД\_НОМЕР (номер отдела), ОТД\_РАЗМ (количество служащих) и ОТД\_СЛУ (набор сотрудников отдела). Для каждого служащего нужно хранить СЛУ\_НОМЕР (номер сотрудника), СЛУ\_ИМЯ (имя сотрудника) и СЛУ\_ЗАРП (заработная плата сотрудника). Как будет показано в гл. 5, при правильном проектировании соответствующей БД в ней появятся два отношения: ОТДЕЛЫ { ОТД\_НОМЕР, ОТД\_РАЗМ } (первичный ключ — {ОТД\_НОМЕР}) и СОТРУДНИКИ { СЛУ\_НОМЕР, СЛУ\_ИМЯ, СЛУ\_ЗАРП, СЛУ\_ОТД\_НОМ } (первичный ключ — {СЛУ\_НОМЕР}).

Как видно, атрибут СЛУ\_ОТД\_НОМ появляется в отношении СЛУЖАЩИЕ не потому, что номер отдела является собственным свойством сотрудника, а лишь для того, чтобы иметь возможность восстановить при необходимости полную сущность ОТДЕЛ. Значение атрибута СЛУ\_ОТД\_НОМ в любом кортеже отношения СЛУЖАЩИЕ должно соответствовать значению атрибута ОТД\_НОМ в некотором кортеже отношения ОТДЕЛЫ. Атрибут такого рода (возможно, составной) называется *внешним ключом (foreign key)*, поскольку его значения однозначно характеризуют сущности, представленные кортежами некоторого другого отношения (т.е. задают значения их первичного ключа). Конечно, внешний ключ может быть составным — состоять из нескольких атрибутов. Говорят, что отношение, в котором определен внешний ключ, ссылается на соответствующее отношение, в котором такой же атрибут является первичным ключом.

В полной форме требование целостности по ссылкам, или требование целостности внешнего ключа, состоит в том, что для каждого значения внешнего ключа, появляющегося в кортеже значения-отношения ссылающейся переменной отношения,

либо в значении-отношении переменной отношения, на которую ведет ссылка, должен найтись кортеж с таким же значением первичного ключа, либо значение внешнего ключа должно быть полностью неопределенным (т.е. ни на что не указывать)\*. Для рассматриваемого примера это означает, что если для сотрудника указан номер отдела, то этот отдел должен существовать.

Заметим, что, как и первичный ключ, внешний ключ должен специфицироваться при определении переменной отношения и представляет собой ограничение на допустимые значения-отношения этой переменной. Другими словами, определение внешнего ключа представляет собой определение ограничения целостности базы данных.

Ограничения целостности сущности и ссылочной целостности должны поддерживаться СУБД. Для соблюдения целостности сущности достаточно гарантировать отсутствие в любой переменной отношения значений-отношений, содержащих кортежи с одним и тем же значением первичного ключа (и запрещать вхождение в значение первичного ключа неопределенных значений). С целостностью по ссылкам дело обстоит несколько сложнее.

Понятно, что при обновлении ссылающегося отношения (вставке новых кортежей или модификации значения внешнего ключа в существующих кортежах) достаточно следить за тем, чтобы не появлялись некорректные значения внешнего ключа. Но как быть при удалении кортежа из отношения, на которое ведет ссылка?

Здесь существуют три подхода, каждый из которых поддерживает целостность по ссылкам. Первый подход заключается в том, что вообще запрещается производить удаление кортежа, для которого существуют ссылки (т.е. сначала нужно либо удалить ссылающиеся кортежи, либо соответствующим образом изменить значения их внешнего ключа). При втором подходе при удалении кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа автоматически становится неопределенным. Наконец, третий подход (каскадное удаление) состоит в том, что при удалении кортежа из отношения, на которое ведет ссылка, из ссылающегося отношения автоматически удаляются все ссылающиеся кортежи.

---

\* В языке SQL допускается несколько вариантов определения внешнего ключа, только один из которых полностью соответствует классическому подходу (подробнее см. гл. 11).

В развитых реляционных СУБД обычно можно выбрать способ поддержания целостности по ссылкам для каждой отдельной ситуации определения внешнего ключа. Конечно, для принятия такого решения необходимо анализировать требования конкретной прикладной области.

\* \* \*

С очень большой вероятностью читатели этой книги работают или будут работать с какой-либо SQL-ориентированной СУБД. Любая компания, производящая подобные СУБД, называет их реляционными системами. Важно отчетливо понимать, какие свойства таких систем действительно являются реляционными, а что в них в действительности отходит от исходных, ясных и строгих идей реляционного подхода и даже противоречит этим идеям. При наличии такого понимания можно более правильно организовывать базы данных и строить приложения в среде SQL-ориентированной СУБД.

В ч. V этой книги обсуждаются возможности языка, специфицированные в стандартах SQL:1999 и SQL:2003. Но до этого читателям предлагается материал, который представляет реляционный подход в первозданном и чистом виде. В данной главе вводится понятийная основа реляционного подхода, определяются основные термины, исследуются фундаментальные следствия базовых определений. Определяемая реляционная модель данных предназначена, прежде всего, для оценки соответствия различных реализаций СУБД общему реляционному подходу.



Как отмечалось в предыдущих главах, в манипуляционной составляющей реляционной модели данных определяются два базовых механизма манипулирования реляционными данными — основанная на теории множеств реляционная алгебра и базирующееся на математической логике (точнее, на исчислении предикатов первого порядка) реляционное исчисление. В свою очередь, обычно выделяются два вида реляционного исчисления — исчисление кортежей и исчисление доменов.

Все эти механизмы обладают одним важным свойством: они замкнуты относительно понятия отношения. Это означает, что выражения реляционной алгебры и формулы реляционного исчисления определяются над отношениями реляционных БД и результатом их «вычисления» также являются отношения (конечно, здесь имеются в виду значения-отношения). В результате любое выражение или формула могут интерпретироваться как отношения, что позволяет использовать их в других выражениях или формулах.

Далее будет показано, что алгебра и исчисление обладают большой выразительной мощностью: очень сложные запросы к базе данных могут быть выражены с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления. Именно по этой причине такие механизмы включены в реляционную модель данных. Конкретный язык манипулирования реляционными БД называется реляционно-полным, если любой запрос, формулируемый с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления, может быть сформулирован с помощью одного оператора этого языка.

Известно (здесь не будем это доказывать), что механизмы реляционной алгебры и реляционного исчисления эквивалентны, т.е. для любого допустимого выражения реляционной алгебры можно построить эквивалентную (т.е. производящую такой же результат) формулу реляционного исчисления, и наоборот. Почему же в реляционной модели данных традиционно присутствуют оба эти механизма?

Дело в том, что они различаются уровнем процедурности. Выражения реляционной алгебры строятся на основе алгебраических операций (высокого уровня), и подобно тому, как интерпретируются арифметические и логические выражения, выражение реляционной алгебры также имеет процедурную интерпретацию. Другими словами, запрос, представленный на языке реляционной алгебры, может быть вычислен на основе выполнения элементарных алгебраических операций с учетом их приоритетов и возможного наличия скобок. Для формулы реляционного исчисления однозначная вычислительная интерпретация, вообще говоря, отсутствует. Формула только ставит условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются в большей степени непроцедурными, или декларативными.

Поскольку механизмы реляционной алгебры и реляционного исчисления эквивалентны, в конкретной ситуации для проверки степени реляционности некоторого языка БД можно пользоваться любым из этих механизмов.

Заметим, что крайне редко алгебра или исчисление принимается в качестве полной основы какого-либо языка БД. Обычно (например, в случае языка SQL) язык основывается на некоторой смеси алгебраических и логических конструкций. Тем не менее знание алгебраических и логических основ языков баз данных часто бывает полезным на практике.

## 4.1. Алгебра Кодда

Основная идея реляционной алгебры состоит в том, что коль скоро отношения являются множествами, средства манипулирования отношениями могут базироваться на традиционных теоретико-множественных операциях, дополненных некоторыми специальными операциями, специфичными для реляционных баз данных.

Существует много подходов к определению реляционной алгебры, которые различаются наборами операций и способами их интерпретации, но, в принципе, являются более или менее равносильными. В данном подразделе рассмотрим немного расширенный начальный вариант алгебры\*, который был предложен Коддом (будем называть ее «алгеброй Кодда»).

---

\* Как отмечалось в гл. 2, строго говоря, этот набор операций не является алгеброй на множестве отношений, поскольку двуместные операции этого набора применимы не к любым парам отношений. Тем не менее исторически в области баз данных это принято называть алгеброй.

#### 4.1.1. Общая характеристика

В расширенном варианте реляционной алгебры набор основных алгебраических операций состоит из восьми операций, которые делятся на два класса — теоретико-множественные операции и специальные реляционные операции. В состав теоретико-множественных операций входят следующие:

- объединения отношений (**UNION**);
- пересечения отношений (**INTERSECT**);
- вычитания отношений (**MINUS**);
- взятия декартова произведения отношений (**TIMES**).

Специальные реляционные операции объединяют:

- ограничение отношения (**WHERE**);
- проекцию отношения (**PROJECT**);
- соединение отношений (**TIMES**);
- деление отношений (**DIVIDE BY**).

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

Поскольку результатом любой реляционной операции (кроме операции присваивания, которая не вырабатывает значения)

| Операция  | Приоритет |
|-----------|-----------|
| RENAME    | 4         |
| WHERE     | 3         |
| PROJECT   | 3         |
| TIMES     | 2         |
| JOIN      | 2         |
| INTERSECT | 2         |
| DIVIDE BY | 2         |
| UNION     | 1         |
| MINUS     | 1         |

Рис. 4.1. Таблица приоритетов операций традиционной реляционной алгебры

является некое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение. В построении реляционного выражения могут участвовать все реляционные операции, кроме операции присваивания. Вычисление выражения производится слева направо с учетом приоритетов операций и скобок. Приоритеты операций показаны на рис. 4.1.

#### **4.1.2. Замкнутость реляционной алгебры и операция переименования**

Как отмечалось в предыдущих главах, каждое значение-отношение характеризуется заголовком и телом (или множеством кортежей). Поэтому, если действительно требуется алгебра, операции которой замкнуты относительно понятия отношения, то каждая операция должна производить отношение в полном смысле, т. е. оно должно обладать и телом, и заголовком. Только в этом случае можно будет строить вложенные выражения.

Заголовок отношения представляет собой множество пар `<имя_атрибута, имя_типа_или_домена>`. Если посмотреть на общий обзор реляционных операций, приведенный в начале этого подраздела, то видно, что домены или типы атрибутов результирующего отношения однозначно определяются доменами или типами атрибутов отношений-операндов. Однако с именами атрибутов результата не всегда все так просто.

Например, представим себе, что у отношений-операндов операции декартова произведения имеются одноименные атрибуты с одинаковыми доменами. Каким был бы заголовок результирующего отношения? Поскольку это множество, в нем не должны содержаться одинаковые элементы. Но и потерять атрибут в результате недопустимо. А это значит, что в таком случае вообще невозможно корректно выполнить операцию декартова произведения.

Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для разрешения проблем в число операций реляционной алгебры вводится операция переименования. Ее следует применять в том случае, когда возникает конфликт именования атрибутов в отношениях-операндах одной реляционной операции. Тогда к одному из операндов сначала применяется операция переименования, а затем основная операция выполняется уже без всяких проблем. Более строго мы определим

операцию переименования в следующем подразделе, а пока лишь заметим, что результатом этой операции является отношение, совпадающее во всем с отношением-операндом, кроме того, что имя указанного атрибута изменено на заданное имя.

В дальнейшем изложении будем предполагать применение операции переименования во всех конфликтных ситуациях.

### **4.1.3. Особенности теоретико-множественных операций реляционной алгебры**

Хотя в основе теоретико-множественной части реляционной алгебры Кодда лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями.

**Операции объединения, пересечения, взятия разности. Совместимость по объединению.** Начнем с операции *объединения отношений* (все, что будет представлено по поводу объединения, верно и для операций пересечения и вычитания отношений). Смысл операции объединения в реляционной алгебре в целом остается теоретико-множественным. Но если в теории множеств операция объединения осмысленна для любых двух множеств-операндов, то в случае реляционной алгебры результатом операции объединения должно являться отношение. Если в реляционной алгебре допустить возможность теоретико-множественного объединения двух произвольных отношений (с разными заголовками), то, конечно, результатом операции будет множество, но это множество будет содержать разнотипные кортежи, т. е. не будет отношением. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной.

Эти соображения приводят к понятию *совместимости отношений по объединению*: *два отношения совместимы по объединению* (т. е. для них осмысленна операция объединения отношений) *в том и только том случае, когда обладают одинаковыми заголовками*. В развернутой форме это означает, что в заголовках обоих отношений содержится один и тот же набор имен атрибутов, и одноименные атрибуты определены на одном и том же домене или типе (эта развернутая формулировка, вообще говоря, является излишней, но она пригодится нам в следующем абзаце).

Если два отношения совместимы по объединению, то при выполнении над ними обычных операций объединения, пере-

сечения и вычитания результатом операции является отношение с корректно определенным заголовком, совпадающим с заголовком каждого из отношений-операндов. Напомним, что если два отношения «почти» совместимы по объединению, т. е. совместимы во всем, кроме имен атрибутов, то до выполнения операции типа объединения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования.

Для иллюстрации операций объединения, пересечения и взятия разности предположим, что в базе данных имеются два отношения СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 и СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 с одинаковыми заголовками {СЛУ\_НОМЕР, СЛУ\_ИМЯ, СЛУ\_ЗАРП, СЛУ\_ОТД\_НОМЕР} (имена доменов опущены по причине очевидности). Каждое из отношений содержит данные о служащих, участвующих в соответствующем проекте. На рис. 4.2 показаны возможные значения каждого из двух отношений (некоторые служащие участвуют в обоих проектах).

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22400.00 | 625           |
| 4435      | Петров  | 29600.00 | 625           |
| 4415      | Сидоров | 23000.00 | 636           |
| 4436      | Федоров | 20000.00 | 625           |
| 4440      | Иванова | 22000.00 | 640           |

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|-----------|----------|---------------|
| 4434      | Иванов    | 22400.00 | 625           |
| 4435      | Петров    | 29600.00 | 625           |
| 4441      | Сидоренко | 18000.00 | 640           |
| 4416      | Федоренко | 20000.00 | 636           |
| 4417      | Иваненко  | 22000.00 | 636           |

Рис. 4.2. Возможные значения отношений СЛУЖАЩИЕ \_В\_ПРОЕКТЕ\_1 и СЛУЖАЩИЕ \_В\_ПРОЕКТЕ\_2

Тогда выполнение операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 UNION СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 позволит получить информацию обо всех служащих, участвующих в обоих проектах. Выполнение операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 INTERSECT СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 позволит получить данные о служащих, которые одновременно участвуют в двух проектах. Наконец, операция СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 MINUS СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 выработает отношение, содержащее кортежи служащих, которые участвуют только в первом проекте. Результаты этих операций показаны на рис. 4.3.

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 UNION СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|-----------|----------|---------------|
| 4434      | Иванов    | 22400.00 | 625           |
| 4435      | Петров    | 29600.00 | 625           |
| 4441      | Сидоренко | 18000.00 | 640           |
| 4416      | Федоренко | 20000.00 | 636           |
| 4417      | Иваненко  | 22000.00 | 636           |
| 4415      | Сидоров   | 23000.00 | 636           |
| 4436      | Федоров   | 20000.00 | 625           |
| 4440      | Иванова   | 22000.00 | 640           |

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 INTERSECT СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22400.00 | 625           |
| 4435      | Петров  | 29600.00 | 625           |

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 MINUS СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4415      | Сидоров | 23000.00 | 636           |
| 4436      | Федоров | 20000.00 | 625           |
| 4440      | Иванова | 22000.00 | 640           |

Рис. 4.3. Результаты выполнения операций UNION, INTERSECT и MINUS

Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является, очевидно, избыточным, поскольку, например, операция пересечения выражается через операцию вычитания\*. Тем не менее, Кодд в свое время решил включить все три операции, исходя из интуитивных потребностей далекого от математики потенциального пользователя системы реляционных БД.

**Операция расширенного декартова произведения и совместимость отношений относительно этой операции.** Другие проблемы связаны с операцией взятия декартова произведения двух отношений. В теории множеств *декартово произведение* может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Если говорить более точно, декартовым произведением множеств  $S_1 \{s_1\}$  и  $S_2 \{s_2\}$  является такое множество пар  $S \{<s_1, s_2>\}$ , что  $<s_1, s_2> \in S$  в том и только том случае, когда  $s_1 \in S_1$  и  $s_2 \in S_2$ .

Поскольку отношения являются множествами, для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением! Элементами результата будут не кортежи, а пары кортежей.

Поэтому в реляционной алгебре используется специализированная форма операции взятия декартова произведения — *расширенное декартово произведение* отношений. При выполнении операции взятия расширенного декартова произведения двух отношений элементом результирующего отношения является кортеж, который представляет собой объединение одного кортежа первого отношения и одного кортежа второго отношения.

Приведем более точное определение операции расширенного декартова произведения. Пусть имеются два отношения  $R_1$  с заголовком  $H_{R1} = \{X_1, X_2, \dots, X_n\}$  и  $R_2$  с заголовком  $H_{R2} = \{Y_1, Y_2, \dots, Y_m\}$ \*\*. Тогда результатом операции  $R_1$  TIMES  $R_2$  является отношение  $R$  с заголовком  $H_R = \{X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m\}$ , такое что кортеж  $\{<X_1, v_{x1}>, <X_2, v_{x2}>, \dots, <X_n, v_{xn}>, <Y_1, v_{y1}>, <Y_2, v_{y2}>, \dots, <Y_m, v_{ym}>\} \in B_R$  в том и только том случае, когда  $\{<X_1, v_{x1}>, <X_2, v_{x2}>, \dots, <X_n, v_{xn}>\} \in B_{R1}$  и  $\{<Y_1, v_{y1}>, <Y_2, v_{y2}>, \dots, <Y_m, v_{ym}>\} \in B_{R2}$ .

Но теперь возникает вторая проблема — как получить корректно сформированный заголовок отношения-результата? Поскольку схема результирующего отношения является объедине-

---

\* Легко убедиться, что  $A \text{ INTERSECT } B = A \text{ MINUS } (A \text{ MINUS } B) = B \text{ MINUS } (B \text{ MINUS } A)$ .

\*\* Домены атрибутов здесь не существенны.



ПРОЕКТЫ

|             |            |
|-------------|------------|
| ПРОЕКТ_НАЗВ | ПРОЕКТ_РУК |
| ПРОЕКТ 1    | Иванов     |
| ПРОЕКТ 2    | Иваненко   |

СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 TIMES ПРОЕКТЫ

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР | ПРОЕКТ_НАЗВ | ПРОЕКТ_РУК |
|-----------|---------|----------|---------------|-------------|------------|
| 4434      | Иванов  | 22400.00 | 625           | ПРОЕКТ 1    | Иванов     |
| 4435      | Петров  | 29600.00 | 625           | ПРОЕКТ 1    | Иванов     |
| 4415      | Сидоров | 23000.00 | 636           | ПРОЕКТ 1    | Иванов     |
| 4436      | Федоров | 20000.00 | 625           | ПРОЕКТ 1    | Иванов     |
| 4440      | Иванова | 22000.00 | 640           | ПРОЕКТ 1    | Иванов     |
| 4434      | Иванов  | 22400.00 | 625           | ПРОЕКТ 2    | Иваненко   |
| 4435      | Петров  | 29600.00 | 625           | ПРОЕКТ 2    | Иваненко   |
| 4415      | Сидоров | 23000.00 | 636           | ПРОЕКТ 2    | Иваненко   |
| 4436      | Федоров | 20000.00 | 625           | ПРОЕКТ 2    | Иваненко   |
| 4440      | Иванова | 22000.00 | 640           | ПРОЕКТ 2    | Иваненко   |

Рис. 4.4. Отношение ПРОЕКТЫ и результат операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 TIMES ПРОЕКТЫ

нием схем отношений-операндов, то очевидной проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами.

Это соображение приводит к введению понятия *совместимости по взятию расширенного декартова произведения*: два отношения совместимы по взятию расширенного декартова произведения в том и только том случае, если пересечение множеств имен атрибутов, взятых из их заголовков отношений, пусто. Любые два отношения всегда можно сделать совместимыми по взятию декартова произведения, если применить операцию переименования к одному из этих отношений.

Для примера предположим, что в придачу к введенным ранее отношениям `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1` и `СЛУЖАЩИЕ_В_ПРОЕКТЕ_2` в базе данных содержится еще и отношение `ПРОЕКТЫ` с заголовком `{ПРОЕКТ_НАЗВ, ПРОЕКТ_РУК}` (имена доменов снова опущены) и телом, показанным на рис. 4.4. На этом же рисунке показан результат операции `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 TIMES ПРОЕКТЫ`.

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

По поводу теоретико-множественных операций реляционной алгебры следует также заметить, что все четыре операции являются ассоциативными, т.е. если обозначить через `OP` любую из четырех операций, то  $(R_1 \text{ OP } R_2) \text{ OP } R_3 = R_1 \text{ OP } (R_2 \text{ OP } R_3)$ , и, следовательно, без внесения двусмысленности можно записать:  $R_1 \text{ OP } R_2 \text{ OP } R_3$  ( $R_1$ ,  $R_2$  и  $R_3$  — отношения, обладающие свойствами, необходимыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т.е.  $R_1 \text{ OP } R_2 = R_2 \text{ OP } R_1$ .

#### 4.1.4. Специальные реляционные операции

В этом подразделе рассмотрим специальные реляционные операции реляционной алгебры, такие как ограничение, проекция, соединение и деление.

**Операция ограничения.** Операция ограничения `WHERE` требует наличия двух операндов: ограничиваемого отношения

и простого условия ограничения. Простое условие ограничения может иметь один из следующих видов:

- $(X_1 \text{ comp-op } X_2)$ , где  $X_1$  и  $X_2$  — имена атрибутов ограничиваемого отношения; атрибуты  $X_1$  и  $X_2$  должны быть определены на одном и том же домене, для значений базового типа данных которого поддерживается операция сравнения `comp_op`, или на базовых типах данных, над значениями которых можно выполнять эту операцию сравнения;
- $(X \text{ comp-op } \text{const})$ , где  $X$  — имя атрибута ограничиваемого отношения, а `const` — литерально заданная константа того же домена или типа данных; атрибут  $X$  должен быть определен на домене или базовом типе, для значений которого поддерживается операция сравнения `comp_op`.

Операцией сравнения `comp-op` могут быть «=», «≠», «>», «≥», «<», «≤». Простые условия вычисляются в трехзначной логике (см. подразд. 3.3.2), и в результате выполнения операции ограничения производится отношение, заголовок которого совпадает с заголовком отношения-операнда, а в тело входят те кортежи отношения-операнда, для которых значением условия ограничения является *true*. Тем самым, если в некоторых кортежах содержатся неопределенные значения и по данной причине вычисление простого условия дает значение *unknown*, то эти кортежи не войдут в результирующее отношение.

Для обозначения вызова операции ограничения будем использовать конструкцию *R WHERE comp*, где *R* — ограничиваемое отношение, а *comp* — простое условие сравнения. Пусть *comp1* и *comp2* — два простых условия ограничения. Тогда по определению:

- *R WHERE (comp1 AND comp2)* обозначает то же самое, что и *(R WHERE comp1) INTERSECT (R WHERE comp2)*;
- *R WHERE (comp1 OR comp2)* обозначает то же самое, что и *(R WHERE comp1) UNION (R WHERE comp2)*;
- *R WHERE NOT comp1* обозначает то же самое, что и *R MINUS (R WHERE comp1)*.

Эти соглашения позволяют использовать операции ограничения, в которых условием ограничения является произвольное булевское выражение, составленное из простых условий с использованием логических связок AND, OR, NOT и скобок.

Результат выполнения операции `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 WHERE (СЛУ_ЗАРП > 20000.00 AND (СЛУ_ОТД_НОМ = 625 OR СЛУ_ОТД_НОМ = 640))` (получить данные из отношения `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1` о служащих, работающих в отделах 310 и 315 и получающих заработную плату, превышающую 20000.00 руб.) показан на рис. 4.5.

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22400.00 | 625           |
| 4435      | Петров  | 29600.00 | 625           |
| 4440      | Иванова | 22000.00 | 640           |

Рис. 4.5. Результат операции `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 WHERE (СЛУ_ЗАРП > 20000.00 AND (СЛУ_ОТД_НОМ = 625 OR СЛУ_ОТД_НОМ = 640))`

На интуитивном уровне операцию ограничения лучше всего представлять как взятие некоторой «горизонтальной» вырезки из отношения-операнда (выборки некоторых строк из таблицы).

**Операция взятия проекции.** Операция взятия проекции также требует наличия двух операндов — проецируемого отношения  $R$  и подмножества множества имен атрибутов, входящих в заголовок отношения  $R$ . Результатом операции `PROJECT R {X1, X2, ..., Xn}` (проекции отношения  $R$  на множество атрибутов  $\{X_1, X_2, \dots, X_n\}$ ;  $\{X_1, X_2, \dots, X_n\} \subset H_R$ ) является отношение, заголовок которого определяется множеством атрибутов  $\{X_1, X_2, \dots, X_n\}$  и в тело которого входит кортеж  $\langle X_1, T_1, v_1 \rangle, \langle X_2, T_2, v_2 \rangle, \dots, \langle X_n, T_n, v_n \rangle$  в том и только том случае, когда в теле отношения  $R$  имеется хотя бы один кортеж, атрибут  $X_1$  которого имеет значение  $v_1$ , атрибут  $X_2$  — значение  $v_2$ , ..., атрибут  $X_n$  — значение  $v_n$ . Тем самым, при выполнении операции проекции выделяется «вертикальная» вырезка отношения-операнда с естественным уничтожением потенциально возникающих кортежей-дубликатов.

Заметим, что потенциальная потребность удаления дубликатов очень сильно усложняет реализацию операции проекции, поскольку в общем случае для удаления дубликатов требуется сортировка промежуточного результата операции. Основная сложность состоит в том, что этот промежуточный результат в общем случае может быть очень большим, и для сортировки требуется применять дорогостоящие алгоритмы *внешней сортировки*, выполняемые с применением обменов с внешней памятью. (Под «стоимостью» действия понимается время его выполнения.)

Результат операции `PROJECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 {СЛУ_ОТД_НОМ}` (в каких отделах работают служащие, данные о которых содержатся в отношении `СЛУЖАЩИЕ_В_ПРОЕКТЕ_1`?) показан на рис. 4.6.

| СЛУ_ОТД_НОМЕР |
|---------------|
| 625           |
| 636           |
| 640           |

Рис. 4.6. Результат операции `PROJECT СЛУЖАЩИЕ_В_ПРОЕКТЕ_1 {СЛУ_ОТД_НОМ}`

**Операция соединения отношений.** Общая операция соединения ( $\Theta$ -JOIN, соединение по условию  $\Theta$ ) требует наличия двух операндов — соединяемых отношений и третьего операнда — простого условия. Пусть соединяются отношения  $R_1$  и  $R_2$ . Как и в случае операции ограничения, условие соединения *comp* имеет вид либо  $X \text{ comp-оп } Y$ , либо  $X \text{ comp-оп } \textit{const}$ , где  $X$  и  $Y$  — имена атрибутов отношений  $R_1$  и/или  $R_2$ , определенные на одном и том же домене или базовом типе, *const* — литерально заданная константа, и *comp-оп* — допустимая в данном контексте операция сравнения.

Тогда по определению результатом операции соединения  $R_1 \text{ JOIN } R_2 \text{ WHERE } \textit{comp}$  отношений  $R_1$  и  $R_2$ , совместимых по взятию расширенного декартова произведения, является отношение, получаемое путем выполнения операции ограничения по условию *comp* расширенного декартова произведения отношений  $R_1$  и  $R_2$  ( $A \text{ JOIN } B \text{ WHERE } \textit{comp} \equiv (A \text{ TIMES } B) \text{ WHERE } \textit{comp}$ ).

Если тщательно осмыслить это определение, то станет ясно, что в общем случае применение условия соединения существенно уменьшит мощность результата промежуточного декартова произведения отношений-операндов только в том случае, если условие соединения имеет вид  $X \text{ comp-оп } Y$ , где  $X$  и  $Y$  — имена атрибутов разных отношений-операндов. Поэтому на практике обычно считают реальными операциями соединения именно те операции, которые основываются на условии соединения приведенного вида.

В начале подраздела (см. операции ограничения) была определена трактовка использования в качестве ограничивающего условия произвольного булевского выражения, которое составлено из простых условий над атрибутами отношения-операнда и литеральными константами. Конечно же, и в операции соединения может задаваться произвольное логическое выражение, составленное из простых условий над атрибутами отношений-операндов и константами. Операцию соединения с та-

ким условием *comp* разумно считать операцией *действительно* соединения, если оно имеет вид (или может быть преобразовано к виду)  $comp1 \text{ AND } (R_1 \text{ comp-ор } R_2)$ , где  $R_1$  и  $R_2$  — имена атрибутов разных отношений-операндов.

Для иллюстрации операций соединения немного изменим заголовки и тела отношений, которые использовались ранее в примерах этой главы. Пусть теперь имеются отношения **СЛУЖАЩИЕ** с заголовком {СЛУ\_НОМЕР, СЛУ\_ИМЯ, СЛУ\_ЗАРП, ПРО\_НОМ} (атрибут ПРО\_НОМ содержит номера проектов, в которых участвует каждый служащий) и **ПРОЕКТЫ** с заголовком {ПРО\_НОМ, ПРОЕКТ\_РУК, ПРО\_ЗАРП} (ПРО\_НОМ — номер проекта, ПРОЕКТ\_РУК — имя служащего-руководителя проекта, ПРО\_ЗАРП — средняя заработная плата служащих, участвующих в проекте). Возможные значения отношений **СЛУЖАЩИЕ** и **ПРОЕКТЫ** показаны на рис. 4.7.

#### СЛУЖАЩИЕ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 1       |
| 4435      | Петров    | 29600.00 | 1       |
| 4415      | Сидоров   | 23000.00 | 1       |
| 4436      | Федоров   | 20000.00 | 1       |
| 4440      | Иванова   | 22000.00 | 1       |
| 4434      | Иванов    | 22400.00 | 2       |
| 4435      | Петров    | 29600.00 | 2       |
| 4441      | Сидоренко | 18000.00 | 2       |
| 4416      | Федоренко | 20000.00 | 2       |
| 4417      | Иваненко  | 22000.00 | 2       |

#### ПРОЕКТЫ

| ПРО_НОМ | ПРОЕКТ_РУК | ПРО_ЗАРП |
|---------|------------|----------|
| 1       | Иванов     | 23400.00 |
| 2       | Иваненко   | 22400.00 |

Рис. 4.7. Возможные значения отношений **СЛУЖАЩИЕ** и **ПРОЕКТЫ**

Тогда осмысленной операцией соединения общего вида будет `СЛУЖАЩИЕ JOIN (RENAME ПРОЕКТЫ (ПРО_НОМ, ПРО_НОМ1)) WHERE (СЛУ_ЗАРП > ПРО_ЗАРП)` (выдать данные о служащих, получающих заработную плату, превышающую среднюю заработную плату какого-либо проекта). Результаты этого запроса показаны на рис. 4.8.

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

Существует важный частный случай соединения — *эквисоединение* (`EQUIJOIN`) и простое, но важное расширение операции эквисоединения — *естественное соединение* (`NATURAL JOIN`). Операция соединения называется *операцией эквисоединения*, если условие соединения имеет вид  $(X = Y)$ , где  $X$  и  $Y$  — атрибуты разных операндов соединения. Этот случай важен потому, что он чаще всего встречается на практике и для него существуют наиболее эффективные алгоритмы реализации.

Операция естественного соединения применяется к паре отношений  $R_1$  и  $R_2$ , обладающих (возможно, составным\*) общим атрибутом  $Z$ . Результатом этой операции по определению является спроецированный на  $H_{R_1} \cup H_{R_2}$  результат эквисоединения  $R_1$  и  $R_2$  по условию  $R_1.Z = R_2.Z^{**}$  ( $R_1 \text{ NATURAL JOIN } R_2 \equiv \text{PROJECT}(H_{R_1} \cup H_{R_2})(R_1 \text{ JOIN } R_2 \text{ WHERE } R_1.Z = R_2.Z)$ ). Хотя операция естественного соединения выражается через операции переименования, соединения общего вида и проекции, для нее обычно используется сокращенная форма, называемая `NATURAL JOIN`.

На рис. 4.9 приведены результаты операций `СЛУЖАЩИЕ JOIN (ПРОЕКТЫ RENAME (ПРО_НОМ, ПРО_НОМ1)) WHERE (СЛУ_ЗАРП = ПРО_ЗАРП)` (эквисоединение отношений `СЛУЖАЩИЕ` и `ПРОЕКТЫ`:

---

\* Здесь и далее под *составным атрибутом*  $Z$  отношения  $R$  понимается произвольное подмножество  $H_R$ . При определении естественного соединения в алгебре Кодда подразумевается, что  $Z \neq \emptyset$ . В следующем подразделе это ограничение будет снято.

\*\* Здесь  $R_1.Z$  и  $R_2.Z$  представляют собой так называемые квалифицированные (уточненные) имена атрибутов (часто такой способ именования называют точечной нотацией). Мы будем использовать подобную нотацию в тех случаях, когда требуется явно показать, заголовку какого отношения принадлежит данный атрибут.

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | ПРО_НОМ | ПРО_НОМ1 | ПРОЕКТ_РУК | ПРО_ЗАРП |
|-----------|---------|----------|---------|----------|------------|----------|
| 4435      | Петров  | 29600.00 | 1       | 1        | Иванов     | 23400.00 |
| 4435      | Петров  | 29600.00 | 1       | 2        | Иваненко   | 22400.00 |
| 4415      | Сидоров | 23000.00 | 1       | 2        | Иваненко   | 22400.00 |

Рис. 4.8. Результат операции СЛУЖАЩИЕ JOIN RENAME (ПРОЕКТЫ (ПРО\_НОМ, ПРО\_НОМ1)) WHERE (СЛУ\_ЗАРП > ПРО\_ЗАРП)

СЛУЖАЩИЕ JOIN (ПРОЕКТЫ RENAME (ПРО\_НОМ, ПРО\_НОМ1)) WHERE (СЛУ\_ЗАРП = ПРО\_ЗАРП)

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | ПРО_НОМ | ПРО_НОМ1 | ПРОЕКТ_РУК | ПРО_ЗАРП |
|-----------|---------|----------|---------|----------|------------|----------|
| 2934      | Иванов  | 22400.00 | 1       | 2        | Иваненко   | 22400.00 |
| 2934      | Иванов  | 22400.00 | 2       | 2        | Иваненко   | 22400.00 |

СЛУЖАЩИЕ NATURAL JOIN ПРОЕКТЫ

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК | ПРО_ЗАРП |
|-----------|---------|----------|---------|------------|----------|
| 4434      | Иванов  | 22400.00 | 1       | Иванов     | 23400.00 |
| 4435      | Петров  | 29600.00 | 1       | Иванов     | 23400.00 |
| 4415      | Сидоров | 23000.00 | 1       | Иванов     | 23400.00 |
| 4436      | Федоров | 20000.00 | 1       | Иванов     | 23400.00 |
| 4440      | Иванова | 22000.00 | 1       | Иванов     | 23400.00 |
| 4434      | Иванов  | 22400.00 | 1       | Иваненко   | 22400.00 |
| 4435      | Петров  | 29600.00 | 1       | Иваненко   | 22400.00 |
| 4415      | Сидоров | 23000.00 | 1       | Иваненко   | 22400.00 |
| 4436      | Федоров | 20000.00 | 1       | Иваненко   | 22400.00 |
| 4440      | Иванова | 22000.00 | 1       | Иваненко   | 22400.00 |

Рис. 4.9. Результаты операций эквисоединения и естественного соединения отношений СЛУЖАЩИЕ и ПРОЕКТЫ



найти всех служащих, получающих заработную плату, равную средней заработной плате в каком-либо проекте) и **СЛУЖАЩИЕ NATURAL JOIN ПРОЕКТЫ** (естественное соединение — выдать полную информацию о служащих и проектах, в которых они участвуют).

Результат операции **СЛУЖАЩИЕ JOIN (ПРОЕКТЫ RENAME (ПРО\_НОМ, ПРО\_НОМ1)) WHERE (СЛУ\_ЗАРП = ПРО\_ЗАРП)**

Если вспомнить введенное в предыдущих главах определение внешнего ключа отношения, то должно стать понятно, что основной смысл операции естественного соединения состоит в возможности восстановления сложной сущности, декомпозированной по причине требования первой нормальной формы. Операция естественного соединения не включается в состав набора операций данной реляционной алгебры Кодда, но имеет очень важное практическое значение.

**Операция деления отношений.** Пусть заданы два отношения —  $R_1$  с заголовком  $\{ \langle X_1, T_{X1} \rangle, \langle X_2, T_{X2} \rangle, \dots, \langle X_n, T_{Xn} \rangle, \langle Y_1, T_{Y1} \rangle, \langle Y_2, T_{Y2} \rangle, \dots, \langle Y_n, T_{Yn} \rangle \}$  и  $R_2$  с заголовком  $\{ \langle Y_1, T_{Y1} \rangle, \langle Y_2, T_{Y2} \rangle, \dots, \langle Y_n, T_{Yn} \rangle \}$ . Назовем множество атрибутов  $\{ \langle X_i, T_{Xi} \rangle \}$  составным атрибутом  $X$ , а множество атрибутов  $\{ \langle Y_j, D_{Yj} \rangle \}$  — составным атрибутом  $Y$ . После этого будем говорить о реляционном делении «бинарного» отношения  $R_1$  с заголовком  $X \text{ UNION } Y$  на «унарное» отношение  $R_2$  с заголовком  $Y$ .

По определению результатом деления  $R_1$  на  $R_2$  ( $R_1 \text{ DIVIDE BY } R_2$ ) является «унарное» отношение  $R$  с заголовком  $Y$ , тело которого состоит из тех и только тех кортежей  $t_Y$ , что в теле отношения  $R_1$  содержатся кортежи  $t_Y \text{ UNION } t_X$  такие, что множество  $\{t_X\}$  включает  $B_{R1}$ . Операция реляционного деления не является примитивной и выражается через операции декартова произведения, взятия разности и проекции (см. подразд. 4.2).

Для иллюстрации этой операции предположим, что в базе данных служащих поддерживаются следующие отношения: **СЛУЖАЩИЕ** с возможным значением, показанным на рис. 4.7, и унарное отношение **НОМЕРА\_ПРОЕКТОВ {ПРО\_НОМ}** (его возможное значение показано на рис. 4.10). Тогда запрос **СЛУЖАЩИЕ DIVIDE BY НОМЕРА\_ПРОЕКТОВ** выдаст данные обо всех служащих, участвующих во всех проектах (результат операции приведен также на рис. 4.10).

**Заключительные замечания.** В завершение подраздела отметим несколько моментов. Прежде всего, заметим, что алгебра Кодда была здесь представлена не в ее оригинальной форме, а с некоторыми существенными коррективами, внесенными в свое время К. Дейтом. С моей точки зрения, одной из наи-

## НОМЕРА\_ПРОЕКТОВ

| ПРО_НОМ |
|---------|
| 1       |
| 2       |

Результат операции СЛУЖАЩИЕ DIVIDE BY НОМЕРА\_ПРОЕКТОВ

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП |
|-----------|---------|----------|
| 4434      | Иванов  | 22400.00 |
| 4435      | Петров  | 29600.00 |

Рис. 4.10. Пример реляционного деления

более значительных корректив было добавление тривиальной, на первый взгляд, операции переименования атрибутов. Когда Э. Кодд в начале 1970-х гг. впервые опубликовал свою алгебру, основное внимание в ней уделялось тому, как конструируются результирующие множества кортежей, т. е. что представляют собой тела результатов операций. Гораздо меньше внимания уделялось заголовкам отношений-результатов. Фактически Кодд пытался применить для именования атрибутов результатов операций точечную нотацию, используя для уточнения имен атрибутов имена исходных отношений-операндов. При наличии произвольно сложных и длинных алгебраических выражений этот путь, в лучшем случае, вел к порождению длинных и трудных для восприятия имен. Очевидно, что введение операции переименования атрибутов позволяет легко справиться с этой проблемой.

Далее, алгебра Кодда исключительно избыточна. Операции пересечения, декартова произведения и естественного соединения, на самом деле, являются частными случаями одной более общей операции, о которой пойдет речь в следующем подразделе. Введение операции декартова произведения в качестве базовой операции алгебры может ввести в заблуждение неопытных студентов и читателей, не осознающих практическую бессмысленность этой операции.

Почему же мы начали обсуждение базовых манипуляционных механизмов реляционной модели данных с этой небезупречной и несколько устаревшей алгебры? Конечно, прежде всего, из уважения к заслугам доктора Эдгара Кодда, вклад которого

в современную технологию баз данных невозможно переоценить. Практические соображения, повлиявшие на наше решение начать обсуждение с алгебры Кодда, заключались в том, что семантика языка SQL во многом базируется именно на этой алгебре и проще изучать SQL, предварительно познакомившись с ней.

## 4.2. Реляционная алгебра А К. Дейта и Х. Дарвена

Рассматриваемая в подразд. 4.1 алгебра Кодда в большей степени основывается на теории множеств. Базовыми операциями являются переименование атрибутов, объединение, пересечение, вычитание, декартово произведение, проекция и ограничение. Операция соединения общего вида, хотя и включается в алгебру, является вторичной и явно представляется через другие операции. Фундаментальная же в реляционном подходе операция естественного соединения выражается через соединение общего вида и в алгебру не включается. В терминах алгебры Кодда проще всего определяются алгебраические черты языка SQL, в частности общая семантика оператора SELECT.

Базисом предложенной К. Дейтом и Х. Дарвеном Алгебры А [11] являются *операции реляционного отрицания* (дополнения), *реляционной конъюнкции* (или дизъюнкции) и *проекции* (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения, объединения отношений и т. д. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений.

Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда\*. Даже сами авторы Алгебры А, Дейт и Дарвен, в своем учебном языке *Tutorial D* используют не Алгебру А напрямую, а некоторое ее надмножество, в большей степени напоминающее алгебру Кодда.

---

\* В отличие от «алгебры» Кодда, которая, как уже отмечалось, не является алгеброй на множестве отношений в математическом смысле, алгебра А — это «настоящая» алгебра, в которой отсутствуют какие-либо ограничения на операнды.

### 4.2.1. Базовые операции Алгебры А

В этом подразделе материал излагается на несколько более формальном уровне, чем в подразд. 4.1. Используемые понятия определяются так же, как и ранее, но для удобства и обеспечения точности изложения повторим определения.

Пусть  $R$  — значение отношения;  $A$  — имя атрибута отношения  $R$ ;  $T$  — имя соответствующего типа (т.е. типа или домена атрибута  $A$ );  $v$  — значение типа  $T$ . Тогда:

- заголовком  $H_R$  значения отношения  $R$  называется множество атрибутов  $R$ , т.е. упорядоченных пар вида  $\langle A, T \rangle$ . По определению никакие два атрибута в этом множестве не могут содержать одно и то же имя атрибута  $A$ ;
- кортеж  $t_R$ , соответствующий заголовку  $H_R$ , — это множество упорядоченных триплетов вида  $\langle A, T, v \rangle$ , по одному такому триплету для каждого атрибута в  $H_R$ ;
- тело  $B_R$  отношения  $R$  — это множество кортежей  $t_R$ . Заметим, что (в общем случае) могут существовать такие кортежи  $t_R$ , которые соответствуют  $H_R$ , но не входят в  $B_R$ .

Заметим, что заголовок — это множество (упорядоченных пар вида  $\langle A, T \rangle$ ), тело — множество (кортежей  $t_R$ ), кортеж — множество (упорядоченных триплетов вида  $\langle A, T, v \rangle$ ). Элемент заголовка — это атрибут (т.е. упорядоченная пара вида  $\langle A, T \rangle$ ), элемент тела — это кортеж, элемент кортежа — это упорядоченный триплет вида  $\langle A, T, v \rangle$ . Любое подмножество заголовка — это некоторый заголовок, любое подмножество тела — некоторое тело и любое подмножество кортежа — некоторый кортеж.

Определим несколько основных операций Алгебры А (как будет показано далее, некоторые из них избыточны). Каждое из последующих определений состоит из формальной спецификации ограничений (если они имеются), применимых к операндам соответствующей операции; формальной спецификации заголовка результата этой операции; формальной спецификации тела этого результата и неформального обсуждения формальных спецификаций.

Во всех формальных спецификациях *exists* обозначает *квантор существования*; *exists  $t_R$*  — «существует такой  $t_R$ , что». Операции *minus* и *union* являются традиционными теоретико-множественными операциями взятия разности и объединения множеств.

Поскольку некоторые базовые операции Алгебры А имеют названия обычных логических операций, чтобы избежать путаницы, имена реляционных операций окружаются сплошными

стрелками: «NOT», «AND», «OR» и т. д. В исходный базовый набор операций входят операции реляционного дополнения «NOT», удаления атрибута «REMOVE», переименования атрибута «RENAME», реляционной конъюнкции «AND» и реляционной дизъюнкции «OR».

**Операция реляционного дополнения.** Пусть  $S$  обозначает результат операции «NOT»  $R$ . Тогда:

- $H_S = H_R$  (заголовок результата совпадает с заголовком операнда);
- $B_S = \{t_S: \text{exists } t_R (t_R \notin B_R \text{ and } t_S = t_R)\}$  (в тело результата входят все кортежи, соответствующие заголовку и не входящие в тело операнда).

Операция «NOT» производит дополнение  $S$  заданного отношения  $R$ . Заголовком  $S$  является заголовок  $R$ . Тело  $S$  включает все кортежи, соответствующие этому заголовку и не входящие в тело  $R$ .

Видимо следует пояснить, почему реляционный аналог операции логического отрицания называется здесь операцией реляционного дополнения. Во-первых, термин «дополнение» полностью соответствует сути операции «NOT»: тело результата операции «NOT»  $R$  является дополнением  $B_R$  до полного множества кортежей, соответствующих  $H_R$ . Во-вторых, это не противоречит природе булевой операции NOT: у традиционного булевского типа имеются всего два значения — *true* и *false*, и  $\text{NOT } \textit{true} = \textit{false}$ , а  $\text{NOT } \textit{false} = \textit{true}$ . (Кстати, операцию NOT в трехзначной логике (см. гл. 3) уже нельзя считать операцией дополнения.)

Чтобы привести пример использования операции «NOT», предположим, что в состав домена ДОПУСТИМЫЕ\_НОМЕРА\_ПРОЕКТОВ, на котором определен атрибут ПРО\_НОМ отношения НОМЕРА\_ПРОЕКТОВ (рис. 4.11, а), входит всего пять значений {1, 2, 3, 4, 5}. Тогда результат операции «NOT» НОМЕРА\_ПРОЕКТОВ будет таким, как показано на рис. 4.11, б.

| НОМЕРА_ПРОЕКТОВ                                                                | ◀NOT▶ НОМЕРА_ПРОЕКТОВ |   |   |                                                                                                   |         |   |   |   |
|--------------------------------------------------------------------------------|-----------------------|---|---|---------------------------------------------------------------------------------------------------|---------|---|---|---|
| <table><tr><th>ПРО_НОМ</th></tr><tr><td>1</td></tr><tr><td>2</td></tr></table> | ПРО_НОМ               | 1 | 2 | <table><tr><th>ПРО_НОМ</th></tr><tr><td>3</td></tr><tr><td>4</td></tr><tr><td>5</td></tr></table> | ПРО_НОМ | 3 | 4 | 5 |
| ПРО_НОМ                                                                        |                       |   |   |                                                                                                   |         |   |   |   |
| 1                                                                              |                       |   |   |                                                                                                   |         |   |   |   |
| 2                                                                              |                       |   |   |                                                                                                   |         |   |   |   |
| ПРО_НОМ                                                                        |                       |   |   |                                                                                                   |         |   |   |   |
| 3                                                                              |                       |   |   |                                                                                                   |         |   |   |   |
| 4                                                                              |                       |   |   |                                                                                                   |         |   |   |   |
| 5                                                                              |                       |   |   |                                                                                                   |         |   |   |   |
| <i>a</i>                                                                       | <i>б</i>              |   |   |                                                                                                   |         |   |   |   |

Рис. 4.11. Пример операции «NOT» НОМЕРА\_ПРОЕКТОВ

**Операция удаления атрибута.** Пусть  $S$  обозначает результат операции  $R \leftarrow \text{REMOVE} \rightarrow A$ . Для обеспечения возможности выполнения операции требуется, чтобы существовал некоторый тип (или домен)  $T$  такой, что  $\langle A, T \rangle \in H_R$  (т.е. в состав заголовка отношения  $R$  должен входить атрибут  $R$ ). Тогда:

- $H_S = H_R \text{ minus } \{\langle A, T \rangle\}$ , т.е. заголовок результата получается изъятием атрибута  $A$  из заголовка операнда;
- $B_S = \{t_s: \text{exists } t_R \text{ exists } v (t_R \in Br \text{ and } v \in T \text{ and } \langle A, T, v \rangle \in tr \text{ and } t_s = t_R \text{ minus } \{\langle A, T, v \rangle\})\}$ , т.е. в тело результата входят те и только те кортежи операнда, из которых удален триплет атрибута  $A$ .

Операция  $\leftarrow \text{REMOVE} \rightarrow$  производит значение отношения  $S$ , формируемое путем удаления указанного атрибута  $A$  из заданного значения отношения  $R$ . Операция эквивалентна взятию проекции  $R$  на все атрибуты, кроме  $A$ . Заголовок  $S$  получается теоретико-множественным вычитанием из заголовка  $R$  множества из одного элемента  $\{\langle A, T \rangle\}$ . Тело  $S$  состоит из кортежей, соответствующих  $H_S$ , причем каждый из них является подмножеством некоторого кортежа тела отношения  $R$ .

Для примера операции  $\leftarrow \text{REMOVE} \rightarrow$  воспользуемся возможным значением отношения **СЛУЖАЩИЕ**, для удобства повторно показанным на рис. 4.12, а. Результат операции **СЛУЖАЩИЕ**  $\leftarrow \text{REMOVE} \rightarrow \text{ПРО\_НОМ}$  (получить данные о служащих, участвующих в проектах) показан на рис. 4.12, б.

**Операция переименования.** Пусть  $S$  обозначает результат операции  $R \leftarrow \text{RENAME} \rightarrow (A, A')$ . Для обеспечения возможности выполнения операции требуется, чтобы существовал некоторый тип  $T$ , такой, что  $\langle A, T \rangle \in H_R$ , и чтобы не существовал такой тип  $T$ , что  $\langle B, T \rangle \notin H_R$ . (Другими словами, в заголовке отношения  $R$  должен присутствовать атрибут  $A$  и не должен присутствовать атрибут  $B$ .) Тогда:

- $H_S = (H_R \text{ minus } \{\langle A, T \rangle\}) \text{ union } \{\langle A', T \rangle\}$ , т.е. в схеме результата  $A'$  заменяет  $A$ ;
- $B_S = \{t_s: \text{exists } t_R \text{ exists } v (t_R \in Br \text{ and } v \in T \text{ and } \langle A, T, v \rangle \in tr \text{ and } t_s = (t_R \text{ minus } \{\langle A, T, v \rangle\}) \text{ union } \{\langle A', T, v \rangle\})\}$ , т.е. во всех кортежах тела результата вместо триплета  $\langle A, T, v \rangle$  появляется триплет  $\langle A', T, v \rangle$ .

Операция  $\leftarrow \text{RENAME} \rightarrow$  производит отношение  $S$ , которое отличается от заданного отношения  $R$  только именем одного его атрибута, которое изменяется с  $A$  на  $A'$ . Заголовок  $S$  такой же, как заголовок  $R$ , за исключением того, что пара  $\langle A', T \rangle$  заменяет пару  $\langle A, T \rangle$ . Тело  $S$  включает все кортежи тела  $R$ , но в каждом из этих кортежей триплет  $\langle A', T, v \rangle$  заменяет триплет  $\langle A, T, v \rangle$ .

# СЛУЖАЩИЕ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 1       |
| 4435      | Петров    | 29600.00 | 1       |
| 4415      | Сидоров   | 23000.00 | 1       |
| 4436      | Федоров   | 20000.00 | 1       |
| 4440      | Иванова   | 22000.00 | 1       |
| 4434      | Иванов    | 22400.00 | 2       |
| 4435      | Петров    | 29600.00 | 2       |
| 4441      | Сидоренко | 18000.00 | 2       |
| 4416      | Федоренко | 20000.00 | 2       |
| 4417      | Иваненко  | 22000.00 | 2       |

a

## СЛУЖАЩИЕ ◀REMOVE▶ ПРО\_НОМ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП |
|-----------|-----------|----------|
| 4434      | Иванов    | 22400.00 |
| 4435      | Петров    | 29600.00 |
| 4415      | Сидоров   | 23000.00 |
| 4436      | Федоров   | 20000.00 |
| 4440      | Иванова   | 22000.00 |
| 4441      | Сидоренко | 18000.00 |
| 4416      | Федоренко | 20000.00 |
| 4417      | Иваненко  | 22000.00 |

б

Рис. 4.12. Пример операции ◀REMOVE▶

**Операция реляционной конъюнкции.** Пусть  $S$  обозначает результат операции  $R_1 \text{ ◀AND▶ } R_2$ . Для обеспечения возможности выполнения операции требуется, чтобы если  $\langle A, T_1 \rangle \in H_{R_1}$  и  $\langle A, T_2 \rangle \in H_{R_2}$ , то  $T_1 = T_2$ . (Другими словами, если в двух отношениях-операндах имеются одноименные атрибуты, то они

должны быть определены на одном и том же типе (домене.).) Тогда:

- $H_S = H_{R_1} \text{ union } H_{R_2}$ , т.е. заголовок результата получается путем объединения заголовков отношений-операндов, как в операциях TIMES и JOIN алгебры Кодда;
- $B_S = \{t_s: \text{exists } t_{R_1} \text{ exists } t_{R_2} (t_{R_1} \in B_{R_1} \text{ and } t_{R_2} \in B_{R_2} \text{ and } t_s = (t_{R_1} \text{ union } t_{R_2}))\}$ .

Обратите внимание на то, что кортеж результата определяется как объединение кортежей операндов. Поэтому, поскольку степень заголовка, а следовательно, и кортежей результата равняется сумме степеней заголовков, а следовательно, и кортежей операндов, легко убедиться в следующих свойствах:

- если у заголовков отношений-операндов имеется непустое пересечение, то операция **◀AND▶** работает как естественное соединение;
- если пересечение заголовков операндов пусто, то **◀AND▶** работает как расширенное декартово произведение;
- если схемы отношений полностью совпадают, то результатом операции является пересечение двух отношений-операндов.

Операция **◀AND▶** является реляционной *конъюнкцией*, в некоторых случаях выдающей в результате отношение  $S$ , ранее называвшееся естественным соединением двух заданных отношений  $R_1$  и  $R_2$ . Заголовок  $S$  является объединением заголовков  $R_1$  и  $R_2$ . Тело  $S$  включает каждый кортеж, соответствующий заголовку  $S$  и являющийся надмножеством некоторого кортежа из тела  $R_1$  и некоторого кортежа из тела  $R_2$ .

Для иллюстрации этой операции воспользуемся возможными значениями отношений СЛУЖАЩИЕ (рис. 4.12), а также ПРОЕКТЫ, СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 и СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2, которые для удобства воспроизводятся на рис. 4.13.

У отношений СЛУЖАЩИЕ и ПРОЕКТЫ имеется общий атрибут ПРО\_НОМ. Поэтому операция **◀AND▶** работает как операция естественного соединения. Результат операции СЛУЖАЩИЕ **◀AND▶** ПРОЕКТЫ показан на рис. 4.14.

Пересечение заголовков отношений СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 и ПРОЕКТЫ пусто, и поэтому в результате реляционной конъюнкции производится расширенное декартово произведение этих отношений. На рис. 4.15 показан результат операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 **◀AND▶** ПРОЕКТЫ.

Наконец, заголовки отношений СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 и СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 совпадают, и телом операции **◀AND▶** является пересечение тел отношений-операндов. Результат



### ПРОЕКТЫ

| ПРО_НОМ | ПРОЕКТ_РУК | ПРО_ЗАРП |
|---------|------------|----------|
| 1       | Иванов     | 23400.00 |
| 2       | Иваненко   | 22400.00 |

### СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22400.00 | 625           |
| 4435      | Петров  | 29600.00 | 625           |
| 4415      | Сидоров | 23000.00 | 636           |
| 4436      | Федоров | 20000.00 | 625           |
| 4440      | Иванова | 22000.00 | 640           |

### СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|-----------|----------|---------------|
| 4434      | Иванов    | 22400.00 | 625           |
| 4435      | Петров    | 29600.00 | 625           |
| 4441      | Сидоренко | 18000.00 | 640           |
| 4416      | Федоренко | 20000.00 | 636           |
| 4417      | Иваненко  | 22000.00 | 636           |

Рис. 4.13. Возможные значения отношений ПРОЕКТЫ, СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 и СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2

операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 ◀AND▶ СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 показан на рис. 4.16.

**Операция реляционной дизъюнкции.** Пусть  $S$  обозначает результат операции  $R_1$  ◀OR▶  $R_2$ . Для обеспечения возможности выполнения операции требуется, чтобы если  $\langle A, T_1 \rangle \in H_{R_1}$  и  $\langle A, T_2 \rangle \in H_{R_2}$ , то  $T_1 = T_2$  (одноименные атрибуты, то они должны быть определены на одном и том же типе (домене)). Тогда:

- $H_S = H_{R_1} \cup H_{R_2}$  (из схемы результата удаляются атрибуты-дубликаты);
- $B_S = \{t_S: \text{exists } t_{R_1} \text{ exists } t_{R_2} ((t_{R_1} \in B_{R_1} \text{ or } t_{R_2} \in B_{R_2}) \text{ and } t_S = (t_{R_1} \cup t_{R_2}))\}$ .

# СЛУЖАЩИЕ «AND» ПРОЕКТЫ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК |
|-----------|-----------|----------|---------|------------|
| 4434      | Иванов    | 22400.00 | 1       | Иванов     |
| 4435      | Петров    | 29600.00 | 1       | Иванов     |
| 4415      | Сидоров   | 23000.00 | 1       | Иванов     |
| 4436      | Федоров   | 20000.00 | 1       | Иванов     |
| 4440      | Иванова   | 22000.00 | 1       | Иванов     |
| 4434      | Иванов    | 22400.00 | 2       | Иваненко   |
| 4435      | Петров    | 29600.00 | 2       | Иваненко   |
| 4441      | Сидоренко | 18000.00 | 2       | Иваненко   |
| 4416      | Федоренко | 20000.00 | 2       | Иваненко   |
| 4417      | Иваненко  | 22000.00 | 2       | Иваненко   |

Рис. 4.14. Пример операции «AND» как естественного соединения

# СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 «AND» ПРОЕКТЫ

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР | ПРО_НОМ | ПРОЕКТ_РУК |
|-----------|---------|----------|---------------|---------|------------|
| 4434      | Иванов  | 22400.00 | 625           | 1       | Иванов     |
| 4435      | Петров  | 29600.00 | 625           | 1       | Иванов     |
| 4415      | Сидоров | 23000.00 | 636           | 1       | Иванов     |
| 4436      | Федоров | 20000.00 | 625           | 1       | Иванов     |
| 4440      | Иванова | 22000.00 | 640           | 1       | Иванов     |
| 4434      | Иванов  | 22400.00 | 625           | 2       | Иваненко   |
| 4435      | Петров  | 29600.00 | 625           | 2       | Иваненко   |
| 4415      | Сидоров | 23000.00 | 640           | 2       | Иваненко   |
| 4436      | Федоров | 20000.00 | 636           | 2       | Иваненко   |
| 4440      | Иванова | 22000.00 | 636           | 2       | Иваненко   |

Рис. 4.15. Пример операции «AND» как расширенного декартова произведения

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | СЛУ_ОТД_НОМЕР |
|-----------|---------|----------|---------------|
| 4434      | Иванов  | 22400.00 | 625           |
| 4435      | Петров  | 29600.00 | 625           |

Рис. 4.16. Пример операции ◀AND▶ как пересечения

Как видно, определение операции ◀OR▶ отличается от определения операции ◀AND▶ тем, что для вхождения в результат операции объединения кортежей, соответствующих заголовкам операндов, в случае ◀OR▶ требуется, чтобы хотя бы один из этих кортежей принадлежал телу операнда (в случае операции ◀AND▶ требовалось, чтобы оба эти кортежа принадлежали телам операндов). Очевидно, что при этом:

- если у операндов нет общих атрибутов, то в тело результирующего отношения входят все такие кортежи  $t_s$ , которые являются объединением кортежей  $t_{R1}$  и  $t_{R2}$ , соответствующих заголовкам отношений-операндов, и хотя бы один из этих кортежей принадлежит телу одного из операндов;
- если у операндов имеются общие атрибуты, то в тело результирующего отношения входят все такие кортежи  $t_s$ , которые являются объединением кортежей  $t_{R1}$  и  $t_{R2}$ , соответствующих заголовкам отношений-операндов, если хотя бы один из этих кортежей принадлежит телу одного из операндов, и значения общих атрибутов  $t_{R1}$  и  $t_{R2}$  совпадают;
- если же заголовки отношений-операндов совпадают, то тело отношения-результата является объединением тел операндов.

Операция ◀OR▶ является реляционной *дизъюнкцией* и обобщением того, что ранее называлось объединением. Заголовок  $S$  есть объединение заголовков  $t_{R1}$  и  $t_{R2}$ . Тело  $S$  состоит из всех кортежей, соответствующих заголовку  $S$  и являющихся надмножеством *либо* некоторого кортежа из тела  $t_{R1}$ , *либо* некоторого кортежа из тела  $t_{R2}$ .

Предположим, что имеются значения отношений ПРОЕКТЫ\_1 {ПРОЕКТ\_НАЗВ, ПРОЕКТ\_РУК} и НОМЕРА\_ПРОЕКТОВ {ПРО\_НОМ} (рис. 4.17). Предположим также, что домен атрибута ПРОЕКТ\_НАЗВ включает значения ПРОЕКТ\_1, ПРОЕКТ\_2, ПРОЕКТ\_3, домен атрибута ПРОЕКТ\_РУК ограничен значениями Иванов, Иваненко, а доменом атрибута ПРО\_НОМ является множество {1, 2, 3}. Результат операции ПРОЕКТЫ ◀OR> НОМЕРА\_ПРОЕКТОВ показан на рис. 4.17, б.

# ПРОЕКТЫ\_1

| ПРОЕКТ_НАЗВ | ПРОЕКТ_РУК |
|-------------|------------|
| ПРОЕКТ 1    | Иванов     |
| ПРОЕКТ 2    | Иваненко   |

# НОМЕРА\_ПРОЕКТОВ

| ПРО_НОМ |
|---------|
| 1       |
| 2       |

*a*

# ПРОЕКТЫ <OR> НОМЕРА\_ПРОЕКТОВ

| ПРОЕКТ_НАЗВ | ПРОЕКТ_РУК | ПРО_НОМ |
|-------------|------------|---------|
| ПРОЕКТ 1    | Иванов     | 1       |
| ПРОЕКТ 2    | Иванов     | 1       |
| ПРОЕКТ 3    | Иванов     | 1       |
| ПРОЕКТ 1    | Иваненко   | 1       |
| ПРОЕКТ 2    | Иваненко   | 1       |
| ПРОЕКТ 3    | Иваненко   | 1       |
| ПРОЕКТ 1    | Иванов     | 2       |
| ПРОЕКТ 2    | Иванов     | 2       |
| ПРОЕКТ 3    | Иванов     | 2       |
| ПРОЕКТ 1    | Иваненко   | 2       |
| ПРОЕКТ 2    | Иваненко   | 2       |
| ПРОЕКТ 3    | Иваненко   | 2       |
| ПРОЕКТ 1    | Иванов     | 3       |
| ПРОЕКТ 2    | Иваненко   | 3       |

*б*

Рис. 4.17. Пример операции «OR» с операндами без общих атрибутов

# ПРОЕКТЫ\_2

| ПРО_НОМ | ПРОЕКТ_РУК |
|---------|------------|
| 1       | Иванов     |
| 2       | Иваненко   |

*a*

# ПРОЕКТЫ\_2 <OR> НОМЕРА\_ПРОЕКТОВ

| ПРО_НОМ | ПРОЕКТ_РУК |
|---------|------------|
| 1       | Иванов     |
| 2       | Иваненко   |
| 2       | Иванов     |
| 1       | Иваненко   |

*б*

Рис. 4.18. Пример операции «OR» с операндами, имеющими общие атрибуты

Как показывает рис. 4.17, операция «OR» при наличии операндов с заголовками без общих атрибутов производит результат, гораздо более мощный, чем результат операции взятия расширенного декартова произведения, и еще менее осмысленный с практической точки зрения.

Для иллюстрации операции «OR» с операндами, у заголовков которых имеется непустое пересечение, воспользуемся значениями отношений ПРОЕКТЫ\_2 {ПРО\_НОМ, ПРОЕКТ\_РУК} (рис. 4.18) и НОМЕРА\_ПРОЕКТОВ (см. рис. 4.17). Будем предполагать, что множества значений доменов атрибутов такие же, как в предыдущем примере. Результат операции ПРОЕКТЫ\_2 «OR» НОМЕРА\_ПРОЕКТОВ показан на рис. 4.18, б.

Как уже отмечалось, при совпадении схем отношений-операндов результатом выполнения над ними операции «OR» является объединение отношений. Это непосредственно следует из спецификации операции. Если этот факт кажется неочевидным, еще раз внимательно посмотрите на спецификацию. Иллюстрирующий примером может служить рис. 4.3, на котором результат операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 UNION СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2 в точности совпадает с результатом операции СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_1 «OR» СЛУЖАЩИЕ\_В\_ПРОЕКТЕ\_2.

#### 4.2.2. Полнота Алгебры А

Покажем, что Алгебра А является полной, т. е. на основе введенных операций выражаются все операции алгебры Кодда, рассмотренной в подразд. 4.1.

В состав базовых операций Алгебры А входят операция **◀REMOVE▶**, которая является аналогом операции PROJECT, а также операция переименования атрибутов **◀RENAME▶**. UNION является частным случаем операции **◀OR▶**, TIMES, INTERSECT и NATURAL JOIN — частные случаи операции **◀AND▶**. Нам осталось показать, что через операции Алгебры А выражаются операции вычитания MINUS, ограничения (WHERE), соединения общего вида (JOIN) и реляционного деления (DIVIDE BY).

**Выводимость операции взятия разности.** Покажем, что для любых двух значений отношений  $R_1$  и  $R_2$ , совместимых относительно операции объединения, т. е. обладающих общими заголовками, верно следующее соотношение:  $R_1 \text{ MINUS } R_2 = R_1 \text{ ◀AND▶ ◀NOT▶ } R_2$ . Обозначим  $R_1 \text{ MINUS } R_2$  как  $S_1$ ,  $R_1 \text{ ◀AND▶ ◀NOT▶ } R_2$  — как  $S_2$  и  $◀NOT▶ R_2$  — как  $S_3$ .

Прежде всего, очевидно, что  $H_{S_1} = H_{S_2}$ , поскольку в обоих случаях заголовков результата совпадает с заголовками каждого из операндов. Покажем, что совпадают и тела результатов, т. е.  $B_{S_1} = B_{S_2}$ .

Сначала покажем, что если  $t_{S_1} \in B_{S_1}$ , то  $t_{S_1} \in B_{S_2}$ . По определению операции MINUS кортеж  $t_{S_1} \in B_{S_1}$  в том и только том случае, когда истинно следующее условие:  $t_{S_1} \in B_{R_1}$  and  $t_{S_1} \notin B_{R_2}$ . Но, по определению операции **◀NOT▶**, если  $t_{S_1} \notin B_{R_2}$ , то  $t_{S_1} \in B_{S_3}$ , а значит (по определению операции **◀AND▶**),  $t_{S_1} \in B_{S_2}$ .

Теперь убедимся в том, что если  $t_{S_2} \in B_{S_2}$ , то  $t_{S_2} \in B_{S_1}$ . По определению операции **◀AND▶** для случая операндов с совпадающими заголовками  $t_{S_2} \in B_{S_2}$  в том и только том случае, когда истинно следующее условие  $t_{S_2} \in B_{R_1}$  and  $t_{S_2} \in B_{S_3}$ . Но по определению операции **◀NOT▶**  $t_{S_2} \in B_{S_3}$  в том и только том случае, когда  $t_{S_2} \notin B_{R_2}$ . Следовательно, для  $t_{S_2}$  выполнено условие операции MINUS, и  $t_{S_2} \in B_{S_1}$ . ▮

**Интерпретация операции ограничения.** В подразд. 4.1 определялась операция ограничения  $R \text{ WHERE } \text{comp}$ , где  $R$  — значение отношения, а  $\text{comp}$  — простое условие ограничения вида  $X_1 \text{ comp-op } X_2$ , где  $X_1$  и  $X_2$  — имена атрибутов ограничиваемого отношения, для которых осмыслена операция сравнения  $\text{comp-op}$ , либо вида  $X \text{ comp-op } \text{const}$ , где  $X$  — имя атрибута ограничиваемого отношения;  $\text{const}$  — литерально заданная константа. Операцией сравнения  $\text{comp-op}$  может быть «=», «≠», «>», «<», «≥», «≤». По-

кажем на нескольких примерах, как можно выразить операцию ограничения с помощью базовых операций Алгебры А для всех простых допустимых условий.

В примерах этого подраздела будем использовать значение отношения СЛУЖАЩИЕ\_1 {СЛУ\_НОМЕР, СЛУ\_ИМЯ, СЛУ\_ЗАРП, РУК\_НОМ} (возможное значение показано на рис. 4.19, а). Атрибут РУК\_НОМ содержит уникальные номера служащих, являющихся руководителями проектов, и определен на том же домене, что и атрибут СЛУ\_НОМЕР. Для упрощения примеров предположим,

СЛУЖАЩИЕ\_1

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | РУК_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 4434    |
| 4435      | Петров    | 29600.00 | 4434    |
| 4415      | Сидоров   | 23000.00 | 4434    |
| 4436      | Федоров   | 20000.00 | 4434    |
| 4440      | Иванова   | 22000.00 | 4417    |
| 4441      | Сидоренко | 18000.00 | 4417    |
| 4416      | Федоренко | 20000.00 | 4417    |
| 4417      | Иваненко  | 22000.00 | 4417    |

а

ЗАРП\_2000

| СЛУ_ЗАРП |
|----------|
| 20000.00 |

б

СЛУЖАЩИЕ\_1 ◀AND▶ ЗАРП\_20000

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | РУК_НОМ |
|-----------|-----------|----------|---------|
| 4436      | Федоров   | 20000.00 | 4434    |
| 4416      | Федоренко | 20000.00 | 4417    |

в

Рис. 4.19. Выражение WHERE X = const через ◀AND▶

что множества значений доменов, на которых определены атрибуты отношения **СЛУЖАЩИЕ\_1**, ограничены значениями, содержащимися в теле этого отношения. Начнем с обсуждения операции **WHERE** с условием вида  $X \text{ comp\_op } const$ .

Предположим, что нам нужно найти всех служащих с заработной платой, равной 20000.00 руб. Возьмем отношение **ЗАРП\_20000 {СЛУ\_ЗАРП}**\* (оно показано на рис. 4.19, б). Как показано на рис. 4.19, в, результат операции **СЛУЖАЩИЕ\_1 ◀AND▶ ЗАРП\_20000** в точности совпадает с результатом операции алгебры Кодда **СЛУЖАЩИЕ\_1 WHERE СЛУ\_ЗАРП = 20000.00**.

Если требуется найти служащих, заработная плата которых превышает 20000.00 руб., то возьмем отношение **ЗАРП\_БОЛЬШЕ\_20000** (рис. 4.20, а)\*\* . Тогда снова результат операции **СЛУЖАЩИЕ\_1 ◀AND▶ ЗАРП\_БОЛЬШЕ\_20000.00** будет совпадать с результатом операции **СЛУЖАЩИЕ\_1 WHERE СЛУ\_ЗАРП > 20000.00** (рис. 4.20, б).

Понятно, что аналогичным образом выражаются через **◀AND▶** операции ограничения с условиями вида  $X \text{ comp\_op } const$ , в которых **comp\_op** является «<», «≤» или «≥». Некоторый особый случай представляет условие вида  $X \neq const$ , и мы проиллюстрируем этот случай на примере запроса «Выбрать всех служащих, не получающих заработную плату в размере 22000.00 руб.». Возьмем отношение **ЗАРП\_НЕ\_22000** (рис. 4.21, а)\*\*\*. Результат операции **СЛУЖАЩИЕ\_1 <AND> ЗАРП\_НЕ\_22000** совпадает с результатом операции **СЛУЖАЩИЕ\_1 WHERE СЛУ\_ЗАРП ≠ 22000.00** (рис. 4.21, б).

---

\* Необходимо пояснить, что отношение **ЗАРП\_20000** в действительности представляет собой литеральную константу соответствующего типа отношений. По своей природе отношение **ЗАРП\_20000** и числовой литерал 20000.00 не различаются.

\*\* Отношение **ЗАРП\_БОЛЬШЕ\_20000** это тоже литеральная константа того же типа отношения, что и **ЗАРП\_20000**, однако мощность тела этого литерального отношения в общем случае (если бы мы не ввели ограничения на множество значений домена **СЛУ\_ЗАРП**) могла бы быть очень большой.

\*\*\* Особенность этого случая состоит в том, что число кортежей в теле литеральной константы **ЗАРП\_НЕ\_22000** всего лишь на единицу меньше мощности множества значений домена **СЛУ\_ЗАРП**. Конечно, эта мощность конечна, поскольку мы имеем дело с компьютерными типами данных, но в общем случае может быть очень большой. Поэтому принципиальная возможность выражения операции ограничения через операцию реляционной конъюнкции не означает, что было бы разумно реализовывать ее таким образом на практике.



# ЗАРП\_БОЛЬШЕ\_20000

| СЛУ_ЗАРП |
|----------|
| 22400.00 |
| 29600.00 |
| 23000.00 |
| 22000.00 |
| 22000.00 |

*a*

## СЛУЖАЩИЕ\_1 ◀AND▶ ЗАРП\_БОЛЬШЕ\_20000.00

| СЛУ_НОМЕР | СЛУ_ИМЯ  | СЛУ_ЗАРП | РУК_НОМ |
|-----------|----------|----------|---------|
| 4434      | Иванов   | 22400.00 | 4434    |
| 4435      | Петров   | 29600.00 | 4434    |
| 4415      | Сидоров  | 23000.00 | 4434    |
| 4440      | Иванова  | 22000.00 | 4417    |
| 4417      | Иваненко | 22000.00 | 4417    |

*б*

Рис. 4.20. Выражение WHERE  $X > \text{const}$  через ◀AND▶

Теперь обратимся к ограничениям с простым условием вида  $X_1 \text{ comp-op } X_2$ . Опять начнем со случая, когда  $\text{comp-op} = \text{«}=\text{»}$ . Предположим, что требуется найти данные о служащих, являющихся руководителями проектов, т. е. выполнить операцию СЛУЖАЩИЕ\_1 WHERE СЛУ\_НОМЕР = РУК\_НОМ. И эту операцию можно выразить через операцию ◀AND▶, используя «константное» отношение. Воспользуемся отношением СЛУ\_НОМЕР\_РУК\_НОМ (рис. 4.22, *a*)\*. После выполнения операции СЛУЖАЩИЕ\_1 ◀AND▶ СЛУ\_НОМЕР\_РУК\_НОМ получается тот же результат, что и у операции СЛУЖАЩИЕ\_1 WHERE СЛУ\_НОМЕР = РУК\_НОМ (рис. 4.22, *б*).

Чтобы показать возможность выполнения операции ограничения вида  $R \text{ WHERE } X_1 > X_2$ , предположим, что имеется отношение СЛУЖАЩИЕ\_2 {СЛУ\_НОМЕР, СЛУ\_ИМЯ, СЛУ\_ЗАРП, СЛУ\_ПРЕМ}

\* Конечно, в общем случае мощность тела такого константного отношения будет равна мощности соответствующего домена.

| ЗАРП_НЕ_20000 |  |
|---------------|--|
| СЛУ_ЗАРП      |  |
| 22400.00      |  |
| 29600.00      |  |
| 23000.00      |  |
| 20000.00      |  |
| 18000.00      |  |

*a*

СЛУЖАЩИЕ\_1 ◀AND▶ ЗАРП\_НЕ\_20000.00

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | РУК_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 4434    |
| 4435      | Петров    | 29600.00 | 4434    |
| 4415      | Сидоров   | 23000.00 | 4434    |
| 4436      | Федоров   | 20000.00 | 4434    |
| 4441      | Сидоренко | 18000.00 | 4417    |
| 4416      | Федоренко | 20000.00 | 4417    |

*б*

Рис. 4.21. Выражение WHERE  $X \neq \text{const}$  через ◀AND▶

(возможное значение показано на рис. 4.23, *a*), причем атрибут СЛУ\_ПРЕМ содержит значения премиального вознаграждения служащего. Естественно, атрибуты СЛУ\_ЗАРП и СЛУ\_ПРЕМ определены на одном и том же домене (напомним, что в примерах этого пункта предполагается, что множество значений доменов ограничено значениями, содержащимися в теле примерного значения отношения). Пусть нас интересуют данные о служащих, получающих дополнительные вознаграждения в размере, превышающем размер основной заработной платы, т.е. требуется нужен результат операции СЛУЖАЩИЕ\_2 WHERE (СЛУ\_ПРЕМ > СЛУ\_ЗАРП).

Возьмем значение отношения ПРЕМ\_БОЛЬШЕ\_ЗАРП {СЛУ\_ПРЕМ, СЛУ\_ЗАРП}, тело которого включает все соответствующие заголовку кортежи {<СЛУ\_ПРЕМ,  $T$ ,  $v_1$ >, <СЛУ\_ЗАРП,  $T$ ,  $v_2$ >} такие, что  $v_1 > v_2$  (рис. 4.23, *б*). Отношение ПРЕМ\_БОЛЬШЕ\_ЗАРП снова

СЛУ\_НОМЕР РУК\_НОМ

| СЛУ_НОМЕР | РУК_НОМ |
|-----------|---------|
| 4434      | 4434    |
| 4435      | 4435    |
| 4415      | 4415    |
| 4436      | 4436    |
| 4440      | 4440    |
| 4441      | 4441    |
| 4416      | 4416    |
| 4417      | 4417    |

*a*

СЛУЖАЩИЕ\_1 ◀AND▶ СЛУ\_НОМЕР\_РУК\_НОМ

| СЛУ_НОМЕР | СЛУ_ИМЯ  | СЛУ_ЗАРП | РУК_НОМ |
|-----------|----------|----------|---------|
| 4434      | Иванов   | 22400.00 | 4434    |
| 4417      | Иваненко | 22000.00 | 4417    |

*б*

Рис. 4.22. Выражение WHERE  $X_1 = X_2$  через ◀AND▶

является литеральной константой типа отношения с двумя атрибутами СЛУ\_ПРЕМ и СЛУ\_ЗАРП. Конечно, даже в случае нашего примера мощность тела этого отношения достаточно велика\*.

Результат выполнения операции СЛУЖАЩИЕ\_2 ◀AND▶ ПРЕМ\_БОЛЬШЕ\_ЗАРП показан на рис. 4.23, в. Как видно, он совпадает с результатом операции СЛУЖАЩИЕ\_2 WHERE (СЛУ\_ПРЕМ > СЛУ\_ЗАРП).

Аналогичным образом через операцию ◀AND▶ выражаются операции ограничения, условия сравнения которых вида  $X_1 \text{ соп-ор } X_2$  базируются на операциях сравнения «<», «≥», «≤», «≠».

**Соединения общего вида.** При наличии того факта, что операция взятия расширенного декартова произведения TIMES

\* Легко убедиться, что в общем случае, если мощность общего домена атрибутов  $X_1$  и  $X_2$  равняется  $n$ , то мощность тела константного отношения  $X_1\_БОЛЬШЕ\_X_2$  будет составлять  $(n - 1)n/2$ .

СЛУЖАЩИЕ\_2

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | СЛУ_ПРЕМ |
|-----------|-----------|----------|----------|
| 4434      | Иванов    | 22400.00 | 18000.00 |
| 4435      | Петров    | 29600.00 | 22000.00 |
| 4415      | Сидоров   | 23000.00 | 20000.00 |
| 4436      | Федоров   | 20000.00 | 22000.00 |
| 4440      | Иванова   | 22000.00 | 20000.00 |
| 4441      | Сидоренко | 18000.00 | 22000.00 |
| 4416      | Федоренко | 20000.00 | 20000.00 |
| 4417      | Иваненко  | 22000.00 | 20000.00 |

*a*

ПРЕМ\_БОЛЬШЕ\_ЗАРП

| СЛУ_ПРЕМ | СЛУ_ЗАРП |
|----------|----------|
| 20000.00 | 18000.00 |
| 22000.00 | 18000.00 |
| 22000.00 | 20000.00 |
| 22400.00 | 18000.00 |
| 22400.00 | 20000.00 |
| 22400.00 | 22000.00 |
| 23000.00 | 18000.00 |
| 23000.00 | 20000.00 |
| 23000.00 | 22000.00 |
| 23000.00 | 22400.00 |
| 29600.00 | 18000.00 |
| 29600.00 | 20000.00 |
| 29600.00 | 22000.00 |
| 29600.00 | 22400.00 |
| 29600.00 | 23000.00 |

*б*

Рис. 4.23. Выражение WHERE  $X_1 > X_2$  через ◀AND▶ (см. также с. 124)

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | СЛУ_ПРЕМ |
|-----------|-----------|----------|----------|
| 4436      | Федоров   | 20000.00 | 22000.00 |
| 4441      | Сидоренко | 18000.00 | 22000.00 |

6

Рис. 4.23. Окончание

является частным случаем операции ◀AND▶, после того как мы научились с помощью Алгебры *A* выполнять ограничения, становится очевидно, что через операции этой алгебры выражаются и соединения общего вида. В общем случае, чтобы получить результат соединения общего вида произвольных отношений  $R_1$  и  $R_2$ , нужно:

- выполнить над одним из отношений одну или несколько операций ◀RENAME▶, чтобы избавиться от общих имен атрибутов;
- выполнить над полученными отношениями операцию ◀AND▶, производящую расширенное декартово произведение;
- над этим промежуточным результатом выполнить одну или несколько операций ◀AND▶ с отношениями-константами, чтобы должным образом ограничить его.

**Реляционное деление.** Пусть имеются отношения  $R_1 \{X, Y\}$  и  $R_2 \{Y\}$ . Утверждается, что результат  $R_1 \text{ DIVIDE BY } R_2$  совпадает с результатом выражения  $(R_1 \text{ PROJECT } X) \text{ MINUS } (((R_2 \text{ TIMES } (R_1 \text{ PROJECT } X)) \text{ MINUS } R_1) \text{ PROJECT } X)$  в терминах операций реляционной алгебры Кодда или  $(R_1 \text{ <REMOVE> } Y) \text{ <AND> <NOT> } (((R_2 \text{ <AND> } (R_1 \text{ <REMOVE> } Y)) \text{ <AND> <NOT> } R_1) \text{ <REMOVE> } Y)$  в терминах операций Алгебры *A*.

Действительно, результатом выполнения операции  $R_1 \text{ PROJECT } X$  является унарное отношение со схемой  $\{X\}$ , кортежи тела которого содержат все значения атрибута  $X$  из тела отношения  $R_1$ . Результат выражения  $R_2 \text{ TIMES } (R_1 \text{ PROJECT } X)$  — это бинарное отношение со схемой  $\{X, Y\}$ , в тело которого входят все возможные комбинации значений атрибута  $Y$  в теле отношения  $R_2$  и атрибута  $X$  в теле отношения  $R_1$ . В теле результата вычисления выражения  $(R_2 \text{ TIMES } (R_1 \text{ PROJECT } X)) \text{ MINUS } R_1$  останутся только кортежи с таким значением атрибута  $X$ , что значение атрибута  $Y$ , принадлежащее телу  $R_2$ , не является значением атрибута  $Y$  ни в одном кортеже тела отношения  $R_1$ . Следовательно, если мы возьмем проекцию результата выражения  $(R_2 \text{ TIMES}$

( $R_1$  PROJECT  $X$ )) MINUS  $R_1$  на атрибут  $X$ , то в результирующем унарном отношении останутся только те значения  $X$ , которые не должны попасть в результат операции  $R_1$  DIVIDE BY  $R_2$ . После выполнения завершающей операции MINUS мы получим желаемый результат. ■

Тем самым, мы показали, что пяти операций Алгебры  $A$  достаточно для выражения всех операций алгебры Кодда. Но на самом деле число операций можно еще более сократить, что мы и продемонстрируем в следующем подразделе.

### 4.2.3. Избыточность Алгебры $A$

В формальной математической логике стандартным базисом для выражения всех возможных булевских функций является набор {NOT, AND, OR} (отрицание, дизъюнкция и конъюнкция). Известно, что этот набор традиционен, но избыточен, поскольку верны тождества  $A$  AND  $B$  NOT (NOT  $A$  OR NOT  $B$ ) и  $A$  OR  $B$  NOT (NOT  $A$  AND NOT  $B$ ). (Эти тождества легко проверяются по таблицам истинности операций.) Оказывается (и это тоже легко проверить, опираясь на определения операций), что аналогичные тождества справедливы для операций ◀NOT▶, ◀AND▶ и ◀OR▶ Алгебры  $A$ . Тем самым, в наборе базовых операций Алгебры  $A$  можно оставить операции ◀AND▶ и ◀NOT▶ (или ◀OR▶ и ◀NOT▶).

**Реляционные аналоги штриха Шеффера и стрелки Пирса.** Более того, в алгебре логики существуют две операции, через каждую из которых выражаются все три «базовые» операции: «штрих Шеффера» —  $sh(A, B) \equiv \text{NOT } A \text{ OR NOT } B$  — и «стрелка Пирса» —  $pi(A, B) \equiv \text{NOT } A \text{ AND NOT } B$ . Легко видеть, что:

- $sh(A, A) \equiv \text{NOT } A$ ;
- $sh(\text{NOT } A, \text{NOT } B) \equiv A \text{ OR } B$ ;
- $\text{NOT } sh(A, B) \equiv A \text{ AND } B$ .

Аналогично:

- $pi(A, A) \equiv \text{NOT } A$ ;
- $pi(\text{NOT } A, \text{NOT } B) \equiv A \text{ AND } B$ ;
- $\text{NOT } pi(A, B) \equiv A \text{ OR } B$ .

Снова нетрудно проверить, что аналогичные тождества справедливы для реляционных вариантов штриха Шеффера (◀sh▶ ( $R_1, R_2$ )  $\equiv$  ◀NOT▶  $R_1$  ◀OR▶ ◀NOT▶  $R_2$ ) и стрелки Пирса (◀pi▶ ( $R_1, R_2$ )  $\equiv$  ◀NOT▶  $R_1$  ◀AND▶ ◀NOT▶  $R_2$ ).

Поэтому можно свести набор операций Алгебры  $A$  к трем операциям: ◀sh▶ (или ◀pi▶), ◀RENAME▶ и ◀REMOVE▶.

**Избыточность операции переименования атрибутов.** Наконец, покажем, что избыточна и операция «RENAME» на примере, в котором снова воспользуемся отношением СЛУЖАЩИЕ, возможное значение которого показано на рис. 4.12, а. Пусть нам требуется результат операции СЛУЖАЩИЕ «RENAME» (ПРО\_НОМ, НОМЕР\_ПРОЕКТА) (по-прежнему предполагаем, что множество значений домена атрибута ПРО\_НОМ ограничено значениями, представленными в теле значения отношения СЛУЖАЩИЕ). Возьмем бинарное отношение ПРО\_НОМ\_НОМЕР\_ПРОЕКТА (рис. 4.24, а), в котором каждый из кортежей содержит два одинаковых значения номера проекта, и в тело отношения входят все значения домена атрибута

ПРО\_НОМ\_НОМЕР\_ПРОЕКТА

| ПРО_НОМ | НОМЕР_ПРОЕКТА |
|---------|---------------|
| 1       | 1             |
| 2       | 2             |

а

(СЛУЖАЩИЕ «AND» ПРО\_НОМ\_НОМЕР\_ПРОЕКТА) «REMOVE» (ПРО\_НОМ)

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | НОМЕР_ПРОЕКТА |
|-----------|-----------|----------|---------------|
| 4434      | Иванов    | 22400.00 | 1             |
| 4435      | Петров    | 29600.00 | 1             |
| 4415      | Сидоров   | 23000.00 | 1             |
| 4436      | Федоров   | 20000.00 | 1             |
| 4440      | Иванова   | 22000.00 | 1             |
| 4434      | Иванов    | 22400.00 | 2             |
| 4435      | Петров    | 29600.00 | 2             |
| 4441      | Сидоренко | 18000.00 | 2             |
| 4416      | Федоренко | 20000.00 | 2             |
| 4417      | Иваненко  | 22000.00 | 2             |

б

Рис. 4.24. Выражение операции «RENAME» через «AND» и «REMOVE»

ПРО\_НОМ\*. Тогда вычисление выражения (СЛУЖАЩИЕ «AND» ПРО\_НОМ\_НОМЕР\_ПРОЕКТА) «REMOVE» (ПРО\_НОМ) приводит к желаемому результату (рис. 4.24, б).

Тем самым можно сократить набор операций Алгебры А до двух операций: «sh» (или «pi») и «REMOVE»\*\*.

#### 4.2.4. Заключительные замечания

Базисом Алгебры А, являющейся алгеброй отношений в строгом математическом смысле, являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения и объединения отношений. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда.

В методическом отношении Алгебра А важна, прежде всего, тем, что в ней реляционная операция естественного соединения является одной из базовых операций, в отличие от алгебры Кодда, где эта операция имела второстепенное значение. Это важно по той причине, что, как будет показано в гл. 5 и 6, операция естественного соединения играет первостепенную роль в классическом подходе к проектированию реляционных баз данных на основе нормализации.

### 4.3. Реляционное исчисление кортежей

Предположим, что мы работаем с базой данных, которая состоит из переменных отношений СЛУЖАЩИЕ {СЛУ\_НОМ, СЛУ\_ИМЯ, СЛУ\_ЗАРП, ПРО\_НОМ} и ПРОЕКТЫ {ПРО\_НОМ, ПРОЕКТ\_РУК, ПРО\_ЗАРП} (в отношении ПРОЕКТЫ атрибут ПРОЕКТ\_РУК содер-

---

\* Это «константное» отношение, тело которого не зависит от текущего содержания тела отношения СЛУЖАЩИЕ.

\*\* И конечно, в Алгебре А, как и в алгебре Кодда, должна присутствовать операция присваивания переменной отношения.



жит имена служащих, являющихся руководителями проектов, а атрибут ПРО\_ЗАРП — среднее значение заработной платы, получаемой участниками проекта), и хотим узнать имена и номера служащих, которые являются руководителями проектов со средней заработной платой, превышающей 18000 руб.

Если бы для формулировки такого запроса использовалась реляционная алгебра, то мы получили бы, например, следующее алгебраическое выражение:

```
(СЛУЖАЩИЕ JOIN (ПРОЕКТЫ WHERE ПРО_ЗАРП > 18000.00)
(СЛУ_ИМЯ = ПРОЕКТ_РУК AND)) ПРОЕКТ (СЛУ_ИМЯ,
СЛУ_НОМ)
```

Это выражение можно было бы прочитать, например, следующим образом:

- ограничить отношение ПРОЕКТЫ по условию ПРО\_ЗАРП > 18000.00;
- выполнить эквисоединение отношения СЛУЖАЩИЕ и результата предыдущего ограничения по условию СЛУ\_НОМ = ПРОЕКТ\_РУК;
- спроецировать результат эквисоединения на множество атрибутов {СЛУ\_ИМЯ, СЛУ\_НОМ}.

На основе алгебраического выражения была сформулирована последовательность шагов выполнения запроса, каждый из которых соответствует одной реляционной операции.

Если сформулировать тот же запрос с использованием реляционного исчисления кортежей, которому посвящается этот подраздел, то мы получили бы два определения переменных:

RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ

и

RANGE ПРОЕКТ IS ПРОЕКТЫ

и выражение

```
СЛУЖАЩИЙ.СЛУ_ИМЯ, СЛУЖАЩИЙ.СЛУ_НОМ
WHERE EXISTS (СЛУЖАЩИЙ.СЛУ_ИМЯ = ПРОЕКТ.ПРОЕКТ_РУК
AND ПРОЕКТ.ПРО_ЗАРП > 18000.00) .
```

Это выражение можно прочитать, например, следующим образом: выдать значения атрибутов СЛУ\_ИМЯ и СЛУ\_НОМ для каждого кортежа служащих такого, что существует кортеж проектов со значением атрибута ПРОЕКТ\_РУК, совпадающим со значением СЛУ\_НОМ этого кортежа служащих, и значением ПРО\_ЗАРП, большим 18000.00.

Во второй формулировке указаны лишь характеристики результирующего отношения, но ничего не говорится о способе его формирования. В этом случае система сама должна решить, какие операции и в каком порядке нужно выполнить над отношениями **СЛУЖАЩИЕ** и **ПРОЕКТЫ**. Обычно говорят, что алгебраическая формулировка является процедурной, т.е. задающей последовательность действий для выполнения запроса, а логическая — описательной (или декларативной), поскольку она всего лишь описывает свойства желаемого результата. Как указывалось в начале этой главы, на самом деле эти два механизма эквивалентны, и существуют не слишком сложные правила преобразования одного формализма в другой.

Реляционное исчисление является прикладной ветвью формального механизма исчисления предикатов первого порядка\*. В основе исчисления лежат понятие переменной с определенной для нее областью допустимых значений и понятие правильно построенной формулы, опирающейся на переменные, предикаты и кванторы.

В зависимости от того, что является областью определения переменной, различают исчисление кортежей и исчисление доменов. В исчислении кортежей областями определения переменных являются тела отношений базы данных, т.е. допустимым значением каждой переменной является кортеж тела некоторого отношения. В исчислении доменов областями определения переменных являются домены, на которых определены атрибуты отношений базы данных, т.е. допустимым значением каждой переменной является значение некоторого домена. В этом подразделе мы сравнительно подробно рассмотрим исчисление кортежей, а в следующем подразделе коротко опишем особенности исчисления доменов.

Как и в подразделах, посвященных реляционной алгебре, нам не удастся избежать использования некоторого конкретного синтаксиса, который мы тем не менее формально определять не будем. Те или иные синтаксические конструкции будут вводиться по мере необходимости. В совокупности используемый синтаксис близок, но не полностью совпадает с синтаксисом языка баз данных QUEL, который долгое время являлся основным языком известной реляционной СУБД Ingres.

---

\* Это совсем не означает, что для понимания этого материала требуется знание исчисления предикатов. Автор стремился к тому, чтобы материал был в основном самодостаточным.

### 4.3.1. Кортежные переменные

Для определения кортежной переменной используется оператор **RANGE**. Например, для того чтобы определить переменную **СЛУЖАЩИЙ**, областью определения которой является значение отношения **СЛУЖАЩИЕ**, нужно употребить конструкцию

**RANGE СЛУЖАЩИЙ IS СЛУЖАЩИЕ**

Как уже отмечалось, из этого определения следует, что в любой момент времени переменная **СЛУЖАЩИЙ** представляет некоторый кортеж отношения **СЛУЖАЩИЕ**. При использовании кортежных переменных в формулах можно ссылаться на значение атрибута переменной (это аналогично тому, как, например, при программировании на языке **C** можно сослаться на значение поля структурной переменной). Например, для того чтобы сослаться на значение атрибута **СЛУ\_ИМЯ** переменной **СЛУЖАЩИЙ**, нужно употребить конструкцию **СЛУЖАЩИЙ.СЛУ\_ИМЯ**.

### 4.3.2. Правильно построенные формулы

Правильно построенная формула (**Well-Formed Formula**, **WFF**) служит для выражения условий, накладываемых на кортежные переменные.

**Простые условия.** Основой **WFF** являются простые условия, представляющие собой операции сравнения скалярных значений (значений атрибутов переменных или литерально заданных констант). Например, конструкции

**СЛУЖАЩИЙ.СЛУ\_НОМ = 4434**

и

**СЛУЖАЩИЙ.СЛУ\_НОМ = ПРОЕКТ.ПРОЕКТ\_РУК**

являются простыми условиями. Первое условие принимает значение *true* в том и только том случае, когда значение атрибута **СЛУ\_НОМ** кортежной переменной **СЛУЖАЩИЙ** равно 4434. Второе условие принимает значение *true* в том и только том случае, когда значения атрибутов **СЛУ\_НОМ** и **ПРОЕКТ\_РУК** переменных **СЛУЖАЩИЙ** и **ПРОЕКТ** совпадают.

По определению, простое сравнение является **WFF**, а **WFF**, заключенная в круглые скобки, представляет собой простое сравнение.

Более сложные варианты **WFF** строятся из **WFF** и простых сравнений с помощью логических связок **NOT**, **AND**, **OR**

# СЛУЖАЩИЕ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 1       |
| 4435      | Петров    | 29600.00 | 1       |
| 4415      | Сидоров   | 23000.00 | 1       |
| 4436      | Федоров   | 20000.00 | 1       |
| 4440      | Иванова   | 22000.00 | 1       |
| 4434      | Иванов    | 22400.00 | 2       |
| 4435      | Петров    | 29600.00 | 2       |
| 4441      | Сидоренко | 18000.00 | 2       |
| 4416      | Федоренко | 20000.00 | 2       |
| 4417      | Иваненко  | 22000.00 | 2       |

## ПРОЕКТЫ

| ПРО_НОМ | ПРОЕКТ_РУК | ПРО_ЗАРП |
|---------|------------|----------|
| 1       | Иванов     | 23400.00 |
| 2       | Иваненко   | 22400.00 |

## НОМЕРА\_ПРОЕКТОВ

| ПРО_НОМ |
|---------|
| 1       |
| 2       |

Рис. 4.25. Возможные значения отношений СЛУЖАЩИЕ, ПРОЕКТЫ и НОМЕРА\_ПРОЕКТОВ

и IF ... THEN\* с учетом обычных приоритетов операций (NOT > AND > OR) и возможности расстановки скобок. Так, если *form* — WFF, а *comp* — простое сравнение, то NOT *form*, *comp* AND *form*, *comp* OR *form* и IF *comp* THEN *form* являются WFF.

Для примеров воспользуемся возможными значениями отношений СЛУЖАЩИЕ, ПРОЕКТЫ и НОМЕРА\_ПРОЕКТОВ из предыдущих разделов (для удобства они еще раз показаны на рис. 4.25).

---

\* Через IF ... THEN здесь обозначается одна из важных логических функций — импликация. По определению, IF *a* THEN *b*  $\equiv$  NOT *a* OR *b*. Хотя операция импликации является избыточной, она явно вводится в реляционное исчисление, поскольку часто требуется на практике для выражения условий.

Правильно построенной является следующая формула: (1)

```
IF СЛУЖАЩИЙ.СЛУ_ИМЯ = 'Иванов'  
THEN (СЛУЖАЩИЙ.СЛУ_ЗАРП >= 22400.00 AND  
СЛУЖАЩИЙ.ПРО_НОМ = 1)
```

Эта формула будет принимать значение *true* для множества значений кортежной переменной **СЛУЖАЩИЙ**, показанного на рис. 4.26.

Конечно, нужно представлять себе какой-нибудь способ реализации системы, которая сможет по заданной WFF при существующем состоянии базы данных произвести такой результат. И таким очевидным способом является следующий: в некотором порядке просмотреть область определения переменной и к каждому очередному кортежу применить условие. Результатом будет то множество кортежей, для которых при вычислении условия производится значение *true*. Очевидно, что результат подобной *интерпретации* формулы эквивалентен результату выполнению алгебраической операции **СЛУЖАЩИЕ WHERE (NOT (СЛУЖАЩИЙ.СЛУ\_ИМЯ = 'Иванов') OR (СЛУЖАЩИЙ.СЛУ\_ЗАРП >= 22400.00 AND СЛУЖАЩИЙ.ПРО\_НОМ = 1))** над отношением, тело которого представляет собой область определения кортежной переменной.

Пусть имеется следующее определение кортежной переменной **ПРОЕКТ**:

RANGE ПРОЕКТ IS ПРОЕКТЫ

| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 1       |
| 4435      | Петров    | 29600.00 | 1       |
| 4415      | Сидоров   | 23000.00 | 1       |
| 4436      | Федоров   | 20000.00 | 1       |
| 4440      | Иванова   | 22000.00 | 1       |
| 4435      | Петров    | 29600.00 | 2       |
| 4441      | Сидоренко | 18000.00 | 2       |
| 4416      | Федоренко | 20000.00 | 2       |
| 4417      | Иваненко  | 22000.00 | 2       |

Рис. 4.26. Область истинности формулы (1)

Вот еще пример правильно построенной формулы: (2)

СЛУЖАЩИЙ.СЛУ\_ИМЯ = ПРОЕКТ.ПРОЕКТ\_РУК

Эта формула будет принимать значение *true* для множества пар значений кортежных переменных СЛУЖАЩИЙ и ПРОЕКТ, показанного на рис. 4.27.

Очевидный способ реализации системы, которая по заданной WFF при существующем состоянии базы данных производит такой результат, заключается в следующем:

- в некотором порядке просматривать область определения (например) переменной СЛУЖАЩИЙ;
- для каждого текущего кортежа из области определения переменной СЛУЖАЩИЙ просматривать область определения переменной ПРОЕКТ;
- оставлять в области истинности те пары кортежей, для которых формула принимает значение *true*.

Возможен и альтернативный подход: начать просмотр с области определения переменной ПРОЕКТ, и для каждого кортежа ПРОЕКТ просматривать область определения СЛУЖАЩИЙ.

Здесь нужно сделать несколько замечаний. Во-первых, если бы в данном случае формула была тождественно истинной, например, имела вид

(СЛУЖАЩИЙ.СЛУ\_ИМЯ = СЛУЖАЩИЙ.СЛУ\_ИМЯ) AND  
(ПРОЕКТ.ПРОЕКТ\_РУК = ПРОЕКТ.ПРОЕКТ\_РУК) ,

то областью истинности этой формулы являлось бы декартово произведение (в строгом математическом смысле) тел отношений СЛУЖАЩИЙ и ПРОЕКТ. В реляционном исчислении кортежей, как и в реляционной алгебре, принято иметь дело с операцией расширенного декартова произведения, и поэтому считается, что в подобных случаях областью истинности WFF является отношение, заголовок которого представляет собой объединение заголовков отношений, на телах которых определены кортежные переменные, а кортежи получаются путем объединения соответствующих кортежей из областей определения переменных. При этом имя атрибута результирующего отношения уточняется именем соответствующей переменной. Поэтому правильное изображать область истинности формулы СЛУЖАЩИЙ.СЛУ\_ИМЯ = ПРОЕКТ.ПРОЕКТ\_РУК так, как показано на рис. 4.28.

Во-вторых, как видно, показанное результирующее отношение в точности совпадает с результатом алгебраической операции СЛУЖАЩИЕ JOIN ПРОЕКТЫ WHERE СЛУ\_ИМЯ = ПРОЕКТ\_РУК с учетом особенности именования атрибутов результирующего

| СЛУЖАЩИЕ  |          |          |         | ПРОЕКТЫ |            |          |
|-----------|----------|----------|---------|---------|------------|----------|
| СЛУ_НОМЕР | СЛУ_ИМЯ  | СЛУ_ЗАРП | ПРО_НОМ | ПРО_НОМ | ПРОЕКТ_РУК | ПРО_ЗАРП |
| 4434      | Иванов   | 22400.00 | 1       | 1       | Иванов     | 23400.00 |
| 4434      | Иванов   | 22400.00 | 2       | 1       | Иванов     | 23400.00 |
| 4417      | Иваненко | 22000.00 | 2       | 2       | Иваненко   | 22400.00 |

Рис. 4.27. Область истинности формулы (2)

| СЛУЖАЩИЙ.<br>СЛУ_НОМЕР | СЛУЖАЩИЙ.<br>СЛУ_ИМЯ | СЛУЖАЩИЙ.<br>СЛУ_ЗАРП | СЛУЖАЩИЙ.<br>ПРО_НОМ | ПРОЕКТ.<br>ПРО_НОМ | ПРОЕКТ.<br>ПРОЕКТ_РУК | ПРОЕКТ.<br>ПРО_ЗАРП |
|------------------------|----------------------|-----------------------|----------------------|--------------------|-----------------------|---------------------|
| 4434                   | Иванов               | 22400.00              | 1                    | 1                  | Иванов                | 23400.00            |
| 4434                   | Иванов               | 22400.00              | 2                    | 1                  | Иванов                | 23400.00            |
| 4417                   | Иваненко             | 22000.00              | 2                    | 2                  | Иваненко              | 22400.00            |

Рис. 4.28. Область истинности формулы (2) в виде нормализованного отношения

отношения. Наконец, заметим, что описанный ранее способ реализации, который приводит к получению области истинности рассмотренной формулы, в действительности является наиболее общим (и зачастую неоптимальным) способом выполнения операций соединения (он называется методом *вложенных циклов* — *nested loops join*).

**Кванторы, свободные и связанные переменные.** При построении WFF допускается использование кванторов существования (EXISTS) и всеобщности (FORALL). Если *form* — это WFF, в которой участвует переменная *var*, то конструкции EXISTS *var* (*form*) и FORALL *var* (*form*) представляют собой WFF. По определению, формула EXISTS *var* (*form*) принимает значение *true* в том и только том случае, если в области определения переменной *var* найдется хотя бы одно значение (кортеж), для которого WFF *form* принимает значение *true*. Формула FORALL *var* (*form*) принимает значение *true*, если для всех значений переменной *var* из ее области определения WFF *form* принимает значение *true*.

Переменные, входящие в WFF, могут быть *свободными* или *связанными*. По определению, все переменные, входящие в WFF, при построении которой не использовались кванторы, являются свободными. Фактически, это означает, что если для какого-то набора значений свободных кортежных переменных при вычислении WFF получено значение *true*, то эти значения кортежных переменных могут входить в результирующее отношение. Если же имя переменной использовано сразу после квантора при построении WFF вида EXISTS *var* (*form*) или FORALL *var* (*form*), то в этой WFF и во всех WFF, построенных с ее участием, *var* является связанной переменной. Это означает, что такая переменная не видна за пределами минимальной WFF, связавшей эту переменную. При вычислении значения такой WFF используется не одно значение связанной переменной, а вся область ее определения.

Пусть здесь и далее в этом разделе СЛУ1 и СЛУ2 представляют собой две кортежные переменные, определенные на отношении СЛУЖАЩИЕ. Тогда WFF (3)

EXISTS СЛУ2 (СЛУ1.СЛУ\_ЗАРП > СЛУ2.СЛУ\_ЗАРП)

для текущего кортежа переменной СЛУ1 принимает значение *true* в том и только том случае, если во всем отношении СЛУЖАЩИЕ найдется такой кортеж (ассоциированный с переменной СЛУ2), чтобы значение его атрибута СЛУ\_ЗАРП удовлетворяло внутреннему условию сравнения. Легко видеть, что эта формула



| СЛУ_НОМЕР | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|-----------|----------|---------|
| 4434      | Иванов    | 22400.00 | 1       |
| 4435      | Петров    | 29600.00 | 1       |
| 4415      | Сидоров   | 23000.00 | 1       |
| 4436      | Федоров   | 20000.00 | 1       |
| 4440      | Иванова   | 22000.00 | 1       |
| 4434      | Иванов    | 22400.00 | 2       |
| 4435      | Петров    | 29600.00 | 2       |
| 4416      | Федоренко | 20000.00 | 2       |
| 4417      | Иваненко  | 22000.00 | 2       |

Рис. 4.29. Область истинности формулы (3)

принимает значение *true* для тех и только тех значений кортежной переменной СЛУ1, которые соответствуют служащим, не получающим минимальную заработную плату. Соответствующее множество кортежей показано на рис. 4.29 (для тела отношения СЛУЖАЩИЕ, показанного на рис. 4.25).

Правильно построенная формула (4)

FORALL СЛУ2 (СЛУ1.СЛУ\_ЗАРП  $\geq$  СЛУ2.СЛУ\_ЗАРП)

для текущего кортежа переменной СЛУ1 принимает значение *true* в том и только том случае, если для всех кортежей отношения СЛУЖАЩИЕ (связанных с переменной СЛУ2) значения атрибута СЛУ\_ЗАРП удовлетворяют условию сравнения. Снова легко видеть, что формула принимает значение *true* только для тех значений кортежной переменной СЛУ1, которые соответствуют служащим, получающим максимальную заработную плату. Соответствующее множество кортежей показано на рис. 4.30.

Очевидно, что представленные на рис. 4.30 и 4.31 отношения соответствуют условиям обеих формул. Но как в данном случае можно реализовать систему, которая по заданной формуле про-

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП | ПРО_НОМ |
|-----------|---------|----------|---------|
| 4435      | Петров  | 29600.00 | 1       |
| 4435      | Петров  | 29600.00 | 2       |

Рис. 4.30. Область истинности формулы (4)

| СЛУ_НОМЕР | СЛУ_ИМЯ | СЛУ_ЗАРП |
|-----------|---------|----------|
| 4434      | Иванов  | 22400.00 |
| 4435      | Петров  | 29600.00 |

Рис. 4.31. Реляционное деление средствами реляционного исчисления

изводит правильный результат? Наиболее очевидным является следующий способ интерпретации обеих обсуждавшихся ранее формул:

- в некотором порядке просматривать область определения свободной кортежной переменной СЛУ1;
- для каждого очередного кортежа из области определения СЛУ1 просматривать область определения связанной переменной СЛУ2 до тех пор, пока не будет установлено истинностное значение формулы для данного кортежа СЛУ1 (в случае наличия квантора существования процесс просмотра для СЛУ2 можно остановить после нахождения первого кортежа, для которого значением подформулы, находящейся под знаком квантора, станет *true*; при наличии квантора всеобщности необходимо просмотреть всю область определения СЛУ2).

Заметим, что здесь вновь получаем два цикла, как и при интерпретации WFF с двумя свободными переменными. Но в данном случае во внешнем цикле обязательно просматривается область определения свободной переменной.

На самом деле, правильнее говорить не о свободных и связанных переменных, а о свободных и связанных вхождениях переменных. Если переменная *var* является связанной в WFF *form*, то во всех WFF, включающих *form*, вне *form* может использоваться вхождение того же имени переменной *var*, которое может быть свободным или связанным, но в любом случае не имеет никакого отношения к вхождению переменной *var* в WFF *form*. Вот пример:

```
EXISTS СЛУ2 (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
AND СЛУ1.СЛУ_НОМЕР ≠ СЛУ2.СЛУ_НОМЕР)
AND FORALL СЛУ2 (IF СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ
THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

Эта формула принимает значение *true* только для тех значений переменной СЛУ1, которые соответствуют служащим, участвующим в проектах с более чем одним участником, причем все участники проекта получают одну и ту же заработную

плату. Здесь имеем два связанных вхождения переменной СЛУ2 с совершенно разным смыслом. Грубо говоря, для текущего значения переменной СЛУ1 переменная СЛУ2 два раза «пробежит» свою область определения — первый раз при вычислении части формулы с квантором существования, а второй при вычислении части с квантором всеобщности. Кстати, к тому же результату приведет следующая формула с одним квантором всеобщности:

```
FORALL СЛУ2 (IF (СЛУ1.ПРО_НОМ = СЛУ2.ПРО_НОМ AND  
СЛУ1.СЛУ_НОМЕР ≠ СЛУ2.СЛУ_НОМЕР)  
THEN СЛУ1.СЛУ_ЗАРП = СЛУ2.СЛУ_ЗАРП)
```

Легко заметить, что кванторы можно трактовать как булевские функции (функции, принимающие значения *true* или *false*) над множеством значений связанной кортежной переменной. С тем же успехом можно ввести в реляционное исчисление числовые функции над множествами, такие как **MIN** (минимальное значение), **MAX** (максимальное значение), **AVG** (среднее значение) и т. д.

В этом случае можно было бы написать, например, WFF

```
СЛУ1.СЛУ_ЗАРП > MIN СЛУ2.СЛУ_ЗАРП (СЛУ1.ПРО_НОМ =  
СЛУ2.ПРО_НОМ),
```

в области истинности которой содержатся все кортежи отношения **СЛУЖАЩИЕ**, соответствующие тем служащим, которые получают заработную плату, превышающую минимальную заработную плату служащих, участвующих в том же проекте. Понятно, что для получения результирующего отношения можно интерпретировать формулу таким же образом, как в обсуждавшемся ранее случае наличия кванторов.

### 4.3.3. Целевые списки и выражения реляционного исчисления

Итак, WFF обеспечивают средства формулировки условия выборки из отношений базы данных. Чтобы можно было использовать исчисление для реальной работы с базами данных, требуется еще один компонент, который определяет набор и имена атрибутов результирующего отношения. Этот компонент называется *целевым списком* (*target list*).

Целевой список строится из целевых элементов, каждый из которых может представляться одним из следующих способов:

- *var.attr*, где *var* — имя свободной переменной соответствующей WFF, а *attr* — имя атрибута отношения, на котором определена переменная *var*;
- *var*, что эквивалентно наличию подписка *var.attr1*, *var.attr2*, ..., *var.attrn*, где множество {*attr1*, *attr2*, ..., *attrn*} включает имена всех атрибутов определяющего отношения;
- *new\_name = var.attr*; *new\_name* — новое имя соответствующего атрибута результирующего отношения.

Последний вариант требуется в тех случаях, когда в WFF используется несколько свободных переменных с одинаковой областью определения. Фактически применение целевого списка к области истинности WFF эквивалентно действию алгебраической операции проекции, а последний из приведенных вариантов представляет собой некоторую разновидность алгебраической операции переименования атрибута.

*Выражением реляционного исчисления кортежей* называется конструкция вида **target\_list WHERE WFF**. Значением выражения является отношение, тело которого определяется WFF, а множество атрибутов и их имена — целевым списком.

В качестве простого примера покажем выражение реляционного исчисления кортежей, результат которого совпадает с результатом операции **СЛУЖАЩИЕ DIVIDE BY НОМЕРА\_ПРОЕКТОВ** (см. рис. 4.10):

```

СЛУ1, СЛУ2 RANGE IS СЛУЖАЩИЕ
НОМЕР_ПРОЕКТА RANGE IS НОМЕРА_ПРОЕКТОВ
СЛУ1.СЛУ_НОМЕР, СЛУ1.СЛУ_ИМЯ, СЛУ1.СЛУ_ЗАРП
WHERE FORALL НОМЕР_ПРОЕКТА EXISTS СЛУ2
(СЛУ1.СЛУ_НОМЕР = СЛУ2.СЛУ_НОМЕР AND
СЛУ1.ПРО_НОМ = НОМЕР_ПРОЕКТА.ПРО_НОМ)

```

Легко убедиться в том, что результатом этого выражения является отношение, показанное на рис. 4.31 и совпадающее с результатом операции **СЛУЖАЩИЕ DIVIDE BY НОМЕРА\_ПРОЕКТОВ**, которое демонстрировалось на рис. 4.10.

## 4.4. Реляционное исчисление доменов

В исчислении доменов областью определения переменных являются не отношения, а домены. В каждый момент времени переменная исчисления домена принимает некоторое значение из множества значений этого домена. Применительно к базе данных **СЛУЖАЩИЕ-ПРОЕКТЫ** можно говорить, например, о до-

менных переменных **ИМЯ** (значения — допустимые имена) или **НОСЛУ** (значения — допустимые номера служащих).

#### 4.4.1. Условия членства

Основным формальным отличием исчисления доменов от исчисления кортежей является наличие дополнительного множества предикатов, позволяющих выражать так называемые условия членства. Если  $R$  — это  $n$ -арное отношение с заголовком  $\{<X_1, T_1>, <X_2, T_2>, \dots, <X_n, T_n>\}$ , то условие членства имеет вид  $R(X_{i_1} : v_{i_1}, X_{i_2} : v_{i_2}, \dots, X_{i_m} : v_{i_m})$  ( $m \leq n$ ,  $\{X_{ij}\} \subset \{X_k\}$ ), где  $v_{ij}$  — это либо литерально задаваемая константа домена (или типа)  $T_{ij}$ , либо имя некоторой доменной переменной, определенной на домене (или типе)  $T_{ij}$ .

Условие членства принимает значение *true* в том и только том случае, если в отношении  $R$  существует кортеж, содержащий указанные значения указанных атрибутов. Если  $v_{ij}$  — константа, то на атрибут  $X_{ij}$  накладывается жесткое условие, не зависящее от текущих значений доменных переменных; если же  $v_{ij}$  — имя доменной переменной, то условие членства может принимать разные значения при разных значениях этой переменной.

Для большей ясности приведем пару примеров. Для простоты будем считать, что определены доменные переменные, имена которых совпадают с именами атрибутов отношения **СЛУЖАЩИЕ**, а в случае, когда требуется несколько доменных переменных, определенных на одном домене, будем добавлять в конце имени цифры. WFF исчисления доменов

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',  
СЛУ_ЗАРП:22400.00, ПРО_НОМ:1)
```

примет значение *true* в том и только том случае, когда в теле отношения **СЛУЖАЩИЕ** содержится кортеж  $\{<СЛУ\_НОМ, 2934>, <СЛУ\_ИМЯ, 'Иванов'>, <СЛУ\_ЗАРП, 22400.00>, <ПРО\_НОМ, 1>\}$  (имена доменов опущены). Соответствующие значения доменных переменных образуют область истинности этой WFF. С другой стороны, WFF

```
СЛУЖАЩИЕ (СЛУ_НОМ:2934, СЛУ_ИМЯ:'Иванов',  
СЛУ_ЗАРП:22400.00, ПРО_НОМ:ПРО_НОМ)
```

будет принимать значение *true* для всех комбинаций явно заданных значений и допустимых значений переменной **ПРО\_НОМ**, которые соответствуют кортежам, входящим в тело отношения **СЛУЖАЩИЕ**. При наличии значения отношения **СЛУЖАЩИЕ**, по-

казанного на рис. 4.25, областью истинности этой WFF являются два следующих набора значений доменных переменных: <2934, 'Иванов', 22400.00, 1> и <2934, 'Иванов', 22400.00, 2>.

#### 4.4.2. Выражения исчисления доменов

Во всех остальных отношениях формулы и выражения исчисления доменов выглядят похожими на формулы и выражения исчисления кортежей. В частности, формулы могут включать кванторы, и различаются свободные и связанные вхождения доменных переменных.

Для примера выражения исчисления доменов сформулируем с использованием исчисления доменов запрос «Выдать номера и имена служащих, не получающих минимальную заработную плату»:

```
СЛУ_НОМ, СЛУ_ИМЯ WHERE EXISTS СЛУ_ЗАРП1  
(СЛУЖАЩИЕ (СЛУ_ЗАРП:СЛУ_ЗАРП1) AND  
СЛУЖАЩИЕ (СЛУ_НОМ:СЛУ_НОМ, СЛУ_ИМЯ:СЛУ_ИМЯ, СЛУ_  
ЗАРП:СЛУ_ЗАРП) AND  
СЛУ_ЗАРП > СЛУ_ЗАРП1)
```

Реляционное исчисление доменов является основой большинства языков запросов, основанных на использовании форм. В частности, на этом исчислении базировался известный язык Query-by-Example [14], который был первым (и наиболее интересным) языком в семействе языков, основанных на табличных формах.

\* \* \*

В этой главе рассматривалась манипуляционная составляющая реляционной модели данных. Были представлены два варианта реляционной алгебры. Конечно, с формальной точки зрения можно было бы обойтись одним из вариантов, поскольку их выразительные средства эквивалентны. Но алгебра Кодда в большей степени базируется на теории множеств. Базовыми операциями являются переименование атрибутов, объединение, пересечение, взятие разности, декартово произведение, проекция и ограничение. Операция соединения общего вида, хотя и включается в алгебру, является вторичной и явно представляется через другие операции. Фундаментальная же в реляционном подходе операция естественного соединения выражается через

соединение общего вида и в алгебру не включается. В терминах алгебры Кодда проще всего определяются алгебраические черты языка SQL, в частности общая семантика оператора SELECT (см. гл. 12).

Базисом Алгебры А являются операции реляционного отрицания (дополнения), реляционной конъюнкции (или дизъюнкции) и проекции (удаления атрибута). Реляционные аналоги логических операций определяются в терминах отношений на основе обычных теоретико-множественных операций и позволяют выражать напрямую операции пересечения, декартова произведения, естественного соединения, объединения отношений. Путем комбинирования базовых операций выражаются операции переименования атрибутов, соединения общего вида, взятия разности отношений. Алгебра А позволяет лучше осознать логические основы реляционной модели, хотя, безусловно, является в меньшей степени ориентированной на практическое применение, чем алгебра Кодда.

Реляционному исчислению мы отвели меньше места, поскольку не ставили перед собой задачу определить какой-либо полноценный логический язык запросов. Цель состояла в том, чтобы показать возможность декларативной логической формулировки запросов. В этом случае выполнение запроса происходит путем интерпретации логической формулы, а не вычисления алгебраического выражения. Были рассмотрены два варианта реляционного исчисления, первый из которых — реляционное исчисление кортежей — был определен сравнительно полно, а для второго — реляционного исчисления доменов — были только отмечены и проиллюстрированы основные отличительные черты.

### Глава 5

#### ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ НА ОСНОВЕ УЧЕТА ФУНКЦИОНАЛЬНЫХ ЗАВИСИМОСТЕЙ. ВТОРАЯ И ТРЕТЬЯ НОРМАЛЬНЫЕ ФОРМЫ ОТНОШЕНИЙ, НОРМАЛЬНАЯ ФОРМА БОЙСА—КОДДА

При проектировании баз данных решаются две основные проблемы:

- Каким образом отобразить объекты предметной области в абстрактные объекты модели данных, чтобы это отображение не противоречило семантике предметной области и было, по возможности, наилучшим (эффективным, удобным и т. д.)? Эту проблему называют проблемой *логического проектирования* баз данных\*.
- Как обеспечить эффективность выполнения запросов к базе данных, т. е. каким образом, имея в виду особенности конкретной СУБД, расположить данные во внешней памяти, создания каких дополнительных структур (например, индексов) потребовать и т. д.? Эту проблему обычно называют проблемой *физического проектирования* баз данных.

В случае реляционных баз данных трудно предложить какие-либо общие рецепты по части физического проектирования. Здесь слишком многое зависит от используемой СУБД. Поэтому мы ограничимся вопросами логического проектирования реляционных баз данных, которые существенны при использовании любой реляционной СУБД.

Более того, здесь не будем касаться очень важного аспекта проектирования — определения ограничений целостности общего

---

\* Часто этапу логического проектирования предшествует этап *концептуального моделирования* предметной области, на котором используются *семантические* модели данных и вырабатывается *концептуальная схема* будущей базы данных, впоследствии отображаемая в модель данных, которая поддерживается целевой СУБД. Этот подход кратко обсуждается в гл. 7.



вида (за исключением ограничений, задаваемых функциональными и многозначными зависимостями, а также зависимостями проекции/соединения). Дело в том, что при использовании СУБД с развитыми механизмами определения и поддержки ограничений целостности (например, SQL-ориентированных систем) трудно предложить какой-либо универсальный подход к определению ограничений целостности. Эти ограничения могут иметь произвольно сложную форму, и их формулировка пока относится скорее к области искусства, чем инженерного мастерства.

Поэтому будем считать, что проблема проектирования реляционной базы данных состоит в обоснованном принятии решений о том, из каких отношений должна состоять БД и какие атрибуты должны быть у этих отношений. В этой и следующей главах будет рассмотрен классический подход, при котором весь процесс проектирования базы данных осуществляется в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений.

Этому подходу посвящена бóльшая часть теории реляционных баз данных, и он описывается практически во всех современных учебниках по технологии баз данных, хотя наиболее авторитетными считаются книги Дейта (например, [2]) и Ульмана (например, [15, 16]). Здесь, как и в других своих книгах [40, 41], автор стремится совместить строгость изложения, свойственную книгам Ульмана, с живостью изложения, присущей книгам Дейта.

При применении классического подхода исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих «улучшенными» свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами, в некотором смысле, лучшими, чем предыдущая.

Каждой нормальной форме соответствует определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером может служить ограничение первой нормальной формы — значения всех атрибутов отношения атомарны\*. Поскольку требование первой нормальной формы

---

\* Напомним, что *атомарность* значения трактуется в том смысле, что значение типизировано и с этим значением можно работать только с помощью операций соответствующего типа данных.

является базовым требованием классической реляционной модели данных, мы будем считать, что исходный набор отношений уже соответствует этому требованию.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);
- третья нормальная форма (3NF);
- нормальная форма Бойса—Кодда (BCNF);
- четвертая нормальная форма (4NF);
- пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Приведем основные свойства нормальных форм:

- каждая последующая нормальная форма в некотором смысле лучше предыдущей нормальной формы;
- при переходе к следующей нормальной форме характеристики предыдущих нормальных форм сохраняются.

В основе процесса проектирования лежит *метод нормализации*, т.е. декомпозиции отношения, находящегося в предыдущей нормальной форме, на два или более отношений, которые удовлетворяют требованиям следующей нормальной формы.

В этой главе обсудим первые шаги процесса нормализации, в которых учитываются функциональные зависимости между атрибутами отношений. Хотя мы и называем эти шаги первыми, именно они имеют основную практическую важность, поскольку позволяют получить схему реляционной базы данных, в большинстве случаев удовлетворяющую потребности приложений.

## 5.1. Элементы теории функциональных зависимостей

Несмотря на практическую ориентированность, теория реляционных баз данных является самостоятельным научным направлением, в котором работали (и продолжают работать) многие известные исследователи. Эта книга не посвящена подробному описанию основных результатов в области теории реляционных баз данных (такой материал можно найти, например, в [17]), поэтому приведем только те определения и утверждения, которые необходимы для общего понимания процесса проектирования реляционных баз данных на основе нормализации (и показать, как доказываются эти утверждения).

Поскольку наиболее важные с практической точки зрения свойства реляционных баз данных базируются на понятии *функциональной зависимости*, мы выделили в отдельный подраздел краткое обсуждение соответствующих теоретических вопросов. Среди этих вопросов наибольший интерес представляют замыкания и покрытия множеств функциональных зависимостей, аксиомы Армстронга и теорема Хита о достаточном условии декомпозиции отношения без потерь. Понятия и утверждения данного подраздела действительно нужны для усвоения материала этой главы, но мы стремились еще и продемонстрировать читателям на несложных примерах, что собой представляет теория реляционных баз данных, каков уровень ее сложности и насколько она понятна интуитивно.

### 5.1.1. Базовые определения и утверждения теории функциональных зависимостей

Наиболее важные с практической точки зрения нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*. Для дальнейшего изложения нам потребуется несколько определений и утверждений (по ходу изложения будем пояснять их и иллюстрировать).

Пусть задана переменная отношения  $r$ , и  $X$  и  $Y$  являются произвольными подмножествами заголовка  $r$  («составными» атрибутами).

**Определение 5.1.** В значении переменной отношения  $r$  атрибут  $Y$  *функционально зависит* от атрибута  $X$  в том и только том случае, если каждому значению  $X$  соответствует в точности одно значение  $Y$ . В этом случае говорят также, что атрибут  $X$  *функционально определяет* атрибут  $Y$  ( $X$  является *детерминантом (определителем)* для  $Y$ , а  $Y$  является зависимым от  $X$ ). Будем обозначать это как  $r.X \rightarrow r.Y$ . ■

Для примера будем использовать переменную отношения СЛУЖАЩИЕ\_ПРОЕКТЫ с заголовком {СЛУ\_НОМ, СЛУ\_ИМЯ, СЛУ\_ЗАРП, ПРО\_НОМ, ПРОЕКТ\_РУК} (возможное значение этой переменной показано на рис. 5.1). Очевидно, что если первичным ключом переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ является СЛУ\_НОМ, то для этого отношения справедлива, например, функциональная зависимость (Functional Dependency — FD)  $\text{СЛУ\_НОМ} \rightarrow \text{СЛУ\_ИМЯ}$ .

На самом деле, для значения переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ, показанного на рис. 5.1, выполняются еще и следующие FD (1):

$\text{СЛУ\_НОМ} \rightarrow \text{СЛУ\_ЗАРП}$   
 $\text{СЛУ\_НОМ} \rightarrow \text{ПРО\_НОМ}$   
 $\text{СЛУ\_НОМ} \rightarrow \text{ПРОЕКТ\_РУК}$   
 $\{\text{СЛУ\_НОМ}, \text{СЛУ\_ИМЯ}\} \rightarrow \text{СЛУ\_ЗАРП}$   
 $\{\text{СЛУ\_НОМ}, \text{СЛУ\_ИМЯ}\} \rightarrow \text{ПРО\_НОМ}$   
 $\{\text{СЛУ\_НОМ}, \text{СЛУ\_ИМЯ}\} \rightarrow \{\text{СЛУ\_ЗАРП}, \text{ПРО\_НОМ}\}$

...

$\text{ПРО\_НОМ} \rightarrow \text{ПРОЕКТ\_РУК}$  и т.д.

Поскольку имена всех служащих различны, то выполняются и такие FD (2):

$\text{СЛУ\_ИМЯ} \rightarrow \text{СЛУ\_НОМ}$   
 $\text{СЛУ\_ИМЯ} \rightarrow \text{СЛУ\_ЗАРП}$   
 $\text{СЛУ\_ИМЯ} \rightarrow \text{ПРО\_НОМ}$  и т.д.

Более того, для значения отношения на рис. 5.1 выполняется и FD (3):

$\text{СЛУ\_ЗАРП} \rightarrow \text{ПРО\_НОМ}$

Однако заметим, что природа FD группы (1) отличается от природы FD групп (2) и (3). Логично предположить, что идентификационные номера служащих должны быть всегда различны, а у каждого проекта имеется только один руководитель. Поэтому FD группы (1) должны быть верны для любого допустимого значения переменной отношения **СЛУЖАЩИЕ\_ПРОЕКТЫ** и могут рассматриваться как *инварианты*, или *ограничения целостности* этой переменной отношения.

| СЛУ_НОМ | СЛУ_ИМЯ   | СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК |
|---------|-----------|----------|---------|------------|
| 4434    | Иванов    | 22400.00 | 1       | Иванов     |
| 4435    | Петров    | 29000.00 | 1       | Иванов     |
| 4415    | Сидоров   | 23000.00 | 1       | Иванов     |
| 4436    | Федоров   | 20600.00 | 1       | Иванов     |
| 4440    | Иванова   | 22000.00 | 1       | Иванов     |
| 4441    | Сидоренко | 18000.00 | 2       | Иваненко   |
| 4416    | Федоренко | 20400.00 | 2       | Иваненко   |
| 4417    | Иваненко  | 21600.00 | 2       | Иваненко   |

Рис. 5.1. Возможное значение переменной отношения **СЛУЖАЩИЕ\_ПРОЕКТЫ**

FD группы (2) базируются на менее естественном предположении о том, что имена всех служащих различны. Это соответствует действительности для возможного значения отношения, показанного на рис. 5.1, но возможно, что с течением времени FD группы (2) не будут выполняться для какого-либо значения переменной отношения **СЛУЖАЩИЕ\_ПРОЕКТЫ**.

Наконец, FD группы (3) основана на совсем неестественном предположении, что никакие двое служащих, участвующие в разных проектах, не получают одинаковую заработную плату. Опять же, данное предположение верно для возможного значения отношения, показанного на рис. 5.1, но, скорее всего, это случайное совпадение.

В дальнейшем нас будут интересовать только те функциональные зависимости, которые должны выполняться для всех возможных значений переменных отношений.

Заметим, что если атрибут  $A$  переменной отношения  $r$  является *возможным ключом*, то для любого атрибута  $B$  этого отношения всегда выполняется  $FD\ A \rightarrow B$  (в группе (1) к этим FD относятся все FD, детерминантом которых является атрибут **СЛУ\_НОМ**). Обратите внимание, что наличие в отношении **СЛУЖАЩИЕ\_ПРОЕКТЫ**  $FD\ PRO\_НОМ \rightarrow ПРОЕКТ\_РУК$  приводит к некоторой *избыточности* этого отношения. Имя руководителя проекта является характеристикой проекта, а не служащего, но в нашем случае содержится в теле отношения столько раз, сколько служащих работает над проектом.

Итак, будем иметь дело с FD, которые выполняются для всех возможных состояний тела соответствующего отношения и могут рассматриваться как ограничения целостности. Как показывает (неполный) список (1), таких зависимостей может быть очень много. Поскольку они трактуются как ограничения целостности, за их соблюдением должна следить СУБД. Поэтому важно уметь сократить набор FD до минимума, поддержка которого гарантирует выполнение всех зависимостей (см. далее).

**Определение 5.2.**  $FD\ A \rightarrow B$  называется *тривиальной*, если  $A \supseteq B$  (т.е. множество атрибутов  $A$  включает множество  $B$  или совпадает с множеством  $B$ ). ■

Очевидно, что любая тривиальная FD всегда выполняется. Например, в отношении **СЛУЖАЩИЕ\_ПРОЕКТЫ** всегда выполняется  $FD\ \{СЛУ\_ЗАРП, PRO\_НОМ\} \rightarrow СЛУ\_ЗАРП$ . Частным случаем тривиальной FD является  $A \rightarrow A$ .

Поскольку тривиальные FD выполняются всегда, их нельзя трактовать как ограничения целостности, и поэтому они не пред-

ставляют интереса с практической точки зрения. Однако в теоретических рассуждениях их наличие необходимо учитывать.

**Определение 5.3.** *Замыканием* множества FD  $S$  является множество FD  $S^+$ , включающее все FD, логически выводимые из FD множества  $S$ . ■

Для начала приведем два примера FD, из которых следуют (или *выводятся*) другие FD. Будем снова пользоваться отношением СЛУЖАЩИЕ\_ПРОЕКТЫ. Для этого отношения выполняется, например, FD СЛУ\_НОМ  $\rightarrow$  {СЛУ\_ЗАРП, ОТД\_НОМ}. Из нее выводятся FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП и СЛУ\_НОМ  $\rightarrow$  ОТД\_НОМ.

В отношении СЛУЖАЩИЕ\_ПРОЕКТЫ имеется также пара FD СЛУ\_НОМ  $\rightarrow$  ОТД\_НОМ и ОТД\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК. Из них выводится FD СЛУ\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК. FD вида СЛУ\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК называются *транзитивными*, поскольку ПРОЕКТ\_РУК зависит от СЛУ\_НОМ «транзитивно», через атрибут ПРО\_НОМ.

**Определение 5.4.** FD  $A \rightarrow C$  называется *транзитивной*, если существует такой атрибут  $B$ , что имеются функциональные зависимости  $A \rightarrow B$  и  $B \rightarrow C$  и отсутствует функциональная зависимость  $C \rightarrow A$ . ■

**Аксиомы Армстронга.** Подход к решению проблемы поиска замыкания  $S^+$  множества FD  $S$  впервые предложил Вильям Армстронг\*. Им был предложен набор *правил вывода* новых FD из существующих (эти правила обычно называют *аксиомами Армстронга*, хотя справедливость правил доказывается на основе определения FD). Обычно принято формулировать эти правила вывода в следующей форме. Пусть  $A$ ,  $B$  и  $C$  являются (в общем случае, составными) атрибутами переменной отношения  $r$ . Множества  $A$ ,  $B$  и  $C$  могут иметь непустое пересечение. Для краткости будем обозначать через  $AB$   $A \cup B$ . Тогда для любого значения  $r$ :

- 1) если  $B \subseteq A$ , то выполняется FD  $A \rightarrow B$  (*аксиома рефлексивности*);
- 2) если выполняется FD  $A \rightarrow B$ , то выполняется и FD  $AC \rightarrow BC$  (*аксиома пополнения*);
- 3) если выполняются FD  $A \rightarrow B$  и  $B \rightarrow C$ , то выполняется и FD  $A \rightarrow C$  (*аксиома транзитивности*).

---

\* К сожалению, классическая статья В. Армстронга — *Armstrong W. W. Dependency Structures of Data Base Relationships // Proc. IFIP Congress, Stockholm, Sweden, 1974* — так и не переведена на русский язык (на самом деле, ее нелегко найти и в оригинале). Поэтому я не могу рекомендовать ее для дополнительного чтения, хотя обязан сослаться.

Докажем истинность этих «аксиом». Истинность первой аксиомы Армстронга следует из того, что при  $B \subseteq A$   $FD A \rightarrow B$  является тривиальной.

Справедливость второй аксиомы докажем от противного. Предположим, что  $FD AC \rightarrow BC$  не соблюдается. Это означает, что в некотором допустимом теле значения переменной отношения  $r$  найдутся два кортежа  $t_1$  и  $t_2$ , такие, что  $t_1 \{AC\} = t_2 \{AC\}$  (а), но  $t_1 \{BC\} \neq t_2 \{BC\}$  (б) (здесь  $t \{A\}$  обозначает проекцию кортежа  $t$  на множество атрибутов  $A$ ). По аксиоме рефлексивности из равенства (а) следует, что  $t_1 \{A\} = t_2 \{A\}$ . Поскольку имеется  $FD A \rightarrow B$ , должно соблюдаться равенство  $t_1 \{B\} = t_2 \{B\}$ . Тогда из неравенства (б) следует, что  $t_1 \{C\} \neq t_2 \{C\}$ , что противоречит наличию тривиальной  $FD AC \rightarrow C$ . Следовательно, предположение об отсутствии  $FD AC \rightarrow BC$  не является верным, и справедливость второй аксиомы доказана.

Аналогично докажем истинность третьей аксиомы Армстронга. Предположим, что  $FD A \rightarrow C$  не соблюдается. Это означает, что в некотором допустимом теле отношения найдутся два кортежа  $t_1$  и  $t_2$ , такие что  $t_1 \{A\} = t_2 \{A\}$ , но  $t_1 \{C\} \neq t_2 \{C\}$ . Но из наличия  $FD A \rightarrow B$  следует, что  $t_1 \{B\} = t_2 \{B\}$ , а потому из наличия  $FD B \rightarrow C$  следует, что  $t_1 \{C\} = t_2 \{C\}$ . Следовательно, предположение об отсутствии  $FD A \rightarrow C$  не является верным, и справедливость третьей аксиомы и утверждения в целом доказана. ■

Можно доказать, что система правил вывода Армстронга *полна и совершенна* (*sound and complete*) в том смысле, что для данного множества  $FD S$  любая  $FD$ , потенциально выводимая из  $S$ , может быть выведена на основе аксиом Армстронга, и применение этих аксиом не может привести к выводу лишней  $FD$ . Тем не менее Дейт по практическим соображениям предложил расширить базовый набор правил вывода еще пятью правилами, для которых сразу покажем, что они выводятся из исходных аксиом Армстронга:

4) для любого атрибута  $A$  выполняется  $FD A \rightarrow A$  (*аксиома самодетерминированности*) — прямо следует из аксиомы рефлексивности (1);

5) если выполняется  $FD A \rightarrow BC$ , то выполняются и  $FD A \rightarrow B$  и  $A \rightarrow C$  (*аксиома декомпозиции*) — из аксиомы рефлексивности (1) следует, что выполняется  $FD BC \rightarrow B$ ; из аксиомы транзитивности (3) следует, что выполняется  $FD A \rightarrow B$ ; аналогично, из аксиомы рефлексивности (1) следует, что выполняется  $FD BC \rightarrow C$ , а из аксиомы транзитивности (3) следует, что выполняется  $FD A \rightarrow C$ ;

6) если выполняются  $FD A \rightarrow B$  и  $A \rightarrow C$ , то выполняется и  $FD A \rightarrow BC$  (*аксиома объединения*) — из аксиомы пополнения

(2) следует, что выполняется  $FD A \rightarrow AB$  и  $AB \rightarrow BC$ ; из аксиомы транзитивности (3) следует, что  $A \rightarrow BC$ ;

7) если выполняются  $FD A \rightarrow B$  и  $C \rightarrow D$ , то выполняется и  $FD AC \rightarrow BD$  (аксиома композиции) — из аксиомы пополнения (2) следует, что выполняются  $FD AC \rightarrow BC$  и  $BC \rightarrow BD$ ; из аксиомы транзитивности (3) следует, что выполняется  $FD AC \rightarrow BD$ ;

8) если выполняются  $FD A \rightarrow BC$  и  $B \rightarrow D$ , то выполняется и  $FD A \rightarrow BCD$  (аксиома накопления) — из аксиомы пополнения (2) следует, что выполняется  $FD BC \rightarrow BCD$ ; из аксиомы транзитивности (3) следует, что выполняется  $FD A \rightarrow BCD$ . ■

**Определение 5.5.** Пусть заданы переменная отношения  $r$ , множество  $Z$  атрибутов этого отношения (подмножество  $H_r$ , или составной атрибут  $r$ ) и некоторое множество  $FD S$ , выполняемых для  $r$ . Тогда **замыканием  $Z$  над  $S$**  называется наибольшее множество  $Z^+$  таких атрибутов  $Y$  отношения  $r$ , что  $FD Z \rightarrow Y \notin S^+$ . ■

Алгоритм вычисления  $Z^+$  очень прост. Один из его вариантов показан на рис. 5.2.

Докажем корректность алгоритма по индукции. На нулевом шаге  $Z[0] = Z$ , и  $FD Z \rightarrow Z[k]$ , очевидно, принадлежит  $S^+$  (тривиальная  $FD$  «выводится» из любого множества  $FD$ ). Пусть для некоторого  $k$  выполняется  $FD Z \rightarrow Z[k]$ , и пусть мы нашли в  $S$  такую  $FD A \rightarrow B$ , что  $A \subseteq Z[k]$ . Тогда можно представить  $Z[k]$  в виде  $AC$ , и, следовательно, выполняется  $FD Z \rightarrow AC$ .

Но по аксиоме накопления (8) тогда выполняется  $FD Z \rightarrow ACB$ , т.е.  $FD Z \rightarrow (Z[k] \text{ UNION } B)$  входит во множество  $S^+$ , что переводит нас на следующий шаг индукции. Очевидно, что, поскольку множество атрибутов отношения конечно, то на некотором шаге алгоритм завершит свою работу. ■

Продemonстрируем работу алгоритма на примере. Пусть имеется отношение с заголовком  $\{A, B, C, D, E, F\}$  и заданным множеством  $FD S = \{A \rightarrow D, AB \rightarrow E, BF \rightarrow E, CD \rightarrow F, E \rightarrow C\}$ . Пусть требуется найти  $\{AE\}^+$  над  $S$ . На первом проходе тела

```

k := 0; Z[0] := Z;
DO
k := k+1;
Z[k] := Z[k-1];
FOR EACH FD A → B IN S DO
IF A ⊆ Z[k] THEN Z[k] := (Z[k] UNION B) END DO;
UNTIL Z[k] = Z[k-1];
Z+ := Z[k];

```

Рис. 5.2. Алгоритм построения замыкания атрибутов над заданным множеством  $FD$



цикла DO  $Z[1]$  равно  $AE$ . В теле цикла FOR EACH будут найдены  $FD A \rightarrow D$  и  $E \rightarrow C$ , и в конце цикла  $Z[1]$  станет равным  $ACDE$ . На втором проходе тела цикла DO при  $Z[2]$ , равном  $ACDE$ , в теле цикла FOR EACH будет найдена  $FD CD \rightarrow F$ , и в конце цикла  $Z[2]$  станет равным  $ACDEF$ . Следующий проход тела цикла DO не изменит  $Z[3]$ , и  $Z^* (\{AE\}^+)$  будет равно  $ACDEF$ .

Алгоритм построения замыкания множества атрибутов  $Z$  над заданным множеством  $FD S$  помогает легко установить, входит ли заданная  $FD Z \rightarrow B$  в замыкание  $S^*$ . Очевидно, что необходимым и достаточным условием для этого является  $B \subseteq Z^*$ , т.е. вхождение составного атрибута  $B$  в замыкание  $Z^*$ .

**Определение 5.6.** *Суперключом* переменной отношения  $r$  называется любое подмножество  $K$  заголовка  $H_r$ , включающее, по меньшей мере, хотя бы один возможный ключ  $r$ . ■

Одно из следствий этого определения состоит в том, что подмножество  $K$  заголовка  $H_r$  является суперключом тогда и только тогда, когда для любого атрибута  $A$  (возможно, составного) из заголовка отношения  $r$  выполняется  $FD K \rightarrow A$ . В терминах замыкания множества атрибутов  $K$  является суперключом тогда и только тогда, когда  $K^*$  совпадает с  $H_r$ .

**Определение 5.7.** Множество  $FD S_2$  называется *покрытием множества*  $FD S_1$ , если любая  $FD$ , выводимая из  $S_1$ , выводится также и из  $S_2$ . ■

Легко заметить, что  $S_2$  является покрытием  $S_1$  тогда и только тогда, когда  $S_1^+ \subseteq S_2^+$ . Два множества  $FD S_1$  и  $S_2$  называются *эквивалентными*, если каждое из них является покрытием другого, т.е.  $S_1^+ = S_2^+$ .

**Определение 5.8.** Множество  $FD S$  называется *минимальным* в том и только том случае, когда удовлетворяет следующим свойствам:

- правая часть любой  $FD$  из  $S$  является множеством из одного атрибута (простым атрибутом);
- детерминант каждой  $FD$  из  $S$  обладает свойством *минимальности*, т.е. удаление любого атрибута из детерминанта приводит к изменению замыкания  $S^+$ , т.е. порождению множества  $FD$ , не эквивалентного  $S^{**}$ ;

---

\* Мы используем здесь обозначения операций проверки включения множеств, что не совсем корректно, поскольку если, например, множество  $B$  состоит из одного элемента, то для его обозначения используется имя соответствующего атрибута, и в этом случае правильнее было бы использовать символ « $\notin$ » (проверка вхождения элемента во множество).

\*\*  $FD$  с минимальным детерминантом называется *минимальной слева*.

- удаление любой FD из  $S$  приводит к изменению  $S^+$ , т.е. порождению множества FD, не эквивалентного  $S$ . ■

Чтобы продемонстрировать минимальные и не минимальные множества FD, вернемся к примеру переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ {СЛУ\_НОМ, СЛУ\_ИМЯ, СЛУ\_ЗАРП, ПРО\_НОМ, ПРОЕКТ\_РУК}, возможное значение которой показано на рис. 5.1. Если считать, что единственным возможным ключом этого отношения является атрибут СЛУ\_НОМ, то множество FD {СЛУ\_НОМ  $\rightarrow$  СЛУ\_ИМЯ, СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП, СЛУ\_НОМ  $\rightarrow$  ПРО\_НОМ, ПРО\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК} будет минимальным. Действительно, в правых частях FD этого множества находятся множества, состоящие ровно из одного атрибута; каждый из детерминантов тоже является множеством из одного атрибута, удаление которого, очевидно, недопустимо; удаление каждой FD явно приводит к изменению замыкания множества FD, поскольку утрачиваемая информация не выводится с помощью аксиом Армстронга.

С другой стороны, множества FD

(a) {СЛУ\_НОМ  $\rightarrow$  {СЛУ\_ИМЯ, СЛУ\_ЗАРП}, СЛУ\_НОМ  $\rightarrow$  ПРО\_НОМ, СЛУ\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК, ПРО\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК};

(b) {СЛУ\_НОМ  $\rightarrow$  СЛУ\_ИМЯ, {СЛУ\_НОМ, СЛУ\_ИМЯ}  $\rightarrow$  СЛУ\_ЗАРП, СЛУ\_НОМ  $\rightarrow$  ПРО\_НОМ, СЛУ\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК, ПРО\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК};

(c) {СЛУ\_НОМ  $\rightarrow$  СЛУ\_НОМ, СЛУ\_НОМ  $\rightarrow$  СЛУ\_ИМЯ, СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП, СЛУ\_НОМ  $\rightarrow$  ПРО\_НОМ, СЛУ\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК, ПРО\_НОМ  $\rightarrow$  ПРОЕКТ\_РУК}

не являются минимальными. Для множества (a) в правой части первой FD присутствует множество из двух элементов. Для множества (b) удаление атрибута СЛУ\_ИМЯ из детерминанта FD {СЛУ\_НОМ, СЛУ\_ИМЯ}  $\rightarrow$  СЛУ\_ЗАРП не меняет замыкание множества FD. Для множества (c) удаление FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_НОМ не приводит к изменению замыкания. Эти примеры показывают, что для определения минимальности множества FD не всегда требуется явное построение замыкания данного множества.

Интересным и важным является тот факт, что для любого множества FD  $S$  существует (и даже может быть построено) эквивалентное ему минимальное множество  $S^-$ .

Приведем общую схему построения  $S^-$  по заданному множеству FD  $S$ . Во-первых, используя аксиому декомпозиции, мы можем привести множество  $S$  к эквивалентному множеству FD

$S_1$ , правые части FD которого содержат только одноэлементные множества (простые атрибуты). Далее, для каждой FD из  $S_1$ , детерминант  $L \{L_1, L_2, \dots, L_n\}$  которой содержит более одного атрибута, будем пытаться удалять атрибуты  $L_i$  ( $i = 1, 2, \dots, n$ ), получая множество FD  $S_2$ . Если после удаления атрибута  $L_i$  множество  $S_2$  оказывается эквивалентным множеству  $S_1$ , то этот атрибут удаляется и пробуется следующий атрибут. Назовем  $S_3$  множество FD, полученное путем допустимого удаления атрибутов из всех детерминантов FD множества  $S_1$ . Наконец, для каждой FD  $f$  из множества  $S_3$  будем проверять эквивалентность множеств  $S_3$  и  $S_3 \text{ MINUS } \{f\}$ . Если эти множества оказываются эквивалентными, удалим  $f$  из множества  $S_3$ , и в заключение получим множество  $S_4$ , которое минимально и по построению эквивалентно исходному множеству FD  $S$ . ■

Пусть, например, имеется переменная отношения  $r$  с заголовком  $\{A, B, C, D\}$  и задано множество FD  $S = \{A \rightarrow B, A \rightarrow BC, AB \rightarrow C, AC \rightarrow D, B \rightarrow C\}$ . По аксиоме декомпозиции (5)  $S$  эквивалентно множеству  $\{A \rightarrow B, A \rightarrow C, AB \rightarrow C, AC \rightarrow D, B \rightarrow C\}$ . В детерминанте FD  $AC \rightarrow D$  можно удалить атрибут  $C$ , поскольку по аксиоме пополнения (2) из наличия FD  $A \rightarrow C$  следует наличие FD  $A \rightarrow AC$ ; по аксиоме транзитивности выводится FD  $A \rightarrow D$ , и поэтому атрибут  $C$  в детерминанте FD  $AC \rightarrow D$  является избыточным. FD  $AB \rightarrow C$  может быть удалена, поскольку может быть выведена из FD  $A \rightarrow C$  (по аксиоме пополнения из этой FD выводится FD  $AB \rightarrow BC$ , а по аксиоме декомпозиции далее выводится  $AB \rightarrow C$ ). Наконец, FD  $A \rightarrow C$  тоже выводится по аксиоме транзитивности из FD  $A \rightarrow B$  и  $B \rightarrow C$ . Таким образом, мы получаем множество FD  $\{A \rightarrow B, A \rightarrow D, B \rightarrow C\}$ , которое является минимальным и эквивалентно  $S$  по построению.

**Определение 5.9.** *Минимальным покрытием множества FD  $S$  называется любое минимальное множество FD  $S_1$ , эквивалентное  $S$ .* ■

Поскольку для каждого множества FD существует эквивалентное минимальное множество FD, у каждого множества FD имеется хотя бы одно минимальное покрытие, причем для его нахождения не обязательно находить замыкание исходного множества.

### 5.1.2. Декомпозиция без потерь и функциональные зависимости

Как уже отмечалось, начиная с подразд. 5.2, будем обсуждать подход к проектированию реляционных баз данных на основе

нормализации, т. е. декомпозиции (разбиения путем проецирования), отношения, находящегося в предыдущей нормальной форме, на два или более отношений, удовлетворяющих требованиям следующей нормальной формы. Считаются правильными такие декомпозиции отношения, которые являются обратимыми, т. е. имеется возможность собрать исходное отношение из декомпозированных отношений без потери информации. Такие декомпозиции называются *декомпозициями без потерь*.

**Определение 5.10.** Декомпозиция путем проецирования отношения  $R$  на отношения  $R_1, R_2, \dots, R_n$  называется *декомпозицией без потерь*, если  $R = R_1 \text{ NATURAL JOIN } R_2 \dots \text{ NATURAL JOIN } R_n$ . ■

**Корректные и некорректные декомпозиции отношений.** На рис. 5.3 приведены две возможные декомпозиции отношения СЛУЖАЩИЕ\_ПРОЕКТЫ (для экономии места мы сократили и слегка изменили значение отношения, показанное на рис. 5.1).

Анализ рис. 5.3 показывает, что в случае декомпозиции (1) мы не потеряли информацию о служащих — про каждого из них можно узнать имя, размер заработной платы, номер выполняемого проекта и имя руководителя проекта. Вторая декомпозиция не дает возможности получить данные о проекте служащего, поскольку Иванов и Иваненко получают одинаковую заработную плату; следовательно, эта декомпозиция приводит к потере информации. Что же привело к тому, что одна декомпозиция является декомпозицией без потерь, а вторая — нет?

Заметим, что при выполнении декомпозиции была использована операция взятия проекции. Каждое из значений отношений СЛУЖ, СЛУ\_ПРО и ЗАРП\_ПРО является проекцией исходного значения отношения СЛУЖАЩИЕ\_ПРОЕКТЫ. В случае декомпозиции (1) отсутствие потери информации означает, что в результате естественного соединения значений отношений СЛУЖ и СЛУ\_ПРО будет гарантированно получено значение отношения, заголовков и тело которого совпадают с заголовком и телом значения отношения СЛУЖАЩИЕ\_ПРОЕКТЫ. Следует отметить, что это произойдет для любых допустимых (и согласованных) значений переменных отношений СЛУЖАЩИЕ\_ПРОЕКТЫ, СЛУЖ и СЛУ\_ПРО, поскольку у всех этих переменных атрибут СЛУ\_НОМ является возможным ключом. Однако если выполнить естественное соединение значений отношений СЛУ и ЗАРП\_ПРО, то будет получено значение отношения, показанное на рис. 5.4.

Схема этого отношения, естественно (поскольку соединение естественное), совпадает со схемой отношения СЛУЖАЩИЕ\_ПРО-

# СЛУЖАЩИЕ\_ПРОЕКТЫ

| СЛУ_НОМ | СЛУ_ИМЯ  | СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК |
|---------|----------|----------|---------|------------|
| 4434    | Иванов   | 22400.00 | 1       | Иванов     |
| 4417    | Иваненко | 22400.00 | 2       | Иваненко   |

Декомпозиция (1). Отношения СЛУЖ и СЛУ\_ПРО

| СЛУ_НОМ | СЛУ_ИМЯ  | СЛУ_ЗАРП |
|---------|----------|----------|
| 4434    | Иванов   | 22400.00 |
| 4417    | Иваненко | 22400.00 |

| СЛУ_НОМ | ПРО_НОМ | ПРОЕКТ_РУК |
|---------|---------|------------|
| 4434    | 1       | Иванов     |
| 4417    | 2       | Иваненко   |

Декомпозиция (2). Отношения СЛУЖ и ЗАРП\_ПРО

| СЛУ_НОМ | СЛУ_ИМЯ  | СЛУ_ЗАРП |
|---------|----------|----------|
| 4434    | Иванов   | 22400.00 |
| 4417    | Иваненко | 22400.00 |

| СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК |
|----------|---------|------------|
| 22400.00 | 1       | Иванов     |
| 22400.00 | 2       | Иваненко   |

Рис. 5.3. Две возможные декомпозиции отношения СЛУЖАЩИЕ\_ПРОЕКТЫ

| СЛУ_НОМ | СЛУ_ИМЯ  | СЛУ_ЗАРП | ПРО_НОМ | ПРОЕКТ_РУК |
|---------|----------|----------|---------|------------|
| 4434    | Иванов   | 22400.00 | 1       | Иванов     |
| 4417    | Иваненко | 22400.00 | 2       | Иваненко   |
| 4434    | Иванов   | 22400.00 | 2       | Иваненко   |
| 4417    | Иваненко | 22400.00 | 1       | Иванов     |

Рис. 5.4. Результат естественного соединения значений отношений СЛУ и ЗАРП\_ПРО

ЕКТЫ, но в теле появились лишние кортежи, наличие которых и приводит к утрате исходной информации. Интуитивно понятно, что это происходит потому, что в отношении ЗАРП\_ПРО отсутствуют функциональные зависимости СЛУ\_ЗАРП  $\rightarrow$  ПРО\_НОМ и СЛУ\_ЗАРП  $\rightarrow$  ПРОЕКТ\_РУК, но точнее причину потери информации в данном случае мы объясним несколько позже. Корректность же декомпозиции 1 следует из теоремы Хита.

**Теорема Хита.** Пусть задана переменная отношения  $r$  с заголовком  $\{A, B, C\}$  ( $A, B$  и  $C$ , в общем случае, являются составными атрибутами), и для любого значения  $r$  выполняется  $FD\ A \rightarrow B$ . Тогда  $r = (r\ PROJECT\ \{A, B\})\ NATURAL\ JOIN\ (r\ PROJECT\ \{A, C\})$ .

*Доказательство.* Пусть  $R$  — некоторое произвольное значение переменной  $r$ . Обозначим результат операции  $R\ PROJECT\ \{A, B\}$  через  $R_1$ , результат операции  $R\ PROJECT\ \{A, C\}$  через  $R_2$ , а результат естественного соединения  $R_1\ NATURAL\ JOIN\ R_2$  через  $R_3$ . Прежде всего докажем, что в  $B_{R_3}$  содержатся все кортежи, содержащиеся в  $B_R$ . Действительно, пусть для некоторых значений  $a, b$  и  $c$  кортеж  $\{<A, a>, <B, b>, <C, c>\} \in B_R$ . Тогда по определению операции взятия проекции  $\{<A, a>, <B, b>\} \in B_{R_1}$  и  $\{<A, a>, <C, c>\} \in B_{R_2}$ . Следовательно,  $\{<A, a>, <B, b>, <C, c>\} \in B_{R_3}$ .

Теперь докажем, что в теле результата естественного соединения нет лишних кортежей, т.е. что если кортеж  $\{<A, a>, <B, b>, <C, c>\} \in B_{R_3}$ , то  $\{<A, a>, <B, b>, <C, c>\} \in B_R$ . Действительно, если  $\{<A, a>, <B, b>, <C, c>\} \in B_{R_3}$ , то существуют кортежи  $\{<A, a>, <B, b>\} \in B_{R_1}$  и  $\{<A, a>, <C, c>\} \in B_{R_2}$ . Последнее условие может выполняться в том и только том случае, когда существует такое значение  $b_1$ , что кортеж  $\{<A, a>, <B, b_1>, <C, c>\} \in B_R$ . Но поскольку выполняется  $FD\ A \rightarrow B$ , то  $b = b_1$  и, следовательно,  $\{<A, a>, <B, b>, <C, c>\} = \{<A, a>, <B, b_1>, <C, c>\}$ . Тем самым, теорема Хита полностью доказана. ■

Для иллюстрации общего случая применения теоремы Хита рассмотрим отношение СЛУЖАЩИЕ\_ОТДЕЛЫ\_ПРОЕКТЫ с заголовком  $\{СЛУ\_НОМ, СЛУ\_ОТД, ПРО\_НОМ\}$  (возможное значение показано на рис. 5.5). Атрибут СЛУ\_ОТД содержит номера отделов, в которых работают служащие, а ПРО\_НОМ — номера проектов, в которых служащие принимают участие. Каждый служащий работает только в одном отделе, т.е. имеется  $FD\ СЛУ\_НОМ \rightarrow СЛУ\_ОТД$ , но один служащий может участвовать в нескольких проектах.

В отношении СЛУЖАЩИЕ\_ОТДЕЛЫ\_ПРОЕКТЫ атрибут СЛУ\_НОМ не является возможным ключом, но, как показано на рис. 5.5, наличия  $FD\ СЛУ\_НОМ \rightarrow СЛУ\_ОТД$  оказывается достаточно для декомпозиции этого отношения без потерь.

# СЛУЖАЩИЕ\_ОТДЕЛЫ\_ПРОЕКТЫ

| СЛУ_НОМ | СЛУ_ОТД | ПРО_НОМ |
|---------|---------|---------|
| 4434    | 625     | 1       |
| 4434    | 625     | 2       |
| 4417    | 636     | 1       |
| 4417    | 636     | 2       |

Декомпозиция: отношения СЛУЖ\_ОТДЕЛЫ и СЛУЖ\_ПРОЕКТЫ

| СЛУ_НОМ | СЛУ_ОТД |
|---------|---------|
| 4434    | 625     |
| 4417    | 636     |

| СЛУ_НОМ | ПРО_НОМ |
|---------|---------|
| 4434    | 1       |
| 4434    | 2       |
| 4417    | 1       |
| 4417    | 2       |

Рис. 5.5. Декомпозиция без потерь по теореме Хита

Для дальнейшего изложения нам потребуется ввести еще одно определение и сделать пару замечаний.

**Определение 5.11.** Атрибут *B* *минимально зависит* от атрибута *A*, если выполняется минимальная слева FD  $A \rightarrow B$ . ■

Например, в отношении СЛУЖАЩИЕ\_ПРОЕКТЫ (см. рис. 5.1) выполняются FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП и {СЛУ\_НОМ, СЛУ\_ИМЯ}  $\rightarrow$  СЛУ\_ЗАРП. Первая FD является минимальной слева, а вторая — нет. Поэтому СЛУ\_ЗАРП минимально зависит от СЛУ\_НОМ, а для {СЛУ\_НОМ, СЛУ\_ИМЯ} свойство минимальной зависимости не выполняется.

**Диаграммы функциональных зависимостей.** Для иллюстраций в следующих подразделах нам пригодятся диаграммы FD, с помощью которых можно наглядно представлять минимальные множества FD. Например, на рис. 5.6 приведена диаграмма минимального множества FD отношения СЛУЖАЩИЕ\_ПРОЕКТЫ.

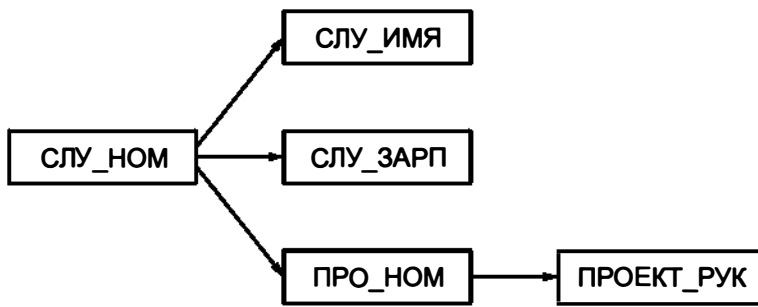


Рис. 5.6. Диаграмма минимального множества FD отношения

В левой части диаграммы все стрелки начинаются с атрибута СЛУ\_НОМ, который является единственным возможным (и, следовательно, первичным) ключом отношения СЛУЖАЩИЕ\_ПРОЕКТЫ. Обратите внимание на отсутствие стрелки от СЛУ\_НОМ к ПРОЕКТ\_РУК. Конечно, поскольку СЛУ\_НОМ является возможным ключом, должна выполняться и FD СЛУ\_НОМ → ПРОЕКТ\_РУК. Но эта FD является транзитивной (через ПРО\_НОМ) и поэтому не входит в минимальное множество FD. Заметим, что в процессе нормализации, к рассмотрению которого мы приступим в следующем подразделе, из диаграмм множества FD удаляются стрелки, начинающиеся не от возможных ключей.

## 5.2. Минимальные функциональные зависимости и вторая нормальная форма

Пусть имеется переменная отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ с заголовком {СЛУ\_НОМ, СЛУ\_УРОВ, СЛУ\_ЗАРП, ПРО\_НОМ, СЛУ\_ЗАДАН}. Новые атрибуты СЛУ\_УРОВ и СЛУ\_ЗАДАН содержат, соответственно, данные о разряде служащего и о задании, которое выполняет служащий в данном проекте. Будем считать, что разряд служащего определяет размер его заработной платы и что каждый служащий может участвовать в нескольких проектах, но в каждом проекте выполняет только

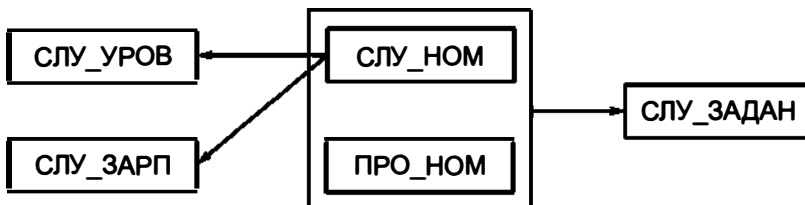


Рис. 5.7. Диаграмма множества FD отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ



| СЛУ_НОМ | СЛУ_УРОВ | СЛУ_ЗАРП | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|----------|----------|---------|-----------|
| 4434    | 2        | 22400.00 | 1       | A         |
| 4435    | 3        | 29600.00 | 1       | B         |
| 4415    | 1        | 20000.00 | 1       | C         |
| 4436    | 1        | 20000.00 | 1       | D         |
| 4434    | 2        | 22400.00 | 2       | D         |
| 4435    | 3        | 29600.00 | 2       | C         |
| 4415    | 1        | 20000.00 | 2       | B         |
| 4436    | 1        | 20000.00 | 2       | A         |

Рис. 5.8. Возможное значение переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ

одно задание. Тогда очевидно, что единственно возможным ключом отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ является составной атрибут {СЛУ\_НОМ, ПРО\_НОМ}. Диаграмма минимального множества FD показана на рис. 5.7, а возможное значение отношения — на рис. 5.8.

### 5.2.1. Аномалии обновления, возникающие из-за наличия неминимальных функциональных зависимостей

В множество FD отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ входит несколько FD, в которых детерминантом является не возможный ключ отношения (соответствующие стрелки в диаграмме начинаются не с {СЛУ\_НОМ, ПРО\_НОМ}, т. е. некоторые функциональные зависимости атрибутов от возможного ключа не являются минимальными). Это приводит к появлению так называемых *аномалий обновления*. Под аномалиями обновления понимаются трудности, с которыми приходится сталкиваться при выполнении операций добавления кортежей в отношение (INSERT), удаления кортежей (DELETE) и модификации кортежей (UPDATE).

Обсудим сначала аномалии обновления, вызываемые наличием FD СЛУ\_НОМ → СЛУ\_УРОВ. Эти аномалии связаны с избыточностью хранения значений атрибутов СЛУ\_УРОВ и СЛУ\_ЗАРП в каждом кортеже, описывающем задание служащего в некотором проекте.

- *Добавление кортежей.* Невозможно дополнить отношение СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ данными о служащем, кото-

рый в данное время еще не участвует ни в одном проекте (ПРО\_НОМ является частью первичного ключа и не может содержать неопределенных значений). Между тем часто бывает, что сначала служащего принимают на работу, устанавливают его разряд и размер заработной платы, а лишь потом назначают для него проект.

- *Удаление кортежей.* Невозможно сохранить в отношении СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ данные о служащем, завершившем участие в своем последнем проекте (по той причине, что значение атрибута ПРО\_НОМ для этого служащего стало бы неопределенным). Между тем характерна ситуация, когда между проектами возникают перерывы, не приводящие к увольнению служащих.
- *Модификация кортежей.* Чтобы изменить разряд служащего, придется модифицировать все кортежи с соответствующим значением атрибута СПУ\_НОМ. В противном случае будет нарушена естественная FD СПУ\_НОМ → СПУ\_УРОВ (у одного служащего имеется только один разряд).

### 5.2.2. Возможная декомпозиция

Для преодоления трудностей, перечисленных в подразд. 5.2.1, можно произвести декомпозицию переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ на две переменных отношений — СЛУЖ с заголовком {СПУ\_НОМ, СПУ\_УРОВ, СПУ\_ЗАРП} и СЛУЖ\_ПРО\_ЗАДАН с заголовком {СПУ\_НОМ, ПРО\_НОМ, СПУ\_ЗАДАН}. На основании теоремы Хита эта декомпозиция является декомпозицией без потерь, поскольку в исходном отношении имела FD {СПУ\_НОМ, ПРО\_НОМ} → СПУ\_ЗАДАН. На рис. 5.9 показаны диаграммы множеств FD этих отношений, а на рис. 5.10 — их значения, соответствующие возможному значению переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ с рис. 5.8.

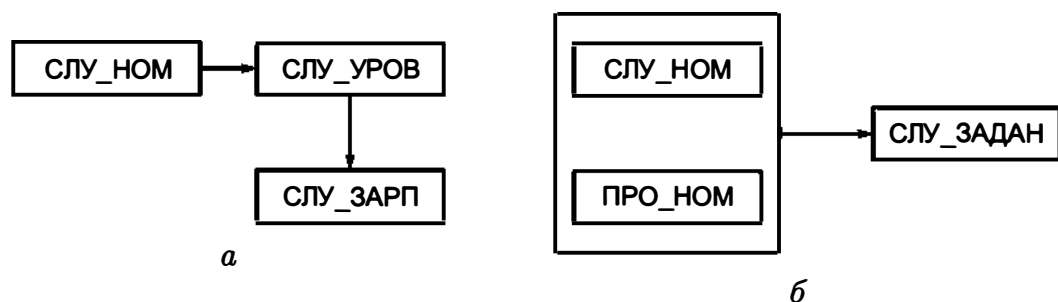


Рис. 5.9. Диаграммы FD в переменных отношениях СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН

СЛУЖ

| СЛУ_НОМ | СЛУ_УРОВ | СЛУ_ЗАРП |
|---------|----------|----------|
| 4434    | 2        | 22400.00 |
| 4435    | 3        | 29600.00 |
| 4415    | 1        | 20000.00 |
| 4436    | 1        | 20000.00 |

СЛУЖ\_ПРО\_ЗАДАН

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 4434    | 1       | A         |
| 4435    | 1       | B         |
| 4415    | 1       | C         |
| 4436    | 1       | D         |
| 4434    | 2       | D         |
| 4435    | 2       | C         |
| 4415    | 2       | B         |
| 4436    | 2       | A         |

Рис. 5.10. Значения переменных отношений СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН

Теперь можно легко справиться с операциями обновления:

- *добавление кортежей.* Чтобы сохранить данные о принятом на работу служащем, который еще не участвует ни в каком проекте, достаточно добавить соответствующий кортеж в отношение СЛУЖ;
- *удаление кортежей.* Если некоторый служащих прекращает работу в некотором проекте, достаточно удалить соответствующий кортеж из отношения СЛУЖ\_ПРО\_ЗАДАН. При увольнении служащего нужно удалить кортежи с соответствующим значением атрибута СЛУ\_НОМ из отношений СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН;
- *модификация кортежей.* Если у служащего меняется разряд (и, следовательно, размер заработной платы), достаточно модифицировать один кортеж в отношении СЛУЖ.

### 5.2.3. Вторая нормальная форма

Как показывает рис. 5.9, в отношениях СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН отсутствуют FD, не являющиеся *минимальными*. Наличие таких FD в переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ вызывало аномалии обновления. В действительности проблема заключалась в том, что атрибут СЛУЖ\_УРОВ относился к сущности *служащий*, в то время как первичный ключ идентифицировал сущность *задание\_служащего\_в\_проекте*.

**Определение 5.12.** Переменная отношения находится во *второй нормальной форме* (2NF) тогда и только тогда, когда она находится в первой нормальной форме, и каждый ее неключевой атрибут<sup>\*</sup> минимально функционально зависит от первичного ключа<sup>\*\*</sup>. ■

Переменные отношений СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН находятся в 2NF (все неключевые атрибуты отношений минимально зависят от первичных ключей СЛУ\_НОМ и {СЛУ\_НОМ, ПРО\_НОМ} соответственно). Переменная отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ не находится в 2NF (например, FD {СЛУ\_НОМ, ПРО\_НОМ} → СЛУ\_УРОВ не является минимальной). Любая переменная отношения, находящаяся в 1NF, но не находящаяся в 2NF, может быть приведена к набору из переменных отношений, находящихся в 2NF.

В результате декомпозиции мы получаем набор проекций исходной переменной отношения, естественное соединение любых значений которых воспроизводит допустимое значение исходной переменной отношения (т. е. это декомпозиция без потерь). Для переменных отношений СЛУЖ и СЛУЖ\_ПРО\_ЗАДАН исходное отношение СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ воспроизводится их естественным соединением по общему атрибуту СЛУ\_НОМ.

Заметим, что допустимое значение переменной отношения СЛУЖ может содержать кортежи, информационное наполнение которых выходит за пределы допустимых значений переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ. Например, в теле отношения СЛУЖ может находиться кортеж с данными о служащем с номером 4438, который еще не участвует ни в одном проекте. Наличие такого кортежа не влияет на результат естественного соединения, тело которого все равно будет совпадать с телом допустимого значения переменной отношения СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ.

---

<sup>\*</sup> *Неключевым атрибутом* называется атрибут, не входящий ни в один возможный ключ.

<sup>\*\*</sup> В определении предполагается, что у отношения имеется только один возможный ключ.

### 5.3. Нетранзитивные функциональные зависимости и третья нормальная форма

В произведенной в подразд. 5.2 декомпозиции переменной отношения **СЛУЖАЩИЕ\_ПРОЕКТЫ\_ЗАДАНИЯ** множество **FD** переменной отношения **СЛУЖ\_ПРО\_ЗАДАН** предельно просто — детерминантом единственной нетривиальной функциональной зависимости является возможный ключ (рис. 5.9, б). При использовании этой переменной отношения какие-либо аномалии обновления не возникают. Однако, как мы покажем далее, переменная отношения **СЛУЖ** не является такой же совершенной.

#### 5.3.1. Аномалии обновления, возникающие из-за наличия транзитивных функциональных зависимостей

Функциональные зависимости переменной отношения **СЛУЖ** по-прежнему порождают некоторые аномалии обновления. Они вызываются наличием транзитивной **FD** **СЛУ\_НОМ** → **СЛУ\_ЗАРП** (через **FD** **СЛУ\_НОМ** → **СЛУ\_УРОВ** и **СЛУ\_УРОВ** → **СЛУ\_ЗАРП**, рис. 5.9, а). Эти аномалии связаны с избыточностью хранения значения атрибута **СЛУ\_ЗАРП** в каждом кортеже, характеризующем служащих с одним и тем же разрядом.

- *Добавление кортежей.* Невозможно сохранить данные о новом разряде (и соответствующем ему размере заработной платы), пока не появится служащий с новым разрядом. (Атрибут **СЛУ\_НОМ**, входящий в состав первичного ключа, не может содержать неопределенные значения.)
- *Удаление кортежей.* При увольнении последнего служащего с данным разрядом будет утрачена информация о наличии такого разряда и соответствующем размере заработной платы.
- *Модификация кортежей.* При изменении размера заработной платы, соответствующей некоторому разряду, придется изменить значение атрибута **СЛУ\_ЗАРП** в кортежах всех служащих, которым назначен этот разряд (иначе не будет выполняться **FD** **СЛУ\_УРОВ** → **СЛУ\_ЗАРП**).

#### 5.3.2. Возможная декомпозиция

Для преодоления этих трудностей произведем декомпозицию переменной отношения **СЛУЖ** на две переменных отношений — **СЛУЖ1** с заголовком {**СЛУ\_НОМ**, **СЛУ\_УРОВ**} и **УРОВ** с за-



Рис. 5.11. Диаграммы FD в отношениях СЛУЖ1 и УРОВ

головком {СЛУ\_УРОВ, СЛУ\_ЗАРП}. По теореме Хита, это снова декомпозиция без потерь по причине наличия, например, FD СЛУ\_НОМ → СЛУ\_УРОВ. На рис. 5.11 показаны диаграммы FD этих переменных отношений, а на рис. 5.12 — их значения, соответствующие значению переменной отношения СЛУЖ, показанному на рис. 5.10.

Как иллюстрирует рис. 5.12, это преобразование является обратимым, т. е. любое допустимое значение исходной переменной отношения СЛУЖ является естественным соединением допустимых значений отношений СЛУЖ1 и УРОВ. Также можно заметить, что мы избавились от трудностей при выполнении операций обновления:

- *добавление кортежей.* Чтобы сохранить данные о новом разряде, достаточно добавить соответствующий кортеж к отношению УРОВ;
- *удаление кортежей.* При увольнении последнего служащего, обладающего данным разрядом, удаляется соответствующий кортеж из отношения СЛУЖ1, но данные о разряде сохраняются в отношении УРОВ;

СЛУЖ1

| СЛУ_НОМ | СЛУ_УРОВ |
|---------|----------|
| 4434    | 2        |
| 4435    | 3        |
| 4415    | 1        |
| 4436    | 1        |

УРОВ

| СЛУ_УРОВ | СЛУ_ЗАРП |
|----------|----------|
| 2        | 22400.00 |
| 3        | 29600.00 |
| 1        | 20000.00 |

Рис. 5.12. Возможные значения переменных отношений СЛУЖ1 и УРОВ

- *модификация кортежей*. При изменении размера заработной платы, соответствующей некоторому разряду, изменяется значение атрибута СЛУ\_ЗАРП ровно в одном кортеже отношения УРОВ.

### 5.3.3. Третья нормальная форма

Аномалии обновления переменной отношения СЛУЖ были связаны с наличием в этой переменной транзитивной FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП. Наличие этой FD на самом деле означало, что атрибут СЛУ\_ЗАРП характеризовал не сущность *служащий*, а сущность *разряд*.

**Определение 5.13.** Переменная отношения находится в *третьей нормальной форме* (3NF) в том и только том случае, когда она находится во второй нормальной форме и каждый неключевой атрибут нетранзитивно\* функционально зависит от первичного ключа\*\*.

Отношения СЛУЖ1 и УРОВ оба находятся в 3NF (все неключевые атрибуты нетранзитивно зависят от первичных ключей СЛУ\_НОМ и СЛУ\_УРОВ, рис. 5.11). Отношение СЛУЖ не находится в 3NF (FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП является транзитивной (см. рис. 5.9, а). Любое отношение, находящееся в 2NF, но не находящееся в 3NF, может быть приведено к набору отношений, находящихся в 3NF. При этом получается набор проекций исходного отношения, естественное соединение которых воспроизводит исходное отношение (т.е. это декомпозиция без потерь). Для отношений СЛУЖ1 и УРОВ исходное отношение СЛУЖ воспроизводится их естественным соединением по общему атрибуту СЛУ\_УРОВ.

Заметим, что допустимые значения отношения УРОВ могут содержать кортежи, информационное наполнение которых выходит за пределы тела отношения СЛУЖ. Например, в теле отношения УРОВ может находиться кортеж с данными о разряде 4, который еще не присвоен ни одному служащему. Наличие такого кортежа не влияет на результат естественного соединения, который все равно будет являться допустимым значением отношения СЛУЖ.

---

\* Функциональная зависимость называется нетранзитивной тогда и только тогда, когда она не является транзитивной.

\*\* В этом определении опять предполагается, что у отношения имеется только один возможный ключ.

### 5.3.4. Независимые проекции отношений. Теорема Риссанена

Заметим, что для переменной отношения СЛУЖ с заголовком {СЛУ\_НОМ, СЛУ\_УРОВ, СЛУ\_ЗАРП} кроме декомпозиции на переменные отношений СЛУЖ1 с заголовком {СЛУ\_НОМ, СЛУ\_УРОВ} и УРОВ с заголовком {СЛУ\_УРОВ, СЛУ\_ЗАРП} возможна и декомпозиция на отношения СЛУЖ1 с заголовком {СЛУ\_НОМ, СЛУ\_УРОВ} и СЛУЖ\_ЗАРП с заголовком {СЛУ\_НОМ, СЛУ\_ЗАРП}<sup>\*</sup>. Обе переменные отношения, полученные путем второй декомпозиции, находятся в 3NF, и эта декомпозиция также является декомпозицией без потерь. Тем не менее вторая декомпозиция, в отличие от первой, не устраняет проблемы, связанные с обновлением отношения СЛУЖ. Например, по-прежнему невозможно сохранить данные о разряде, которым не обладает ни один служащий. Посмотрим, с чем это связано.

Переменные отношений СЛУЖ1 и УРОВ могут обновляться независимо (являются *независимыми проекциями*), и при этом результат их естественного соединения всегда будет таким, как если бы обновлялось исходная переменная отношения СЛУЖ. Это происходит потому, что функциональные зависимости отношения СЛУЖ трансформировались в индивидуальные ограничения первичного ключа отношений СЛУЖ1 и УРОВ. При второй декомпозиции FD СЛУ\_УРОВ  $\rightarrow$  СЛУ\_ЗАРП трансформируется в ограничение целостности сразу для двух отношений (такого рода ограничения целостности называются ограничениями базы данных, и их поддержка гораздо более накладна с технической точки зрения). Понятно, что в процессе нормализации декомпозиция отношения на независимые проекции является предпочтительной. Необходимые и достаточные условия независимости проекций отношения обеспечивает теорема Риссанена.

**Теорема Риссанена.** Проекции  $r_1$  и  $r_2$  переменной отношения  $r$  являются независимыми тогда и только тогда, когда:

- каждая FD в отношении  $r$  логически следует<sup>\*\*</sup> из FD в  $r_1$  и  $r_2$ ;

---

<sup>\*</sup> Теоретически возможная третья декомпозиция отношения СЛУЖ на отношения СЛУЖ2 с заголовком {СЛУ\_НОМ, СЛУ\_ЗАРП} и УРОВ с заголовком {СЛУ\_УРОВ, СЛУ\_ЗАРП} не является декомпозицией без потерь. Чтобы убедиться в этом, рассмотрите случай, когда для двух разных разрядов сотрудников назначен один и тот же размер заработной платы. Покажите также, что для этой декомпозиции не выполняются условия теоремы Хита.

<sup>\*\*</sup> Т.е. выводится на основе аксиом Армстронга.



- общие атрибуты  $r_1$  и  $r_2$  образуют возможный ключ хотя бы для одного из этих отношений.

Доказательство теоремы приводить не будем, но продемонстрируем ее верность на примере двух показанных ранее декомпозиций отношения СЛУЖ. В первой декомпозиции (на проекции СЛУЖ1 и УРОВ) общий атрибут СЛУ\_УРОВ является возможным (и первичным) ключом отношения УРОВ, а единственная дополнительная FD отношения СЛУЖ (СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП) логически следует из FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_УРОВ и СЛУ\_УРОВ  $\rightarrow$  СЛУ\_ЗАРП, выполняемых для отношений СЛУЖ1 и УРОВ соответственно. Вторая декомпозиция удовлетворяет второму условию теоремы Риссанена (СЛУ\_НОМ является первичным ключом в каждом из отношений СЛУЖ1 и СЛУ\_ЗАРП), но FD СЛУ\_УРОВ  $\rightarrow$  СЛУ\_ЗАРП не выводится из FD СЛУ\_НОМ  $\rightarrow$  СЛУ\_УРОВ и СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП.

*Атомарной* переменной отношения называется такая переменная, которую невозможно декомпозировать на независимые проекции. Далеко не всегда для неатомарных (не являющихся атомарными) переменных отношений требуется декомпозиция на атомарные проекции. Например, переменная отношения СЛУЖ2 с заголовком {СЛУ\_НОМ, СЛУ\_ЗАРП, ПРО\_НОМ} и множеством FD {СЛУ\_НОМ  $\rightarrow$  СЛУ\_ЗАРП, СЛУ\_НОМ  $\rightarrow$  ПРО\_НОМ} не является атомарной (возможна декомпозиция на независимые проекции СЛУЖ3 с заголовком {СЛУ\_НОМ, СЛУ\_ЗАРП} и СЛУЖ4 с заголовком {СЛУ\_НОМ, ПРО\_НОМ}). Но эта декомпозиция не улучшает свойства переменной отношения СЛУЖ2 и поэтому не является осмысленной. Другими словами, при выборе способа декомпозиции нужно стремиться к получению независимых, но не обязательно атомарных проекций.

## 5.4. Перекрывающиеся возможные ключи и нормальная форма Бойса — Кодда

До сих пор в определениях нормальных форм предполагалось, что у декомпозируемого отношения имеется только один возможный ключ. На практике чаще всего бывает именно так. Но имеется один частный случай, который почти удовлетворяет требованиям 2NF и 3NF, но, тем не менее, порождает аномалии обновления. Это тот случай, когда у отношения имеется несколько возможных ключей и некоторые из этих возможных ключей «перекрываются», т. е. содержат общие атрибуты.

5.4.1. Аномалии обновлений, связанные с наличием перекрывающихся возможных ключей

Пусть имеется, например, переменная отношения СЛУЖ\_ПРО\_ЗАДАН1 с заголовком {СЛУ\_НОМ, СЛУ\_ИМЯ, ПРО\_НОМ, СЛУ\_ЗАДАН} и множеством FD, показанным на рис. 5.13.

В отношении СЛУЖ\_ПРО\_ЗАДАН1 служащие уникально идентифицируются как по номерам удостоверений, так и по именам. Следовательно, существуют FD СЛУ\_НОМ → СЛУ\_ИМЯ и СЛУ\_ИМЯ → СЛУ\_НОМ. Но один служащий может участвовать в нескольких проектах, поэтому возможными ключами являются {СЛУ\_НОМ, ПРО\_НОМ} и {СЛУ\_ИМЯ, ПРО\_НОМ}. На рис. 5.14 показано возможное значение переменной отношения СЛУЖ\_ПРО\_ЗАДАН1.

В отношении СЛУЖ\_ПРО\_ЗАДАН1 все FD неключевых атрибутов от возможных ключей являются минимальными и транзитивные FD отсутствуют, но, тем не менее, этому отношению свойственны аномалии обновления. Например, в случае изменения имени служащего требуется обновить атрибут СЛУ\_ИМЯ во всех корте-

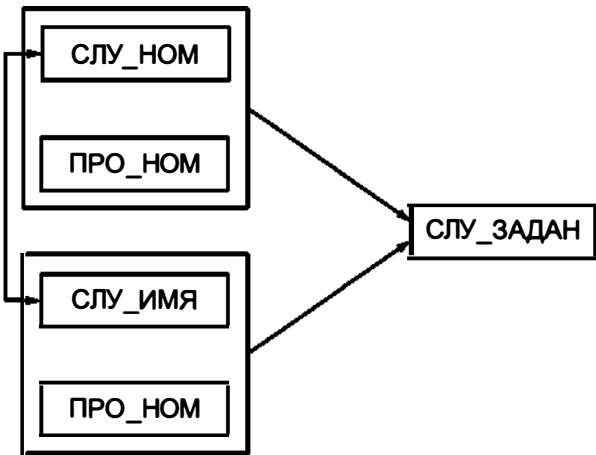


Рис. 5.13. Диаграмма FD переменной отношения СЛУЖ\_ПРО\_ЗАДАН1

| СЛУ_НОМ | СЛУ_ИМЯ  | ПРО_НОМ | ПРО_ЗАДАН |
|---------|----------|---------|-----------|
| 4434    | Иванов   | 1       | А         |
| 4417    | Иваненко | 2       | В         |
| 4434    | Иванов   | 2       | В         |
| 4417    | Иваненко | 1       | А         |

Рис. 5.14. Возможное значение переменной отношения СЛУЖ\_ПРО\_ЗАДАН1

жах отношения СЛУЖ\_ПРО\_ЗАДАН1, соответствующих данному служащему. Иначе будет нарушена FD СЛУ\_НОМ → СЛУ\_ИМЯ и база данных окажется в несогласованном состоянии.

#### 5.4.2. Нормальная форма Бойса—Кодда

Причиной отмеченных аномалий является то, что в требованиях 2NF и 3NF не требовалась минимальная функциональная зависимость от первичного ключа атрибутов, являющихся компонентами других возможных ключей. Проблему решает нормальная форма, которую исторически принято называть нормальной формой Бойса—Кодда и которая является уточнением 3NF в случае наличия нескольких перекрывающихся возможных ключей.

**Определение 5.14.** Переменная отношения находится в *нормальной форме Бойса—Кодда* (BCNF) в том и только том случае, когда детерминантом любой выполняемой для этой переменной нетривиальной и минимальной FD является некоторый возможный ключ данного отношения. ■

Заметим, что если в переменной отношения имеется только один возможный ключ, то любое отношение, находящееся в нормальной форме Бойса—Кодда, одновременно находится во второй и третьей нормальных формах. Это утверждение легко доказывается методом «от противного» на основе определений 2NF и 3NF.

Переменная отношения СЛУЖ\_ПРО\_ЗАДАН1 может быть приведена к BCNF путем одной из двух декомпозиций:

- СЛУЖ\_НОМ\_ИМЯ с заголовком {СЛУ\_НОМ, СЛУ\_ИМЯ} и СЛУЖ\_НОМ\_ПРО\_ЗАДАН с заголовком {СЛУ\_НОМ, ПРО\_НОМ, СЛУ\_ЗАДАН} с множеством FD и значениями, показанными на рис. 5.15 и 5.16 соответственно;
- СЛУЖ\_НОМ\_ИМЯ с заголовком {СЛУ\_НОМ, СЛУ\_ИМЯ} и СЛУЖ\_ИМЯ\_ПРО\_ЗАДАН с заголовком {СЛУ\_ИМЯ, ПРО\_НОМ, СЛУ\_ЗАДАН} (FD и значения результирующих переменных отношений выглядят аналогично).

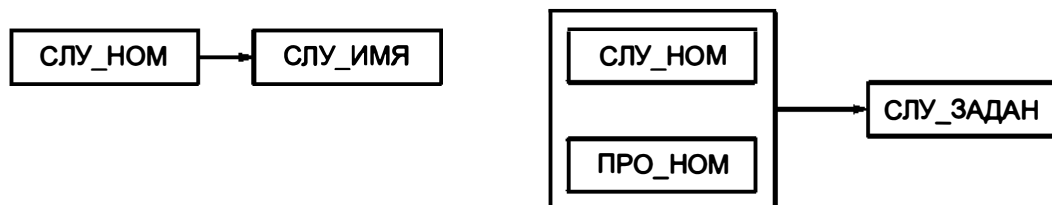


Рис. 5.15. Диаграммы FD отношений СЛУЖ\_НОМ\_ИМЯ и СЛУЖ\_НОМ\_ПРО\_ЗАДАН

### СЛУЖ\_НОМ\_ИМЯ

| СЛУ_НОМ | СЛУ_ИМЯ  |
|---------|----------|
| 4434    | Иванов   |
| 4417    | Иваненко |

### СЛУЖ\_ПРО\_ЗАДАН

| СЛУ_НОМ | ПРО_НОМ | ПРО_ЗАДАН |
|---------|---------|-----------|
| 4434    | 1       | А         |
| 4417    | 2       | В         |
| 4434    | 2       | В         |
| 4417    | 1       | А         |

Рис. 5.16. Значения переменных отношений СЛУЖ\_НОМ\_ИМЯ и СЛУЖ\_НОМ\_ПРО\_ЗАДАН

Очевидно, что каждая из декомпозиций устраняет трудности, связанные с обновлением отношения СЛУЖ\_ПРО\_ЗАДАН1.

### 5.4.3. Всегда ли следует стремиться к BCNF?

Предположим теперь, что в организации все проекты включают разные задания и по-прежнему каждый служащий может участвовать в нескольких проектах, но может выполнять в каждом проекте только одно задание. Одно задание в каждом проекте могут выполнять несколько служащих. Тогда переменная отношения СЛУЖ\_ПРО\_ЗАДАН имеет множество FD, показанное на рис. 5.17, и может содержать значение, представленное рис. 5.18.

В этой переменной отношения существуют два возможных ключа: {СЛУ\_НОМ, ПРО\_НОМ} и {СЛУ\_НОМ, СЛУ\_ЗАДАН}. Отношение удовлетворяет требованиям 3NF (если не учитывать то,

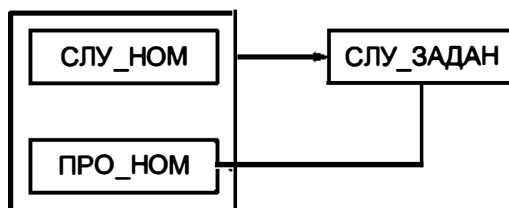


Рис. 5.17. Диаграмма FD отношения СЛУЖ\_ПРО\_ЗАДАН (новый вариант)

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 4434    | 1       | A         |
| 4417    | 1       | A         |
| 4435    | 2       | B         |
| 4434    | 2       | C         |
| 4417    | 1       | D         |

Рис. 5.18. Возможное значение переменной отношения СЛУЖ\_ПРО\_ЗАДАН (еще один вариант)

что в нем имеются два возможных ключа): отсутствуют не минимальные FD неключевых атрибутов от возможных ключей (поскольку нет неключевых атрибутов) и отсутствуют транзитивные FD. Однако из-за наличия FD  $\text{СЛУ\_ЗАДАН} \rightarrow \text{ПРО\_НОМ}$  это отношение не находится в BCNF. Поэтому отношению СЛУ\_ПРО\_ЗАДАН снова свойственны аномалии обновления. Например (поскольку СЛУ\_НОМ является компонентом обоих возможных ключей), невозможно удалить данные о единственном служащем, выполняющем задание в некотором проекте, не утратив информацию об этом задании.

Можно привести отношение СЛУЖ\_ПРО\_ЗАДАН к BCNF, выполнив его декомпозицию на отношения СЛУЖ\_НОМ\_ЗАДАН с заголовком {СЛУ\_НОМ, СЛУ\_ЗАДАН}\* и ПРО\_НОМ\_ЗАДАН с заголовком {СЛУ\_ЗАДАН, ПРО\_НОМ}, и эта декомпозиция решает указанные ранее проблемы (теперь можно хранить данные о задании проекта, не выполняемом ни одним служащим). Значения переменных отношений СЛУЖ\_НОМ\_ЗАДАН и ПРО\_НОМ\_ЗАДАН показаны на рис. 5.19.

Однако возникают новые трудности. Например, система должна запретить добавление в отношение СЛУЖ\_НОМ\_ЗАДАН кортежа {<СЛУ\_НОМ, 4434>, <СЛУ\_ЗАДАН, D>}, поскольку задание D относится к проекту 1, а служащий с номером 2934 уже выполняет задание в этом проекте. Так происходит потому, что исходная FD {СЛУ\_НОМ, ПРО\_НОМ}  $\rightarrow$  СЛУ\_ЗАДАН не выводится из единственной (нетривиальной) действующей для этих проекций FD  $\text{СЛУ\_ЗАДАН} \rightarrow \text{ПРО\_НОМ}$ , и соответствующее ограничение целостности становится ограничением базы данных.

---

\* Единственным возможным ключом отношения СЛУЖ\_НОМ\_ЗАДАН является {СЛУ\_НОМ, СЛУ\_ЗАДАН}, и в этом отношении отсутствуют нетривиальные FD.

| СЛУЖ_НОМ_ЗАДАН |           |
|----------------|-----------|
| СЛУ_НОМ        | СЛУ_ЗАДАН |
| 4434           | A         |
| 4417           | A         |
| 4435           | B         |
| 4434           | C         |
| 4417           | D         |

| ПРО_НОМ_ЗАДАН |           |
|---------------|-----------|
| ПРО_НОМ       | СЛУ_ЗАДАН |
| 1             | A         |
| 2             | B         |
| 2             | C         |
| 1             | D         |

Рис. 5.19. Значения переменных отношений СЛУЖ\_НОМ\_ЗАДАН и ПРО\_НОМ\_ЗАДАН

Тем самым, проекции СЛУЖ\_НОМ\_ЗАДАН и ПРО\_НОМ\_ЗАДАН не являются независимыми, а переменная отношения СЛУЖ\_ПРО\_ЗАДАН атомарна, хотя и не находится в BCNF. Из этого следует, что при проектировании реляционной базы данных приведение отношения к BCNF не должно быть самоцелью. Нужно внимательно оценивать положительные и отрицательные последствия нормализации.

Наконец, приведем пример, когда наличие двух перекрывающихся возможных ключей не мешает отношению находиться в BCNF. Предположим, что в организации проекты включают одни и те же задания, каждый служащий может участвовать в нескольких проектах, но может выполнять в каждом проекте только одно задание.

Тогда переменная отношения СЛУЖ\_НОМ\_ЗАДАН имеет множество FD, показанное на рис. 5.20.

В третьем варианте отношения СЛУЖ\_НОМ\_ЗАДАН имеются перекрывающиеся возможные ключи ( $\{СЛУ\_НОМ, ПРО\_НОМ\}$  и  $\{ПРО\_НОМ, СЛУ\_ЗАДАН\}$ ), однако оно находится в BCNF, поскольку эти ключи являются единственными детерминантами

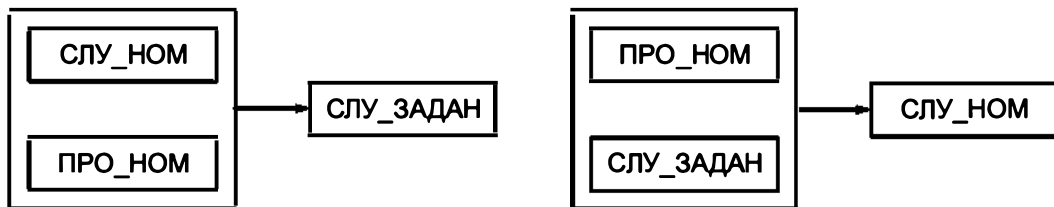


Рис. 5.20. Диаграмма FD отношения СЛУЖ\_НОМ\_ЗАДАН (третий вариант)

имеющихся нетривиальных и минимальных FD. Легко убедиться, что отношению СЛУЖ\_НОМ\_ЗАДАН аномалии обновления не свойственны.

\* \* \*

В первом подразделе этой главы было введено понятие функциональной зависимости и исследовались важные свойства функциональных зависимостей. Одна из целей состояла в том, чтобы на основе некоторого множества функциональных зависимостей суметь построить минимальное эквивалентное множество функциональных зависимостей. Мы начали обсуждение с понятия замыканий множества функциональных зависимостей и аксиом Амстронга, теоретически позволяющих построить такое замыкание. Замыкание множества функциональных зависимостей содержит все функциональные зависимости, выводимые из функциональных зависимостей заданного множества. Рассмотренный далее алгоритм построения замыкания множества атрибутов над заданным множеством функциональных зависимостей упрощает задачу, позволяя определить принадлежность заданной функциональной зависимости к замыканию заданного множества функциональных зависимостей без потребности в реальном построении замыкания.

Далее мы занялись покрытиями множеств функциональных зависимостей и минимальными множествами функциональных зависимостей. Наиболее важным результатом этого подраздела является доказательство существования и наметки алгоритма построения минимального покрытия заданного множества функциональных зависимостей — минимального множества функциональных зависимостей, эквивалентного исходному множеству.

Наконец, был обсужден критерий декомпозиции отношений без потерь, т.е. такой способ проецирования заданной переменной отношения на две переменных отношений, при котором

результат естественного соединения значений переменных-проекций в точности совпадает со значением исходной переменной отношения. Достаточное (и очень естественное) условие декомпозиции без потерь обеспечивает теорема Хита.

В следующих трех подразделах были рассмотрены три начальные нормальные формы переменных отношений — вторая и третья нормальные формы и нормальная форма Бойса — Кодда, которые производятся путем декомпозиции без потерь исходной переменной отношения на две переменные-проекции, в которых отсутствуют аномалии изменений, существовавшие в исходной переменной отношения по причине наличия функциональных зависимостей с нежелательными свойствами.

Нормализация схемы базы данных способствует более эффективному выполнению системой управления базами данных операций обновления базы данных, поскольку сокращается число проверок и вспомогательных действий, поддерживающих целостность базы данных. При проектировании реляционной базы данных почти всегда добиваются второй нормальной формы всех входящих в базу данных отношений. В часто обновляемых базах данных обычно стараются обеспечить третью нормальную форму отношений. На нормальную форму Бойса — Кодда внимание обращают гораздо реже, поскольку на практике ситуации, в которых у отношения имеется несколько составных перекрывающихся возможных ключей, встречаются нечасто.



## ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ: ДАЛЬНЕЙШАЯ НОРМАЛИЗАЦИЯ

Функциональные зависимости, о которых говорилось в гл. 5, и нормальные формы, основанные на учете «аномальных» функциональных зависимостей, являются естественными и легко понимаемыми, поскольку в их основе лежит понятие функционального отображения, интуитивно понятного даже людям, далеким от математики.

Конечно, было бы замечательно, если бы ликвидация в ходе нормализации аномальных функциональных зависимостей гарантировала отсутствие аномалий обновления отношений. К сожалению, эта гарантия в общем случае не обеспечивается. Иногда в переменных отношениях требуется поддержка более сложных ограничений целостности, для выражения которых понятие функции оказывается недостаточным.

Класс зависимостей, опирающихся на понятие *функционала* — обобщение понятия функции, обнаружил в 1970-е гг. Рональд Фейджин. Он назвал такие зависимости *многозначными*, поскольку в них одному значению детерминанта соответствует множество значений зависимого атрибута. Наличие в переменной отношения многозначных зависимостей, не являющихся функциональными зависимостями от возможного ключа, приводит к аномалиям обновления таких отношений. Фейджин показал, что и в этом случае возможна декомпозиция данных отношений на две проекции, для которых подобные аномалии обновления не проявляются. Такие проекции находятся в *четвертой нормальной форме*.

Позже было установлено, что при наличии некоторых естественных ограничений, являющихся обобщением ограничений многозначных зависимостей, и в отношениях, которые находятся в четвертой нормальной форме, проявляются аномалии обновления. Более того, эти аномалии невозможно устранить путем декомпозиции отношения на две проекции, требуется декомпозиция на три или большее число отношений. Такие ограничения получили название зависимостей *проекции/соединения*. Отношение, в котором существует нетривиальная зависимость проекции/

соединения, может быть декомпозировано на три или большее число проекций, в которых зависимости проекции/соединения следуют из возможного ключа. Такие проекции находятся в *пятой нормальной форме*, или *нормальной форме проекции/соединения*. В отношениях, находящихся в пятой нормальной форме, отсутствуют аномалии обновления, которые можно было бы устранить путем декомпозиции, и поэтому при достижении пятой нормальной формы процесс проектирования реляционной базы данных на основе нормализации естественным образом завершается.

**6.1. Многозначные зависимости и четвертая нормальная форма**

Чтобы перейти к вопросам дальнейшей нормализации, рассмотрим еще одну возможную интерпретацию переменной отношения СЛУЖ\_ПРО\_ЗАДАН. Предположим, что каждый сотрудник может участвовать в нескольких проектах, но в каждом

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | A         |
| 2934    | 1       | B         |
| 2934    | 2       | A         |
| 2934    | 2       | B         |
| ....    | ....    | ....      |
| 2941    | 1       | A         |
| 2941    | 1       | D         |

Рис. 6.1. Возможное значение переменной отношения СЛУЖ\_ПРО\_ЗАДАН (и еще один вариант)

проекте, в котором он участвует, им должны выполняться одни и те же задания. Возможное значение этого варианта переменной отношения СЛУЖ\_ПРО\_ЗАДАН показано на рис. 6.1.

**6.1.1. Аномалии обновлений при наличии многозначных зависимостей и возможная декомпозиция**

В новом варианте переменной отношения единственно возможным ключом является заголовок отношения {СЛУ\_НОМ,

ПРО\_НОМ, СЛУ\_ЗАДАН}. Кортёж {<СЛУ\_НОМ, *сн*>, <ПРО\_НОМ, *пн*>, <СЛУ\_ЗАДАН, *сз*>} входит в тело отношения в том и только том случае, когда служащий с номером *сн* выполняет в проекте *пн* задание *сз*.

Поскольку для каждого служащего указываются все проекты, в которых он участвует, и все задания, которые он должен выполнять в этих проектах, для каждого допустимого значения переменной отношения СЛУЖ\_ПРО\_ЗАДАН должно выполняться следующее ограничение ( $V_{спз}$  обозначает тело значения переменной отношения СЛУЖ\_ПРО\_ЗАДАН):

IF ({<СЛУ\_НОМ, *сн*>, <ПРО\_НОМ, *пн1*>, <СЛУ\_ЗАДАН, *сз1*>}  $\in V_{спз}$   
AND {<СЛУ\_НОМ, *сн*>, <ПРО\_НОМ, *пн2*>, <СЛУ\_ЗАДАН, *сз2*>}  $\in V_{спз}$ )  
THEN ({<СЛУ\_НОМ, *сн*>, <ПРО\_НОМ, *пн1*>, <СЛУ\_ЗАДАН, *сз2*>}  $\in V_{спз}$   
AND {<СЛУ\_НОМ, *сн*>, <ПРО\_НОМ, *пн2*>, <СЛУ\_ЗАДАН, *сз1*>}  $\in V_{спз}$ )

Наличие такого ограничения (как будет показано далее, это ограничение порождается наличием многозначной зависимости) приводит к тому, что при работе с отношением СЛУЖ\_ПРО\_ЗАДАН проявляются следующие аномалии обновления:

- *добавление кортежа*. Если уже участвующий в проектах служащий присоединяется к новому проекту, то к телу значения переменной отношения СЛУЖ\_ПРО\_ЗАДАН требуется добавить столько кортежей, сколько заданий выполняет этот служащий;
- *удаление кортежей*. Если служащий прекращает участие в проектах, то отсутствует возможность сохранить данные о заданиях, которые он может выполнять;
- *модификация кортежей*. При изменении одного из заданий служащего необходимо изменить значение атрибута СЛУ\_ЗАДАН в стольких кортежах, в скольких проектах участвует служащий.

Трудности, связанные с обновлением переменной отношения СЛУЖ\_ПРО\_ЗАДАН, решаются путем его декомпозиции на две переменных отношений: СЛУЖ\_ПРО\_НОМ {СЛУ\_НОМ, ПРО\_НОМ} и СЛУЖ\_ЗАДАНИЕ {СЛУ\_НОМ, СЛУ\_ЗАДАН}. Значения этих переменных отношений, соответствующие значению переменной отношения СЛУЖ\_ПРО\_ЗАДАН с рис. 6.1, показаны на рис. 6.2.

Легко видеть, что декомпозиция, представленная на рис. 6.2, является декомпозицией без потерь и что эта декомпозиция решает перечисленные ранее проблемы с обновлением переменной отношения СЛУЖ\_ПРО\_ЗАДАН:

- *добавление кортежа*. Если некоторый уже участвующий в проектах служащий присоединяется к новому проекту,

### СЛУЖ\_ПРО\_НОМ

| СЛУ_НОМ | ПРО_НОМ |
|---------|---------|
| 2934    | 1       |
| 2934    | 2       |
| ....    | ....    |
| 2941    | 1       |

### СЛУЖ\_ЗАДАНИЕ

| СЛУ_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 2934    | A         |
| 2934    | B         |
| ....    | ....      |
| 2941    | A         |
| 2941    | D         |

Рис. 6.2. Значения переменных отношений СЛУЖ\_ПРО\_НОМ и СЛУЖ\_ЗАДАНИЕ

то к телу значения переменной отношения СЛУЖ\_ПРО\_НОМ требуется добавить один кортеж, соответствующий новому проекту;

- *удаление кортежей.* Если служащий прекращает участие в проектах, то данные о заданиях, которые он может выполнять, остаются в отношении СЛУЖ\_ЗАДАНИЕ;
- *модификация кортежей.* При изменении одного из заданий сотрудника достаточно изменить значение атрибута СЛУ\_ЗАДАН в одном кортеже отношения СЛУЖ\_ЗАДАНИЕ.

### 6.1.2. Многозначные зависимости. Теорема Фейджина. Четвертая нормальная форма

Заметим, что последний вариант переменной отношения СЛУЖ\_ПРО\_ЗАДАН находится в BCNF, поскольку все атрибуты заголовка отношения входят в состав единственно возможного ключа. В этом отношении вообще отсутствуют нетривиальные FD. Поэтому ранее обсуждавшиеся принципы нормализации здесь неприменимы, но, тем не менее, мы получили полезную декомпозицию.

Все дело в том, что в случае этого варианта отношения СЛУЖ\_ПРО\_ЗАДАН мы имеем дело с новым видом зависимости, впервые обнаруженным Роном Фейджином в 1971 г. Фейджин назвал зависимости этого вида *многозначными* (multi-valued dependency — MVD). Многозначная зависимость атрибута  $A$  от атрибута  $B$  обозначается так:  $A \twoheadrightarrow B$ . Как будет показано далее, MVD является обобщением понятия FD.

В отношении СЛУЖ\_ПРО\_ЗАДАН выполняются две MVD: СЛУ\_НОМ  $\twoheadrightarrow$  ПРО\_НОМ и СЛУ\_НОМ  $\twoheadrightarrow$  СЛУ\_ЗАДАН. Первая MVD означает, что каждому значению атрибута СЛУ\_НОМ соответствует определяемое только этим значением множество значений атрибута ПРО\_НОМ. Другими словами, в результате вычисления алгебраического выражения

(СЛУЖ\_ПРО\_ЗАДАН WHERE (СЛУ\_НОМ =  $сн$  AND СЛУ\_ЗАДАН =  $сз$ ))  
PROJECT {ПРО\_НОМ}

для фиксированного допустимого значения  $сн$  и любого допустимого значения  $сз$  мы всегда получим одно и то же множество значений атрибута ПРО\_НОМ. Аналогично трактуется вторая MVD.

**Определение 6.1.** В переменной отношения  $r$  с атрибутами  $A, B, C$  (в общем случае, составными) имеется *многозначная зависимость  $B$  от  $A$*  ( $MVD A \twoheadrightarrow B$ ) в том и только том случае, когда множество значений атрибута  $B$ , соответствующее паре значений атрибутов  $A$  и  $C$ , зависит от значения  $A$  и не зависит от значения  $C$ . ■

Многозначные зависимости обладают интересным свойством «двойственности», которое демонстрирует следующая лемма.

**Лемма Фейджина.** В отношении  $r$   $\{A, B, C\}$  выполняется  $MVD A \twoheadrightarrow B$  в том и только том случае, когда выполняется  $MVD A \twoheadrightarrow C$ .

*Доказательство.* Докажем достаточность условия леммы. Пусть выполняется  $MVD A \twoheadrightarrow B$ . Пусть  $R$  — некоторое значение переменной отношения  $r$ , удовлетворяющее этой зависимости. Пусть  $a$  обозначает значение атрибута  $A$  в некотором кортеже  $B_R$ ,  $\{b\}$  и  $\{c\}$  — множества значений атрибутов  $B$  и  $C$  соответственно, взятых из всех кортежей тела  $B_R$ , в которых значением атрибута  $A$  является  $a$ . Предположим, что для этого значения  $a$   $MVD A \twoheadrightarrow C$  не выполняется. Это означает, что существует значение  $c \in \{c\}$ , что для него найдется такое значение  $b \in \{b\}$ , что кортеж  $\langle A, a, \langle B, b \rangle, \langle C, c \rangle \rangle \notin B_R$ . Но это противоречит наличию  $MVD A \twoheadrightarrow B$  ( $\{b\}$  зависит только от  $a$ ). Следовательно, если выполня-

ется  $MVD A \twoheadrightarrow B$ , то выполняется и  $MVD A \twoheadrightarrow C$ . Аналогично можно доказать необходимость условия леммы. ▮

Таким образом,  $MVD A \twoheadrightarrow B$  и  $A \twoheadrightarrow C$  всегда составляют пару. Поэтому обычно их представляют вместе в форме  $A \twoheadrightarrow B \mid C$ .

FD является частным случаем MVD, когда множество значений зависимого атрибута обязательно состоит из одного элемента. Таким образом, если выполняется  $FD A \rightarrow B$ , то выполняется и  $MVD A \rightarrow B$ .

Видим, что отношения СЛУЖ\_ПРО\_НОМ и СЛУЖ\_ЗАДАНИЕ не содержат MVD, отличных от FD, и именно в этом выигрывает декомпозиция из рис. 6.2. Правомочность этой декомпозиции доказывается теоремой Фейджина, которая является уточнением и обобщением теоремы Хита.

**Теорема Фейджина.** Пусть имеется переменная отношения  $r$  с атрибутами  $A, B, C$  (в общем случае, составными). Тогда  $r = (r \text{ PROJECT } \{A, B\}) \text{ NATURAL JOIN } (r \text{ PROJECT } \{A, C\})$  в том и только том случае, когда для любого значения  $r$  выполняется  $MVD A \twoheadrightarrow B \mid C$ .

*Доказательство.* Сначала докажем достаточность условия теоремы, т. е., что если для любого значения  $r$  выполняется  $MVD A \twoheadrightarrow B \mid C$ , то  $r = (r \text{ PROJECT } \{A, B\}) \text{ NATURAL JOIN } (r \text{ PROJECT } \{A, C\})$ . Пусть  $R$  является некоторым допустимым значением переменной отношения  $r$ . Обозначим результат операции  $R \text{ PROJECT } \{A, B\}$  через  $R_1$ , результат операции  $R \text{ PROJECT } \{A, C\}$  через  $R_2$ , а результат естественного соединения  $R_1 \text{ NATURAL JOIN } R_2$  через  $R_3$ . Пусть  $a$  обозначает значение атрибута  $A$  в некотором кортеже  $B_R$ ,  $\{b\}$  и  $\{c\}$  — множества значений атрибутов  $B$  и  $C$  соответственно, взятых из всех кортежей тела  $B_R$ , в которых значением атрибута  $A$  является  $a$ . Тогда очевидно, что в  $B_{R_1}$  будут входить все кортежи вида  $\langle A, a \rangle, \langle B, b \rangle$ , где  $b \in \{b\}$ , и если некоторый кортеж  $\langle A, a \rangle, \langle B, b \rangle$  входит в  $B_{R_1}$ , то  $b \in \{b\}$ . Аналогичные рассуждения применимы к  $B_{R_2}$ . Следовательно, для данного значения  $a$  в  $B_{R_3}$  будут входить те и только те кортежи  $\langle A, a \rangle, \langle B, b \rangle, \langle C, c \rangle$ , для которых  $b \in \{b\}$  и  $c \in \{c\}$ . Но по определению MVD и в  $B_R$  для данного значения входят те и только те кортежи  $\langle A, a \rangle, \langle B, b \rangle, \langle C, c \rangle$ , для которых  $b \in \{b\}$  и  $c \in \{c\}$ . Следовательно,  $R = R_3$  и достаточность условия теоремы доказана.

Докажем необходимость условия теоремы, т. е. что если для произвольного значения  $R$  переменной отношения  $r$  выполняется соотношение  $r = (r \text{ PROJECT } \{A, B\}) \text{ NATURAL JOIN } (r \text{ PROJECT } \{A, C\})$ , то в нем существует MVD  $A \twoheadrightarrow B \mid C$ . Другими словами, нам требуется показать, что в  $B_R$  поддерживается следующее ограничение:

IF  $\{ \langle A, a \rangle, \langle B, b1 \rangle, \langle C, c1 \rangle \} \in B_R$  AND  $\{ \langle A, a \rangle, \langle B, b2 \rangle, \langle C, c2 \rangle \} \in B_R$   
 THEN  $\{ \langle A, a \rangle, \langle B, b1 \rangle, \langle C, c2 \rangle \} \in B_R$  AND  $\{ \langle A, a \rangle, \langle B, b2 \rangle, \langle C, c1 \rangle \} \in B_R$

Действительно, пусть  $\{ \langle A, a \rangle, \langle B, b1 \rangle, \langle C, c1 \rangle \} \in B_R$  и  $\{ \langle A, a \rangle, \langle B, b2 \rangle, \langle C, c2 \rangle \} \in B_R$ . Предположим, что  $\{ \langle A, a \rangle, \langle B, b1 \rangle, \langle C, c2 \rangle \} \notin B_R$  OR  $\{ \langle A, a \rangle, \langle B, b2 \rangle, \langle C, c1 \rangle \} \notin B_R$ . Если воспользоваться введенными ранее обозначениями, то, очевидно, что  $\{ \langle A, a \rangle, \langle B, b1 \rangle \} \in B_{R1}$  и  $\{ \langle A, a \rangle, \langle B, b2 \rangle \} \in B_{R1}$ , а также  $\{ \langle A, a \rangle, \langle C, c1 \rangle \} \in B_{R2}$  и  $\{ \langle A, a \rangle, \langle C, c2 \rangle \} \in B_{R2}$ . По свойствам операции естественного соединения  $\{ \langle A, a \rangle, \langle B, b1 \rangle, \langle C, c2 \rangle \} \in B_{R3}$  и  $\{ \langle A, a \rangle, \langle B, b2 \rangle, \langle C, c1 \rangle \} \in B_{R3}$ . Поскольку по условию теоремы  $R = R_3$ , это противоречит предположению об отсутствии, по крайней мере, одного из этих кортежей в  $B_R$ . Тем самым, теорема Фейджина полностью доказана. ■

Обсудим теперь, почему и в каком отношении теорема Фейджина является обобщением теоремы Хита. В соответствии с теоремой Хита, достаточным условием декомпозиции без потерь переменной отношения  $r \{A, B, C\}$  на проекции  $r \text{ PROJECT } \{A, B\}$  и  $r \text{ PROJECT } \{A, C\}$  является наличие функциональной зависимости  $A \rightarrow B$ . Но поскольку, как уже отмечалось, функциональная зависимость является частным случаем многозначной зависимости, то по лемме Фейджина в переменной отношения, удовлетворяющей условию теоремы Хита, имеется  $MVD A \twoheadrightarrow B \mid C$ , и, следовательно, теорема Хита является следствием теоремы Фейджина. Из теоремы же Фейджина следует, что теорема Хита задает только достаточное условие декомпозиции без потерь, потому что из того, что для произвольного значения  $R$  переменной отношения  $r$  выполняется соотношение  $r = (r \text{ PROJECT } \{A, B\}) \text{ NATURAL JOIN } (r \text{ PROJECT } \{A, C\})$ , выводится наличие  $MVD A \twoheadrightarrow B \mid C$ , но совсем не обязательно  $FD A \rightarrow B$ .

Теорема Фейджина обеспечивает основу для декомпозиции отношений, удаляющей «аномальные» многозначные зависимости, с приведением отношений в *четвертую нормальную форму*.

**Определение 6.2.** Переменная отношения  $r$  находится в *четвертой нормальной форме* (4NF) в том и только том случае, когда она находится в BCNF, и все MVD  $r$  являются FD с детерминантами — возможными ключами отношения  $r$ . ■

В сущности, 4NF является BCNF, в которой многозначные зависимости вырождаются в функциональные. Понятно, что отношение СЛУЖ\_ПРО\_ЗАДАН не находится в 4NF, поскольку

детерминант  $MVD\ \text{СЛУ\_НОМ} \twoheadrightarrow \text{ПРО\_НОМ}$  и  $\text{СЛУ\_НОМ} \twoheadrightarrow \text{СЛУ\_ЗАДАН}$  не является возможным ключом, и эти  $MVD$  не являются функциональными. С другой стороны, отношения  $\text{СЛУЖ\_ПРО\_НОМ}$  и  $\text{СЛУЖ\_ЗАДАНИЕ}$  находятся в  $BCNF$  и не содержат  $MVD$ , отличных от  $FD$  с детерминантом — возможным ключом. Поэтому они находятся в  $4NF$ .

## 6.2. Зависимости проекции/соединения и пятая нормальная форма

Приведение отношения к  $4NF$  предполагает его декомпозицию без потерь на две проекции (как и в случае  $2NF$ ,  $3NF$  и  $BCNF$ ). Однако бывают (хотя и нечасто) случаи, когда декомпозиция без потерь на две проекции невозможна, но можно произвести декомпозицию без потерь на большее число проекций. Будем называть  $n$ -декомпозируемым отношением отношение, которое может быть декомпозировано без потерь на  $n$  проекций. До сих пор мы имели дело с 2-декомпозируемыми отношениями.

### 6.2.1. $N$ -декомпозируемые отношения

Начнем с еще одного определения.

**Определение 6.3.** В переменной отношения  $r$  с атрибутами (возможно, составными)  $A$  и  $B$   $MVD\ A \twoheadrightarrow B$  называется *тривиальной*, если либо  $A \subseteq B$ , либо  $A \cup B$  совпадает с заголовком отношения  $r$ . ■

Тривиальная  $MVD$  всегда удовлетворяется. При  $A \subseteq B$  она вырождается в тривиальную  $FD$ . В случае  $A \cup B = H$ , требования многозначной зависимости соблюдаются очевидным образом.

Для примера  $n$ -декомпозируемого отношения при  $n > 2$  рассмотрим следующий вариант переменной отношения  $\text{СЛУЖ\_ПРО\_ЗАДАН}$ , в которой имеется единственный возможный ключ  $\{\text{СЛУ\_НОМ}, \text{ПРО\_НОМ}, \text{СЛУ\_ЗАДАН}\}$  и отсутствуют нетривиальные  $MVD$ . Пример значения переменной отношения ( $R_{\text{СПС}}$ ) приведен на рис. 6.3. На этом же рисунке показаны результаты применения к  $R_{\text{СПС}}$  операций проекций на все три возможных подмножества заголовка, включающие по два атрибута, а также результат естественного соединения двух проекций.

Как показано на рис. 6.3, тело результата естественного соединения проекций  $R_{\text{СП}}$  и  $R_{\text{ПС}}$  почти совпадает с телом исходного



$$R_{СПС}$$

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | А         |
| 2934    | 1       | В         |
| 2934    | 2       | А         |
| 2941    | 1       | А         |

$$R_{СП} = (R_{СПЗ} \text{ ПРОЕКТ } \{\text{СЛУ\_НОМ, ПРО\_НОМ}\})$$

| СЛУ_НОМ | ПРО_НОМ |
|---------|---------|
| 2934    | 1       |
| 2934    | 2       |
| 2941    | 1       |

$$R_{ПС} = (R_{СПЗ} \text{ ПРОЕКТ } \{\text{ПРО\_НОМ, СЛУ\_ЗАДАН}\})$$

| ПРО_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 1       | А         |
| 1       | В         |
| 2       | А         |

$$R_{СС} = (R_{СПЗ} \text{ ПРОЕКТ } \{\text{СЛУ\_НОМ, СЛУ\_ЗАДАН}\})$$

| СЛУ_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 2934    | А         |
| 2934    | В         |
| 2941    | А         |

$$R_{СП} \text{ NATURAL JOIN } R_{ПС}$$

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | А         |
| 2934    | 1       | В         |
| 2934    | 2       | А         |
| 2941    | 1       | А         |
| 2941    | 1       | В         |

← Лишний кортеж

Рис. 6.3. Соединение с потерями

значения-отношения  $R_{СПС}$ , но в нем присутствует один лишний кортеж, который исчезнет после выполнения заключительного естественного соединения с проекцией  $R_{СС}$ . Легко убедиться, что исходное отношение будет восстановлено при любом порядке естественного соединения трех проекций.

### 6.2.2. Зависимость проекции/соединения

Утверждение о том, что значение-отношение  $R_{СПС}$  восстанавливается без потерь путем естественного соединения его проекций  $R_{СП}$ ,  $R_{ПС}$  и  $R_{СС}$  эквивалентно следующему утверждению:

IF ( $\{ \langle \text{СПУ\_НОМ}, \text{сн} \rangle, \langle \text{ПРО\_НОМ}, \text{пн} \rangle \} \in B_{R_{СП}}$   
AND  $\{ \langle \text{ПРО\_НОМ}, \text{пн} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз} \rangle \} \in B_{R_{ПС}}$   
AND  $\{ \langle \text{СПУ\_НОМ}, \text{сн} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз} \rangle \} \in B_{R_{СС}}$ )  
THEN  $\{ \langle \text{СПУ\_НОМ}, \text{сн} \rangle, \langle \text{ПРО\_НОМ}, \text{пн} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз} \rangle \} \in B_{R_{СПС}}$

Чтобы возможность восстановления без потерь значения  $R_{СПС}$  переменной отношения СЛУЖ\_ПРО\_ЗАДАН путем естественного соединения значений ее проекций  $R_{СП}$ ,  $R_{ПС}$  и  $R_{СС}$  существовала при любом допустимом  $R_{СПС}$ , для значений этой переменной должно поддерживаться следующее ограничение:

IF ( $\{ \langle \text{СПУ\_НОМ}, \text{сн1} \rangle, \langle \text{ПРО\_НОМ}, \text{пн1} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз2} \rangle \} \in B_{R_{СПС}}$   
AND  $\{ \langle \text{СПУ\_НОМ}, \text{сн2} \rangle, \langle \text{ПРО\_НОМ}, \text{пн1} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз1} \rangle \} \in B_{R_{СПС}}$   
AND  $\{ \langle \text{СПУ\_НОМ}, \text{сн1} \rangle, \langle \text{ПРО\_НОМ}, \text{пн1} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз1} \rangle \} \in B_{R_{СПС}}$ )  
THEN  $\{ \langle \text{СПУ\_НОМ}, \text{сн1} \rangle, \langle \text{ПРО\_НОМ}, \text{пн1} \rangle, \langle \text{СПУ\_ЗАДАН}, \text{сз1} \rangle \} \in B_{R_{СПС}}$

Это обычное ограничение реального мира, которое для переменной отношения СЛУЖ\_ПРО\_ЗАДАН может быть сформулировано на естественном языке следующим образом:

*Если служащий с номером сн участвует в проекте пн, и в проекте пн выполняется задание сз, и служащий с номером сн выполняет задание сз, то служащий с номером сн выполняет задание сз в проекте пн.*

В общем виде такое ограничение называется *зависимостью проекции/соединения*. Вот формальное определение.

**Определение 6.4.** Пусть задана переменная отношения  $r$ , и  $A, B, \dots, Z$  являются произвольными подмножествами заголовка  $r$  (составными, перекрывающимися атрибутами). В переменной отношения  $r$  удовлетворяется *зависимость проекции/соединения* (Project-Join Dependency — PJD)  $*(A, B, \dots, Z)$  тогда и только тогда, когда любое допустимое значение  $r$  можно получить путем естественного соединения проекций этого значения на атрибуты  $A, B, \dots, Z$ . ■

### 6.2.3. Аномалии, вызываемые наличием зависимости проекции/соединения

Предположим, что для переменной отношения СЛУЖ\_ПРО\_ЗАДАН выполняется  $RJD *(\{СЛУ\_НОМ, ПРО\_НОМ\}, \{ПРО\_НОМ, СЛУ\_ЗАДАН\}, \{СЛУ\_НОМ, СЛУ\_ЗАДАН\})$ . Наличие такой RJD обеспечивает возможность декомпозиции этой переменной отношения на три проекции, но возникает вопрос, зачем это нужно? Чем плохо исходное отношение СЛУЖ\_ПРО\_ЗАДАН? Ответ обычный: этому отношению свойственны аномалии обновления. Для примера предположим, что значением переменной отношения СЛУЖ\_ПРО\_ЗАДАН является отношение  $R_{cnc1}$ , показанное на рис. 6.4.

Возможное значение переменной отношения СЛУЖ\_ПРО\_ЗАДАН  $R_{cnc1}$

Тогда имеют место следующие аномалии:

- *добавление кортежей*. Если к переменной со значением  $R_{cnc1}$  (рис. 6.4) добавляется кортеж {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>}, то в новом значении переменной должен быть добавлен и кортеж {<СЛУ\_НОМ, 2934>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>}. Действительно, после вставки в значение переменной отношения появятся кортежи {<СЛУ\_НОМ, 2934>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, В>}, {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>} и {<СЛУ\_НОМ, 2934>, <ПРО\_НОМ, 2>, <СЛУ\_ЗАДАН, А>}. Легко видеть, что ограничение целостности требует включения и кортежа {<СЛУ\_НОМ,

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | В         |
| 2934    | 2       | А         |

Значение СЛУЖ\_ПРО\_ЗАДАН после добавления к  $R_{cnc1}$  кортежа {< СЛУ\_НОМ, 2941>, < ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>} ( $R_{cnc2}$ )

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | В         |
| 2934    | 2       | А         |
| 2941    | 1       | А         |
| 2934    | 1       | А         |

Рис. 6.4. Иллюстрации аномалий обновления переменной СЛУЖ\_ПРО\_ЗАДАН при наличии зависимости соединения

2934>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>}. Интересно, что добавление кортежа {<СЛУ\_НОМ, 2934>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>} не нарушает ограничение целостности и, тем самым, не требует добавления кортежа {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>};

- *удаление кортежей*. Если из переменной со значением  $R_{сн2}$  удаляется кортеж {<СЛУ\_НОМ, 2934>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>}, то должен быть удален и кортеж {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>}, поскольку в соответствии с ограничением целостности наличие второго кортежа означает наличие первого. Интересно, что удаление кортежа {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>, <СЛУ\_ЗАДАН, А>} не нарушает ограничения целостности и не требует дополнительных удалений.

#### 6.2.4. Устранение аномалий обновления в 3-декомпозиции

После выполнения декомпозиции трудности с обновлением автоматически снимаются. Действительно, декомпозируем переменную отношения СЛУЖ\_ПРО\_ЗАДАН на три переменных

СЛУЖ\_ПРО\_НОМ ( $R_{сн1}$ )

| СЛУ_НОМ | ПРО_НОМ |
|---------|---------|
| 2934    | 1       |
| 2934    | 2       |

СЛУЖ\_ЗАДАНИЕ ( $R_{сн1}$ )

| СЛУ_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 2934    | В         |
| 2934    | А         |

ПРО\_НОМ\_ЗАДАН ( $R_{сн1}$ )

| ПРО_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 1       | В         |
| 2       | А         |

а

Рис. 6.5. Иллюстрация декомпозиции переменной отношения с зависимостью соединения (см. также с. 188)

Добавление данных о служащем с номером 2941,  
выполняющем задание А в проекте 1

СЛУЖ\_ПРО\_НОМ ( $R_{cn2}$ )

| СЛУ_НОМ | ПРО_НОМ |
|---------|---------|
| 2934    | 1       |
| 2934    | 2       |
| 2941    | 1       |

СЛУЖ\_ЗАДАНИЕ ( $R_{cc2}$ )

| СЛУ_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 2934    | В         |
| 2934    | А         |
| 2941    | А         |

ПРО\_НОМ\_ЗАДАН ( $R_{nc2}$ )

| ПРО_НОМ | СЛУ_ЗАДАН |
|---------|-----------|
| 1       | В         |
| 2       | А         |
| 1       | А         |

б

$R_{cn2}$  NATURAL JOIN  $R_{cc2}$  NATURAL JOIN  $R_{nc2}$

| СЛУ_НОМ | ПРО_НОМ | СЛУ_ЗАДАН |
|---------|---------|-----------|
| 2934    | 1       | В         |
| 2934    | 2       | А         |
| 2941    | 1       | А         |
| 2934    | 1       | А         |

в

Рис. 6.5. Окончание

отношения: СЛУЖ\_ПРО\_НОМ {СЛУ\_НОМ, ПРО\_НОМ}, СЛУЖ\_ЗАДАНИЕ {СЛУ\_НОМ, СЛУ\_ЗАДАН} и ПРО\_НОМ\_ЗАДАН {ПРО\_НОМ, СЛУ\_ЗАДАН}. Результат декомпозиции значения переменной отношения СЛУЖ\_ПРО\_ЗАДАН с телом  $R_{снс1}$  показан на рис. 6.5, а.

Теперь, если мы хотим добавить данные о сотруднике с номером 2941, выполняющем задание А в проекте 1, то, естественно, вставим кортеж {<СЛУ\_НОМ, 2941>, <ПРО\_НОМ, 1>} в отношение СЛУЖ\_ПРО\_НОМ, кортеж {<СЛУ\_НОМ, 2941>, <СЛУ\_ЗАДАН, А>} в отношение СЛУЖ\_ЗАДАНИЕ и кортеж {<СЛУ\_НОМ, 1>, <СЛУ\_ЗАДАН, А>} в отношение ПРО\_НОМ\_ЗАДАН. Результат этих операций показан на рис. 6.5, б.

Но если выполнить естественное соединение значений декомпозированных переменных с телами, полученными после добавления данных о сотруднике с номером 2941, выполняющем задание А в проекте 1, то будет получено значение-отношение с заголовком отношения СЛУЖ\_ПРО\_ЗАДАН и телом  $R_{снс2}$  (рис. 6.5, в). Тем самым, проведенная декомпозиция позволила избежать сложностей при выполнении добавления кортежей с получением корректных результатов.

Аналогично можно проиллюстрировать простоту и корректность операций удаления кортежей.

### 6.2.5. Пятая нормальная форма

Переменные отношений СЛУЖ\_ПРО\_НОМ, СЛУЖ\_ЗАДАНИЕ и ПРО\_НОМ\_ЗАДАН находятся в пятой нормальной форме, но, прежде чем привести ее определение, нам требуется ввести еще два важных понятия.

**Определение 6.5** (зависимость проекции/соединения, подразумеваемая возможными ключами). В переменной отношения  $r$   $PJD^*(A, B, \dots, Z)$  называется *подразумеваемой возможными ключами* в том и только том случае, когда каждый составной атрибут  $A, B, \dots, Z$  является суперключом  $r$ , т.е. включает хотя бы один возможный ключ  $r$ . ■

**Определение 6.6** (тривиальная зависимость проекции/соединения). В переменной отношения  $r$   $PJD^*(A, B, \dots, Z)$  называется *тривиальной*, если хотя бы один из составных атрибутов  $A, B, \dots, Z$  совпадает с заголовком  $r$ . ■

Легко убедиться, что нетривиальные  $PJD$ , подразумеваемые возможными ключами, существуют во всех отношениях с арностью, большей двух, первичный ключ которых не совпадает с заголовком отношения. Например, если в переменной отношения СЛУЖ\_ПРО\_ЗАДАН атрибут СЛУ\_НОМ является первичным

ключом, то, очевидно, имеется  $RJD *(\{СЛУ\_НОМ, ПРО\_НОМ\}, \{СЛУ\_НОМ, СЛУ\_ЗАДАН\})$  (это следует из теоремы Хита). Но такие зависимости проекции/соединения неинтересны с точки зрения проектирования базы данных, поскольку не порождают аномалий обновления. Поэтому общепринятое определение пятой нормальной формы выглядит следующим образом.

**Определение 6.7.** Переменная отношения  $r$  находится в *пятой нормальной форме*, или в нормальной форме проекции/соединения (5NF, или PJ/NF — Project-Join Normal Form) в том и только том случае, когда каждая **нетривиальная**  $RJD$  в  $r$  подразумевается возможными ключами  $r$ . ■

Таким образом, чтобы распознать, что данная переменная отношения  $r$  находится в 5NF, необходимо знать все возможные ключи  $r$  и все  $RJD$  этой переменной отношения. Обнаружение всех зависимостей соединения является нетривиальной задачей, и для ее решения нет общих методов. Поэтому на практике проектирование реляционных баз методом нормализации обычно завершается после достижения 4NF, и отношения, находящиеся в 4NF, как правило, находятся и в 5NF. Зачем же тогда была введена эта туманная и труднодостижимая пятая нормальная форма?

Ответ на этот естественный вопрос состоит в том, что 5NF является «окончательной» нормальной формой, которой можно достичь в процессе нормализации на основе проекций. «Окончателность» понимается в том смысле, что у отношения, находящегося в 5NF, отсутствуют аномалии обновлений, которые можно было бы устранить путем его декомпозиции. Другими словами, такие отношения далее нормализовать бессмысленно.

\* \* \*

Процесс проектирования реляционной базы на основе метода нормализации преследует две основных цели:

- избежать избыточности хранения данных;
- устранить аномалии обновления отношений.

Рассмотрим, насколько эти цели актуальны в современных условиях, когда объемы доступных носителей внешней памяти непрерывно возрастают, стоимость их падает, а современные серверы реляционных (точнее, SQL-ориентированных) баз данных способны автоматически поддерживать целостность БД.

Начнем с того, что теория реляционных баз данных и методы их проектирования на основе нормализации разрабатывались в расчете на реляционную модель данных. Подавляющее боль-

шинство существующих в настоящее время баз данных и средств управления ими опирается на модель данных SQL. Поэтому, прежде всего, следует обсудить, насколько применимы методы нормализации при проектировании БД, основанных на модели данных SQL.

Как видно из краткого описания модели данных SQL, приведенного в гл. 2, основным ее отличием от реляционной модели данных является то, что таблица модели SQL может содержать мультимножества строк, и поэтому для таблицы, вообще говоря, может быть не определен никакой возможный ключ. Если потребовать от проектируемой SQL-базы данных наличия хотя бы одного возможного ключа для каждой таблицы целевой базы данных, то практически все методы нормализации остаются пригодными. Здесь требуется сделать два уточнения.

1. Для применимости методов нормализации требуется, чтобы в составе значений любого возможного ключа не допускалось присутствие неопределенных значений (в модели данных SQL это не является обязательным).

2. Для любого определяемого пользователями типа данных необходимо наличие операции сравнения значений этого типа по равенству. Другими словами, SQL-ориентированную базу данных можно проектировать как реляционную базу данных, если должным образом ограничить используемые средства модели данных SQL. Далее под реляционными базами данных будут пониматься SQL-ориентированные базы данных, не противоречащие требованиям реляционной модели данных.

С учетом этого замечания следует отметить два важных обстоятельства. Во-первых, теория реляционных баз данных и методы их проектирования активно развивались уже несколько десятилетий. Ситуация в области технологии аппаратуры и программного обеспечения в начале развития была совсем иной, чем в настоящее время, и хорошо нормализованные реляционные базы данных в значительной степени способствовали росту эффективности приложений.

Во-вторых, в то время реляционные базы преимущественно использовались в информационных системах оперативной обработки транзакций (On-Line Transaction Processing — OLTP). Характерные примеры таких систем отмечалось в гл. 1 — банковские системы, системы резервирования билетов и мест в гостиницах. Системам категории OLTP свойственны частые обновления базы данных, поэтому аномалии обновлений, даже если их корректировка производится СУБД автоматически, могут заметно снижать эффективность приложения.



В настоящее время на переднем крае приложений баз данных находятся системы категории оперативной аналитической обработки (On-Line Analytical Processing — OLAP). В подобных системах, в частности, системах поддержки принятия решений, базы данных в основном используются для выборки данных, поэтому аномалиями обновлений можно пренебречь, а объем этих баз настолько огромен, что можно пренебречь и избыточностью хранения.

Значит ли это, что подход к проектированию реляционных баз данных методом нормализации утратил свою роль? Нет!

Мир приложений баз данных в настоящее время огромен. Сегодня любое мало-мальски приличное предприятие использует хотя бы одно приложение баз данных — бухгалтерские, складские, кадровые системы. Это системы категории OLTP с частым обновлением данных и умеренными запросами к базе данных, не вызывающими соединений многих отношений. Для небольших компаний равно важны как эффективность информационных систем, так и стоимость используемых аппаратно-программных средств. Правильно спроектированные, хорошо нормализованные реляционные базы данных помогают решению корпоративных проблем.

Да, любое правильно развивающееся предприятие рано или поздно приходит к использованию систем категории OLAP, например, некоторой разновидности систем поддержки принятия решений (Decision Support System — DSS). В базах данных таких систем обновления очень редки, а запросы могут иметь произвольную сложность, включая соединения многих отношений. Но, во-первых, технологически правильно для системы OLAP поддерживать отдельную базу данных (обычно подобные базы данных называют *хранилищами данных* — *Data Warehouse*), а во-вторых, основными источниками данных для построения таких хранилищ данных являются базы данных систем OLTP. Так что актуальность правильно спроектированных баз данных OLTP-систем не уменьшается, а постоянно возрастает.

Следует ли из этого, что принципы нормализации непригодны для проектирования баз данных OLAP-приложений? И снова в ответ категорическое НЕТ! Возможно, окончательная схема такой базы данных должна быть денормализована из соображений повышения эффективности выполнения запросов. Но чтобы получить правильную денормализованную схему, нужно сначала понять, как выглядит нормализованная схема.

Основной вывод этой и предыдущей глав можно сформулировать следующим образом. Пока мы остаемся в мире реляцион-

ных баз данных, для правильного проектирования базы данных необходимо понимать принципы нормализации, воспринимая их не как догму, а как руководство к действию.

Кстати, это относится не только к «ручному» проектированию реляционных баз данных, но и к их проектированию с применением семантических моделей данных и CASE-средств, которые будут рассмотрены в гл. 7.

### ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ ДИАГРАММ «СУЩНОСТЬ-СВЯЗЬ» И ДИАГРАММ КЛАССОВ ЯЗЫКА UML

Широкое распространение реляционных (SQL-ориентированных, см. заключение к гл. 6) СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования разнообразных предметных областей. Однако проектирование реляционных баз данных в терминах отношений на основе кратко рассмотренного в гл. 5 и 6 механизма нормализации часто представляет собой очень сложный и неудобный для проектировщика процесс.

При использовании в проектировании ограниченность реляционной модели проявляется в следующих аспектах. (Начиная с этой главы вместо строгих реляционных терминов *отношение*, *атрибут* и *таблица* будут в основном использоваться термины *таблица*, *строка* и *столбец*, поскольку на практике приходится проектировать SQL-ориентированные базы данных, для которых эта упрощенная терминология более естественна.)

- Модель не обеспечивает достаточных средств для представления смысла данных. Семантика реальной предметной области должна независимым от модели способом представляться в голове проектировщика. В частности, это относится к проблеме представления ограничений целостности, выходящих за пределы ограничений первичного и внешнего ключа.
- Во многих прикладных областях трудно моделировать предметную область на основе плоских таблиц. В ряде случаев на самой начальной стадии проектирования дизайнеру приходится нелегко, поскольку от него требуется описать предметную область в виде одной (возможно, даже ненормализованной) таблицы.
- Хотя весь процесс проектирования происходит на основе учета функциональных и других зависимостей, реляционная модель не предоставляет какие-либо формализованные средства для представления этих зависимостей.
- Несмотря на то, что процесс проектирования начинается с выделения некоторых существенных для приложения

объектов предметной области («сущностей») и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо механизма для разделения сущностей и связей\*.

## 7.1. Семантические модели данных

Потребность проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области привела к появлению семантических моделей данных. Основным назначением семантических моделей является обеспечение возможности выражения семантики данных.

Прежде чем перейти к рассмотрению особенностей двух распространенных семантических моделей, остановимся на возможных областях их применения. Чаще всего на практике семантическое моделирование используется на первой стадии проектирования базы данных. В терминах семантической модели производится концептуальная схема базы данных, которая затем вручную преобразуется к реляционной (или какой-либо другой) схеме. Этот процесс выполняется под управлением методик, в которых достаточно четко оговорены все этапы такого преобразования. Основным достоинством данного подхода является отсутствие потребности в дополнительных программных средствах, поддерживающих семантическое моделирование. Требуется только знание основ выбранной семантической модели и правил преобразования концептуальной схемы в реляционную схему.

Следует заметить, что многие начинающие проектировщики баз данных недооценивают важность семантического моделирования вручную. Зачастую это воспринимается как дополнительная и излишняя работа. Эта точка зрения абсолютно неверна. Во-первых, построение мощной и наглядной концептуальной схемы БД позволяет более полно оценить специфику моделируемой предметной области и избежать возможных ошибок

---

\* Многие сторонники реляционного подхода считают отсутствие раздельного представления сущностей и связей преимуществом реляционной модели данных, мотивируя это тем, что зачастую то, что вчера считалось сущностью, сегодня разумнее принять за связь, и наоборот. Это, безусловно, верно с точки зрения поддержки и модификации существующих реляционных баз данных, но отнюдь не так с точки зрения проектирования базы данных.

на стадии проектирования схемы реляционной БД. Во-вторых, на этапе семантического моделирования производится важная документация (хотя бы в виде вручную нарисованных диаграмм и комментариев к ним), которая может оказаться очень полезной не только при проектировании схемы реляционной БД, но и при эксплуатации, сопровождении и развитии уже заполненной БД.

Неоднократно приходилось и приходится наблюдать ситуации, в которых отсутствие такого рода документации существенно затрудняет внесение даже небольших изменений в схему существующей реляционной БД. Конечно, это относится к случаям, когда проектируемая БД содержит не очень малое число таблиц. Скорее всего, без семантического моделирования можно обойтись, если число таблиц не превышает 10, но оно совершенно необходимо, если БД включает более 100 таблиц. Для справедливости заметим, что процедура создания концептуальной схемы вручную с ее последующим преобразованием в реляционную схему БД затруднительна в случае больших БД (содержащих несколько сотен таблиц). Причины, по всей видимости, не требуют пояснений.

История систем автоматизации проектирования баз данных (CASE-средств проектирования БД) началась с автоматизации процесса рисования диаграмм, проверки их формальной корректности, обеспечения средств долговременного хранения диаграмм и другой проектной документации. Конечно, компьютерная поддержка работы с диаграммами для проектировщика БД очень полезна. Наличие электронного архива проектной документации помогает при эксплуатации, администрировании, сопровождении и реинжиниринге систем баз данных.

Но система, которая ограничивается поддержкой рисования диаграмм, проверкой их корректности и хранением, напоминает текстовый редактор, поддерживающий ввод, редактирование и проверку синтаксической корректности конструкций некоторого языка программирования, но существующий отдельно от компилятора. Кажется естественным желание расширить такой редактор функциями компилятора, и это действительно возможно, поскольку известна техника компиляции конструкций языка программирования в коды целевого компьютера. Но коль скоро имеется четкая методика преобразования концептуальной схемы БД в реляционную схему, то почему бы не выполнить программную реализацию соответствующего «компилятора» и не включить ее в состав системы проектирования баз данных?

Эта идея, естественно, показалаcь разумной производителям CASE-средств проектирования БД. Подавляющее большинство подобных систем, представленных на рынке, обеспечивает автоматизированное преобразование диаграммных концептуальных схем баз данных, представленных в той или иной семантической модели данных, в реляционные схемы, специфицированные чаще всего на языке SQL. У читателя может возникнуть вопрос, почему в предыдущем предложении говорится про «автоматизированное», а не про «автоматическое» преобразование? Все дело в том, что в типичной схеме SQL-ориентированной БД могут содержаться определения многих объектов (ограничений целостности общего вида, триггеров и хранимых процедур и т. д.), которые невозможно сгенерировать автоматически на основе концептуальной схемы. Поэтому на завершающем этапе проектирования реляционной схемы снова требуется ручная работа проектировщика.

Еще раз обратите внимание на то, какой ход рассуждений привел нас к выводу о возможности автоматизации процесса преобразования концептуальной схемы БД в реляционную схему. Если создатели семантической модели данных предоставляют методику преобразования концептуальных схем в реляционные схемы, то почему бы не реализовать программу, которая производит те же преобразования, следуя той же методике? Зададимся теперь другим, но, по существу, схожим вопросом. Если создатели семантической модели данных предоставляют язык (например, диаграммный), используя который проектировщики БД на основе исходной информации о предметной области могут сформировать концептуальную схему БД, то почему бы не реализовать программу, которая сама генерирует концептуальную схему БД в соответствующей семантической модели, используя исходную информацию о предметной области?

Хотя мне не известны коммерческие CASE-средства проектирования БД, поддерживающие такой подход, экспериментальные системы успешно существовали. Они представляли собой интегрированные системы проектирования с автоматизированным созданием концептуальной схемы на основе интервью с экспертами предметной области и последующим преобразованием концептуальной схемы в реляционную схему.

Как правило, CASE-средства, автоматизирующие преобразование концептуальной схемы БД в реляционную, производят реляционную схему базы данных в третьей нормальной форме. Нормализация более высокого уровня усложняет программную реализацию и редко требуется на практике.

## 7.2. Семантическая модель Entity-Relationship (Сущность-Связь)

В этом подразделе мы кратко рассмотрим некоторые черты одной из наиболее популярных семантических моделей данных — модель «Сущность-Связь» (часто ее называют кратко ER-моделью (от *Entity-Relationship*)).

Прежде всего, требуется сделать замечание по поводу терминологии. Оба термина *relation* и *relationship* могут быть переведены на русский язык как *отношение*. Поэтому в русскоязычной литературе ER-модель иногда называют моделью *сущность-отношение*, а иногда и реляционной семантической моделью. Наверное, в этом нет ничего страшного, если говорить о ER-модели в отрыве от тематики проектирования реляционных баз данных.

Но если требуется одновременно использовать термины ER-модели и реляционной модели данных, то, безусловно, требуется применять для терминов *relation* и *relationship* разные русские эквиваленты. За этими терминами стоят весьма разные понятия. В реляционной модели отношение (*relation*) — это единственная родовая структура данных. С помощью этого же механизма представляются «связанные» сущности (вспомните, например, про внешние ключи). Как мы увидим немного позже, в ER-модели для представления схемы базы данных используются два равноправных понятия — *сущность* и *связь*. Связи в ER-модели играют роль, отличную от той, какую играют отношения в реляционной модели данных.

Кроме того, в русскоязычную терминологию вошла и чистая транслитерация термина *relation* именно в смысле *отношение*. Мы говорим, например, про реляционную модель данных, реляционную алгебру и т. д., понимая модель данных, основанную на отношениях, алгебру отношений и т. п. По этому поводу, по крайней мере, в контексте баз данных, разумно окончательно зарезервировать термины *relation* и *отношение* для обозначения понятий реляционной модели данных, а для термина *relationship* использовать другой допустимый русскоязычный эквивалент — *связь*.

На использовании разных вариантов ER-модели основано большинство современных подходов к проектированию баз данных (главным образом, реляционных). Модель была предложена Питером Ченом (Peter Chen) в 1976 г. [15]. Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число различных компонентов. Простота и наглядность представления

концептуальных схем баз данных в ER-модели привели к ее широкому распространению в CASE-системах, поддерживающих автоматизированное проектирование реляционных баз данных. Среди множества разновидностей ER-моделей одна из наиболее популярных и развитых применяется в CASE-системе компании Oracle. В этом подразделе описывается упрощенный вариант этой диаграммной модели, достаточный для понимания основных особенностей проектирования реляционных баз данных с использованием ER-моделей.

### 7.2.1. Основные понятия ER-модели

Основными понятиями ER-модели являются *сущность*, *связь* и *атрибут*. *Сущность* — это реальный или представляемый объект, информация о котором должна сохраняться и быть доступной. Это «определение» позаимствовано из одного из ранних вариантов документации CASE-системы Oracle. Понятно, что на самом деле мы имеем дело с тавтологией, поскольку, во-первых, пытаемся определить термин *сущность* через не определенный термин *объект*, а во-вторых, попытки определения термина *объект* настолько же безнадежны. Обычно авторы пытаются оправдываться тем, что в подобном контексте имеется в виду «житейское», а не сколько-нибудь формализованное понятие *объекта*. Конечно, от этого не становится легче, поскольку понятие *сущности* должно пониматься в достаточно точном смысле. Но эта тавтология традиционна для области семантического моделирования. В этой области стремятся максимально избегать формальностей.

В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности. При этом имя сущности — это имя типа, а не некоторого конкретного экземпляра этого типа. Хотя было бы правильнее всегда использовать термины *тип сущности* и *экземпляр типа сущности*, во избежание многословности (и следуя традиции) в тех случаях, где это не приводит к двусмысленности, мы будем использовать термин *сущность* в значении *типа сущности*.

Для большей выразительности и лучшего понимания имя сущности может сопровождаться примерами конкретных экземпляров этого типа. На рис. 7.1 изображена сущность АЭРОПОРТ с примерными экземплярами «Шереметьево» и «Хитроу». Эта примитивная диаграмма тем не менее несет важную



Рис. 7.1. Пример типа сущности



информацию. Во-первых, она показывает, что в базе данных будут содержаться однотипные структуры данных (экземпляры сущности), описывающие аэропорты. Во-вторых, поскольку в жизни существует несколько точек зрения на аэропорты (например, точка зрения пилота, точка зрения пассажира, точка зрения администратора) и этим точкам зрения соответствуют разные структуры данных, то приведенные примеры аэропортов позволяют несколько сузить допустимый набор точек зрения. В нашем случае приведены примеры международных аэропортов, так что, скорее всего, имеется точка зрения пассажира или пилота международных авиарейсов.

При определении типа сущности необходимо гарантировать, что каждый экземпляр типа сущности может быть отличим от любого другого экземпляра того же типа сущности. Это требование в некотором роде аналогично требованию отсутствия коротежей-дубликатов в реляционных таблицах.

*Связь* — это графически изображаемая ассоциация, устанавливаемая между двумя типами сущностей. Как и сущность, связь — это типовое понятие, все экземпляры обоих связываемых типов сущностей подчиняются устанавливаемым правилам связывания. Поэтому правильнее говорить о типе связи, устанавливаемой между типами сущности, и об экземплярах типа связи, устанавливаемых между экземплярами типа сущности. Тем не менее, как и в случае типа сущности, для краткости мы будем часто использовать термин *связь* в значении *типа связи*.

В обсуждаемом здесь варианте ER-модели связи всегда являются бинарными, т.е. соединяющими два типа сущности, и они могут существовать между двумя разными типами сущностей или между типом сущности и им же самим (рекурсивная связь). В любой связи выделяются два конца (в соответствии с существующей парой связываемых сущностей), на каждом из которых указываются имя конца связи, степень конца связи (сколько экземпляров данного типа сущности должно присутствовать в каждом экземпляре данного типа связи), обязательность связи (т.е. любой ли экземпляр данного типа сущности должен участвовать в некотором экземпляре данного типа связи). Заметим, что в некоторых вариантах ER-модели конец связи называют ролью связи в данной сущности. Тогда можно говорить об *имени роли*, *степени роли* и *обязательности роли* связи в данной сущности.

Связь представляется в виде ненаправленной линии, соединяющей две сущности или ведущей от сущности к ней же самой.



Рис. 7.2. Пример типа связи

При этом в месте «стыковки» связи с сущностью используются:

- трехточечный вход в прямоугольник сущности, если для этой сущности в связи могут (или должны) использоваться много (*many*) экземпляров сущности;
- одноточечный вход, если в связи может (или должен) участвовать только один (*one*) экземпляр сущности.

Обязательный конец связи изображается сплошной линией, а необязательный — прерывистой линией. Связь между сущностями **БИЛЕТ** и **ПАССАЖИР**, показанная на рис. 7.2, связывает билеты и пассажиров. Конец связи с именем «для» позволяет связывать с одним пассажиром более одного билета, причем каждый билет должен быть связан с каким-либо пассажиром. Конец связи с именем «имеет» показывает, что каждый билет может принадлежать только одному пассажиру, причем пассажир не обязан иметь хотя бы один билет.

Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

- каждый **БИЛЕТ** предназначен для одного и только одного **ПАССАЖИРА**;
- каждый **ПАССАЖИР** может иметь один или более **БИЛЕТОВ**.

На рис. 7.3 изображена рекурсивная связь, связывающая сущность **МУЖЧИНА** с ней же самой. Конец связи с именем «сын» определяет тот факт, что несколько мужчин могут быть сыновьями одного отца. Конец связи с именем «отец» означает, что не у каждого мужчины должны быть сыновья.

Лаконичная устная трактовка изображенной диаграммы состоит в следующем:

- каждый **МУЖЧИНА** является сыном одного и только одного **МУЖЧИНЫ**;
- каждый **МУЖЧИНА** может являться отцом одного или более **МУЖЧИН**.

Атрибутом сущности является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Имена

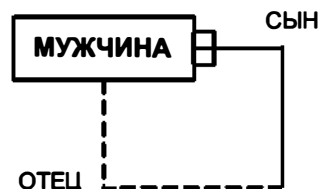


Рис. 7.3. Пример рекурсивного типа связи

атрибутов заносятся в прямоугольник, изображающий сущность, под именем сущности и изображаются малыми буквами, возможно, с примерами. Некоторые атрибуты могут помечаться как *необязательные*. Значения таких атрибутов не обязаны присутствовать во всех экземплярах данного типа сущности.

Пример типа сущности **ЧЕЛОВЕК** с указанными атрибутами показан на рис. 7.4. С технической точки зрения атрибуты типа сущности в ER-модели похожи на атрибуты отношения в реляционной модели данных. И в том, и в другом случаях введение именованных атрибутов вводит некоторую типовую структуру данных, имя которой совпадает с именем типа сущности в случае ER-модели или с именем переменной отношения в случае реляционной модели. Этой типовой структуре должны следовать все экземпляры типа сущности или все кортежи отношения. Но имеется и важное отличие.

Напомним, что в реляционной модели данных атрибут определяется как упорядоченная пара <имя\_атрибута, имя\_домена> (или <имя\_атрибута, имя\_базового\_типа\_данных>, если понятие домена не поддерживается). Заголовок отношения, определяемый как множество таких пар, представляет собой полный аналог структурного типа данных в языках программирования. При определении атрибутов типа сущности в ER-модели указание домена атрибута не является обязательным, хотя это и возможно (см. далее). Обсудим, чем вызвана эта возможность «ослабленного» определения атрибутов.

Прежде всего, как отмечалось в начале главы, семантические модели данных используются для построения концептуальных схем БД, и эти схемы преобразуются в реляционные схемы БД, которые поддерживаются той или иной СУБД. Несмотря на то, что в настоящее время типовые возможности РСУБД в основном стандартизованы (на основе стандарта языка SQL), детали базового набора типов данных и средств определения доменов в разных системах могут различаться. Поскольку производители CASE-средств проектирования реляционных БД стремятся не связывать обеспечиваемые ими возможности семантического моделирования с конкретной реализацией СУБД, они стиму-

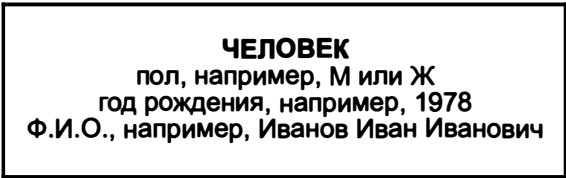


Рис. 7.4. Пример типа сущности с атрибутами

лируют откладывание строгого определения типов атрибутов до стадии полного определения реляционной схемы.

Кроме того, напомним, что при определении заголовка отношения допускается использование имен атрибутов, совпадающих с именами своих доменов (это два разных пространства имен, и наличие одинаковых имен у атрибутов и доменов не вызывает коллизий). Поэтому при определении атрибутов типов сущности можно так подбирать их имена, что они в дальнейшем будут подсказывать, какие домены у этих атрибутов имеются в виду. Пониманию предполагаемой сути доменов способствует и возможность указания примеров значений атрибутов. Например, на рис. 7.4 имеется атрибут **год рождения**, в качестве примерного значения которого указано «1978». Это подсказывает, что в реляционной схеме при определении соответствующего атрибута наиболее естественным базовым типом данных будет темпоральный тип «ДАТА», значения которого задают дату с точностью до года.

### **7.2.2. Уникальные идентификаторы типов сущности**

Как отмечалось в предыдущем подразделе, при определении типа сущности необходимо гарантировать, что каждый экземпляр сущности является отличным от любого другого экземпляра той же сущности. Поскольку сущность является абстракцией реального или представляемого объекта внешнего мира, это требование нужно иметь в виду уже при выборе кандидата в типы сущности.

Например, предположим, что проектируется база данных для поддержки работы книжного склада. На складе могут храниться произвольные части тиража любого издания любой книги. Может ли в этом случае индивидуальная книга являться прообразом типа сущности? Утверждается, что нет, поскольку отсутствует возможность различения книг одного издания. Для книжного склада прообразом типа сущности будет набор одноименных книг одного автора, вышедших в одном издании. Одним из атрибутов этого типа сущности будет число книг в наборе. Но когда книга поступает в библиотеку и ей присваивается уникальный библиотечный номер, она становится разумным прообразом типа сущности. Плохо устроены библиотеки, в которых не различаются индивидуальные книги (даже одноименные книги одного автора, вышедшие в одном издании).

Но при проектировании базы данных мало того, чтобы проектировщик убедился в правильном выборе типов сущности,

**КНИГА**  
автор  
название  
номер издания  
издательство  
год издания  
ISBN  
число книг на складе

Рис. 7.5. Тип сущности, экземпляры которого идентифицируются атрибутами

гарантирующем различие экземпляров каждого типа сущности. Необходимо сообщить системе автоматизации проектирования БД, каким образом будут различаться эти экземпляры, т.е. сообщить, как конструируются уникальные идентификаторы экземпляров каждого типа сущности. В ER-модели у экземпляра типа сущности не может быть назначаемого пользователем имени или назначаемого системой внешнего уникального идентификатора. Экземпляр типа сущности может идентифициро-

ваться только своими индивидуальными характеристиками, а они представляются значениями атрибутов и экземплярами типов связи, связывающими данный экземпляр типа сущности с экземплярами других типов сущности или этого же типа сущности. Поэтому уникальным идентификатором сущности может быть атрибут, комбинация атрибутов, связь, комбинация связей или комбинация связей и атрибутов, уникально отличающая любой экземпляр сущности от других экземпляров сущности того же типа.

Приведем несколько примеров. На рис. 7.5 показан тип сущности **КНИГА**, пригодный для использования в базе данных книжного склада. При издании любой книги в издательстве ей присваивается уникальный номер — ISBN. Понятно, что значение атрибута isbn будет уникально идентифицировать партию книг на складе. Кроме того, конечно, в качестве уникального идентификатора годится и комбинация атрибутов <автор, название, номер издания, издательство, год издания>.

На рис. 7.6 диаграмма включает два связанных типа сущности. У каждого обычного взрослого человека имеется один и только один паспорт (мы не берем в расчет особый случай, когда у одного человека имеется несколько паспортов, хотя это не изменило бы ситуацию), и каждый паспорт может принадлежать только одному взрослому человеку (некоторые уже готовые паспорта могут быть еще никому не выданы). Тогда



Рис. 7.6. Тип сущности, экземпляры которой идентифицируются связью

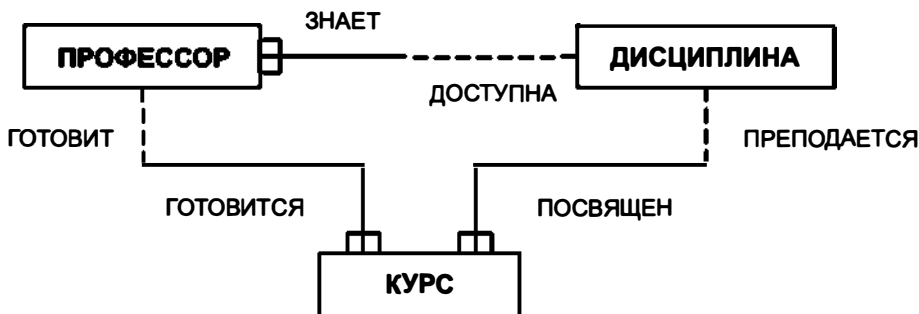


Рис. 7.7. Тип сущности, экземпляры которой идентифицируются комбинацией связей

связь человека с его паспортом (конец связи **ИМЕЕТ**) уникально идентифицирует взрослого человека, т.е., грубо говоря, паспорт определяет взрослого человека. Поскольку могут существовать паспорта, еще не выданные какому-либо человеку, эта связь не является уникальным идентификатором сущности **ПАСПОРТ**.

На диаграмме, приведенной на рис. 7.7, имеются три связанных типа сущности. Профессора обладают знаниями в нескольких учебных дисциплинах. Преподавание каждой дисциплины доступно нескольким профессорам. Другими словами, между сущностями **ПРОФЕССОР** и **ДИСЦИПЛИНА** определена связь «многие ко многим». Каждый профессор может готовить курсы по любой доступной ему дисциплине. Каждой дисциплине может быть посвящено несколько учебных курсов. Но каждый профессор может готовить только один курс по любой доступной ему дисциплине, и каждый курс может быть посвящен только одной дисциплине. Тем самым, каждый экземпляр типа сущности **КУРС** уникально идентифицируется экземпляром сущности **ПРОФЕССОР** и экземпляром сущности **ДИСЦИПЛИНА**, т.е. парой связей с именами концов **ГОТОВИТСЯ** и **ПОСВЯЩЕН** на стороне сущности **КУРС**. Заметим, что сущности **ПРОФЕССОР** и **ДИСЦИПЛИНА** связями не идентифицируются.

Наконец, на рис. 7.8 приведен пример типа сущности, уникальный идентификатор которого является комбинацией атрибутов и связей. Это несколько уточненный вариант сущности с рекурсивной связью с рис. 7.3. У каждого человека могут быть дети, и у каждого человека имеется отец. Тогда, если предположить, что близнецам, появившимся на свет одновре-

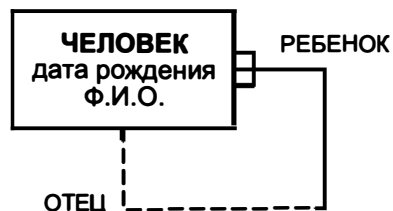


Рис. 7.8. Тип сущности, экземпляры которой идентифицируются комбинацией атрибутов и связей

менно, не дают одинаковых имен, то уникальным идентификатором типа сущности **ЧЕЛОВЕК** может быть комбинация атрибутов <дата рождения, ФИО> и связь с именем конца **РЕБЕНОК**.

### 7.2.3. Нормальные формы ER-диаграмм

Как и в случае схем реляционных баз данных, для ER-диаграмм вводится понятие нормальных форм, причем их смысл очень близко соответствует смыслу нормальных форм отношений. Заметим, что определения нормальных форм ER-диаграмм делают более понятным смысл нормализации схем отношений. Мы приведем только очень краткие и неформальные определения трех первых нормальных форм. Конечно, можно было бы ввести дальнейшие нормальные формы ER-диаграмм, аналогичные нормальной форме Бойса—Кодда, 4NF и 5NF, но на практике к такой нормализации обычно не прибегают, а общие идеи должны быть понятны после ознакомления с гл. 7.

**Первая нормальная форма ER-диаграммы.** В первой нормальной форме ER-диаграммы устраняются атрибуты, содержащие множественные значения, т. е. производится выявление неявных сущностей, «замаскированных» под атрибуты.

На рис. 7.9, а показана диаграмма, в которой тип сущности **АЭРОДРОМ** не удовлетворяет требованию первой нормальной формы. Здесь несущественны атрибуты сущности **АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ**, но сущность **АЭРОДРОМ** помимо атрибутов, отражающих собственные характеристики аэродромов (длина взлетно-посадочной полосы, число ангаров и т. д.), содержит атрибут, множественное значение которого характеризует самолеты, приписанные к этому аэродрому. Очевидно, что самолеты нуждаются в ремонте, т. е. должны обслуживаться некоторым авиаремонтным предприятием. Но поскольку самолеты являются частью сущности **АЭРОДРОМ**, единственным способом фиксации этого факта на диаграмме является проведение связи «многие ко многим» между типами сущности **АЭРОДРОМ** и **АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ**. Таким образом выражается то соображение, что для ремонта разных самолетов, приписанных к одному аэродрому, могут использоваться разные авиаремонтные предприятия, и каждое авиаремонтное предприятие может обслуживать несколько аэродромов.

Чем плоха эта ситуация? Прежде всего, тем, что скрывается тот факт, что авиаремонтное предприятие ремонтирует самолеты, а не аэродромы. Связь же между типами сущности **АЭРОДРОМ** и **АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ** на самом деле

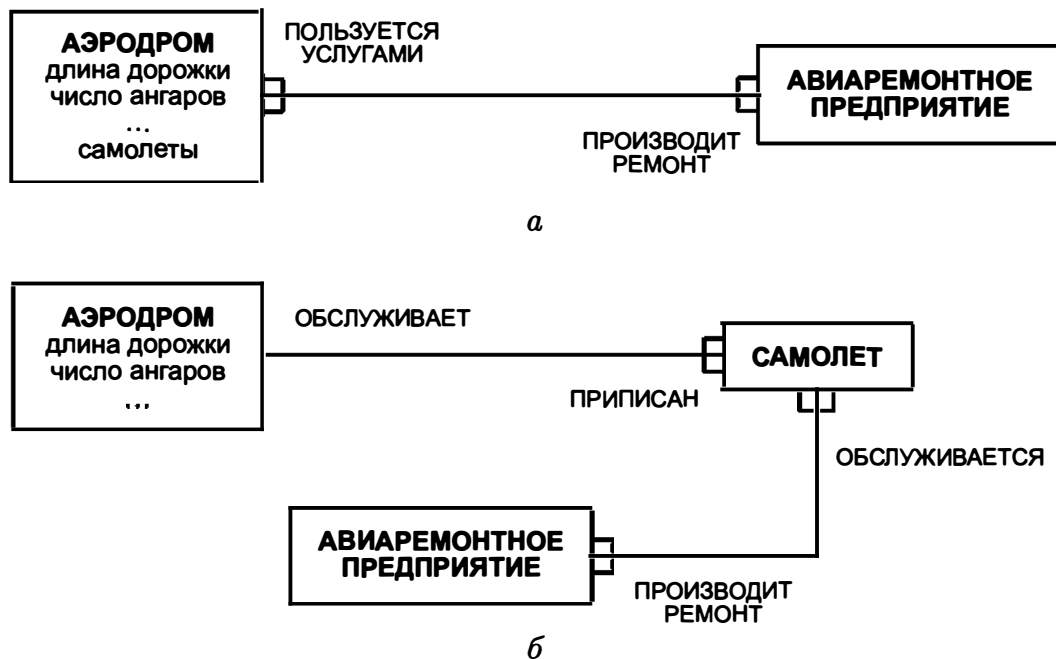


Рис. 7.9. Пример приведения ER-диаграммы к первой нормальной форме:

*а* — ER-диаграмма, не находящаяся в первой нормальной форме; *б* — аналог ER-диаграммы после приведения к первой нормальной форме

означает, что любой аэродром из группы аэродромов обслуживается любым авиаремонтным предприятием из группы таких предприятий. Проблема состоит именно в том, что значением атрибута «самолеты» является множество экземпляров типа сущности **САМОЛЕТ**, и этот тип сущности сам обладает атрибутами и связями.

Ситуацию исправляет ER-диаграмма, показанная на рис. 7.9, б. Здесь мы выделили тип сущности **САМОЛЕТ**. Связь между сущностями **АЭРОПОРТ** и **САМОЛЕТ** показывает, что к одному аэродрому приписывается несколько самолетов. Связь между сущностями **САМОЛЕТ** и **АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ** означает, что каждый самолет из группы самолетов (группу самолетов могут составлять, например, все самолеты одного типа) обслуживается любым транспортным предприятием из некоторой группы таких предприятий. ER-диаграмма на рис. 7.9, б находится в первой нормальной форме и, как мы видим, правильнее отображает реальную ситуацию.

**Вторая нормальная форма ER-диаграммы.** Во второй нормальной форме устраняются атрибуты, зависящие только от части уникального идентификатора. Эта часть уникального идентификатора определяет отдельную сущность.



На рис. 7.10, *а* показана диаграмма, на которой тип сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** не удовлетворяет требованиям второй нормальной формы. На этой диаграмме у сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** имеются следующие свойства. Элементы расписания предназначены для сохранения данных о рейсах самолетов, вылетающих в течение дня. Некоторыми важными характеристиками рейса являются номер рейса, аэропорт вылета, аэропорт назначения, дата и время вылета, бортовой номер самолета, тип самолета. Если говорить про российские авиационные компании, то, во-первых, у каждого рейса имеется заранее приписанный ему номер (уникальный среди всех других имеющихся номеров рейсов), во-вторых, не все рейсы совершаются каждый день, поэтому характеристикой конкретного рейса является дата и время его совершения, в-третьих, бортовой номер самолета определяется парой <номер рейса, дата-время вылета>. Имеется связь «многие к одному» между сущностями **ЭЛЕМЕНТ РАСПИСАНИЯ** и **ГОРОД** (каждый день много рейсов прибывает в один и тот же город). Экземпляры типа сущности **ГОРОД** характеризуют город, в который прибывает данный рейс.

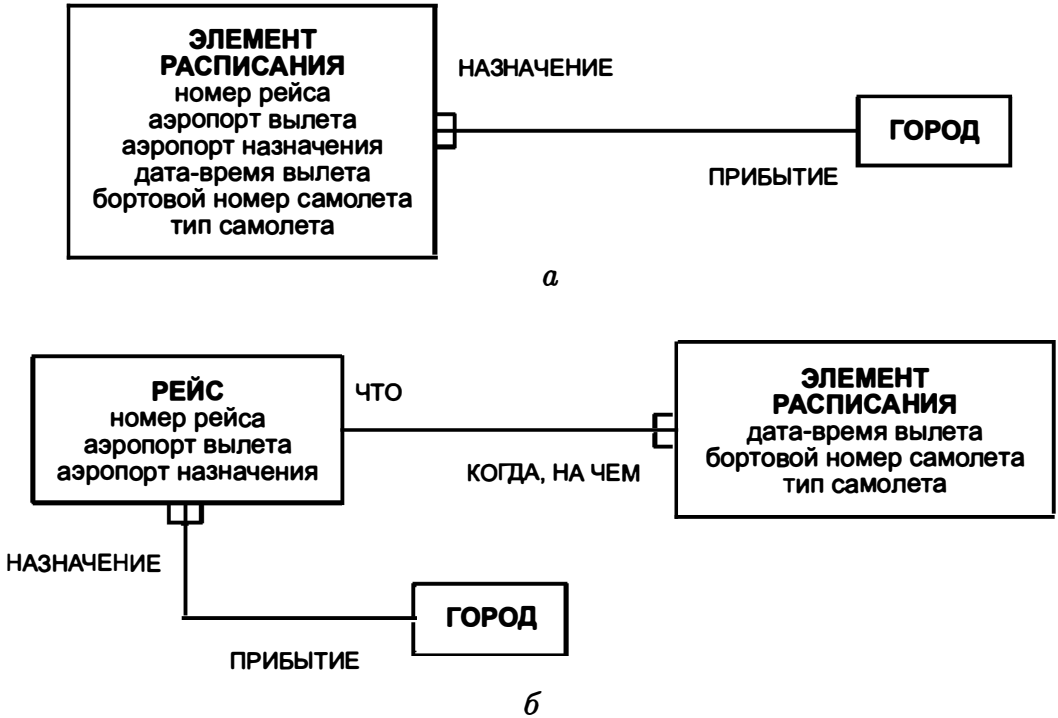


Рис. 7.10. Пример приведения ER-диаграммы ко второй нормальной форме:

*а* — ER-диаграмма, не находящаяся во второй нормальной форме; *б* — аналог ER-диаграммы после приведения ко второй нормальной форме

Уникальным идентификатором типа сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** является пара атрибутов <номер рейса, дата-время вылета>. Если вернуться к терминам функциональных зависимостей, то между атрибутами этой сущности имеются следующие FD:

- {номер рейса, дата-время вылета} → бортовой номер самолета;
- номер рейса → аэропорт вылета;
- номер рейса → аэропорт назначения;
- бортовой номер самолета → тип самолета.

Кроме того, очевидно, что каждый экземпляр связи с сущностью **ГОРОД** также определяется значением атрибута **номер рейса**. Налицо нарушение требования второй нормальной формы. Мы получаем не только избыточное хранение значений атрибутов аэропорт вылета и аэропорт назначения в каждом экземпляре типа сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** с одним и тем же значением номера рейса. Искажается и затемняется смысл связи с сущностью **ГОРОД**. Можно подумать, что в разные дни один и тот же рейс прибывает в разные города.

На рис. 7.10, б показан нормализованный вариант диаграммы, в котором все сущности находятся во второй нормальной форме. Теперь имеются три типа сущности: **РЕЙС** с атрибутами номер рейса, дата-время вылета, аэропорт назначения; **ЭЛЕМЕНТ РАСПИСАНИЯ** с атрибутами дата-время вылета, бортовой номер самолета, тип самолета; **ГОРОД**. Уникальным идентификатором сущности **РЕЙС** является атрибут номер рейса; уникальный идентификатор **ЭЛЕМЕНТ РАСПИСАНИЯ** состоит из атрибута дата-время вылета и конца связи **КОГДА, НА ЧЕМ**. Мы видим, что ни в одном типе сущности больше нет атрибутов, определяемых частью уникального идентификатора. Свойства второй нормальной формы удовлетворяются, и мы имеем более качественную диаграмму.

**Третья нормальная форма 3NF-диаграммы.** В третьей нормальной форме устраняются атрибуты, зависящие от атрибутов, не входящих в уникальный идентификатор. Эти атрибуты являются основой отдельной сущности.

Посмотрим еще раз на тип сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** (см. рис. 7.10, б). Конечно, каждый день каждый рейс выполняется только одним самолетом, поэтому бортовой номер самолета полностью зависит от уникального идентификатора. Но бортовой номер является уникальной характеристикой каждого самолета, и от этой характеристики зависят все остальные характеристики, в частности тип самолета. Другими словами, между уникальным идентификатором и другими атрибутами

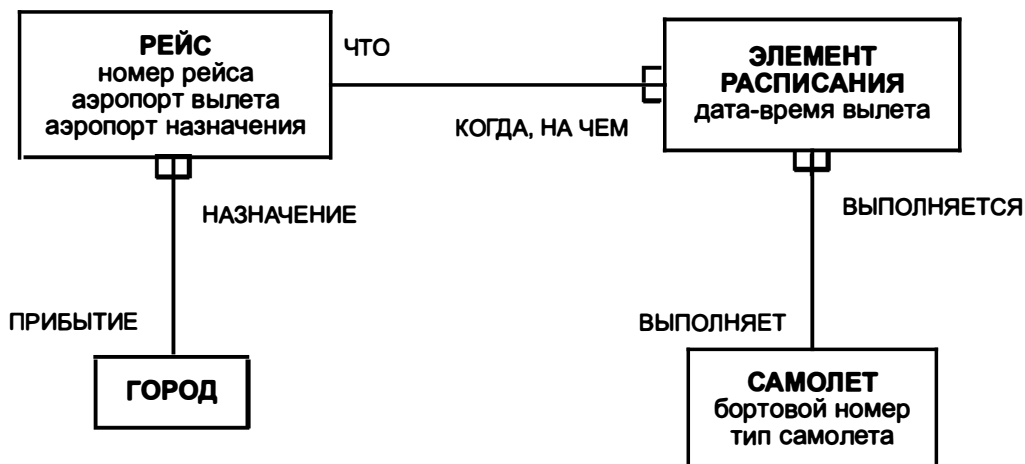


Рис. 7.11. Пример приведения ER-диаграммы к третьей нормальной форме

типа сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** имеются следующие функциональные зависимости:

- {КОГДА, НА ЧЕМ, дата-время вылета} → бортовой номер самолета;
- {КОГДА, НА ЧЕМ, дата-время вылета} → тип самолета;
- бортовой номер самолета → тип самолета.

Как видно, имеется транзитивная FD {КОГДА, НА ЧЕМ, дата-время вылета} → тип самолета, и наличие этой FD вызывает нарушение требования третьей нормальной формы. На самом деле, тип сущности **ЭЛЕМЕНТ РАСПИСАНИЯ** на рис. 7.10, б включает в себя (по крайней мере, частично) тип сущности **САМОЛЕТ**. Это вызывает избыточность хранения и затуманивает смысл диаграммы. На рис. 7.11 показан нормализованный вариант диаграммы, в котором все типы сущности находятся в третьей нормальной форме.

#### 7.2.4. Более сложные элементы ER-модели

До сих пор рассматривались только самые основные и наиболее очевидные понятия ER-модели данных. К числу некоторых более сложных элементов модели относятся следующие.

- *Подтипы и супертипы сущностей.* Подобно тому, как это делается в языках программирования с развитыми типовыми системами (например, в языках объектно-ориентированного программирования), в ER-модели поддерживается возможность наследования типа сущности от одного супертипа сущности. Механизм наследования в ER-модели обладает несколькими особенностями: в частности, инте-

ресные нюансы связаны с необходимостью графического изображения этого механизма (более подробно механизм наследования рассматривается далее в этом подразделе).

- *Уточняемые степени связи.* Иногда бывает полезно определить возможное количество экземпляров сущности, участвующих в данной связи (примером может служить то ограничение, что служащему разрешается участвовать не более чем в трех проектах одновременно). Для выражения этого семантического ограничения разрешается указывать на конце связи ее максимально допустимую или обязательную степень.
- *Взаимно исключающие связи.* Для заданного типа сущности можно определить такой набор типов связи с другими типами сущности, что для каждого экземпляра заданного типа сущности может (если набор связей является необязательным) или должен (если набор связей обязателен) существовать экземпляр только одной связи из этого набора.
- *Каскадные удаления экземпляров сущностей.* Некоторые связи бывают настолько сильными (конечно, в случае связи «один ко многим»), что при удалении опорного экземпляра сущности (соответствующего концу связи «один») нужно удалить и все экземпляры сущности, соответствующие концу связи «многие». Соответствующее требование каскадного удаления можно специфицировать при определении связи.
- *Домены.* Как и в случае реляционной модели данных, в некоторых случаях полезна возможность определения потенциально допустимого множества значений атрибута сущности (домена).

Приведенные и другие усложненные элементы ER-модели делают ее более мощной, но одновременно несколько затрудняют ее использование. Конечно, при реальном применении ER-диаграмм для проектирования баз данных необходимо ознакомиться со всеми возможностями. Далее будет подробно разобрана суть механизма наследования в ER-модели, а также приведем пример типа сущности с взаимно исключающими связями.

**Наследование типов сущности и типов связи.** Тип сущности может быть расщеплен на два или более взаимно исключающих подтипов, каждый из которых включает общие атрибуты и/или связи. Эти общие атрибуты и/или связи явно определяются один раз на более высоком уровне. В подтипах могут определяться собственные атрибуты и/или связи. В прин-

ципе, подтипизация может продолжаться на более низких уровнях, но опыт использования ER-модели при проектировании баз данных показывает, что в большинстве случаев оказывается достаточно двух-трех уровней.

Особенности механизма наследования в ER-модели определяются следующими правилами. Если у типа сущности  $A$  имеются подтипы  $B_1, B_2, \dots, B_n$ , то:

а) любой экземпляр типа сущности  $B_1, B_2, \dots, B_n$  является экземпляром типа сущности  $A$  (включение);

б) если  $a$  является экземпляром типа сущности  $A$ , то  $a$  является экземпляром некоторого подтипа сущности  $B_i$  ( $i = 1, 2, \dots, n$ ) (отсутствие собственных экземпляров у супертипа сущности);

в) ни для каких подтипов  $B_i$  и  $B_j$  ( $i, j = 1, 2, \dots, n$ ) не существует экземпляра, типом которого одновременно являются типы сущности  $B_i$  и  $B_j$  (разъединенность подтипов).

Тип сущности, на основе которого определяются подтипы, называется супертипом. Как мы видели ранее, объединение множества экземпляров подтипов должно образовывать полное множество экземпляров супертипа, т. е. любой экземпляр супертипа должен относиться к некоторому подтипу. Иногда для обеспечения такой полноты приходится определять дополнительный подтип ПРОЧIE.

На рис. 7.12 показан пример супертипа ЛЕТАТЕЛЬНЫЙ АППАРАТ и его подтипов АЭРОПЛАН, ВЕРТОЛЕТ, ПТИЦЕЛЕТ и ПРОЧIE. У подтипа АЭРОПЛАН имеются два собственных подтипа — ПЛАНЕР и МОТОРНЫЙ САМОЛЕТ. Для супертипа сущности ЛЕТАТЕЛЬНЫЙ АППАРАТ определен атрибут максимальная дальность полета и необязательная связь «многие ко многим» с типом сущности ПИЛОТ. Эти атрибут и связь наследуются всеми подтипами этого супертипа сущности. У непосредственного подтипа сущности АЭРОПЛАН определяется один дополнительный атрибут, так что в совокупности у данного типа сущности имеются два атрибута максимальная дальность полета и размах крыльев и одна унаследованная связь с типом сущности ПИЛОТ. У подтипа второго уровня МОТОРНЫЙ САМОЛЕТ супертипа АЭРОПЛАН определяется один дополнительный атрибут мощность мотора и одна дополнительная (обязательная) связь с типом сущности АЭРОДРОМ. Тем самым, у типа сущности МОТОРНЫЙ САМОЛЕТ имеются три атрибута: два унаследованных — максимальная дальность полета и размах крыльев и один собственный — мощность мотора, а также две связи: одна унаследованная — с типом сущности ПИЛОТ и одна собственная — с типом сущности АЭРОДРОМ. И так далее. Понятно, что для типа сущности ПРОЧIE, скорее всего, бессмысленно определять

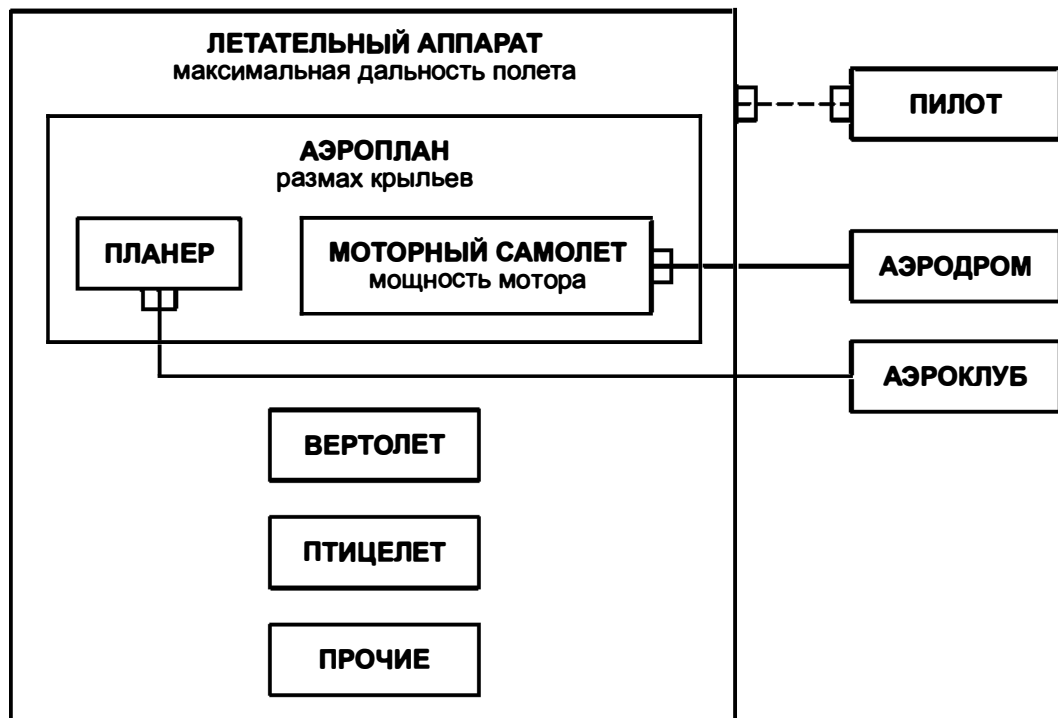


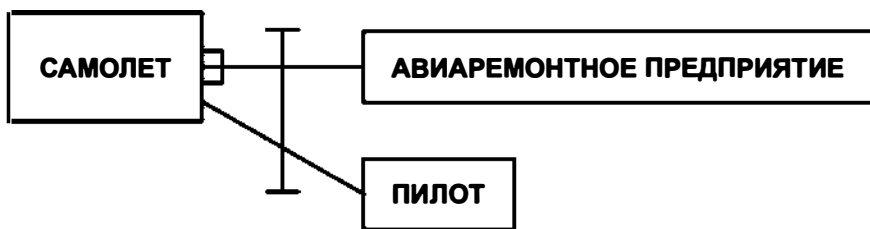
Рис. 7.12. Супертипы и подтипы сущности

собственные атрибуты и связи, так что свойства этого типа будут совпадать со свойствами его супертипа.

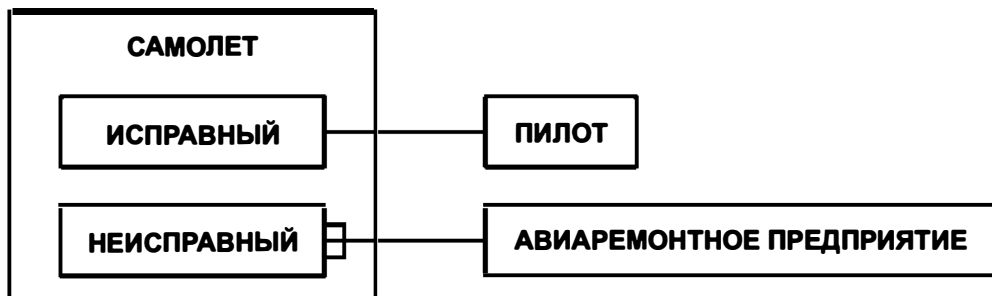
Как же следует понимать диаграмму, представленную на рис. 7.12? Если начинать от супертипа, то диаграмма изображает **ЛЕТАТЕЛЬНЫЙ АППАРАТ**, который должен быть **АЭРОПЛАНОМ**, **ВЕРТОЛЕТОМ**, **ПТИЦЕЛЕТОМ** или **ДРУГИМ ЛЕТАТЕЛЬНЫМ АППАРАТОМ**. Если начинать от подтипа (например, сущности **ВЕРТОЛЕТ**), то это **ВЕРТОЛЕТ**, который относится к типу **ЛЕТАТЕЛЬНОГО АППАРАТА**. Если начинать от подтипа, который является одновременно супертипом, то это **АЭРОПЛАН**, который относится к типу **ЛЕТАТЕЛЬНОГО АППАРАТА** и должен быть **ПЛАНЕРОМ** или **МОТОРНЫМ САМОЛЕТОМ**.

В механизме наследования ER-модели допускается наличие двух или более разбиений сущности на подтипы. Например, тип сущности **ЧЕЛОВЕК** может быть расщеплен на подтипы по профессиональному признаку (**ПРОГРАММИСТ**, **ДОЯРКА** и т. д.), а может быть расщеплен и по половому признаку (**МУЖЧИНА**, **ЖЕНЩИНА**).

**Взаимно исключающие связи.** Пример диаграммы из двух сущностей с взаимно исключающими связями показан на рис. 7.13, а. Самолет может находиться в рабочем состоянии, и тогда у него имеется один и только один пилот. Или же само-



*а*



*б*

Рис. 7.13. Пример ER-диаграммы с взаимно исключающими связями:

*а* — ER-диаграмма с взаимно исключающими связями; *б* — аналог ER-диаграммы без взаимно исключающих связей, но с подтипами

лет может быть неисправным, и тогда он находится на ремонте на некотором авиаремонтном предприятии (каждое предприятие может производить ремонт нескольких самолетов).

В данном случае для каждого экземпляра типа сущности **САМОЛЕТ** должен существовать экземпляр одной из указанных связей. Для экземпляров типа сущности **САМОЛЕТ**, соответствующих исправным самолетам, должен существовать экземпляр связи «один к одному» с экземпляром типа сущности **ПИЛОТ**, а экземпляры, соответствующие неисправным самолетам, должны участвовать в экземпляре типа связи «многие ко одному» с экземпляром типа сущности **АВИАРЕМОНТНОЕ ПРЕДПРИЯТИЕ**.

Как показано на рис. 7.13, *б*, диаграмма с взаимно исключающими связями из рис. 7.13, *а* может быть преобразована к диаграмме без взаимно исключающих связей путем введения подтипов. Поскольку любой самолет может быть либо исправным, либо неисправным, можно корректным образом ввести два подтипа супертипа **САМОЛЕТ** — **ИСПРАВНЫЙ САМОЛЕТ** и **НЕИСПРАВНЫЙ САМОЛЕТ**. На уровне супертипа сущности связи не определяются. Для подтипа **ИСПРАВНЫЙ САМОЛЕТ** определяется обязательная связь «один к одному» с типом сущности **ПИЛОТ**, а для подтипа **НЕИСПРАВНЫЙ САМОЛЕТ** определяется

### 7.2.5. Получение реляционной схемы из ER-диаграммы

Опишем типовую многошаговую процедуру преобразования ER-диаграммы в реляционную (более точно, в SQL-ориентированную) схему базы данных. Следует заметить, что в этом подразделе предполагается использование «традиционных» средств определения данных SQL, не включающих возможности определения структурных типов данных с поддержкой механизма наследования типов и типизированных таблиц (см. гл. 2 и 11). Ко времени написания этой книги отсутствует общепризнанная методология проектирования SQL-ориентированных баз данных, в которых используются «объектные» расширения SQL.

**Базовые приемы.** Каждый *простой тип сущности* превращается в таблицу. (Простым типом сущности называется тип сущности, не являющийся подтипом и не имеющий подтипов.) *Имя типа сущности* становится именем таблицы. *Экземплярам* типа сущности соответствуют *строки* соответствующей таблицы.

Каждый *атрибут* становится *столбцом* таблицы с тем же именем; может выбираться более точный формат представления данных. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределенные значения; столбцы, соответствующие обязательным атрибутам, — не могут.

*Компоненты уникального идентификатора сущности* превращаются в *первичный ключ* таблицы. Если имеется несколько возможных уникальных идентификаторов, для первичного ключа выбирается наиболее характерный уникальный идентификатор. Если в состав уникального идентификатора входят связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящейся на дальнем конце связи (этот процесс может продолжаться рекурсивно, и в общем случае может привести к заикливанию). Для именования этих столбцов используются имена концов связей и/или имена парных типов сущностей.

*Связи «многие к одному»* (и «один к одному») становятся внешними ключами, т. е. образуется копия уникального идентификатора сущности на конце связи «один», и соответствующие столбцы составляют внешний ключ таблицы, соответствующей типу сущности на конце связи «многие». Необязательные связи соответствуют столбцам внешнего ключа, допускающим наличие неопределенных значений; обязательные связи — столбцам,



не допускающим неопределенных значений. Если между двумя типами сущности *A* и *B* имеется связь «один к одному», то соответствующий внешний ключ по желанию проектировщика может быть объявлен как в таблице *A*, так и в таблице *B*. Чтобы отразить в определении таблицы ограничение, которое заключается в том, что степень конца связи должна равняться единице, соответствующий (возможно, составной) столбец должен быть дополнительно специфицирован как возможный ключ таблицы (в случае использования языка SQL для этого служит спецификация **UNIQUE**, см. гл. 11).

Для поддержки связи «многие ко многим» между типами сущности *A* и *B* создается дополнительная таблица *AB* с двумя столбцами, один из которых содержит уникальные идентификаторы экземпляров типа сущности *A*, а другой — уникальные идентификаторы экземпляров типа сущности *B*. Обозначим через **УИД(с)** уникальный идентификатор экземпляра некоторого типа сущности *C*. Тогда, если в экземпляре связи «многие ко многим» участвуют экземпляры  $a_1, a_2, \dots, a_n$  типа сущности *A* и экземпляры  $b_1, b_2, \dots, b_m$  типа сущности *B*, то в таблице *AB* должны присутствовать все строки вида **<УИД( $a_i$ ), УИД( $b_j$ )>** для  $i = 1, 2, \dots, n, j = 1, 2, \dots, m$ . Понятно, что, используя таблицы *A*, *B* и *AB*, с помощью стандартных реляционных операций можно найти все пары экземпляров типов сущности, участвующих в данной связи.

**Представление в реляционной схеме супертипов и подтипов сущности.** Если в концептуальной схеме (ER-диаграмме) присутствуют подтипы сущностей, то возможны два способа их представления в реляционной схеме:

- 1) собрать все подтипы в одной таблице;
- 2) для каждого подтипа образовать отдельную таблицу.

При применении первого способа таблица создается для максимального супертипа (типа сущности, не являющегося подтипом), а для подтипов определяются представления (см. гл. 11). Таблица содержит столбцы, соответствующие каждому атрибуту (и связям) каждого подтипа. В таблицу добавляется, по крайней мере, один столбец, содержащий «код типа»; он становится частью первичного ключа. Для каждой строки таблицы значение этого столбца определяет конкретный тип сущности, экземпляру которого соответствует строка. Столбцы этой строки, которые соответствуют атрибутам и связям, отсутствующим в данном типе сущности, должны содержать неопределенные значения.

При использовании второго способа для каждого подтипа первого уровня (непосредственного подтипа максимального супер-

типа) создается отдельная таблица. Для более глубоких уровней наследования применяется первый способ. Супертип воссоздается с помощью объединения проекций таблиц, соответствующих подтипам, на заголовок таблицы супертипа (т.е. из всех таблиц подтипов выбираются общие столбцы — столбцы супертипа).

У каждого способа есть свои достоинства и недостатки. К достоинствам первого способа (общая таблица для супертипа и всех его подтипов) можно отнести следующие:

- соответствие логике супертипов и подтипов: поскольку любой экземпляр любого подтипа является экземпляром супертипа, логично хранить вместе все строки, соответствующие экземплярам супертипа;
- обеспечение простого доступа к экземплярам супертипа и не слишком сложный доступ к экземплярам подтипов;
- возможность обойтись небольшим числом таблиц.

Приведем недостатки первого способа:

- приложения, работающие с одной таблицей супертипа, должны содержать дополнительный программный код для работы с разными наборами столбцов (в зависимости от значения столбца «кода типа») и разными ограничениями целостности (в зависимости от особенностей связей, определенных для подтипа);
- общая для всех подтипов таблица потенциально может стать узким местом при многопользовательском доступе по причине возможности блокировки таблицы целиком;
- потенциально в общей таблице будет содержаться много неопределенных значений, что может привести к непроизводительному расходу внешней памяти.

Достоинства второго способа состоят в следующем:

- действуют более понятные правила работы с подтипами (каждому подтипу соответствует одноименная таблица);
- упрощается логика приложений; каждая программа работает только с нужной таблицей.

Перечислим недостатки второго способа:

- в общем случае требуется слишком много отдельных таблиц;
- работа с экземплярами супертипа на основе представления, объединяющего таблицы супертипов, может оказаться недостаточно эффективной;
- поскольку множество экземпляров супертипа является объединением множеств экземпляров подтипов, не все РСУБД могут обеспечить выполнение операций модификации экземпляров супертипа.

**Представление в реляционной схеме взаимно исключающих связей.** Как отмечалось в подразд. 7.2.4, ER-диаграмму с взаимно исключающими связями можно преобразовать к диаграмме с подтипами исходного типа сущности. Однако можно построить схему SQL-ориентированной базы данных и без такого преобразования.

Существуют два способа формирования схемы реляционной БД при наличии взаимно исключающих связей (имеются в виду связи «один ко многим», причем конец связи «многие» находится на стороне сущности, для которой связи являются взаимно исключающими):

- 1) определение таблицы с одним столбцом для представления всех взаимно исключающих связей, т. е. общее хранение внешних ключей;

- 2) определение таблицы, в которой каждой взаимно исключающей связи соответствует отдельный столбец, т. е. раздельное хранение внешних ключей.

Понятно, что если имеются взаимно исключающие связи упомянутой категории, то в таблице, соответствующей сущности, для которой связи являются взаимно исключающими, необходимо хранить внешние ключи. Если внешние ключи всех потенциально связанных таблиц имеют общий формат, то можно применить первый способ, т. е. создать два столбца, содержащие идентификатор связи и уникальный идентификатор соответствующей сущности (второй столбец может быть составным). Столбец идентификатора связи используется для различения связей, покрываемых дугой исключения.

Если результирующие внешние ключи не относятся к одному домену, то приходится прибегать к использованию второго способа, т. е. создавать для каждой связи, покрываемой дугой исключения, явные столбцы внешних ключей; каждый из этих столбцов может содержать неопределенные значения.

Преимущество первого подхода состоит в том, что в таблице, соответствующей сущности с взаимно исключающими связями, появляется всего два дополнительных столбца. Очевидным недостатком является усложнение выполнения операции соединения: чтобы воспользоваться для соединения одной из альтернативных связей, нужно сначала произвести ограничение таблицы в соответствии с нужным значением столбца, содержащего идентификаторы связей.

При использовании второго подхода соединения являются явными (и естественными). Недостаток состоит в том, что требуется иметь столько столбцов, сколько имеется альтерна-

тивных связей. Кроме того, в каждом из таких столбцов будет содержаться много неопределенных значений, хранение которых может привести к излишнему расходу внешней памяти.

На этом мы заканчиваем краткую экскурсию в семантическое моделирование с использованием ER-диаграмм. Основной целью этого подраздела было ознакомление с семантическими моделями данных на примере упрощенного варианта ER-модели. Представленный вариант ER-модели, с одной стороны, является достаточно развитым, чтобы можно было почувствовать общую специфику семантических моделей данных, а с другой стороны, не перегружен деталями и излишними понятиями, затрудняющими общее понимание подхода.

С практической точки зрения наибольшую пользу могут принести рассмотренные приемы перехода от ER-диаграмм к схеме реляционной базы данных. Особенно могут пригодиться рекомендации по представлению в реляционной схеме связей «многие ко многим», подтипов и супертипов сущности и взаимно исключающих связей.

### **7.3. Диаграммы классов языка UML**

В этом подразделе обсудим основные понятия диаграмм классов языка UML и возможности применения этой диаграммной модели для проектирования реляционных баз данных. Кроме того, будет кратко рассмотрен язык объектных ограничений OCL и приведены примеры формулировок на языке OCL ограничений целостности в терминах концептуальной схемы базы данных.

Языку объектно-ориентированного моделирования UML (Unified Modeling Language) посвящено великое множество книг, многие из которых переведены на русский язык (а некоторые и написаны российскими авторами). Язык UML разработан и развивается консорциумом OMG (Object Management Group) и имеет много общего с объектными моделями, на которых основана технология распределенных объектных систем CORBA, и объектной моделью ODMG (Object Data Management Group).

UML позволяет моделировать разные виды систем: чисто программные, чисто аппаратные, программно-аппаратные, смешанные, явно включающие деятельность людей и т.д. Но, помимо прочего, язык UML активно применяется для проектирования реляционных БД. Для этого используется небольшая

часть языка (диаграммы классов), да и то не в полном объеме. С точки зрения проектирования реляционных БД модельные возможности не слишком отличаются от возможностей ER-диаграмм. Но все же мы кратко опишем диаграммы классов UML, поскольку их использование при проектировании реляционных БД позволяет оставаться в общем контексте UML и применять другие виды диаграмм для проектирования приложений баз данных.

### 7.3.1. Основные понятия диаграмм классов UML

Диаграммой классов в терминологии UML называется диаграмма, на которой показан набор классов (и некоторых других сущностей, не имеющих явного отношения к проектированию БД), а также связей между этими классами. Кроме того, диаграмма классов может включать комментарии и ограничения. Здесь следует сделать два замечания. Во-первых, в этом разделе термин *сущность* используется настолько же неформально, как в предыдущем разделе использовался термин *объект*. UML претендует на обеспечение более точного и формального понятия *объекта* (UML обычно называют языком *объектно-ориентированного моделирования*). В спецификации языка UML даже присутствует определение понятия объекта средствами самого UML. Однако, несмотря на эти попытки, понятие *объекта* в UML остается таким же нечетким, как и понятие *сущности* в ER-модели. По-прежнему приходится опираться в основном на интуицию и здравый смысл. Во-вторых, в UML, как и в модели ER-диаграмм, для родового обозначения связей используется термин *relationship*. Во многих переводах книг про UML на русский язык вместо термина *связь* применяется термин *отношение*. Как и в предыдущем подразделе, используем термин *связь*.

Для диаграмм классов UML могут задаваться ограничения на естественном языке или же на языке объектных ограничений OCL (Object Constraints Language). Язык OCL является частью общей спецификации UML, но, в отличие от других частей языка, имеет не графическую, а линейную нотацию. Более подробно язык OCL обсуждается в подразд. 7.3.2.

**Классы, атрибуты, операции.** *Классом* называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически класс изображается в виде прямоугольника. У каждого класса должно быть имя (текстовая строка), уникально отличающее его от всех других классов. При формировании имен классов в UML до-

Рис. 7.14. Примеры описания классов

пускается использование произвольной комбинации букв, цифр и даже знаков препинания. Однако на практике рекомендуется использовать в качестве имен классов короткие и осмысленные прилагательные и существительные, каждое из которых начинается с заглавной буквы. Примеры описания классов показаны на рис. 7.14.

*Атрибутом класса* называется именованное свойство класса, описывающее множество значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов (в частности, не иметь ни одного атрибута). Свойство, выражаемое атрибутом, является свойством моделируемой сущности, общим для всех объектов данного класса. Так что атрибут является абстракцией состояния объекта.

Имена атрибутов представляются в разделе класса, расположенном под именем класса. Хотя UML не накладывает ограничений на способы создания имен атрибутов (имя атрибута может быть произвольной текстовой строкой), на практике рекомендуется использовать короткие прилагательные и существительные, отражающие смысл соответствующего свойства класса. Первое слово в имени атрибута рекомендуется писать с прописной буквы, а все остальные слова — с заглавной. Пример описания класса с указанными атрибутами показан на рис. 7.15.

*Операцией класса* называется именованная услуга, которую можно запросить у любого объекта этого класса. Операция — это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

Операции класса определяются в разделе, расположенном ниже раздела с атрибутами. При этом можно ограничиться только указанием имен операций, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать глагольные формы, соответствующие ожидаемому поведению объектов данного класса. Описание операции может также содержать ее *сигнатуру*, т. е. имена и типы всех параметров, а если операция является функ-

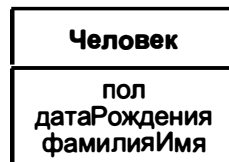


Рис. 7.15. Класс Человек с указанными именами атрибутов

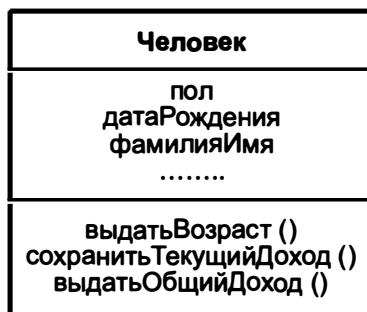


Рис. 7.16. Класс Человек с операциями

цией, то и тип ее значения. Класс **Человек** с определенными операциями показан на рис. 7.16.

Для класса **Человек** определены три операции: **выдатьВозраст**, **сохранитьТекущийДоход**, **выдатьОбщийДоход**. В операции **выдатьВозраст** используются значение атрибута **датаРождения** и значение текущей даты. Операция **сохранитьТекущийДоход** позволяет зафиксировать в состоянии объекта сумму и дату поступления дохода данного человека. Операция **выдатьОбщийДоход** выдает суммарный доход данного человека за указанное время. Заметим, что состояние объекта меняется при выполнении только второй операции. Результаты первой и третьей операций формируются на основе текущего состояния объекта.

**Категории связей. Связь-зависимость.** В диаграмме классов могут участвовать связи трех разных категорий: *зависимость* (dependency), *обобщение* (generalization) и *ассоциация* (association). При проектировании реляционных БД наиболее важны вторая и третья категории связей, поэтому о связях-зависимостях будет сказано только самое основное.

*Зависимостью* называют связь по применению, когда изменение в спецификации одного класса может повлиять на поведение другого класса, использующего первый класс. Чаще всего зависимости применяют в диаграммах классов, чтобы отразить в сигнатуре операции одного класса тот факт, что параметром этой операции могут быть объекты другого класса. Понятно, что если интерфейс второго класса изменяется, это влияет на по-



Рис. 7.17. Диаграмма классов со связью-зависимостью

ведение объектов первого класса. Пример диаграммы классов со связью-зависимостью показан на рис. 7.17. Здесь в классе **РасписаниеЗанятий** определены две операции с очевидной семантикой, параметрами которых являются объекты класса **Курс**. Понятно, что при изменении интерфейса класса **Курс** изменится поведение объектов класса **РасписаниеЗанятий**.

Зависимость показывается прерывистой линией со стрелкой, направленной к классу, от которого имеется зависимость. Очевидно, что связи-зависимости существенны для объектно-ориентированных систем (в том числе и для ООБД). При проектировании традиционных SQL-ориентированных баз данных непонятно, как можно использовать информацию о наличии связей-зависимостей между классами.

**Связи-обобщения и механизм наследования классов в UML.** *Связью-обобщением* называется связь между общей сущностью, называемой *суперклассом*, или *родителем*, и более специализированной разновидностью этой сущности, называемой *подклассом*, или *потомком*. Обобщения иногда называют связями «*is a*», имея в виду, что класс-потомок является частным случаем класса-предка. Класс-потомок наследует все атрибуты и операции класса-предка, но в нем могут быть определены дополнительные атрибуты и операции.

Объекты класса-потомка могут использоваться везде, где могут использоваться объекты класса-предка. Это свойство называют полиморфизмом по включению, имея в виду, что объекты потомка можно считать включаемыми во множество объектов класса-предка. Графически обобщения изображаются в виде сплошной линии с большой незакрашенной стрелкой, направленной к суперклассу. В качестве первой иллюстрации, приведенной на рис. 7.18, воспользуемся классификацией летательных аппаратов (см. рис. 7.12). На рис. 7.18 показан пример иерархии одиночного наследования: у каждого подкласса имеется только один суперкласс. Следует заметить, что в отличие от механизма наследования типов сущностей ER-модели здесь отсутствует класс **ПРОЧИЕ**, т. е. в классе **ЛетательныйАппарат** могут присутствовать «собственные» объекты, не относящиеся ни к классу **Аэроплан**, ни к классу **Вертолет**.

Одиночное наследование является достаточным в большинстве случаев применения связи-обобщения. Однако в UML допускается и множественное наследование, когда один подкласс определяется на основе нескольких суперклассов. В качестве одного из разумных (не слишком распространенных) примеров рассмотрим диаграмму классов на рис. 7.19 (для



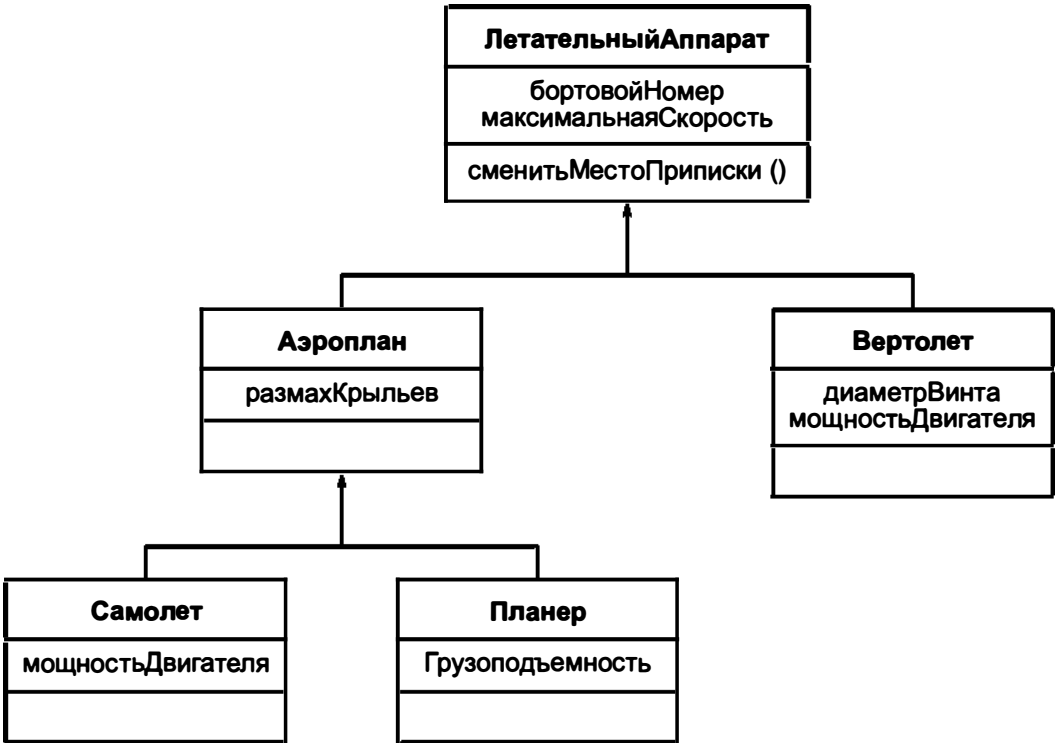


Рис. 7.18. Иерархия одиночного наследования классов

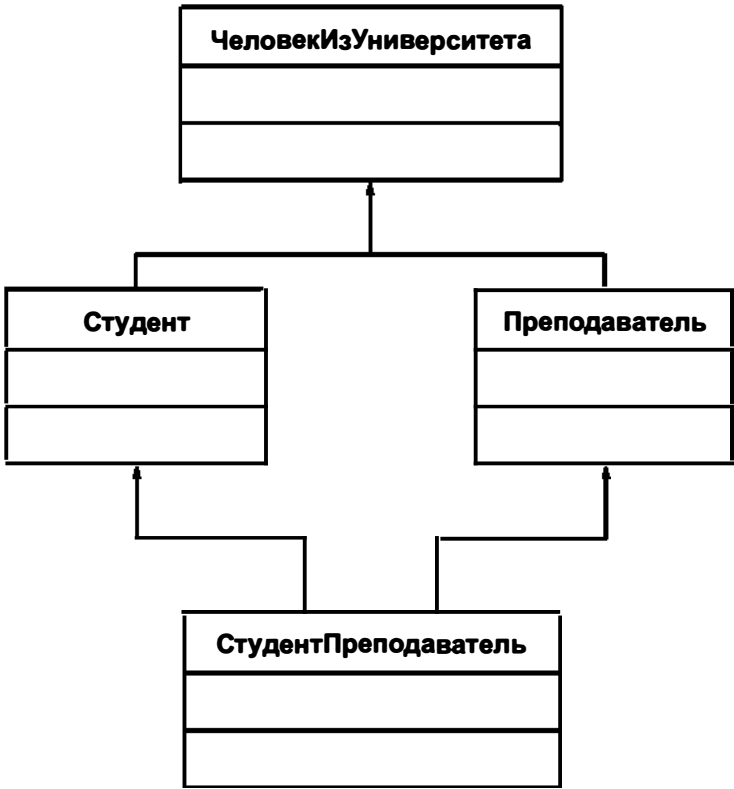


Рис. 7.19. Пример множественного наследования классов

упрощения диаграммы имена атрибутов и операций указывать не будем).

На этой диаграмме классы **Студент** и **Преподаватель** порождены из одного суперкласса **ЧеловекИзУниверситета**. Вообще говоря, к классу **Студент** относятся те объекты класса **ЧеловекИзУниверситета**, которые соответствуют студентам, а к классу **Преподаватель** — объекты класса **ЧеловекИзУниверситета**, соответствующие преподавателям. Но, как это часто случается, многие студенты уже в студенческие годы начинают преподавать, так что могут существовать такие два объекта классов **Студент** и **Преподаватель**, которым соответствует один объект класса **ЧеловекИзУниверситета**. Итак, среди объектов класса **Студент** могут быть преподаватели, а некоторые преподаватели могут быть студентами.

Тогда можно определить класс **СтудентПреподаватель** путем множественного наследования от суперклассов **Студент** и **Преподаватель**. Объект класса **СтудентПреподаватель** обладает всеми свойствами и операциями классов **Студент** и **Преподаватель** и может быть использован везде, где могут применяться объекты этих классов. Так что полиморфизм по включению продолжает работать. Заметим, что для этого пришлось отказаться от еще одного свойства механизма наследования ER-модели — отсутствие общих экземпляров у подтипов одного типа сущности. В данном случае множественное наследование возможно именно потому, что в классы **Студент** и **Преподаватель** входят разные объекты, которым соответствует один и тот же объект суперкласса.

Следует также отметить, что множественное наследование, помимо того, что не слишком часто требуется на практике, порождает ряд проблем, из которых одной из наиболее известных является проблема именования атрибутов и операций в подклассе, полученном путем множественного наследования. Например, предположим, что при образовании подклассов **Студент** и **Преподаватель** в них обоим был определен атрибут с именем **номерКомнаты**. Очень вероятно, что для объектов класса **Студент** значениями этого атрибута будут номера комнат в студенческом общежитии, а для объектов класса **Преподаватель** — номера служебных кабинетов. Как быть с объектами класса **Студент-Преподаватель**, для которых существенны оба одноименных атрибута (у студента-преподавателя могут иметься и комната в общежитии, и служебный кабинет)? На практике применяется одно из следующих решений:

- 1) запретить образование подкласса **СтудентПреподаватель**, пока в одном из суперклассов не будет произведено переименование атрибута **номерКомнаты**;

2) наследовать это свойство только от одного из суперклассов, так что, например, значением атрибута `номерКомнаты` у объектов класса `Студент` `Преподаватель` всегда будут номера служебных кабинетов;

3) унаследовать в подклассе оба свойства, но автоматически переименовать оба атрибута, чтобы прояснить их смысл; назвать их, например, `номерКомнатыСтудента` и `номерКомнатыПреподавателя`.

Ни одно из решений не является полностью удовлетворительным. Первое решение требует возврата к ранее определенному классу, имена атрибутов и операций которого, возможно, уже используются в приложениях. Второе решение нарушает логику наследования, не давая возможности на уровне подкласса использовать все свойства суперклассов. Наконец, третье решение заставляет использовать длинные имена атрибутов и операций, которые могут стать недопустимо длинными, если процесс множественного наследования будет продолжаться от полученного подкласса.

Но, конечно, сложность проблемы именования атрибутов и операций несопоставимо меньше сложности реализации множественного наследования в реляционных БД. Поэтому при использовании UML для проектирования реляционных БД нужно очень осторожно использовать наследование классов вообще и стараться избегать множественного наследования.

**Связи-ассоциации: роли, кратность, агрегация.** Ассоциацией называется структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса. Допускается, чтобы оба конца ассоциации относились к одному классу. В ассоциации могут связываться два класса, и тогда она называется *бинарной*. Допускается создание ассоциаций, связывающих сразу  $n$  классов (они называются  *$n$ -арными* ассоциациями).<sup>\*</sup> Графически ассоциация изображается в виде линии, соединяющей класс сам с собой или с другими классами.

С понятием ассоциации связаны четыре важных дополнительных понятия: *имя*, *роль*, *кратность* и *агрегация*. Во-первых, ассоциации может быть присвоено *имя*, характеризующее

---

<sup>\*</sup> Напомним, что в варианте ER-модели, рассмотренном в предыдущем подразделе, допускались только бинарные связи. В свое время компания Oracle обосновывала это решение тем, что наличие бинарных ассоциаций всегда является достаточным. Здесь мы также ограничимся обсуждением бинарных ассоциаций.

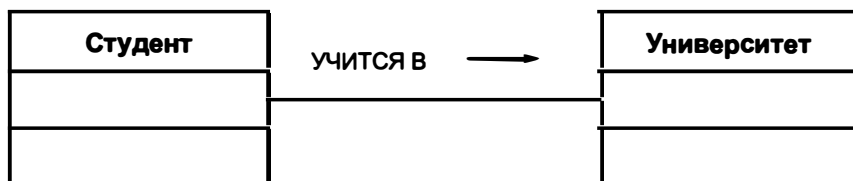


Рис. 7.20. Пример именованной ассоциации

природу связи. Смысл имени уточняется с помощью черного треугольника, который располагается над линией связи справа или слева от имени ассоциации. Этот треугольник указывает направление чтения имя связи. Пример именованной ассоциации показан на рис. 7.20. Треугольник показывает, что именованная ассоциация должна читаться так: «Студент учится в Университете».

Другим способом именованной ассоциации является указание *роли* каждого класса, участвующего в этой ассоциации. Роль класса, как и имя конца связи в ER-модели, задается именем, помещаемым под линией ассоциации ближе к данному классу. На рис. 7.21 показаны две ассоциации между классами **Человек** и **Университет**, в которых эти классы играют разные роли. Как мы видим, объекты класса **Человек** могут выступать в роли **РАБОТНИКОВ** при участии в ассоциации, в которой объекты класса **Университет** играют роль **НАНИМАТЕЛЯ**. В другой ассоциации объекты класса **Человек** играют роль **СТУДЕНТА**, а объекты класса **УНИВЕРСИТЕТ** — роль **ОБУЧАЮЩЕГО**.

В общем случае для ассоциации могут задаваться и ее собственное имя, и имена ролей классов. Это связано с тем, что класс может играть одну и ту же роль в разных ассоциациях, так что в общем случае пара имен ролей классов не идентифицирует ассоциацию. С другой стороны, в простых случаях, когда между двумя классами определяется только одна ассоциация, можно вообще не связывать с ней дополнительные имена.



Рис. 7.21. Две ассоциации с разными ролями классов

*Кратностью* (multiplicity) роли ассоциации называется характеристика, указывающая, сколько объектов класса с данной ролью может или должно участвовать в каждом экземпляре ассоциации. В терминологии UML экземпляр ассоциации называется *соединением* — link, но здесь не будем использовать этот термин, чтобы не создавать путаницу — все-таки трудно одновременно говорить про *связи*, *ассоциации* и *соединения*, имея в виду разные понятия.

Наиболее распространенным способом задания кратности роли ассоциации является указание конкретного числа или диапазона. Например, указание «1» говорит о том, что каждый объект класса с данной ролью должен участвовать в некотором экземпляре данной ассоциации, причем в каждом экземпляре ассоциации может участвовать ровно один объект класса с данной ролью. Указание диапазона «0..1» говорит о том, что не все объекты класса с данной ролью обязаны участвовать в каком-либо экземпляре данной ассоциации, но в каждом экземпляре ассоциации может участвовать только один объект. Аналогично, указание диапазона «1..\*» говорит о том, что все объекты класса с данной ролью должны участвовать в некотором экземпляре данной ассоциации и в каждом экземпляре ассоциации должен участвовать хотя бы один объект (верхняя граница не задана). Толкование диапазона «0..\*» является очевидным расширением случая «0..1».

В более сложных (но крайне редко встречающихся на практике) случаях определения кратности можно использовать списки диапазонов. Например, список «2, 4..6, 8..\*» говорит о том, что все объекты класса с указанной ролью должны участвовать в некотором экземпляре данной ассоциации, и в каждом экземпляре ассоциации должны участвовать два, от четырех до шести или более семи объектов класса с данной ролью.

На диаграмме классов с рис. 7.22 показано, что произвольное (может быть, нулевое) число людей являются сотрудниками произвольного числа университетов. Каждый университет обучает произвольное (может быть, нулевое) число студентов, но каждый студент может быть студентом только одного университета.

Обычная ассоциация между двумя классами характеризует связь между равноправными сущностями: оба класса находятся на одном концептуальном уровне. Но иногда в диаграмме классов требуется отразить тот факт, что ассоциация между двумя классами имеет специальный вид «часть-целое». В этом случае класс «целое» имеет более высокий концептуальный



Рис. 7.22. Ассоциации с указанными кратностями ролей

уровень, чем класс «часть». Ассоциация такого рода называется *агрегатной*. Графически агрегатные ассоциации изображаются в виде простой ассоциации с незакрашенным ромбом на стороне класса-«целого». Пример агрегатной ассоциации показан на рис. 7.23.

Объектами класса **Аудитория** являются студенческие аудитории, в которых проходят занятия. В каждой аудитории должны быть установлены парты. Поэтому в некотором смысле класс **Парта** является «частью» класса **Аудитория**. Мы умышленно сделали роль класса **Парта** в этой ассоциации необязательной, поскольку могут существовать аудитории без парт (например, класс для занятий танцами) и некоторые парты могут находиться на складе. Обратите внимание, что, хотя аудитории, не оснащенные партами, как правило, непригодны для занятий, объекты классов **Аудитория** и **Парта** существуют независимо. Если некоторая аудитория ликвидируется, то находящиеся в ней парты не уничтожаются, а переносятся на склад.

Бывают случаи, когда связь «части» и «целого» настолько сильна, что уничтожение «целого» приводит к уничтожению всех его «частей». Агрегатные ассоциации, обладающие таким свойством, называются *композиционными*, или просто *композициями*. При наличии композиции объект-часть может быть частью только одного объекта-целого (композиита). При обычной агрегатной ассоциации «часть» может одновременно принад-

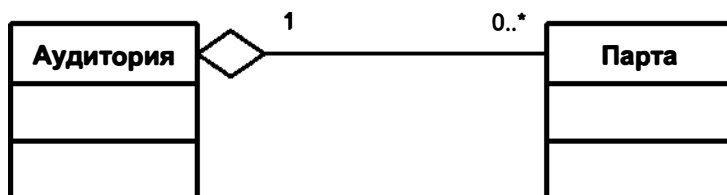


Рис. 7.23. Пример агрегатной ассоциации

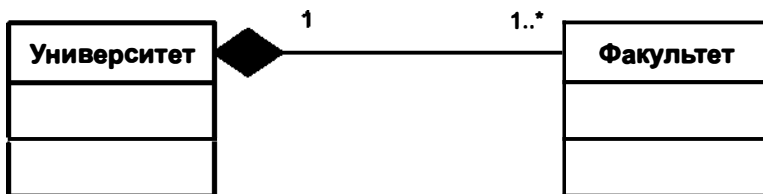


Рис. 7.24. Пример композитной агрегатной ассоциации

лежать несколько «целым». Графически композиция изображается в виде простой ассоциации, дополненной закрашенным ромбом со стороны «целого». Пример композитной агрегатной ассоциации показан на рис. 7.24. Любой факультет является частью одного университета, и ликвидация университета приводит к ликвидации всех существующих в нем факультетов (хотя во время существования университета отдельные факультеты могут ликвидироваться и создаваться).

Заметим, что в контексте проектирования реляционных БД агрегатные и в особенности композитные ассоциации влияют только на способ поддержки ссылочной целостности. В частности, композитная связь является явным указанием на то, что ссылочная целостность между «целым» и «частями» должна поддерживаться путем каскадного удаления частей при удалении целого. Подробнее способы поддержки ссылочной целостности в SQL-ориентированных БД рассматриваются в гл. 11.

При наличии простой ассоциации между двумя классами (например, ассоциации между классами *Студент* и *Университет* с рис. 7.20) предполагается возможность *навигации* между объектами, входящими в один экземпляр ассоциации. Если известен конкретный объект-студент, то должна обеспечиваться возможность узнать соответствующий объект-университет. Если известен конкретный объект-университет, то должна обеспечиваться возможность узнать все соответствующие объекты-студенты. Другими словами, если не оговорено иное, то навигация по ассоциации может проводиться в обоих направлениях\*. Однако бывают случаи, когда желательно ограничить направление навигации для некоторых ассоциаций. В этом случае на линии

---

\* Поскольку UML — это высокоуровневый язык моделирования, в нем не уточняется, что такое навигация в реализационном смысле. Но очевидно, что само появление понятия навигации связано с объектно-ориентированной природой UML. Термин *навигация* является почти ругательным в мире реляционных БД, но для мира объектно-ориентированных БД он вполне естественен, поскольку в этом мире на модельном уровне присутствует понятие ссылки, или указателя.

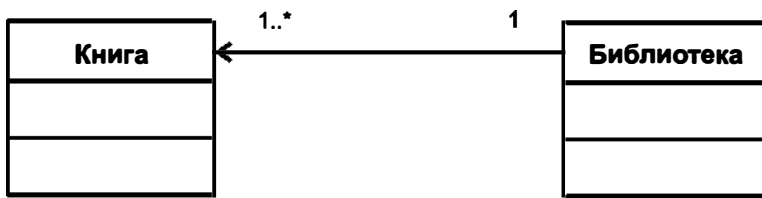


Рис. 7.25. Ассоциация с указанным направлением навигации

ассоциации ставится стрелка, указывающая направление навигации. Пример диаграммы классов с однонаправленной навигацией показан на рис. 7.25.

В библиотеке должно содержаться некоторое количество книг, и каждая книга должна принадлежать некоторой библиотеке. С точки зрения библиотечного хозяйства разумно иметь возможность найти книгу в библиотеке, т.е. произвести навигацию от объекта-библиотеки к связанным с ним объектам-книгам. Однако вряд ли потребуется по данному экземпляру книги узнать, в какой библиотеке она находится\*.

### 7.3.2. Ограничения целостности и язык OCL

Как уже отмечалось, в диаграммах классов могут указываться ограничения целостности, которые должны поддерживаться в проектируемой базе данных. В UML допускаются два способа определения ограничений: на естественном языке и на языке OCL. На рис. 7.26 показана простая диаграмма классов **Студент** и **Университет** с ограничением, выраженным на естественном языке.

В данном случае накладывается ограничение на состояние объектов классов **Студент** и **Университет**, входящих в один экземпляр ассоциации. Объект класса **Студент** может входить в экземпляр ассоциации с объектом класса **Университет** только при условии, что размер стипендии данного студента находится в диапазоне, допустимом в данном университете.

**Общая характеристика языка OCL.** Более точный и лаконичный способ формулировки ограничений обеспечивает язык OCL (Object Constraints Language). Вот, например, формулировка на языке OCL ограничения, показанного на рис. 7.26:

---

\* С точки зрения реляционных БД ассоциации с однонаправленной навигацией можно считать указанием на необходимость ограничения видимости объектов БД. Соответствующую (но существенно более общую) возможность в SQL-ориентированных БД обеспечивает механизм представлений (view). Подробнее об этом см. гл. 11.





Рис. 7.26. Ограничение, выраженное на естественном языке

(Значение атрибута Студент.стипендия должно находиться в диапазоне между значениями атрибутов Университет.минСтипендия и Университет.максСтипендия)

```

context Студент inv:
self.стипендия ≥ self.обучающий.минСтипендия and
self.стипендия ≤ self.обучающий.максСтипендия
  
```

Хотя язык OCL формально считается частью UML, он специфицирован в отдельном документе, в котором присутствуют ссылки на другие части спецификации UML, а также вводятся собственные понятия и определения. Из языка UML в OCL заимствованы следующие понятия:

- класс, атрибут, операция;
- объект (экземпляр класса);
- связь-ассоциация;
- тип данных (включая набор предопределенных типов Boolean, Integer, Real и String);
- значение (экземпляр типа данных).

Для понимания языка OCL существенны определяемые в UML традиционные для объектных моделей данные различия между объектом некоторого класса и значением некоторого типа:

- Объект обладает уникальным идентификатором и может сравниваться с другими объектами только по значению идентификатора. Следствием этого является возможность определения операций над множествами объектов в терминах их идентификаторов. Следует заметить, что ни в спецификации UML, ни в описании какой-либо другой объектной модели никогда прямо не говорится, что в операциях над множествами объектов в действительности участвуют идентификаторы объектов. Но другого понимания не существует.
- Объект может быть ассоциирован через бинарную связь с другими объектами, что позволяет определить в OCL операцию перехода от данного объекта к связанным с ним объектам. Заметим, что хотя в UML допускаются  $n$ -арные

связи, в OCL речь идет только об уже привычном для нас бинарном варианте.

- В то же время значение является «чистым значением» в том смысле, что:
  - при сравнении двух значений проверяются сами эти значения;
  - значения не могут участвовать в связях, поскольку понятие связи определено только для объектов классов.

В дополнение к *скалярным* типам данных, заимствованным из UML, в OCL предопределены *структурные* типы, которые являются разновидностями типов коллекций (*collection*):

- математическое множество (*set*), неупорядоченная коллекция, не содержащая одинаковых элементов;
- *мультимножество* (*bag*), неупорядоченная коллекция, которая может содержать повторяющиеся элементы-дубликаты;
- *последовательность* (*sequence*), упорядоченная коллекция, которая может содержать элементы-дубликаты.

Элементами каждого из трех типов коллекций могут быть объекты или значения одного класса или одного типа соответственно.

Язык OCL предназначен, главным образом, для определения ограничений целостности данных, соответствующих модели, которая представлена в терминах диаграммы классов UML. Язык OCL может применяться для определения ограничений, описывающих пред- и постусловия операций классов, и ограничений, представляющих собой инварианты классов. С точки зрения определения ограничений целостности баз данных более важны средства определения инвариантов классов.

**Инвариант класса.** Под инвариантом класса в OCL понимается условие, которому должны удовлетворять все объекты данного класса. Если говорить более точно, инвариант класса — это логическое выражение, при вычислении которого для любого объекта данного класса в течение всего времени существования этого объекта получается булевское значение *true*. При определении инварианта требуется указать имя класса и выражение, определяющее инвариант указанного класса. Синтаксически это выглядит следующим образом:

```
context <class_name> inv:  
<OCL-выражение>
```

Здесь *<class-name>* является именем класса, для которого определяется инвариант, *inv* — ключевое слово, говорящее о том,

что определяется именно инвариант, а не ограничение другого вида, и **context** — ключевое слово, которое указывает на то, что контекстом следующего после двоеточия OCL-выражения являются объекты класса **<class-name>**, т.е. OCL-выражение должно принимать значение *true* для всех объектов этого класса.

Заметим, что OCL является типизированным языком, поэтому у каждого выражения имеется некоторый тип (тип значения, получаемого при вычислении выражения). Как уже отмечалось, OCL-выражение в инварианте класса должно быть логического типа.

В общем случае OCL-выражение в определении инварианта основывается на композиции операций, которым посвящена большая часть определения языка. В спецификации языка эти операции условно разделены на следующие группы:

- операции над значениями предопределенных в UML (скалярных) типов данных;
- операции над объектами;
- операции над множествами;
- операции над мультимножествами;
- операции над последовательностями.

Последовательно обсудим эти группы операций.

*Операции над значениями предопределенных типов данных.* Как уже отмечалось в начале этого подраздела, в OCL поддерживаются скалярные типы данных Boolean, Integer, Real и String. Общее представление об операциях над значениями этих типов данных дает табл. 7.1.

В основном смысл операций должен быть понятен из их обозначений. Для менее очевидных случаев приведем краткие пояснения. Операция **xor** — это стандартное «исключающее или». Она принимает два параметра булевского типа и вырабатывает значение *true*, если значением одного и только одного параметра является *true*. В противном случае операция вырабатывает зна-

Таблица 7.1. Операции предопределенных типов

| Предопределенный скалярный тип | Список операций                          |
|--------------------------------|------------------------------------------|
| Boolean                        | and, or, xor, not, implies, if-then-else |
| Integer                        | *, +, -, /, abs(), операции сравнения    |
| Real                           | *, +, -, /, floor(), операции сравнения  |
| String                         | concat(), size(), substring()            |

чение *false*. Операция *implies* — это импликация. Она принимает два параметра булевского типа и вырабатывает значение *true*, если значением первого параметра является *false*, или если значениями обоих параметров является *true*. В противном случае операция вырабатывает значение *false*. Операция *floor* вырабатывает наибольшее значение целого типа, меньшее или равное значению параметра операции. Операция *concat* конкатенирует две строки-аргументы, *size* выдает целое значение, равное длине строки-аргументу, *substring* выдает подстроку строки-аргумента с заданными начальной позицией и длиной.

*Операции над объектами.* В OCL определены три операции над объектами:

- получение значения атрибута;
- переход по экземпляру ассоциации,
- вызов операции класса (последняя операция для целей проектирования «традиционных» реляционных БД несущественна).

Для обозначения всех трех этих операций используется «точечная нотация».

Результатом выражения вида

`<объект>.<имя атрибута>`

является текущее значение атрибута с именем *имя атрибута*, если объект имеет такой атрибут. Если атрибут типизирован именем некоторого класса, то результатом вызова операции является некоторый объект этого класса (объектный идентификатор), к которому также применимы операции над объектами. В противном случае использование подобного выражения приводит к возникновению ошибки типа.

Результатом применения к объекту операции перехода по экземпляру связи-ассоциации является коллекция, содержащая все объекты, которые ассоциированы с данным объектом через указываемый экземпляр ассоциации. Этот экземпляр ассоциации идентифицируется именем роли, противоположной по отношению к данному объекту. Таким образом, синтаксис выражения перехода по соединению следующий:

`<объект>.<имя роли, противоположной по отношению к объекту>`

*Операции над множествами, мультимножествами и последовательностями.* В OCL поддерживается обширный набор операций над значениями коллекционных типов данных. Обсудим только те из них, которые являются уместными в контексте

данного раздела. Синтаксически вызовы операций над коллекциями записываются в нотации, аналогичной точечной, но вместо точки используется стрелка ( $\rightarrow$ ). Общий синтаксис применения операции к коллекции выглядит следующим образом:

$\langle \text{коллекция} \rangle \rightarrow \langle \text{имя операции} \rangle (\langle \text{список фактических параметров} \rangle)$

**Операция select.** В OCL определены три одноименных операции **select**, которые обрабатывают заданное множество, мультимножество или последовательность на основе заданного логического выражения над элементами коллекции. Результатом каждой операции является новое множество, мультимножество или последовательность, соответственно, из тех элементов входной коллекции, для которых результатом вычисления логического выражения является *true*.

**Операция collect.** В OCL определены три операции **collect**, параметрами которых являются множество, мультимножество или последовательность и некоторое выражение над элементами соответствующей коллекции. Результатом является мультимножество для операций **collect**, определенных над множествами и мультимножествами, и последовательность для операции **collect**, определенной над последовательностью. При этом результирующая коллекция соответствующего типа (коллекция значений или объектов) состоит из результатов применения выражения к каждому элементу входной коллекции. Операция **collect** используется, главным образом, в тех случаях, когда от заданной коллекции объектов требуется перейти к некоторой другой коллекции объектов, которые ассоциированы с объектами исходной коллекции через некоторый экземпляр ассоциации. В этом случае выражение над элементом исходной коллекции основывается на операции перехода по экземпляру ассоциации.

**Операции exists, forAll, count.** В OCL определены три одноименных операции **exists** над множеством, мультимножеством и последовательностью; дополнительным параметром этих операций является логическое выражение. В результате каждой из этих операций выдается *true* в том и только том случае, когда хотя бы для одного элемента входной коллекции значением логического выражения является *true*. В противном случае результатом операции является *false*. Операции **forAll** отличаются от операций **exist** тем, что в результате каждой из них выдается *true* в том и только том случае, когда для всех элементов входной коллекции результатом вычисления логического выражения является *true*. В противном случае результатом операции будет

*false*. Операция `count` применяется к коллекции и выдает число содержащихся в ней элементов.

**Операции `union`, `intersect`, `symmetricDifference`.** Параметрами двуместных операций `union` (объединение), `intersect` (пересечение), `symmetricDifference` (симметричное вычитание) являются две коллекции, причем в OCL операции определены почти для всех возможных комбинаций типов коллекции. Не будем рассматривать все определения этих операций и кратко упомянем только две из них. Результатом операции `union`, определенной над множеством и мультимножеством, является мультимножество, т.е. из результата объединения таких двух коллекций дубликаты не исключаются. Результатом же операции `union`, определенной над двумя множествами, является множество, т.е. в этом случае возможные дубликаты должны быть исключены.

**Примеры инвариантов.** В заключение обзора языка OCL приведем примеры четырех инвариантов, выраженных на этом языке. Будем основываться на диаграмме классов, показанной на рис. 7.27.

**Пример 7.1.** Определить ограничение «возраст служащих должен быть больше 18 и меньше 100 лет».

```
context Служащий inv:  
self.возраст >18 and self.возраст < 100
```

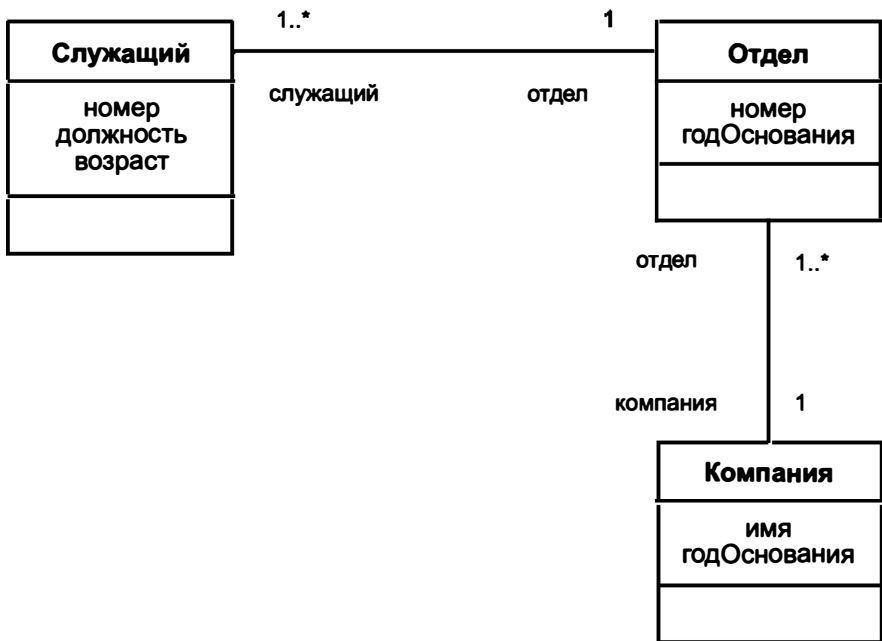


Рис. 7.27. Диаграмма классов, используемая для примеров ограничений на языке OCL

Условие инварианта накладывает требуемое ограничение на значения атрибута **возраст**, определенного в классе **Служащий**. В условном выражении инварианта ключевое слово **self** обозначает текущий объект класса-контекста инварианта. Можно считать, что при проверке данного условия будут перебираться существующие объекты класса **Служащий**, и для каждого объекта будет проверяться, что значения атрибута **возраст** находятся в пределах заданного диапазона. Ограничение удовлетворяется, если условное выражение принимает значение *true* для каждого объекта класса-контекста.

**Пример 7.2.** Выразить на языке OCL ограничение, в соответствии с которым в отделах с номерами больше 5 должны работать сотрудники старше 30 лет.

```
context Отдел inv:  
self.номер ≤ 5 or  
self.служащий → select (возраст ≤ 30) → count ()  
= 0
```

В этом случае условное выражение инварианта будет вычисляться для каждого объекта класса **Отдел**. Подвыражение справа от операции **or** вычисляется слева направо. Сначала вычисляется подвыражение **self.служащий**, значением которого является множество объектов, соответствующих служащим, которые работают в текущем отделе. Далее к этому множеству применяется операция **select (возраст ≤ 30)**, в результате которой вырабатывается множество объектов, соответствующих служащим текущего отдела, возраст которых не превышает 30 лет. Значением операции **count ()** является число объектов в этом множестве. Все выражение принимает значение *true*, если последняя операция сравнения «=0» вырабатывает значение *true*, т.е. если в текущем отделе нет сотрудников младше 31 года. Ограничение в целом удовлетворяется только в том случае, если значением условия инварианта является *true* для каждого отдела.

Тот же инвариант можно сформулировать в контексте класса **Служащий**:

```
context Служащий inv:  
self.возраст > 30 or self.отдел.номер ≤ 5
```

Здесь следует обратить внимание на подвыражение **self.отдел.номер ≤ 5**. Поскольку **отдел** — это имя роли ассоциации, значением подвыражения **self.отдел** является коллекция (множество).

Но кратность роли **отдел** равна единице, т.е. каждому объекту служащего соответствует в точности один объект отдела. Поэтому в OCL допускается сокращенная запись операции **self.отдел.номер**, значением которой является номер отдела текущего служащего.

**Пример 7.3.** Определить ограничение, в соответствии с которым у каждого отдела должен иметься менеджер, и любой отдел должен быть основан не раньше соответствующей компании.

```
context Отдел inv:  
self.служащий → exists (должность = "manager")  
and self.компания.годОснования ≥ self.годОснования
```

Здесь **должность** — атрибут класса **Служащий**, а атрибуты с именем **годОснования** имеются и у класса **Отдел**, и у класса **Компания**. В условном выражении этого инварианта подвыражение **self.служащий → exists (должность = "manager")** эквивалентно выражению **self.служащий → select (должность = "manager") → count () > 1**. Если бы в ограничении мы потребовали, чтобы у каждого отдела был только один менеджер, то следовало бы написать ... **count () = 1**, и это было бы не эквивалентно варианту с **exists**.

Обратите внимание, что в этом случае снова законным является подвыражение **self.компания.годОснования**, поскольку кратность роли **компания** в ассоциации классов **Отдел** и **Компания** равна единице.

**Пример 7.4.** Условие четвертого инварианта ограничивает максимально возможное количество сотрудников компании числом 1000.

```
context Компания inv:  
self.отдел → collect (служащие) → count () < 1000
```

Здесь полезно обратить внимание на использование операции **collect**. Проследим за вычислением условного выражения. В нашем случае в классе **Компания** всего один объект, и он сразу становится текущим. В результате выполнения операции **self.отдел** будет получено множество объектов, соответствующих всем отделам компании. При выполнении операции **collect (служащие)** для каждого объекта-отдела по экземпляру ассоциации с объектами класса **Служащие** будет образовано множество объектов-служащих данного отдела, а в результате будет образовано множество объектов, соответствующих всем служащим всех отделов компании, т.е. всем служащим компании.



**Достоинства и недостатки использования языка OCL при проектировании реляционных баз данных.** Язык OCL позволяет формально и однозначно (без двусмысленностей, свойственных естественным языкам) определять ограничения целостности БД в терминах ее концептуальной схемы. Скорее всего, наличие подобной проектной документации будет полезным для сопровождения БД, даже если придется преобразовывать инварианты OCL в ограничения целостности SQL вручную.

К отрицательным сторонам использования OCL относятся, прежде всего, сложность языка и неочевидность некоторых его конструкций. Кроме того, строгость синтаксиса и линейная форма языка в некотором роде противоречат наглядности и интуитивной ясности диаграммной части UML. Да, в инвариантах OCL используются те же понятия и имена, что и в соответствующей диаграмме классов, но используются совсем в другой манере. И последнее. Трудно доказать или опровергнуть как предположение, что на языке OCL можно выразить любое ограничение целостности, которое можно определить средствами SQL, так и утверждение, что на языке OCL нельзя выразить такой инвариант, для которого окажется невозможным сформулировать эквивалентное ограничение целостности на языке SQL. Мне неизвестны работы, в которых бы сравнивалась выразительная мощность этих языков в связи с ограничениями целостности реляционных БД.

### **7.3.3. Получение схемы реляционной базы данных из диаграммы классов UML**

Если не обращать внимания на различия в терминологии, то здесь выполняются практически те же шаги, что и в случае преобразования в схему реляционной БД ER-диаграммы. Поэтому ограничимся только некоторыми рекомендациями, специфичными для диаграмм классов.

*Рекомендация 1.* Прежде чем определять в классах операции, подумайте, что вы будете делать с этими определениями в среде целевой РСУБД. Если в этой среде поддерживаются хранимые процедуры, то, возможно, некоторые операции могут быть реализованы именно с помощью такого механизма. Но если в среде РСУБД поддерживается механизм определяемых пользователями функций, возможно, он окажется более подходящим.

*Рекомендация 2.* Помните, что сравнительно эффективно в РСУБД реализуются только ассоциации видов «один ко многим» и «многие ко многим». Если в созданной диаграмме классов

имеются ассоциации «один к одному», следует задуматься о целесообразности такого проектного решения. Реализация в среде РСУБД ассоциаций с точно заданными кратностями ролей возможна, но требует определения дополнительных триггеров, выполнение которых понизит эффективность.

*Рекомендация 3.* В спецификации UML говорится о том, что, определяя однонаправленные связи, вы можете способствовать эффективности доступа к некоторым объектам. Для технологии реляционных баз данных поддержка такого объявления вызовет дополнительные накладные расходы и тем самым снизит эффективность.

*Рекомендация 4.* Не злоупотребляйте возможностями OCL. Диаграммы классов UML — это мощный инструмент для создания концептуальных схем баз данных, но, как известно, все хорошо в меру.

\* \* \*

Нельзя сказать, что проектирование баз данных на основе семантических моделей в любом случае ускоряет и/или упрощает процесс проектирования. Все зависит от сложности предметной области, квалификации проектировщика и качества вспомогательных программных средств. Но так или иначе этап диаграммного моделирования обеспечивает следующие преимущества:

- на раннем этапе проектирования до привязки к конкретной РСУБД проектировщик может обнаружить и исправить логические недочеты проекта, руководствуясь наглядным графическим представлением концептуальной схемы;
- окончательный вид концептуальной схемы, полученной непосредственно перед переходом к формированию реляционной схемы, а может быть, и промежуточной версии концептуальной схемы, должен стать частью документации целевой реляционной БД. Наличие этой документации очень полезно для сопровождения и, в особенности, для изменения схемы БД в связи с изменившимися требованиями;
- при использовании CASE-средств концептуальное моделирование БД может стать частью всего процесса проектирования целевой информационной системы, что должно способствовать правильной структуризации процесса, эффективности и повышению качества проекта в целом.

В этой главе мы также стремились показать, что в контексте проектирования реляционных БД структурные методы проектирования, основанные на использовании ER-диаграмм, и объектно-ориентированные методы, основанные на использовании языка UML, различаются, главным образом, лишь терминологией. ER-модель концептуально проще UML, в ней меньше понятий, терминов, вариантов применения. И это понятно, поскольку разные варианты ER-моделей разрабатывались именно для поддержки проектирования реляционных БД, и ER-модели почти не содержат возможностей, выходящих за пределы реальных потребностей проектировщика реляционной БД.

Язык UML принадлежит объектному миру. Этот мир гораздо сложнее (если угодно, непонятнее, запутаннее) реляционного мира. Поскольку UML может использоваться для унифицированного объектно-ориентированного моделирования всего чего угодно, в этом языке содержится масса различных понятий, терминов и вариантов использования, избыточных с точки зрения проектирования реляционных БД. Если вычленишь из общего механизма диаграмм классов то, что действительно требуется для проектирования реляционных БД, то мы получим в точности ER-диаграммы с другой нотацией и терминологией.

Поэтому выбор конкретной концептуальной модели данных — это вопрос вкуса и сложившихся обстоятельств. Понятно, что если в организации уже имеется сложившаяся инфраструктура проектирования приложений, то разумно продолжать ею пользоваться до тех пор, пока это не станет тормозом. При построении же новой инфраструктуры стратегические соображения высшего руководства компании имеют больший вес, чем предпочтения технических специалистов, хотя эти предпочтения тоже обязательно должны учитываться.

## Глава 8

### **ПРИМЕР ОБЩЕЙ ОРГАНИЗАЦИИ СУБД. ФИЗИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ ВО ВНЕШНЕЙ ПАМЯТИ. ИНДЕКСНЫЕ СТРУКТУРЫ**

В 1975—1979 гг. в исследовательской лаборатории компании IBM разрабатывалась система управления реляционными базами данных System R. Эта работа оказала революционизирующее влияние на развитие теории и практики реляционных систем во всем мире. Именно System R практически доказала жизнеспособность реляционного подхода к управлению базами данных.

После успешного завершения работ по созданию этой системы и получения экспериментальных результатов ее использования был разработан целый ряд коммерчески доступных реляционных систем, в том числе и на основе непосредственного развития System R. Исключительно важен опыт, приобретенный при разработке этой системы. Практически во всех более поздних реляционных СУБД в той или иной степени используются методы, примененные в System R. Поэтому главы этой книги, посвященные внутренней организации SQL-ориентированных СУБД, во многом опираются на материалы статей, посвященных System R.

Организации СУБД System R посвящена обширная библиография [16—32] (здесь подобраны публикации, свободно доступные в Internet к моменту написания этого текста). Поскольку публикации появлялись по ходу практической реализации системы, каждая из них отражает состояние дел (идейное и практическое) именно на том этапе работы, когда была написана соответствующая статья. Некоторые идеи и представления, естественно, изменялись по ходу работы. Сравнительно законченное представление о системе в целом дают только заключительные публикации. Имеется обширный обзор литературы, посвященной System R и связанных с ней проектов [33].

## 8.1. Основные понятия, цели и общая организация System R

Поскольку обсуждение принципов внутренней организации реляционных (точнее, SQL-ориентированных) СУБД в этой книге проводится в контексте System R, начнем с рассмотрения основных понятий этой системы.

### 8.1.1. Используемая терминология

Несмотря на то, что при реализации System R использовался подход, несколько отличающийся от реляционного подхода Кодда (отсюда и пошли расхождения между реляционной моделью данных и моделью данных SQL), мы будем активно пользоваться терминами реляционной модели. К таким терминам относятся названия реляционных операций — ограничение, проекция, соединение; названия теоретико-множественных операций — объединение, пересечение, взятие разности и т. д.

В тех случаях, когда терминология System R расходится с реляционной терминологией, предпочтение будет отдаваться терминологии System R. В частности, это касается использования термина «поле таблицы» вместо термина «атрибут отношения». В самой System R при переходе к коммерческим системам также произошла некоторая смена терминологии. В частности, появилась тенденция к употреблению терминов, более привычных в среде пользователей IBM: файл, запись и т. д. Здесь будут использоваться термины System R, более близкие реляционным системам. Опишем некоторые основные термины System R, опираясь в основном не на теоретические соображения, а на практические аспекты соответствующих понятий.

Базовым понятием System R является понятие *таблицы* (приближенный к реализации аналог основного понятия реляционного подхода *отношения*; иногда, в зависимости от контекста, мы будем использовать и этот термин). Таблица — это регулярная структура данных, состоящая из конечного набора однотипных записей — кортежей. Каждый кортеж одной таблицы состоит из конечного (и одинакового) числа полей кортежа, причем *i*-е поле каждого кортежа одной таблицы может содержать данные только одного типа, и набор допустимых типов данных в System R предопределен и фиксирован.

В силу регулярности структуры таблицы понятие поля кортежа расширяется до понятия поля таблицы. Тогда *i*-е поле таблицы можно трактовать как набор одноместных кортежей, по-

лученных выборкой  $i$ -х полей из каждого кортежа этой таблицы, т. е. в общепринятой терминологии как проекцию таблицы на  $i$ -й атрибут. В терминологию System R не входит понятие домена, оно заменяется здесь понятием типа поля, т. е. типом данных, хранение которых в данном поле допускается (это не вполне эквивалентная замена, но такова реальность System R).

Таблицы, составляющие базу данных System R, могут физически храниться в одном или нескольких сегментах, каждому из которых соответствует отдельный файл внешней памяти. Сегменты разбиваются на страницы, в которых располагаются кортежи таблиц и вспомогательные служебные структуры данных — индексы. Соответственно, каждый сегмент содержит две группы страниц — страницы данных и страницы индексной информации. Страницы каждой группы имеют фиксированный размер, но страницы с индексной информацией меньше по размеру, чем страницы данных. В страницах данных могут располагаться кортежи более чем одной таблицы (это очень важное свойство физической организации баз данных System R; следующие из этой организации преимущества разъясним позже).

Этим, конечно, не исчерпывается набор понятий System R, но остальные термины будем пояснять по ходу изложения, поскольку для этого требуется соответствующий понятийный контекст.

### **8.1.2. Цели System R и их связь с общей организацией системы**

При выполнении проекта System R преследовались следующие основные цели:

- обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
- обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
- поддерживать динамически изменяемую среду баз данных, в которой таблицы, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
- обеспечить возможность параллельной работы с одной базой данных многих пользователей, допуская параллельную

модификацию объектов базы данных при наличии необходимых средств защиты целостности базы данных;

- обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
- обеспечить гибкий механизм, позволяющий определять различные представления хранимых данных и ограничивать этими представлениями доступ пользователей к базе данных по выборке и модификации на основе механизма авторизации;
- обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Прежде всего отметим, что при разработке System R поставленные цели в основном были достигнуты. Рассмотрим теперь, какими средствами удалось достичь этих целей и как можно более точно интерпретировать их в контексте System R.

Основой System R является «реляционный» язык SEQUEL (который достаточно быстро был переименован в SQL). Заметим, что разработчики System R искренне считали созданный ими язык реляционным; однако, как отмечалось в предыдущих главах этой книги и будет более детально показано в ее заключительных главах, в этом языке в действительности нарушаются многие важные принципы реляционной модели данных. Иногда его называют языком запросов или языком манипулирования данными, но на самом деле возможности SQL гораздо шире. Средствами SQL (с соответствующей системной поддержкой) решаются многие из поставленных целей. Язык SQL включает средства динамической компиляции запросов, на основе чего возможно построение диалоговых систем обработки запросов. Допускается динамическая параметризация статически откомпилированных запросов, в результате чего возможно построение эффективных (не требующих динамической компиляции) диалоговых систем со стандартными наборами (параметризуемых) запросов. Средствами SQL определяются все доступные пользователю объекты баз данных: таблицы, индексы, представления. Имеются средства уничтожения любого такого объекта. Соответствующие операторы языка могут выполняться в любой момент, и возможность выполнения операции данным пользователем зависит от ранее предоставленных ему прав.

Что касается целостности баз данных, то в System R под целостным состоянием базы данных понимается состояние, удовлетворяющее набору сохраняемых при базе данных предикатов

целостности. Эти предикаты, называемые в System R *утверждениями целостности* (assertion), также задаются средствами языка SQL. Любой оператор языка выполняется в границах некоторой *транзакции* — последовательности операторов языка, неделимой в смысле состояния базы данных. Неделимость означает, что все изменения базы данных, произведенные в пределах одной транзакции, либо целиком отображаются в состоянии базы данных, либо полностью в нем отсутствуют. Последняя возможность возникает при *откате* транзакции, который может произойти по инициативе пользователя (при выполнении соответствующего оператора SQL) или по инициативе системы.

Одной из причин отката транзакции по инициативе системы является как раз нарушение целостности базы данных в результате действий данной транзакции (другие возможные условия отката транзакции по инициативе системы мы рассмотрим далее). Язык SQL System R (так будем называть вариант языка SQL, разработанный в проекте System R, чтобы отличать его от более поздних, «стандартных» вариантов этого языка) содержит средство установки так называемых *точек сохранения* (savepoint). При иницииируемом пользователем откате транзакции можно указать номер точки сохранения, выше которого откат не распространяется. Иницииируемый системой откат транзакции производится до ближайшей точки сохранения, в которой условие, вызвавшее откат, уже отсутствует. В частности, откат транзакции, инициированный по причине нарушения условия целостности, производится до ближайшей точки сохранения, в которой условия целостности соблюдены.

Естественно, что для реального выполнения отката транзакции необходимо запоминать некоторую информацию о выполнении транзакции. В System R для этих и других целей используется специальный набор данных — *журнал*, в который помещаются записи обо всех операциях всех транзакций, изменяющих состояние базы данных. При откате транзакции происходит процесс *обратного выполнения* транзакции (undo), в ходе которого в обратном порядке выполняются все изменения, запомненные в журнале.

В языке SQL System R имеется средство определения так называемых *триггеров* (trigger), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. В SQL System R триггер — это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения. Особенно существенно наличие такого механизма в связи с наличием обсуждаемых далее представ-



лений базы данных, которыми может быть ограничен доступ к базе данных для ряда пользователей. Возможна ситуация, когда такие пользователи просто не могут соблюдать целостность базы данных без автоматического выполнения условных воздействий, поскольку они просто «не видят» всей базы данных и, в частности, не могут представить всех ограничений ее целостности.

Язык SQL содержит средства определения представлений. Представление — это каталогизированный именованный запрос на выборку данных (из одной или нескольких таблиц). Поскольку SQL — это «реляционный» язык, результатом выполнения любого запроса на выборку является таблица, и поэтому концептуально можно относиться к любому представлению как к таблице (при определении представления можно, в частности, присвоить имена полям этой таблицы). В языке допускается использование ранее определенных представлений практически везде, где допускается использование таблиц (с некоторыми ограничениями по поводу возможностей модификации через представления). Наличие возможности определять представления в совокупности с развитой системой авторизации позволяет ограничить доступ некоторых пользователей к базе данных выделенным набором представлений.

Авторизация доступа к базе данных также основана на средствах SQL. При создании любого объекта базы данных пользователь, выполняющий эту операцию, становится полновластным владельцем этого объекта, т. е. может выполнять по отношению к этому объекту любую допустимую операцию SQL. Далее этот пользователь может выполнить оператор SQL, означающий передачу всех его прав на этот объект (или их подмножества) любому другому пользователю. В частности, этому пользователю может быть передано право на передачу всех переданных ему прав (или их части) третьему пользователю и т. д. Одним из прав пользователя по отношению к объекту является право на изъятие у других пользователей всех или некоторых прав, которые ранее им были переданы. Эта операция распространяется транзитивно на всех дальнейших наследников этих прав.

Наличие в языке средств определения представлений и авторизации в принципе позволяет обойтись при эксплуатации System R без традиционного администратора баз данных, поскольку практически все системные действия производятся на основе средств SQL. Тем не менее, если организационно администратор баз данных требуется, то его работа достаточно упрощается за счет унифицированного набора средств управле-

ния. Кроме того, в System R каталоги баз данных поддерживаются также в виде таблиц, и к ним применены все запросы языка SQL. Заметим, что в более поздних SQL-ориентированных СУБД появился ряд дополнительных утилит, не связанных с языком SQL (например, утилиты сбора статистики или массовой загрузки базы данных), и в этих системах, видимо, без администратора базы данных не обойтись.

По части обеспечения параллельной работы многих пользователей с одной базой данных, основной подход System R состоит в том, что пользователь не обязан знать о наличии других пользователей, конкурирующих с ним за доступ к базе данных, т. е. система ответственна за обеспечение изолированности пользователей с гарантией отсутствия их взаимного влияния в пределах транзакций. Из этого следует, во-первых, что в интерфейсе пользователя с системой (т. е. в языке SQL) не должно быть средств регулирования взаимодействий с другими пользователями и, во-вторых, что система должна обеспечить автоматическую сериализацию набора транзакций, т. е. обеспечить режим выполнения этого набора транзакций, эквивалентный по конечному результату некоторому последовательному выполнению этих транзакций. Эта проблема решается в System R за счет автоматического выполнения синхронизационных блокировок всех изменяемых объектов базы данных.

Одним из основных требований к СУБД вообще и к System R в частности является обеспечение надежности баз данных по отношению к различного рода сбоям. К таким сбоям могут относиться программные ошибки прикладного и системного уровня, сбой процессора, поломки внешних носителей и т. д. В частности, к одному из видов сбоев можно отнести упоминавшиеся ранее нарушения целостности базы данных и автоматический иницилируемый системой откат транзакции — это системное средство восстановления базы данных после сбоев такого рода. Как уже отмечалось, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. На информации журнала также основано восстановление базы данных и после сбоев другого рода.

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со спецификой физической организации БД во внешней памяти, использованием техники индексированного доступа к данным, буферизацией используемых страниц базы данных в основной памяти и развитой техникой оптимизации SQL-запросов, производимой на стадии их компиляции.

Структурная организация System R согласуется с поставленными при ее разработке целями и выбранными решениями. Основными структурными компонентами System R являются система управления реляционными данными (Relational Data System — RDS), состоящая, по существу, из компилятора языка SQL и подсистемы поддержки откомпилированных операторов, и система управления реляционной памятью (Relational Storage System — RSS).

RSS обеспечивает интерфейс довольно низкого, но достаточно-го для реализации SQL уровня доступа к хранимым в базе данным (этот внутренний интерфейс System R напоминает внешний интерфейс систем, основанных на модели инвертированных таблиц, см. гл. 2; более подробно он описывается далее). Синхронизация транзакций, журнализация изменений и восстановление баз данных после сбоев также относятся к числу функций RSS.

Компилятор запросов использует интерфейс RSS для доступа к разнообразной справочной информации (каталоги таблиц, индексов, прав доступа, условий целостности, условных воздействий и т. д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса RSS.

Таким образом, система естественно разделяется на два уровня — уровень управления памятью и синхронизацией, фактически не зависящий от базового языка запросов системы, и языковый уровень (уровень SQL), на котором решается большинство проблем System R. Заметим, что эта независимость скорее условная, чем абсолютная: язык SQL можно в принципе заменить каким-либо другим языком, но он должен обладать примерно такой же семантикой.

### **8.1.3. Организация внешней памяти в базах данных System R**

Как уже отмечалось, база данных System R располагается в одном или нескольких сегментах внешней памяти. Каждый сегмент состоит из страниц данных и страниц индексной информации. Размер страницы данных в сегменте может быть выбран равным либо 4, либо 32 килобайтам (кб); размер страницы индексной информации равен 512 б. Кроме того, при работе RSS поддерживается дополнительный набор данных для ведения журнала. Для повышения надежности журнала (а это наиболее критичная информация; при ее потере восстановление базы данных после сбоев невозможно) этот набор данных дублируется на двух внешних носителях.

## Страницы данных и идентификаторы кортежей.

В каждой странице данных хранятся кортежи одной или нескольких таблиц. Фундаментальным понятием RSS является *идентификатор кортежа* (tuple identifier — tid). Гарантируется неизменяемость tid'a во все время существования кортежа в базе данных независимо от перемещений кортежа внутри страницы и даже при перемещении кортежа в другую страницу. Потребность в перемещении кортежей возникает по той причине, что кортеж, занесенный в некоторую таблицу базы данных, вообще говоря, во время своего существования может увеличиваться в размерах (если к этой таблице добавляется новое поле, или если в ней имеется хотя бы одно поле, типом данных которого являются строки символов переменного размера). Реально tid представляет собой пару <номер страницы, индекс описателя кортежа в странице>. При этом кортеж может реально располагаться в данной странице (рис. 8.1, а) или в другой странице (рис. 8.1, б).

Как показывает рис. 8.1, в каждой странице данных имеются две области: область хранения описателей кортежей и область хранения самих кортежей. Один из остроумных приемов, примененных в System R, состоит в том, что обе эти области являются динамическими, т. е. в странице данных заранее не резервируется место под описатели кортежей. Легко видеть, что выделение фиксированной части страницы данных под описатели кортежей

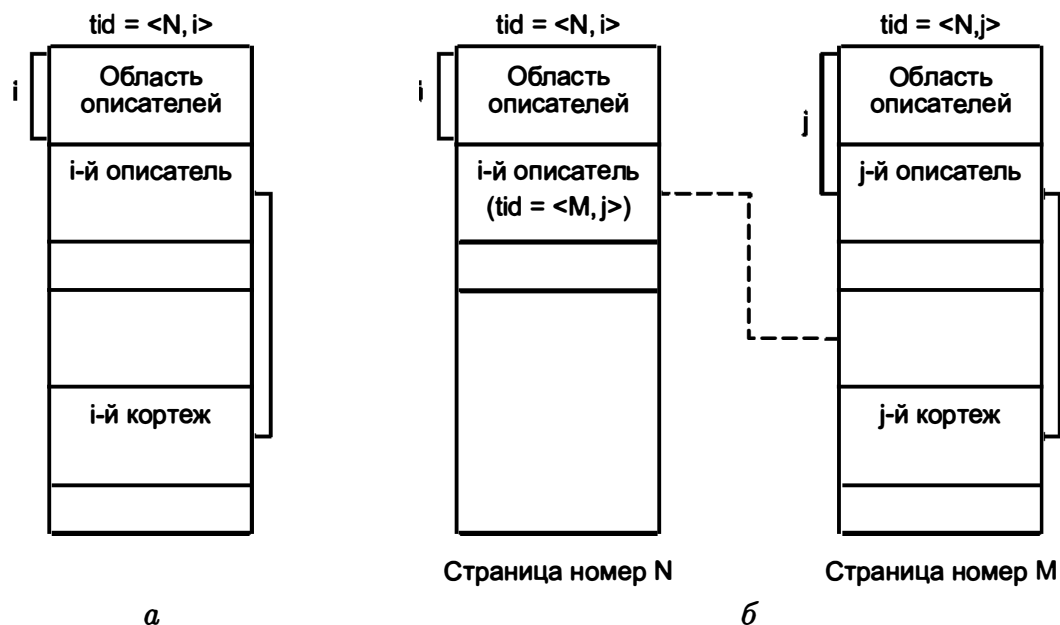


Рис. 8.1. Идентификатор кортежа (а) и расположение кортежа в странице данных (б)

(вмещающей, например,  $k$  описателей) потенциально привело бы к потере памяти в этой странице, поскольку при размещении в ней  $k$  кортежей очень маленького размера пропадало бы место в области хранения кортежей, а при размещении  $p < k$  крупных кортежей полностью заполнялась бы область хранения кортежей, но пропадало бы место в области описателей. Для динамического распределения памяти внутри страницы память на описатели кортежей выделяется вниз от начала страницы, а память для хранения кортежей — вверх от конца страницы.

Второй вариант хранения кортежей возникает в том случае, когда некоторый кортеж после своего создания был размещен системой в странице с номером  $N$ , а после обновления с увеличением размера перестал помещаться в этой странице, и система была вынуждена разместить его в странице с номером  $M$ . Тогда исходный  $tid$  этого кортежа не изменится, но его описатель в странице  $N$  будет содержать не координаты кортежа в данной странице, а новый  $tid$ , указывающий на реальное положение кортежа в странице  $M$ . Легко видеть, что применение такого подхода позволяет ограничиться максимум одним уровнем косвенности (если данный кортеж в какой-то момент времени перестанет помещаться в странице  $M$  и система переместит его в страницу  $P$ , то достаточно будет изменить косвенную ссылку на этот кортеж в странице  $N$ , и его исходный  $tid$  не изменится).

Поскольку допускается нахождение в одной странице данных кортежей разных таблиц, каждый кортеж должен кроме содержательной части включать служебную информацию, идентифицирующую таблицу, которой принадлежит данный кортеж. Кроме того, в System R (точнее, в языке SQL) допускается динамическое добавление полей к существующим таблицам. При этом реально происходит лишь модификация описателя таблицы в таблице-каталоге таблиц. В существующем кортеже таблицы новое поле возникает только при модификации этого кортежа, затрагивающей новое поле. Это позволяет избежать массовой перестройки хранимой таблицы при добавлении к ней новых полей, но, естественно, требует хранения при кортеже дополнительной служебной информации, определяющей реальное число полей в данном кортеже. (Заметим, что удалять существующие поля существующей таблицы в SQL System R не разрешалось.)

**Индексы и кластеризация таблиц.** На основе наличия уникальных, обеспечивающих почти прямой доступ к кортежам и не изменяемых во время существования кортежей  $tid$ 'ов

в System R поддерживаются дополнительные управляющие структуры — индексы. Каждый индекс определяется на одном или нескольких полях таблицы, значения которых составляют его ключ, и позволяет производить прямой поиск по ключу кортежей (их *tid'ов*) и последовательное сканирование таблицы по индексу, начиная с указанного ключа, в порядке возрастания или убывания значений ключа. Некоторые индексы при их создании могут обладать атрибутом уникальности. В таком индексе не допускаются дубликаты ключа. Это единственное средство SQL System R указания системе первичного ключа таблицы (фактически, набора первичного и всех возможных ключей таблицы).

Для организации индексов в System R применяется техника В+-деревьев (более подробно В+-деревья рассматриваются в подразд. 8.2). Каждый индекс занимает отдельный набор страниц, номер корневой страницы запоминается в описателе индекса. Использование В+-деревьев позволяет достичь эффективности при прямом поиске, поскольку они из-за своей сильной ветвистости обладают небольшой глубиной. Кроме того, В+-деревья сохраняют порядок ключей в листовых блоках иерархии, что позволяет производить последовательное сканирование таблицы в порядке возрастания или убывания значений полей, на которых определен индекс. Фундаментальное свойство В+-деревьев — автоматическая балансировка дерева — допускает произведение лишь локальных модификаций индекса при переполнениях и опустошениях страниц индекса. Насколько известно автору, System R была первой системой, в которой для организации индексов использовались В+-деревья. Эту традицию соблюдает большинство реляционных систем, возникших позднее.

Видимо, наиболее важной особенностью физической организации баз данных в System R является возможность обеспечения кластеризации связанных кортежей одной или нескольких таблиц. Под кластеризацией кортежей понимается физически близкое расположение (в пределах одной страницы данных) логически связанных кортежей. Обеспечение соответствующей кластеризации позволяет добиться высокой эффективности системы при выполнении некоторого класса запросов. В силу большой важности понятия кластеризации в System R и ее развитиях рассмотрим историю вопроса более подробно.

В окончательном варианте System R существует только одно средство определения условий кластеризации таблицы — объявить до заполнения таблицы один (и только один) индекс,

определенный на полях этой таблицы, кластеризованным. Тогда, если заполнение таблицы кортежами производится в порядке возрастания или убывания значений полей кластеризации (в зависимости от атрибутики индекса), система физически располагает кортежи в страницах данных в том же порядке.

Кроме того, в каждой странице данных кластеризованной таблицы оставляется некоторое резервное свободное пространство. При последующих вставках кортежей в такую таблицу система стремится поместить каждый кортеж в одну из страниц данных, в которых уже находятся кортежи этой таблицы с такими же (или близкими) значениями полей кластеризации. Естественно, что поддерживать идеальную кластеризацию таблицы можно только до определенного предела, пока не исчерпается резервная память в страницах. Далее этого предела степень кластеризации таблицы начинает уменьшаться, и для восстановления идеальной кластеризации таблицы требуется физическая реорганизация таблицы (ее можно произвести средствами SQL).

Очевидным преимуществом кластеризации таблицы является то, что при последовательном сканировании кластеризованной таблицы с использованием кластеризованного индекса потребуется ровно столько чтений страниц данных из внешней памяти, сколько страниц занимают кортежи этой таблицы. Следовательно, при правильно выбранных критериях кластеризации запросы, связанные с заданием условий на полях кластеризации, можно выполнить почти оптимально.

В ранних версиях System R существовал еще один способ физического доступа к кортежам таблицы и, соответственно, еще один способ указания условия кластеризации с использованием так называемых связей (links). На уровне физического представления связь — это физическая ссылка (tid) из одного кортежа на другой (не обязательно одной таблицы). В языке SEQUEL (до того момента, когда его стали называть SQL) существовали средства определения связей в иерархической манере: можно было объявить некоторую таблицу родительской по отношению к той же или другой таблице-потомку. При этом указывались поля родительской таблицы и таблицы-потомка, в соответствии со значениями которых образовывалась иерархия. Правила построения были очень простыми — проводились связи от кортежа родительской таблицы ко всем кортежам таблицы-потомка с теми же значениями полей связывания. На самом деле, все кортежи таблицы-потомка с общим значением полей связывания образовывали кольцевой список, на который проводилась одна связь из соответствующего кортежа родительской таблицы.

Следует заметить, что этот способ использования механизма связей поддерживался в ранних версиях SEQUEL. В интерфейсе RSS System R этого периода допускалась возможность произвольной установки связей без учета совпадения значений полей связывания. Тем самым, в системе в целом не использовались все возможности RSS, которые с избытком превосходили потребности организации иерархических бинарных связей по совпадению полей связывания.

Для одной таблицы допускалось создание многих связей: кортеж таблицы мог быть родителем нескольких иерархий и входить в несколько других иерархий в качестве потомка. При этом одна связь могла быть объявлена кластеризованной. Тогда система стремилась поместить в одну страницу данных все кортежи одной иерархии. При этом, естественно, использовалась возможность размещения в одной странице данных кортежей нескольких таблиц. Основной смысл такой кластеризации заключался в возможности оптимизации выполнения некоторых запросов, включающих (экви)соединение двух связанных таблиц в соответствии со значениями полей связывания.

В более поздних публикациях, посвященных System R, упоминания о механизме связей исчезли, из чего можно заключить, что разработчики отказались от его использования. Думается, что основными причинами отказа от использования связей были следующие. Во-первых, средства построения связей, обеспечиваемые RSS, были очень низкого уровня, гораздо более низкого, чем средства поддержки индексов. Если при занесении, удалении или обновлении кортежа RSS обеспечивала автоматическую коррекцию всех индексов, то для коррекции связей требовалось выполнить ряд дополнительных обращений к RSS, из-за чего время выполнения этих операций, конечно, увеличивалось.

Во-вторых, при реализации этого механизма возникают дополнительные синхронизационные проблемы нижнего уровня (уровня совместного доступа к страницам данных). В частности, наличие прямых ссылок между страницами данных увеличивает вероятность возникновения синхронизационных тупиков.

В-третьих, все эти дополнительные накладные расходы не окупались преимуществами, предоставляемыми механизмом связей. Действительно, максимального эффекта от использования связей можно достичь только при выполнении операции соединения двух таблиц, кластеризованных по этой связи, если поле соединения совпадает с полем связывания и условия, накладываемые на родительскую таблицу, выделяют в нем ровно один кортеж. Очевидно, что такие запросы на практике редки.



(Отметим, что приведенные соображения принадлежат автору и не излагались в публикациях по System R, так что на самом деле причины могли быть и другими.)

Кроме таблиц и индексов при работе System R во внешней памяти могут располагаться еще и временные объекты — списки (list). *Список* — это временная структура данных, создаваемая с целью оптимизации выполнения SQL-запроса, содержащая некоторые кортежи хранимой таблицы базы данных, не имеющая имени и, следовательно, не видимая на уровне интерфейса SQL. Кортежи списка могут быть упорядочены по возрастанию или убыванию полей соответствующей таблицы. Средства работы со списками имеются в интерфейсе RSS, но их, естественно, нет в SQL. Соответственно, эти средства используются только внутри системы при выполнении запросов (в частности, один из наиболее эффективных алгоритмов выполнения соединений основан на использовании отсортированных списков кортежей). Публикации по System R не дают точного представления о структурах данных, используемых при организации списков, но исходя из здравого смысла можно предположить, что они устроены не так, как таблицы (например, для кортежа, входящего в список, не требуется адресация через tid), и что располагаются они во временных файлах (в случае сбоя системы все временные объекты пропадают).

#### 8.1.4. Интерфейс RSS

Следует заметить, что описываемый в этом подразделе интерфейс RSS не соответствует в точности ни одной из публикаций, посвященных System R, а является скорее некоторой компиляцией, согласующейся с завершающими публикациями.

На уровне RSS отсутствует именование объектов базы данных, употребляемое на уровне SQL. Вместо имен объектов используются их уникальные идентификаторы, являющиеся прямыми или косвенными адресами внутренних описателей объектов во внешней памяти для постоянных объектов или в основной памяти для временных объектов. Замена имен объектов базы данных на их идентификаторы производится компилятором SQL на основе информации, черпаемой им из системных таблиц-каталогов.

Можно выделить следующие группы операций:

- операции сканирования таблиц и списков;
- операции создания и уничтожения постоянных и временных объектов базы данных;

- операции модификации таблиц и списков;
- операция добавления поля к таблице;
- операции управления прохождением транзакций;
- операция явной синхронизации.

**Операции сканирования таблиц и списков.** Операции группы сканирования позволяют последовательно, в порядке, определяемом типом сканирования, прочитать кортежи таблицы или списка, удовлетворяющие требуемым условиям. Группа включает операции **OPEN**, **NEXT** и **CLOSE**, означающие, соответственно, начало сканирования, требование чтения следующего кортежа, удовлетворяющего условиям, и конец сканирования.

Для таблицы возможны два режима сканирования: прямое сканирование и сканирование через индекс. При прямом сканировании единственным параметром операции **OPEN** является идентификатор таблицы (включающий и идентификатор сегмента, в котором эта таблица хранится). По причине того, что в System R допускается размещение в одной странице данных кортежей нескольких таблиц, прямое сканирование предполагает последовательный просмотр всех страниц сегмента с выделением в них кортежей, входящих в данную таблицу; это очень дорогой способ сканирования таблицы. При этом порядок выборки кортежей определяется их физическим размещением в страницах сегмента, т.е. предопределен системой.

При начале сканирования таблицы через индекс в число параметров операции **OPEN** входит идентификатор какого-либо индекса, определенного ранее на полях этой таблицы. Кроме того, можно указать диапазон сканирования в терминах значений поля (полей), составляющего ключ индекса. При открытии сканирования через индекс производится начальная установка указателя сканирования в позицию листа В+-дерева (см. подразд. 8.2) индекса, соответствующую левой границе заданного диапазона. Процесс сканирования состоит в последовательном продвижении по листовым вершинам индекса до достижения правой границы диапазона сканирования с выборкой идентификаторов кортежей и чтением соответствующих кортежей. Легко видеть, что в худшем случае может потребоваться столько чтений страниц данных из внешней памяти, сколько идентификаторов кортежей было встречено, т.е. эффективность сканирования по индексу определяется «широтой» заданного диапазона сканирования. При этом, конечно, имеется то преимущество, что порядок сканирования соответствует порядку возрастания или убывания значений ключа индекса.

Наконец, при сканировании списка, как и при прямом сканировании таблицы, единственным параметром операции **OPEN** является идентификатор списка, но в отличие от прямого сканирования таблицы это сканирование максимально эффективно: читаются только страницы, содержащие кортежи из данного списка, и порядок сканирования совпадает с порядком занесения кортежей в список или порядком списка, если он упорядочен.

В результате успешного выполнения операции открытия сканирования (если нет ошибок в параметрах) вырабатывается и возвращается идентификатор сканирования, который используется в качестве аргумента других операций этой группы.

Операция **NEXT** выполняет чтение следующего кортежа указанного сканирования, удовлетворяющего условию данной операции. Условие представляет собой дизъюнктивную нормальную форму простых условий, накладываемых на значения указанных полей таблицы. Простое условие — это условие вида **номер-поля** **оп** **константа**, где **оп** — операция сравнения **<**, **<=**, **>**, **>=**, **=** или **!=**. Общее условие является параметром операции **NEXT**.

Семантика операции **NEXT** следующая: начиная с текущей позиции сканирования выбираются кортежи таблицы в порядке, определяемом типом сканирования, до тех пор, пока не встретится кортеж, значения полей которого удовлетворяют указанному условию. Этот кортеж и является результатом операции. Если при выборке кортежа достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании через индекс или последний кортеж таблицы или списка при прямом сканировании), то вырабатывается особый признак результата. После этого единственным разумным действием является закрытие сканирования — операция **CLOSE**.

Операция **CLOSE** может быть выполнена в данной транзакции по отношению к любому ранее открытому сканированию независимо от его состояния (т.е. независимо от того, достигнута ли при сканировании правая граница диапазона сканирования). Параметром операции является идентификатор сканирования, и ее выполнение приводит к тому, что этот идентификатор становится недействительным (и, соответственно, уничтожаются служебные структуры памяти **RSS**, относящиеся к данному сканированию).

**Операции создания и уничтожения постоянных и временных объектов базы данных.** Группа операций создания и уничтожения постоянных и временных объектов базы данных включает операции создания таблиц (**CREATE TABLE**), списков (**CREATE LIST**), индексов (**CREATE IMAGE**) и уничтожения любого

из подобных объектов (**DROP TABLE**, **DROP LIST** и **DROP IMAGE**). Входным параметром операций создания таблиц и списков является спецификатор структуры объекта, т. е. число полей объекта и спецификаторы их типов. Кроме того, при спецификации полей таблицы указывается разрешение или запрещение наличия неопределенных значений полей в кортежах этой таблицы или списка. Неопределенные значения кодируются специальным образом. Любая операция сравнения константы данного типа с неопределенным значением по определению вырабатывает значение *false*, кроме операции сравнения на совпадение со специальной литеральной константой **NULL**.

В результате выполнения этих операций заводится описатель в служебной таблице описателей таблиц или основной памяти (в зависимости от того, создается ли постоянный объект или временный) и вырабатывается идентификатор объекта, который служит входным параметром других операций, относящихся к соответствующему объекту (в частности, параметром операции **OPEN** при открытии сканирования объекта).

Входными параметрами операции **CREATE IMAGE** являются идентификатор таблицы, для которой создается индекс, список номеров полей, значения которых составляют ключ индекса, и признаки упорядочения по возрастанию или убыванию для всех полей, составляющих ключ. Кроме того, может быть указан признак уникальности индекса, т. е. запрещения наличия в данном индексе ключей-дубликатов. Если операция выполняется по отношению к пустой в этот момент таблице, то выполнение операции такое же простое, как и для операций создания таблиц и списков: создается описатель в служебной таблице описателей индексов и возвращается идентификатор индекса (который, в частности, используется в качестве аргумента операции открытия сканирования таблицы через индекс).

Если же к моменту создания индекса соответствующая таблица не пуста (а это допускается), то операция становится существенно более дорогостоящей, поскольку при ее выполнении происходит реальное создание В-дерева индекса, что требует, по меньшей мере, одного последовательного просмотра таблицы. При этом, если создаваемый индекс имеет признак уникальности, то это контролируется при создании В-дерева, и если уникальность нарушается, то операция не выполняется (т. е. индекс не создается). Из этого следует, что хотя создание индексов в динамике не запрещается, более эффективно создавать все индексы на данной таблице до ее заполнения. Заметим, что создание кластеризованного индекса для непустой таблицы за-

прещено, поскольку соответствующую кластеризацию таблицы без ее реструктуризации получить невозможно.

Операции **DROP TABLE**, **DROP LIST** и **DROP IMAGE** могут быть выполнены в любой момент независимо от состояния объектов. Выполнение операции приводит к уничтожению соответствующего объекта и, вследствие этого, недействительности его идентификатора.

Следует отметить, что массовые операции над постоянными объектами (**CREATE IMAGE** и **DROP TABLE**) требуют дополнительных накладных расходов в связи с необходимостью обеспечения возможности откатов транзакции, для чего требуется выполнение массовых обратных действий. Особенно сильно это затрагивает операцию уничтожения непустых таблиц, поскольку требует журнализации всех кортежей, содержащихся в них к моменту уничтожения. Поэтому, хотя уничтожение непустых таблиц и не запрещено, нужно иметь в виду, что это очень дорогостоящая операция.

**Операции модификации таблиц и списков.** Группа операций модификации таблиц и списков включает операции вставки кортежа в таблицу или список (**INSERT**), удаления кортежа из таблицы (**DELETE**) и обновления кортежа в таблице (**UPDATE**).

Параметрами операции вставки кортежа являются идентификатор таблицы или списка и набор значений полей кортежа. Среди значений полей могут быть литеральные неопределенные значения **NULL**. Естественно, при выполнении операции контролируется допустимость неопределенных значений в соответствующих полях. При занесении кортежа в кластеризованную таблицу поиск места в сегменте под кортеж производится с использованием кластеризованного индекса: система пытается вставить кортеж в страницу данных, уже содержащую кортежи с теми же или близкими значениями полей кластеризации. При занесении кортежа в некластеризованную таблицу место под кортеж выделяется в первой подходящей странице данных. Наконец, при вставке кортежа в список он помещается в конец списка.

При занесении кортежа в таблицу производится коррекция всех индексов, определенных на этой таблице. Реально это выражается во вставке новой записи во все В-деревья индексов. При этом могут произойти переполнения одной или нескольких страниц индекса, что вызовет переливание части записей в соседние страницы или расщепление страниц. Если индекс определен с атрибутом уникальности, то проверяется соблюдение этого условия, и если оно нарушено, операция вставки

считается невыполненной. Из этого видно, что операция вставки кортежа тем более накладна, чем больше индексов определено для данной таблицы (это относится и к операциям удаления и модификации кортежей).

В результате успешного выполнения операции вставки кортежа в таблицу вырабатывается идентификатор нового кортежа, который выдается в качестве результата операции и может быть в дальнейшем использован как прямой параметр операций удаления и модификации кортежей таблицы. При занесении кортежа в список значение идентификатора кортежа не вырабатывается (для списков допускается только последовательное сканирование и добавление новых кортежей в конец списка; над ними нельзя определить индексов, и поэтому косвенная адресация кортежей списков через их идентификаторы не требуется).

Операции удаления и модификации кортежей допускаются только для кортежей таблиц. Естественно, что для выполнения этих операций необходимо идентифицировать соответствующий кортеж. В интерфейсе RSS допускаются два способа такой идентификации: с помощью идентификатора кортежа (явная адресация) и с использованием идентификатора открытого к этому времени сканирования. Первый вариант возможен, поскольку идентификатор кортежа сообщается как ответный параметр операции занесения кортежа в постоянную таблицу. При идентификации кортежа с помощью идентификатора сканирования имеется в виду кортеж, прочитанный с помощью последней операции **NEXT**. Если при такой идентификации выполняется операция **DELETE** или операция **UPDATE**, задевающая порядок сканирования (т.е. сканирование ведется по индексу и операция модификации меняет поле кортежа, входящее в состав ключа этого индекса), то текущий кортеж сканирования теряется, и его идентификатор нельзя использовать для идентификации кортежа, пока не будет выполнена следующая операция **NEXT**.

Единственным параметром операции **DELETE** является идентификатор кортежа или идентификатор сканирования. Параметры операции **UPDATE** включают, кроме этого, спецификацию изменяемых полей кортежа (список номеров полей и их новых значений). Среди значений могут находиться литеральные изображения неопределенных значений, если соответствующие поля таблицы допускают хранение неопределенных значений. При выполнении операции **DELETE** производится коррекция всех индексов, определенных на данной таблице. Операция **UPDATE** также может повлечь коррекцию индексов, если затрагивает поля, входящие в состав их ключей.

Кроме описанных «атомарных» операций сканирования и модификации таблиц и списков, интерфейс RSS включает одну «макрооперацию» **BUILDLIST**, позволяющую за одно обращение к RSS построить список, отсортированный в соответствии со значениями заданных полей. Эта операция включает сканирование заданной таблицы или списка, создание нового списка, в который включаются указанные поля выбираемых кортежей, и сортировку построенного списка в соответствии со значениями указанных полей. Идентификатор заново построенного отсортированного списка является ответным параметром операции.

Соответственно, параметрами операции **BUILDLIST** являются набор параметров для открытия сканирования (допускается любой способ сканирования), список номеров полей, составляющих кортежи нового списка, и список номеров полей, по которым нужно производить сортировку (как и в случае создания нового индекса, можно отдельно для каждого из этих полей указать требование к сортировке по возрастанию или убыванию значений данного поля). Отдельным параметром операции **BUILDLIST** является признак, в соответствии со значением которого в новом списке допускаются или не допускаются кортежи-дубликаты.

**Операция добавления поля к существующей таблице.** Операция RSS добавления поля к существующей таблице позволяет в динамике изменять схему таблицы. Параметрами операции **CHANGE** являются идентификатор существующей таблицы и спецификация нового поля (его тип). При выполнении операции изменяется только описатель данной таблицы в служебной таблице описателей таблиц. До выполнения первой операции **UPDATE**, затрагивающей новое поле таблицы, реально ни в одном кортеже таблицы память под новое поле выделяться не будет. По умолчанию значения нового поля во всех кортежах таблицы, в которые еще не производилось явное занесение значения, считаются неопределенными. Тем самым, ни для одного поля, динамически добавленного к существующей таблице, не может быть запрещено хранение неопределенных значений.

**Операции управления прохождением транзакций.** Каждая операция RSS выполняется в пределах некоторой транзакции. Интерфейс RSS включает набор операций управления прохождением транзакции: начать транзакцию (**BEGIN TRANSACTION**), закончить транзакцию (**END TRANSACTION**), установить точку сохранения (**SAVE**) и выполнить откат до указанной точки сохранения или до начала транзакции (**RESTORE**).

Это не отмечалось раньше, но на самом деле при вызове любой операции функции RSS, кроме **BEGIN TRANSACTION**, должен ука-

зываться еще один параметр — идентификатор транзакции. Этот идентификатор и вырабатывается при выполнении операции **BEGIN TRANSACTION**, которая сама входных параметров не требует.

В любой точке транзакции до выполнения операции **END TRANSACTION** может быть выполнен откат данной транзакции, т. е. обратное выполнение всех изменений, произведенных в данной транзакции, и восстановление состояния позиций сканирования. Откат может быть произведен до начала транзакции (в этом случае о восстановлении позиций сканирования говорить бессмысленно) или до установленной ранее в транзакции точки сохранения.

Точка сохранения устанавливается с помощью операции **SAVE**. При выполнении этой операции запоминаются состояние сканов данной транзакции, открытых к моменту выполнения **SAVE**, и координаты последней записи об изменениях в базе данных в журнале, произведенной от имени данной транзакции. Ответным параметром операции **SAVE** (а прямых параметров, кроме идентификатора транзакции, она не требует) является идентификатор точки сохранения. Этот идентификатор в дальнейшем может быть использован как аргумент операции **RESTORE**, при выполнении которой производится восстановление базы данных по журналу (с использованием записей о ее изменениях от данной транзакции) до того состояния, в котором находилась база данных к моменту установки указанной точки сохранения. Кроме того, по локальной информации в оперативной памяти, привязанной к транзакции, восстанавливается состояние ее сканов. Откат к началу транзакции инициируется также вызовом операции **RESTORE**, но с указанием некоторого предопределенного идентификатора точки сохранения.

При выполнении своих транзакций пользователи System R изолированы один от другого, т. е. не ощущают того, что система функционирует в многопользовательском режиме. Это достигается за счет наличия в RSS механизма неявной синхронизации. До конца транзакции никакие изменения базы данных, произведенные в пределах этой транзакции, не могут быть использованы в других транзакциях (попытка использования таких данных приводит к временным синхронизационным блокировкам этих транзакций). При выполнении операции **END TRANSACTION** происходит «фиксация» изменений, произведенных в данной транзакции, т. е. они становятся видимыми в других транзакциях. Реально это означает снятие синхронизационных блокировок с объектов базы данных, изменявшихся в транзакции. Из этого следует, что после выполнения **END TRANSACTION** невозможны индивидуальные откаты данной транзакции. RSS просто делает



недействительным идентификатор данной транзакции и после выполнения операции окончания транзакции отвергает все операции с таким идентификатором.

**Операция явной синхронизации.** Последняя операция интерфейса RSS — операция явной синхронизации LOCK. Эта операция позволяет установить явную синхронизационную блокировку указанной таблицы (параметром операции является идентификатор таблицы). Выполнение операции LOCK гарантирует, что никакая другая транзакция до конца данной не сможет изменить эту таблицу (вставить в нее новый кортеж, удалить или модифицировать существующий), если установлена блокировка в режиме чтения, или даже прочитать любой кортеж этой таблицы, если установлена монопольная блокировка.

Из всего, что говорилось ранее по поводу подхода к синхронизации в System R и соответствующего разбиения системы на уровни, следует нелогичность наличия этой операции в интерфейсе RSS. На самом деле, логически эта операция избыточна, т. е. если бы ее не было, можно было бы реализовать SQL с использованием оставшейся части операций. Предварительно (до подробного обсуждения средств управления транзакциями в гл. 9) заметим, что операция LOCK введена в интерфейс RSS для возможности оптимизации выполнения запросов.

Дело в том, что, как видно из описания интерфейса RSS, этот интерфейс является покортёжным. Следовательно, и информация для синхронизации носит достаточно узкий характер. В то же время, на уровне SQL имеется более полная информация. Например, если обрабатывается предложение SQL `DELETE FROM table_name`, то известно, что будут удалены все кортежи указанной таблицы. Понятно, что как бы не реализовывался механизм синхронизации в RSS, в данном случае выгоднее сообщить сразу, что изменения касаются всей таблицы.

Но ситуации, в которых очевидна выгода от использования явной синхронизации, достаточно редки. Пользоваться этим средством можно только очень осмотрительно, потому что неоправданные захваты таких крупных объектов могут резко ограничить степень асинхронности выполнения транзакций.

## **8.2. Общие принципы организации данных во внешней памяти в SQL-ориентированных СУБД**

В этом подразделе кратко обсуждаются основные подходы к организации данных во внешней памяти, принятые в совре-

менных SQL-ориентированных СУБД. В большинстве случаев они основаны на идеях, заложенных в System R, хотя, конечно, в любой развитой системе имеются собственные приемы, которые здесь обсуждаться не будут.

SQL-ориентированные СУБД обладают рядом особенностей, влияющих на организацию внешней памяти. Наиболее важным автору кажутся следующие особенности:

- Наличие двух уровней системы: уровня непосредственного управления данными во внешней памяти (а также обычно управления буферами оперативной памяти, управления транзакциями и журнализацией изменений БД) и языкового уровня (уровня, реализующего язык SQL). При такой организации подсистема нижнего уровня должна поддерживать во внешней памяти набор базовых структур, конкретная интерпретация которых входит в число функций подсистемы верхнего уровня.
- Поддержка таблиц-каталогов. Информация, связанная с именованием объектов базы данных и их конкретными свойствами (например, структура ключа индекса), поддерживается подсистемой языкового уровня. С точки зрения структур внешней памяти таблица-каталог ничем не отличается от обычной таблицы базы данных.
- Регулярность структур данных. Поскольку основным объектом модели данных SQL является плоская таблица, основной набор объектов внешней памяти может иметь очень простую регулярную структуру.
- Необходимость обеспечения возможности эффективного выполнения операторов языкового уровня как над одной таблицей (простые селекция и проекция), так и над несколькими таблицами (наиболее распространено и трудоемко соединение нескольких таблиц). Для этого во внешней памяти должны поддерживаться дополнительные «управляющие» структуры — индексы.
- Наконец, для выполнения требования надежного хранения баз данных необходимо поддерживать избыточность хранения данных, что обычно реализуется в виде журнала изменений базы данных.

Соответственно, возникают следующие разновидности объектов во внешней памяти базы данных:

- строки таблиц — основная часть базы данных, большей частью непосредственно видимая пользователям;
- управляющие структуры — индексы, создаваемые по инициативе пользователя (администратора) или верхнего

уровня системы из соображений повышения эффективности выполнения запросов и обычно автоматически поддерживаемые нижним уровнем системы;

- журнальная информация, поддерживаемая для удовлетворения потребности в надежном хранении данных;
- служебная информация, поддерживаемая для удовлетворения внутренних потребностей нижнего уровня системы (например, информация о свободной памяти).

### 8.2.1. Хранение таблиц

Существуют два принципиальных подхода к физическому хранению таблиц. Наиболее распространенным является по-кортежное хранение таблиц (единицей физического хранения является кортеж). Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одной таблицы и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа.

Альтернативным (менее распространенным) подходом является хранение таблицы по столбцам, т. е. единицей хранения является столбец таблицы с исключенными дубликатами. Естественно, что при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся; за один обмен с внешней памятью в общем случае считывается больше полезной информации. Дополнительным преимуществом является возможность использования значений столбца таблицы для оптимизации выполнения операций соединения. Но при этом требуются существенные дополнительные действия для сборки целого кортежа (или его части).

Поскольку гораздо более распространено хранение по строкам, рассмотрим немного более подробно этот способ хранения таблиц (в дополнение к тому, что говорилось в подразд. 8.1). Типовой, унаследованной от System R, структурой страницы данных является та, которая показана на рис. 8.1.

Эту организацию хранения кортежей можно в целом охарактеризовать следующим образом:

- каждый кортеж обладает уникальным идентификатором (tid), не изменяемым во все время существования кортежа и позволяющим выбрать кортеж в основную память не более чем за два обращения к внешней памяти. Структура tid следует из рис. 8.1;

- обычно каждый кортеж хранится целиком в одной странице. Из этого следует, что максимальная длина кортежа любой таблицы ограничена размерами страницы. Возникает вопрос: как быть с «длинными» данными, которые в принципе не помещаются в одной странице? Применяется несколько методов. Наиболее простым решением является хранение таких данных в отдельных (вне базы данных) файлах с заменой «длинного» данного в кортеже на имя соответствующего файла. В некоторых системах такие данные хранились внутри базы данных в отдельном наборе страниц внешней памяти, связанном физическими ссылками. Оба эти решения сильно ограничивают возможность работы с длинными данными (как, например, удалить несколько байт из середины 2-мегабайтной строки?). В настоящее время все чаще используется метод, предложенный много лет тому назад в проекте Exodus [34], когда «длинные» данные организуются в виде В-деревьев последовательностей байт;
- как правило, в одной странице данных хранятся кортежи только одной таблицы. Существуют, однако, варианты с возможностью хранения в одной странице кортежей нескольких таблиц. Это вызывает некоторые дополнительные расходы по части служебной информации (при каждом кортеже нужно хранить информацию о соответствующей таблице), но зато иногда позволяет резко сократить число обменов с внешней памятью при выполнении соединений;
- изменение схемы хранимой таблицы с добавлением нового поля не вызывает потребности в физической реорганизации таблицы. Достаточно лишь изменить информацию в описателе таблицы и расширять кортежи только при занесении информации в новое поле;
- поскольку таблицы могут содержать неопределенные значения, необходима соответствующая поддержка на уровне хранения. Обычно это достигается путем хранения соответствующей шкалы при каждом кортеже, который в принципе может содержать неопределенные значения;
- проблема распределения памяти в страницах данных связана с проблемами синхронизации и журнализации и не всегда тривиальна. Например, если в ходе выполнения транзакции некоторая страница данных опустошается, то ее нельзя перевести в статус свободных страниц до конца транзакции, поскольку при откате транзакции удаленные при прямом выполнении транзакции и восстановленные при ее откате кортежи должны получить те же самые идентификаторы;

- распространенным способом повышения эффективности СУБД является кластеризация таблицы по значениям одного или нескольких столбцов. Полезной для оптимизации соединений является совместная кластеризация нескольких таблиц;
- в целях использования возможностей распараллеливания обменов с внешней памятью иногда применяют схему декластеризованного хранения таблиц: кортежи с общим значением столбца декластеризации размещают на разных дисковых устройствах, обмены с которыми можно выполнять параллельно.

Что же касается хранения таблицы по столбцам, то основная идея состоит в совместном хранении всех значений одного (или нескольких) столбцов. Для каждого кортежа таблицы хранится кортеж той же степени, состоящий из ссылок на места расположения соответствующих значений столбцов.

### 8.2.2. Индексы

Как бы не были организованы индексы в конкретной СУБД, их основное назначение состоит в обеспечении эффективного прямого доступа к кортежу таблицы по ключу. Обычно индекс определяется для одной таблицы и ключом является значение ее поля (возможно, составного). Если ключом индекса является возможный ключ таблицы, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа таблицы автоматически заводится уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа, так или иначе, требуется индексная поддержка.

Поскольку при выполнении многих операций уровня SQL требуется сортировка кортежей таблиц в соответствии со значениями некоторых полей, полезным свойством индекса является обеспечение последовательного просмотра кортежей таблицы в заданном диапазоне значений ключа в порядке возрастания или убывания значений ключа.

Наконец, одним из способов оптимизации выполнения эквивалентности таблиц (наиболее распространенная из числа дорогостоящих операций) является организация так называемых мультииндексов для нескольких таблиц, обладающих общими

атрибутами. Любой из этих атрибутов (или их набор) может выступать в качестве ключа мультииндекса. Значению ключа сопоставляется набор кортежей всех связанных мультииндексом таблиц, значения выделенных атрибутов которых совпадают со значением ключа.

Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу и последовательный просмотр в порядке возрастания или убывания значений ключа, является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Одна организация индекса отличается от другой, главным образом, в способе поиска ключа с заданным значением.

**В-деревья.** Наиболее популярным подходом к организации индексов в базах данных является использование техники В+-деревьев. Техника В- и В+-деревьев была предложена в начале 1970-х гг. Рудольфом Байером (Rudolf Bayer) и Эдом Маккрейтом (Ed McCreight). С точки зрения внешнего логического представления, В-дерево — это сбалансированное сильно ветвистое дерево во внешней памяти. Сбалансированность означает, что длина пути от корня дерева к любому его листу одна и та же. Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков. С точки зрения физической организации, В-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). В В+-дереве внутренние и листовые страницы обычно имеют разную структуру.

Типовая структура внутренней страницы В+-дерева показана на рис. 8.2.

При этом выдерживаются следующие свойства:

- $ключ_1 \leq ключ_2 \leq \dots \leq ключ_m$ ;
- в странице дерева  $N_m$  находятся ключи  $k$  со значениями  $ключ_m \leq k \leq ключ_{m+1}$ .

Листовая страница обычно имеет структуру, показанную на рис. 8.3.



Рис. 8.2. Типовая структура внутренней страницы В+-дерева

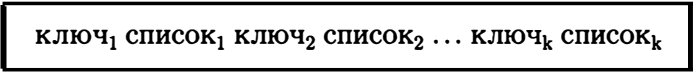


Рис. 8.3. Структура листовой страницы В+-дерева

Листовая страница обладает следующими свойствами:

- $\text{ключ}_1 < \text{ключ}_2 < \dots < \text{ключ}_k$ ;
- $\text{список}_r$  — упорядоченный список идентификаторов кортежей ( $\text{tid}$ ), включающих значение  $\text{ключ}_r$ ;
- листовые страницы связаны одно- или двунаправленным списком.

Поиск в В+-дереве — это прохождение от корня к листу в соответствии с заданным значением ключа. Заметим, что поскольку В+-деревья являются сильно ветвистыми и сбалансированными, для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутренней странице помещается  $n$  ключей, то при хранении  $m$  записей требуется дерево глубиной  $\log_n(m)$ . Если  $n$  достаточно велико (обычный случай), то глубина дерева невелика и производится быстрый поиск.

Основной «изюминкой» В+-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций занесения и удаления записей.

При занесение новой записи выполняются следующие действия:

- поиск листовой страницы. Фактически, производится обычный поиск по ключу. Если в В+-дереве не содержится ключ с заданным значением, то будет получен номер страницы, в которой ему надлежит содержаться, и соответствующие координаты внутри страницы;
- помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер страницы внешней памяти;
- если после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память или временно сохранен в основной памяти в зависимости от политики управления буферами;
- если же возникло переполнение буфера (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы. Для этого запрашива-

ется новая страница внешней памяти, используемая часть буфера разбивается примерно пополам (так, чтобы вторая половина также начиналась с ключа) и вторая половина записывается во вновь выделенную страницу, а в старой странице модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых страниц;

- чтобы обеспечить доступ от корня дерева к заново заведенной странице, необходимо соответствующим образом модифицировать внутреннюю страницу, являющуюся предком ранее существовавшей листовой страницы, т. е. вставить в нее соответствующее значение ключа и ссылку на новую страницу. При выполнении этого действия может снова произойти переполнение теперь уже внутренней страницы, и она будет расщеплена на две. В результате потребуется вставить значение ключа и ссылку на новую страницу во внутреннюю страницу-предка выше по иерархии и т. д.;
- предельным случаем является переполнение корневой страницы В+-дерева. В этом случае она тоже расщепляется на две и заводится новая корневая страница дерева, т. е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия:

- поиск записи по ключу. Если запись не найдена, то удалять ничего не нужно;
- реальное удаление записи в буфере, в который прочитана соответствующая листовая страница;
- если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых страницах, являющихся левым или правым братом данной страницы, больше, чем размер страницы, операция завершается;
- иначе производится слияние с правым или левым братом, т. е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и ее левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц;
- чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы — ее предка. При этом может возникнуть потребность в слиянии этой страницы с ее левым или правым братом и т. д.;



- предельным случаем является полное опустошение корневой страницы дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневая страница освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности В+-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

- упреждающие расщепления, т.е. расщепления страницы не при ее переполнении, а несколько раньше, когда степень заполненности страницы достигает некоторого уровня;
- переливания, т.е. поддержание равновесного заполнения соседних страниц;
- слияния 3-в-2, т.е. порождение двух листовых страниц на основе содержимого трех соседних.

Следует заметить, что при организации мультидоступа к В+-деревам, характерного при их использовании в СУБД, приходится решать ряд нетривиальных проблем. Конечно, грубые решения очевидны, например, возможен монопольный захват В+-дерева (т.е. его корневого блока) на все время выполнения операции модификации. Но существуют и более тонкие решения, рассмотрение которых выходит за пределы данной книги.

**Хэширование.** Альтернативным и достаточно популярным подходом к организации индексов является использование техники хэширования. Это очень обширная тема, которая заслуживает отдельного рассмотрения. Ограничимся здесь лишь несколькими замечаниями. Общей идеей методов хэширования является применение к значению ключа некоторой функции свертки (хэш-функции), вырабатывающей значение меньшего размера. Значение хэш-функции затем используется для доступа к записи.

В самом простом, классическом случае свертка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значения свертки (одним из распространенных видов «хороших» хэш-функций являются функции, выдающие остаток от деления значения ключа на некоторое простое число). При возникновении коллизий (одна и та же свертка для

нескольких значений ключа) образуются цепочки переполнения. Главным ограничением этого метода является фиксированный размер таблицы. Если таблица заполнена слишком сильно или переполнена, но возникнет слишком много цепочек переполнения, и главное преимущество хэширования — доступ к записи почти всегда за одно обращение к таблице — будет утрачено. Расширение таблицы требует ее полной переделки на основе новой хэш-функции (со значением свертки большего размера).

Идея доступа к данным на основе хэширования настолько привлекательна (потенциальная возможность за одно обращение к памяти получить требуемые данные), что от нее невозможно отказаться при работе с данными во внешней памяти. Исходная идея кажется очевидной: если при управлении данными на основе хэширования в основной памяти хэш-функция вырабатывает адрес требуемого элемента, то при обращении к внешней памяти необходимо генерировать номер блока дискового пространства, в котором находится запрашиваемый элемент данных. Основная проблема относится к коллизиям. Если при работе в основной памяти потенциально возникающими потребностями дополнительного поиска информации при возникновении коллизий можно, вообще говоря, пренебречь (поскольку время доступа к основной памяти мало), то при использовании внешней памяти любое дополнительное обращение вызывает существенные накладные расходы. Основные методы хэширования для поиска информации во внешней памяти направлены на решение именно этой задачи.

В основе подхода расширяемого хэширования (Extendible Hashing) [35] лежит принцип использования деревьев цифрового поиска в основной памяти. В основной памяти поддерживается справочник, организованный на основе бинарного дерева цифрового поиска, ключами которого являются значения хэш-функции, а в листовых вершинах хранятся номера блоков записей во внешней памяти. В этом случае любой поиск в дереве цифрового поиска является «успешным», т. е. ведет к некоторому блоку внешней памяти. Входит ли в этот блок искомая запись, обнаруживается уже после прочтения блока в основную память.

Проблема коллизий переформулируется следующим образом. Как таковых, коллизий не существует. Может возникнуть лишь ситуация переполнения блока внешней памяти. Значение хэш-функции указывает на этот блок, но места для включения записи в нем уже нет. Эта ситуация обрабатывается так. Блок расщепляется на два, и дерево цифрового поиска переформи-

руется соответствующим образом. Конечно, при этом может потребоваться расширение самого справочника.

Расширяемое хэширование хорошо работает в условиях динамически изменяемого набора записей в хранимом файле, но требует наличия в основной памяти справочного дерева.

Идея линейного хэширования (Linear Hashing) [36] состоит в том, чтобы можно было обойтись без поддержания справочника в основной памяти. Основой метода является то, что для адресации блока внешней памяти всегда используются младшие биты значения хэш-функции. Если возникает потребность в расщеплении, то записи перераспределяются по блокам так, чтобы адресация осталась правильной.

### **8.2.3. Журнальная информация**

Структура журнала обычно является сугубо частным делом конкретной реализации. Отметим только самые общие свойства.

Журнал обычно представляет собой чисто последовательный файл с записями переменного размера, которые можно просматривать в прямом или обратном порядке. Обмены производятся стандартными порциями (страницами) с использованием буфера оперативной памяти. В грамотно организованных системах структура (и тем более, смысл) журнальных записей известна только компонентам СУБД, ответственным за журнализацию и восстановление.

Поскольку содержимое журнала является критичным при восстановлении базы данных после сбоев, к ведению файла журнала предъявляются особые требования по части надежности. В частности, обычно стремятся поддерживать две идентичные копии журнала на разных устройствах внешней памяти.

### **8.2.4. Служебная информация**

Для корректной работы подсистемы управления данными во внешней памяти необходимо поддерживать информацию, которая используется только этой подсистемой и не видна подсистеме языкового уровня. Набор структур служебной информации зависит от общей организации системы, но обычно требуется поддержание следующих служебных данных:

- внутренние каталоги, описывающие физические свойства объектов базы данных, например, число атрибутов таблицы, их размер и, возможно, типы данных; описание индексов, определенных для данной таблицы и т. д.;

- описатели свободной и занятой памяти в страницах данных. Такая информация требуется для нахождения свободного места при занесении кортежа. Отдельно приходится решать задачу поиска свободного места в случаях некластеризованных и кластеризованных таблиц (в последнем случае приходится дополнительно использовать кластеризованный индекс). Как уже отмечалось, нетривиальной является проблема освобождения страницы в условиях мультидоступа;
- связывание страниц одной таблицы. Если в одном файле внешней памяти могут располагаться страницы нескольких таблиц (обычно к этому стремятся), то нужно каким-то образом связать страницы одной таблицы. Тривиальный способ использования прямых ссылок между страницами часто приводит к затруднениям при синхронизации транзакций (например, особенно трудно освобождать и заводить новые страницы таблицы). Поэтому стараются использовать косвенное связывание страниц с использованием служебных индексов. В частности, известен общий механизм для описания свободной памяти и связывания страниц на основе В-деревьев.

\* \* \*

В этой главе была кратко описана экспериментальная реляционная СУБД System R, оказавшая гигантское влияние на становление современной технологии баз данных. Были рассмотрены основные цели проекта System R, общая архитектура системы, основные структуры данных внешней памяти и интерфейс подсистемы управления внешней памятью.

Далее были обсуждены основные подходы к организации внешней памяти, применяемые в современных СУБД. Конечно, в любой конкретной SQL-ориентированной СУБД используется ряд собственных приемов организации хранения таблиц и индексов, но практически во всех случаях общие принципы похожи на те, которые описаны в подразд. 8.2.

### МЕТОДЫ УПРАВЛЕНИЯ ТРАНЗАКЦИЯМИ. СИХРОНИЗАЦИОННЫЕ БЛОКИРОВКИ, ВРЕМЕННЫЕ МЕТКИ И ВЕРСИИ

Поддержка механизма транзакций — показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей в многопользовательских системах. Часто эти два аспекта рассматриваются по отдельности, но на самом деле они взаимосвязаны, что и будет показано в этой главе.

#### 9.1. Общее понятие транзакции и основные характеристики транзакций

Более точно, в современных СУБД поддерживается понятие транзакции, характеризующее аббревиатурой ACID (Atomicity, Consistency, Isolation и Durability). В соответствии с этим понятием под транзакцией понимается последовательность операций над базой данных, обладающая следующими свойствами:

- *атомарность (Atomicity)*. Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство атомарности, которое часто называют свойством «все или ничего», позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных (в общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения);
- *согласованность (Consistency)*. В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций толь-

ко в том случае, когда действия операций не нарушают *целостность* базы данных, т.е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. Это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки согласованности и явным образом проверять ограничения целостности (с точки зрения автора, в контексте баз данных термины *согласованность* и *целостность* эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности, т.е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии);

- *изоляция (Isolation)*. Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции *T1* не должны быть видны никакой другой транзакции *T2* до тех пор, пока транзакция *T1* не завершится успешным образом;
- *долговечность (Durability)*. После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоя аппаратуры или программного обеспечения. Этому аспекту транзакционных систем посвящается гл. 10.

Заметим, что хотя с точки зрения обеспечения целостности баз данных механизм транзакций следовало бы поддерживать в персональных СУБД, на практике это обычно не выполняется. Поэтому при переходе от персональных к многопользовательским СУБД пользователи сталкиваются с необходимостью четкого понимания природы транзакций.

### 9.1.1. Атомарность транзакций

В этом смысле под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в состоянии базы данных, либо воздействие всех этих операторов полностью отсутствует.

Лозунгом транзакции является «Все или ничего»: при завершении транзакции оператором COMMIT (высокоуровневый аналог операции END TRANSACTION в интерфейсе RSS, см. гл. 8) резуль-

таты гарантированно фиксируются во внешней памяти (смысл термина *commit* состоит в запросе «фиксации» результатов транзакции); при завершении транзакции оператор **ROLLBACK** (высокоуровневый аналог операции **RESTORE** в интерфейсе **RSS**, см. гл. 8) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Каким образом в СУБД поддерживаются индивидуальные откаты транзакций, описывается в гл. 10.

### 9.1.2. Транзакции и целостность баз данных

Понятие транзакции имеет непосредственную связь с понятием целостности базы данных. Очень часто база данных может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения базы данных. Например, в базе данных **СЛУЖАЩИЕ-ОТДЕЛЫ** (см. гл. 1) естественным ограничением целостности является совпадение значения атрибута **ОТД\_РАЗМЕР** в кортеже таблицы **ОТДЕЛЫ**, описывающей данный отдел (например, отдел 625), с числом кортежей таблицы **СЛУЖАЩИЕ**, таких, что значение поля **СЛУ\_ОТД\_НОМЕР** равно 625. Как в этом случае принять на работу в отдел 625 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в таблице **СОТРУДНИКИ** или модификация существующего кортежа в отношении **ОТДЕЛЫ**, после выполнения операции база данных окажется в нецелостном состоянии.

Поэтому для поддержки подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии базы данных и должна оставить это состояние целостными после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат (т.е. вместо оператора **COMMIT** выполняется оператор **ROLLBACK**), и база данных остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Более точно, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно

откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (например, возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: заработная плата сотрудника не может быть увеличена за одну операцию более чем на 100 000 руб. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности — это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора **COMMIT** на оператор **ROLLBACK**. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор **ROLLBACK** с откатом транзакции до ее начала или до установленной ранее точки сохранения или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

Заметим, что концептуально в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако в реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных **СЛУЖАЩИЕ-ОТДЕЛЫ** в ней не выполнялись операторы вставки или удаления кортежей из отношения **СЛУЖАЩИЕ**, то проверять упоминавшееся ранее ограничение целостности не требуется (а для проверки подобных ограничений требуется достаточно большая работа).

Понятно, что описанный механизм поддержки целостности баз данных обеспечивает требуемое свойство транзакций: никакая транзакция не может быть зафиксирована, если ее действия нарушили целостность базы данных. Однако в этом подходе имеются два серьезных дефекта.

Во-первых, если при выполнении транзакции не устанавливать точки сохранения и не проверять периодически соответствие текущего состояния базы данных (с точки зрения данной



транзакции) ограничениям целостности, то долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора **COMMIT**. Конечно, это означает непроизводительный расход системных ресурсов и времени пользователей. Во-вторых, чем длиннее транзакция, модифицирующая состояние базы данных, тем потенциально больше ограничений целостности придется проверять при ее завершении и тем накладнее становится оператор **COMMIT**.

Простое и элегантное решение этой проблемы предлагается в [11]. Авторы предлагают отказаться от откладываемых ограничений целостности базы данных, а вместо этого ввести составные операторы изменения базы данных (нечто наподобие блоков **BEGIN ... END**, поддерживаемых в языках программирования). После выполнения каждого такого блока (или отдельного оператора изменения базы данных, используемого без операторов начала и конца блока) база данных должна находиться в целостном состоянии. Если составной оператор нарушает ограничение целостности, то он целиком отвергается и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. Понятно, что при использовании такого подхода при выполнении оператора **COMMIT** не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии.

Интересно, что для реализации описанного подхода не требуются какие-либо новые механизмы, кроме точек сохранения транзакции, насильственной проверки ограничений целостности и частичных откатов транзакций, а отмеченные ранее проблемы снимаются. К сожалению, насколько известно автору данной книги, этот подход на практике пока не применяется.

### 9.1.3. Изолированность транзакций

В многопользовательских системах с одной базой данных одновременно может работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т. е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с базой данных в одиночку.

В связи со свойством сохранения целостности базы данных транзакции являются подходящими единицами изолированности пользователей. Действительно, если с каждым сеансом работы пользователя или приложений с базой данных ассоциируется

транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.

При соблюдении обязательного требования поддержки целостности базы данных возможно наличие нескольких уровней изолированности транзакций. Заметим, что впервые эти уровни изолированности транзакций были установлены и описаны участниками проекта System R.

**Отсутствие потерянных изменений (первый уровень изолированности).** Рассмотрим сценарий совместного выполнения двух транзакций, показанный на рис. 9.1. В момент времени  $t_1$  транзакция  $T_1$  изменяет объект базы данных  $o$  (выполняет операцию  $W(o)$ ). До завершения транзакции  $T_1$  в момент времени  $t_2 > t_1$  транзакция  $T_2$  также изменяет объект  $o$ . В момент времени  $t_3 > t_2$  транзакция  $T_2$  завершается оператором **ROLLBACK** (например, по причине нарушения ограничений целостности).

Тогда при повторном чтении объекта  $o$  (выполнении операции  $R(o)$ ) в момент времени  $t_4 > t_3$  транзакция  $T_1$  не видит своих изменений этого объекта, произведенных ранее (в частности, из-за этого может не удастся фиксация этой транзакции, что, возможно, повлечет потерю изменений у еще одной транзакции и т.д.).

Такая ситуация называется ситуацией *потерянных изменений*. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции  $T_1$ , требуется, чтобы до завершения транзакции  $T_1$  никакая другая транзакция не могла изменять никакой измененный транзакцией  $T_1$  объект  $o$  (в частности, достаточно заблокировать доступ по изменению к объекту  $o$  до завершения

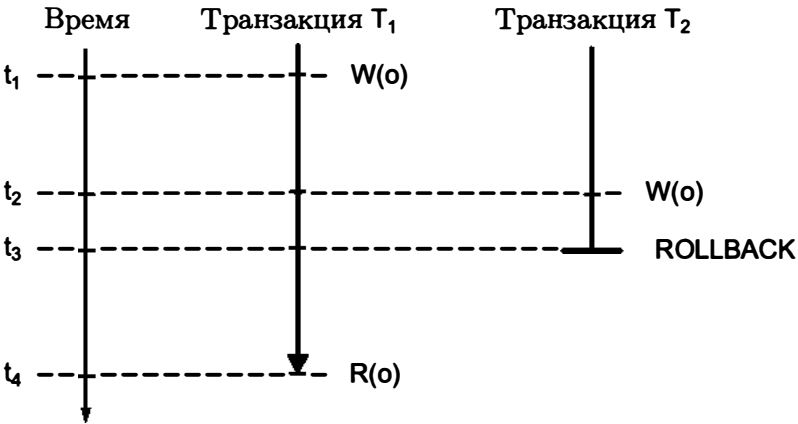


Рис. 9.1. Потерянные изменения

транзакции  $T_1$ ). Отсутствие потерянных изменений является минимальным требованием к СУБД при обеспечении изолированности одновременно выполняемых транзакций.

**Отсутствие чтения «грязных» данных (второй уровень изолированности).** Рассмотрим сценарий совместного выполнения транзакций  $T_1$  и  $T_2$ , показанный на рис. 9.2. В момент времени  $t_1$  транзакция  $T_1$  изменяет объект базы данных  $o$  (выполняет операцию  $W(o)$ ). В момент времени  $t_2 > t_1$  транзакция  $T_2$  читает объект  $o$  (выполняет операцию  $R(o)$ ). Поскольку транзакция  $T_1$  еще не завершена, транзакция  $T_2$  видит несогласованные «грязные» данные. В частности, в момент времени  $t_3 > t_2$  транзакция  $T_1$  может завершиться откатом (например, по причине нарушения ограничений целостности).

Эта ситуация также не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и имеет право видеть только согласованные данные). Чтобы избежать ситуации чтения «грязных» данных, до завершения транзакции  $T_1$ , изменившей объект базы данных  $o$ , никакая другая транзакция не должна читать объект  $o$  (например, достаточно заблокировать доступ по чтению к объекту  $o$  до завершения изменившей его транзакции  $T_1$ ).

**Отсутствие неповторяющихся чтений (третий уровень изоляции).** Рассмотрим сценарий совместного выполнения транзакций  $T_1$  и  $T_2$ , показанный на рис. 9.3. В момент времени  $t_1$  транзакция  $T_1$  читает объект базы данных  $o$  (выполняет операцию  $R(o)$ ). До завершения транзакции  $T_1$  в момент времени  $t_2 > t_1$  транзакция  $T_2$  изменяет объект  $o$  (выполняет операцию  $W(o)$ ) и успешно завершается оператором COMMIT. В момент времени  $t_3 > t_2$  транзакция  $T_1$  повторно читает объект  $o$  и видит его измененное состояние.

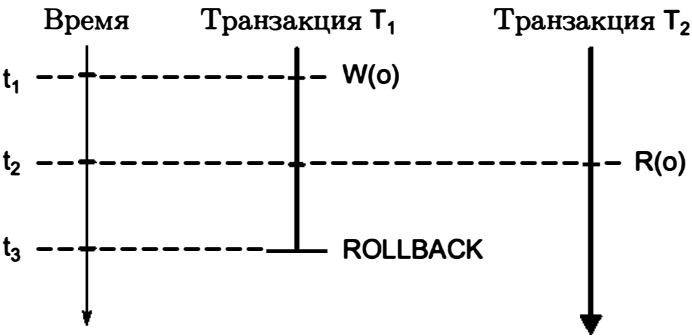


Рис. 9.2. «Грязные» чтения

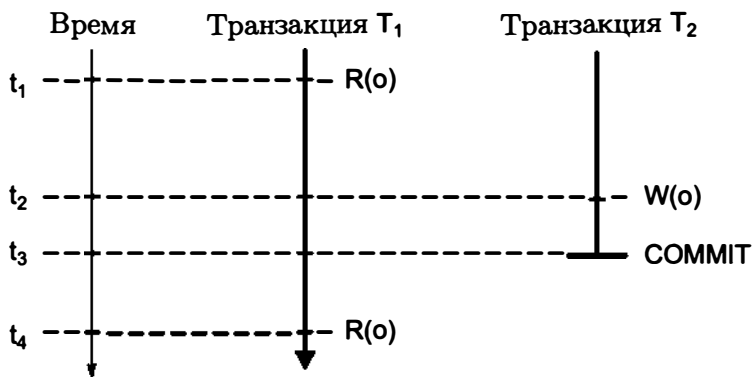


Рис. 9.3. Неповторяющиеся чтения

Чтобы избежать *неповторяющихся чтений*, до завершения транзакции  $T_1$  никакая другая транзакция не должна изменять объект  $o$  (для этого достаточно заблокировать доступ по записи к объекту  $o$  до завершения транзакции  $T_1$ ). Часто это является максимальным требованием к средствам обеспечения изолированности транзакций, хотя, как будет показано далее, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (кстати, соответствующие операторы были предусмотрены уже в стандарте SQL:1992). Как уже отмечалось, для корректного соблюдения ограничений целостности достаточен первый уровень. Существует ряд приложений, которым хватает первого уровня изолированности (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удастся существенно сократить накладные расходы СУБД и повысить общую эффективность.

**Проблема фантомов.** К более тонким проблемам изолированности транзакций относится так называемая проблема *кортежей-«фантомов»*, приводящая к ситуациям, которые также противоречат изолированности пользователей. Рассмотрим сценарий, показанный на рис. 9.4.

В момент времени  $t_1$  транзакция  $T_1$  выполняет оператор выборки кортежей таблицы  $Tab$  с условием выборки  $S$  (т. е. выбирается часть кортежей таблицы  $Tab$ , удовлетворяющих условию  $S$ ). До завершения транзакции  $T_1$  в момент времени  $t_2 > t_1$  транзакция  $T_2$  вставляет в таблицу  $Tab$  новый кортеж  $r$ , удовлетворяющий условию  $S$ , и успешно завершается. В момент времени  $t_3 > t_2$  транзакция  $T_1$  повторно выполняет тот же оператор выборки,

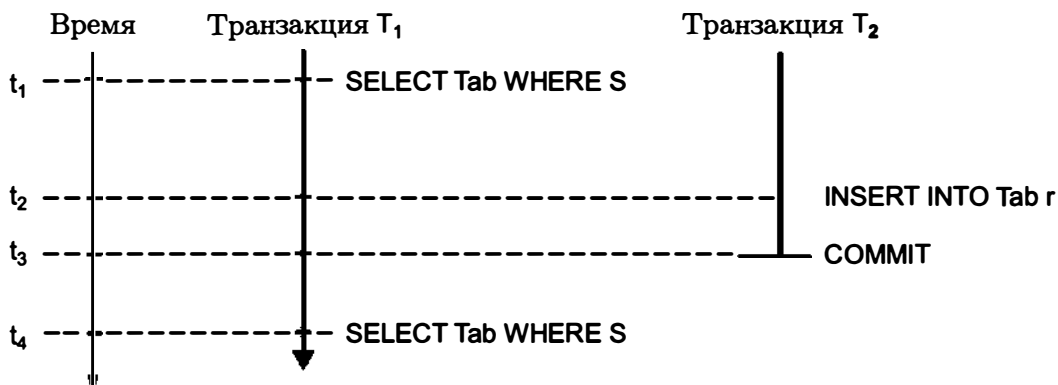


Рис. 9.4. Проблема фантомов

и в результате появляется кортеж, который отсутствовал при первом выполнении оператора.

Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

#### 9.1.4. Сериализация транзакций

Чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.

Пусть в системе одновременно выполняется некоторое множество транзакций  $S = \{T_1, T_2, \dots, T_n\}$ . План (способ) выполнения набора транзакций  $S$  (в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций) называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций ( $T_{i1}, T_{i2}, \dots, T_{in}$ ).

Сериализация транзакций — это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций, обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограни-

чивал бы чередование их операций или реальную параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением свойства сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями  $T_1$  и  $T_2$  могут существовать следующие виды конфликтов:

- W/W — транзакция  $T_2$  пытается изменять объект, измененный не закончившейся транзакцией  $T_1$  (наличие такого конфликта может привести к возникновению ситуации потерянных изменений);
- R/W — транзакция  $T_2$  пытается изменять объект, прочитанный не закончившейся транзакцией  $T_1$  (наличие такого конфликта может привести к возникновению ситуации неповторяющихся чтений);
- W/R — транзакция  $T_2$  пытается читать объект, измененный не закончившейся транзакцией  $T_1$  (наличие такого конфликта может привести к возникновению ситуации «грязного» чтения).

Практические методы сериализации транзакций основаны на учете этих конфликтов.

## 9.2. Методы сериализации транзакций

Существуют два базовых подхода к сериализации транзакций — основанный на синхронизационных захватах объектов базы данных и на использовании временных меток. Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении. Далее эти подходы будут рассмотрены более подробно. Кроме того, кратко обсудим возможности использования версий объектов базы данных для ускорения выполнения «только читающих» транзакций, т. е. транзакций, в которых не выполняются операции изменения базы данных.

Предварительно заметим, что для каждого из подходов имеются две разновидности — пессимистическая и оптимистическая. При применении пессимистических методов, ориентированных на ситуации, когда конфликты возникают часто, конфликты распознаются и разрешаются немедленно при их возникновении. Оптимистические методы основываются на том, что результаты

всех операций модификации базы данных сохраняются в рабочей памяти транзакций. Реальная модификация базы данных производится только на стадии фиксации транзакции. Тогда же проверяется, не возникают ли конфликты с другими транзакциями.

Далее мы ограничимся рассмотрением более распространенных пессимистических разновидностей методов сериализации транзакций. Пессимистические методы сравнительно просто трансформируются в свои оптимистические варианты.

### 9.2.1. Синхронизационные блокировки

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре «клиент-сервер») является подход, основанный на соблюдении двухфазного протокола синхронизационных захватов объектов баз данных (Two-Phase Locking Protocol, 2PL). В общих чертах подход состоит в том, что перед выполнением любой операции в транзакции  $T$  над объектом базы данных  $o$  от имени транзакции  $T$  запрашивается синхронизационная блокировка объекта  $o$  в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных блокировок являются следующие:

- совместный режим —  $S$  (Shared), означающий совместную (по чтению) блокировку объекта и требуемый для выполнения операции чтения объекта;
- монопольный режим —  $X$  (eXclusive), означающий монопольную (по записи) блокировку объекта и требуемый для выполнения операций вставки, удаления и модификации объекта.

Блокировки одних и тех же объектов по чтению несколькими транзакциями совместимы, т. е. нескольким транзакциям допускается одновременно читать один и тот же объект. Блокировка объекта одной транзакцией по чтению не совместима с блокировкой другой транзакцией того же объекта по записи, т. е. никакой транзакции нельзя изменять объект, читаемый некоторой транзакцией (кроме самой этой транзакции), и никакой транзакции нельзя читать объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Блокировки одного и того же объекта по записи разными транзакциями не совместимы, т. е. никакой транзакции нельзя изменять объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Правила совместимости захватов одного объекта разными транзакциями приведены в табл. 9.1.

В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. При этом знак «—» соответствует состоянию объекта, для которого не установлен никакой захват. Транзакция, запросившая синхронизационный захват объекта БД, уже захваченный другой транзакцией в несовместимом режиме, блокируется до тех пор, пока захват с этого объекта не будет снят. Заметим, что слово «нет» (отсутствие совместимости блокировок) в этой таблице соответствует описанным ранее возможным случаям конфликтов транзакций по доступу к объектам базы данных (W/W, R/W, W/R). Совместимость S-блокировок соответствует тому, что конфликт R/R не существует.

Таблица 9.1.  
Совместимость блокировок S и X

|   | X   | S   |
|---|-----|-----|
| — | да  | да  |
| X | нет | нет |
| S | нет | да  |

Для обеспечения сериализации транзакций (третьего уровня изолированности) синхронизационные блокировки объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении (см. примеры сценариев, обсуждавшихся в подразд. 9.1). Это требование порождает двухфазный протокол синхронизационных захватов — 2PL. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

- первая фаза транзакции (выполнение операций над базой данных) — накопление блокировок;
- вторая фаза (фиксация или откат) — снятие блокировок.

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных блокировок действительно обеспечивается сериализация транзакций на третьем уровне изолированности. Также можно видеть, что для обеспечения отсутствия потерянных данных достаточно блокировать в режиме X изменяемые объекты базы данных и удерживать эти блокировки до конца транзакции, а для обеспечения отсутствия чтения «грязных» данных достаточно блокировать в режиме X изменяемые объекты до конца транзакции и блокировать в режиме S читаемые объекты на время выполнения операции чтения.

Основная проблема состоит в том, что следует считать объектом для синхронизационного захвата? В контексте реляционных баз данных возможны следующие альтернативы:

- файл (сегмент в терминах System R) — физический (с точки зрения базы данных) объект, область хранения нескольких таблиц и, возможно, индексов;



- таблица — логический объект, соответствующий множеству кортежей данной таблицы;
- страница данных — физический объект, хранящий кортежи одной или нескольких таблиц, индексную или служебную информацию;
- кортеж — элементарный физический объект базы данных.

На самом деле, любая операция над объектом базы данных фактически воздействует и на объемлющие его объекты. Например, операция над кортежем является и операцией над страницей, в которой этот кортеж хранится, и над соответствующей таблицей, и над файлом, содержащим таблицу. Поэтому действительно имеется выбор уровня объекта блокировки.

Понятно, что для поддержки блокировок требуются системные ресурсы, и что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект — логический или физический), тем меньше синхронизационных блокировок будет поддерживаться в системе, и на это, соответственно, будут тратиться меньшие накладные расходы. Более того, если устанавливать блокировки на уровне файлов или таблиц, то будет решена даже проблема фантомов (если это не ясно сразу, посмотрите еще раз описание проблемы фантомов и определение двухфазного протокола захватов).

Но вся беда в том, что при использовании для блокировок крупных объектов возрастает вероятность конфликтов транзакций и, тем самым, уменьшается допускаемая степень чередования их операций или реального параллельного выполнения. Фактически, при укрупнении объекта синхронизационной блокировки мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, в которых на самом деле их нет.

Разработчики многих систем начинали с использования страничных блокировок, полагая это некоторым компромиссом между стремлениями сократить накладные расходы и сохранить достаточно высокий уровень параллельности транзакций. Но это не очень хороший выбор. Не будем останавливаться на деталях, но заметим, что использование страничных блокировок в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организацию СУБД (коротко говоря, эти проблемы связаны с тем, что страницы приходится блокировать на двух разных уровнях — уровне управления буферами страниц в основной памяти и уровне выполнения логических операций). В большинстве современных систем используются покортежные синхронизационные блокировки.

Но при этом возникает очередной вопрос. Если единицей блокировки является кортеж, то какие синхронизационные блокировки потребуются при выполнении таких операций как уничтожение заполненной таблицы? Было бы довольно нелепо перед выполнением такой операции потребовать блокировки всех существующих кортежей таблицы. Кроме того, это не предотвратило бы возможности параллельной вставки нового кортежа в уничтожаемое отношение в некоторой другой транзакции.

**Гранулированные синхронизационные блокировки.** Подобные рассуждения привели к разработке механизма гранулированных синхронизационных блокировок. При применении этого подхода синхронизационные блокировки могут запрашиваться по отношению к объектам разного уровня: файлам, таблицам и кортежам. Требуемый уровень объекта определяется тем, какая операция выполняется (например, для выполнения операции уничтожения таблицы объектом синхронизационной блокировки должна быть вся таблица, а для выполнения операции удаления кортежа — этот кортеж). Объект любого уровня может быть заблокирован в режиме S или X.

Для согласования блокировок разного уровня вводятся специальный протокол гранулированных блокировок и новые типы блокировок. Коротко говоря, перед установкой блокировки на некоторый объект базы данных в режиме S или X соответствующий объект верхнего уровня должен быть заблокирован в режиме IS, IX или SIX. Что же собой представляют эти режимы блокировок?

Блокировка в режиме IS (Intented for Shared lock) некоторого составного объекта *о* базы данных означает намерение заблокировать некоторый объект *о'*, входящий в *о*, в совместном режиме (режиме S). Например, при намерении читать кортежи из таблицы *Tab* эта таблица должна быть заблокирована в режиме IS (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица *Tab*).

Блокировка в режиме IX (Intented for eXclusive lock) некоторого составного объекта *о* базы данных означает намерение заблокировать некоторый объект *о'*, входящий в *о*, в монопольном режиме (режиме X). Например, для удаления кортежей из таблицы *Tab* эта таблица должна быть заблокирована в режиме IX (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица *Tab*).

Блокировка в режиме SIX (Shared, Intented for eXclusive lock) некоторого составного объекта *о* базы данных означает совместную блокировку всего этого объекта с намерением впо-

следствии блокировать какие-либо входящие в него объекты в монопольном режиме (режиме X). Например, если выполняется длинная операция просмотра таблицы Tab с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего заблокировать таблицу Tab в режиме SIX (а до этого заблокировать в режиме IS файл, в котором располагается таблица Tab).

В табл. 9.2 приведена схема совместимости блокировок S, X, IS, IX и SIX. Немного поясним правила совместимости. Должно быть понятно, что для атомарных объектов разумны только блокировки в режимах S и X, для которых правила совместимости остаются такими же, как были показаны в табл. 9.1. Пусть теперь o — это некоторый составной объект.

Тогда блокировка объекта o в режиме X в транзакции T<sub>1</sub> не совместима с блокировкой этого объекта в режимах X, S, IX, IS или SIX в транзакции T<sub>2</sub>. Действительно, блокировка объекта o в режиме X в транзакции T<sub>1</sub> направлена на то, чтобы изменить объект o целиком. Несовместимость блокировки объекта o в режиме X в транзакции T<sub>1</sub> с его блокировкой в режиме X или IX в транзакции T<sub>2</sub> устраняет конфликты транзакций T<sub>1</sub> и T<sub>2</sub> вида W/W. Несовместимость блокировки объекта o в режиме X в транзакции T<sub>1</sub> с его блокировкой в режиме S или IS в транзакции T<sub>2</sub> устраняет конфликты транзакций T<sub>1</sub> и T<sub>2</sub> вида W/R. Наконец, несовместимость блокировки объекта o в режиме X в транзакции T<sub>1</sub> с его блокировкой в режиме SIX в транзакции T<sub>2</sub> устраняет конфликты транзакций T<sub>1</sub> и T<sub>2</sub> вида W/R и W/W.

Блокировка объекта o в режиме S в транзакции T<sub>1</sub> совместима с блокировкой этого объекта в режимах S или IS в транзакции T<sub>2</sub>, поскольку эти блокировки в транзакциях T<sub>1</sub> и T<sub>2</sub> направлены только на то, чтобы только читать некоторые объекты o',

Таблица 9.2. Совместимость блокировок S, X, IS, IX и SIX

|     | X   | S   | IX  | IS  | SIX |
|-----|-----|-----|-----|-----|-----|
| —   | да  | да  | да  | да  | да  |
| X   | нет | нет | нет | нет | нет |
| S   | нет | да  | нет | да  | нет |
| IX  | нет | нет | да  | да  | нет |
| IS  | нет | да  | да  | да  | да  |
| SIX | нет | нет | нет | да  | нет |

входящие в  $o$ . Блокировка объекта  $o$  в режиме  $S$  в транзакции  $T_1$  не совместима с блокировкой этого объекта в режимах  $X$ ,  $IX$  или  $SIX$  в транзакции  $T_2$ , поскольку любая из этих блокировок направлена на то, чтобы изменять в транзакции  $T_2$  объект  $o$  целиком или какой-либо объект  $o'$ , входящий в  $o$ . Несовместимость блокировки объекта  $o$  в режиме  $S$  в транзакции  $T_1$  с блокировкой этого объекта в режимах  $X$ ,  $IX$  или  $SIX$  в транзакции  $T_2$ , тем самым, устраняет конфликты транзакций  $T_1$  и  $T_2$  вида  $R/W$ .

Блокировка объекта  $o$  в режиме  $IX$  в транзакции  $T_1$  совместима с блокировкой этого же объекта в режимах  $IS$  или  $IX$  в транзакции  $T_2$ . Действительно, блокировка объекта  $o$  в режиме  $IX$  в транзакции  $T_1$  направлена на то, чтобы в этой транзакции изменять какой-либо объект  $o'$ , входящий в  $o$ , а блокировка этого же объекта в режиме  $IS$  в транзакции  $T_2$  — на то, чтобы читать в транзакции  $T_2$  какой-либо объект  $o''$ , входящий в  $o$ . Если объекты  $o'$  и  $o''$  — разные, то конфликт транзакций  $T_1$  и  $T_2$  не возникнет. Если  $o' = o''$ , то перед изменением этот объект будет заблокирован в транзакции  $T_1$  в режиме  $X$ , а перед чтением — в транзакции  $T_2$  в режиме  $S$ . Несовместимость этих блокировок позволит избежать конфликта транзакций  $T_1$  и  $T_2$  вида  $W/R$ , и для этого не требуется несовместимость блокировок  $IX$  и  $IS$  объекта  $o$ . Аналогично обосновывается совместимость блокировок  $IX$  и  $IX$ . Блокировка  $IX$  не совместима с блокировкой  $S$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида  $W/R$ . Блокировка  $IX$  не совместима с блокировкой  $X$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида  $W/W$ . Наконец, блокировка  $IX$  не совместима с блокировкой  $SIX$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида  $W/R$  или  $W/W$ .

Блокировка объекта  $o$  в режиме  $IS$  в транзакции  $T_1$  совместима с блокировкой этого же объекта в режимах  $S$ ,  $IS$ ,  $IX$  или  $SIX$  в транзакции  $T_2$ . Совместимость с блокировкой в режиме  $S$  или  $IS$  уже обосновывалась. Покажем, что блокировка объекта  $o$  в режиме  $IS$  в транзакции  $T_1$  совместима с блокировкой того же объекта в режиме  $IX$  в транзакции  $T_2$ . Действительно, блокировка объекта  $o$  в режиме  $IS$  в транзакции  $T_1$  направлена на то, чтобы в этой транзакции читать какой-либо объект  $o'$ , входящий в  $o$ , а блокировка этого же объекта в режиме  $IX$  в транзакции  $T_2$  — на то, чтобы в транзакции  $T_2$  изменять какой-либо объект  $o''$ , входящий в  $o$ . Если объекты  $o'$  и  $o''$  — разные, то конфликт транзакций не возникнет. Если  $o' = o''$ , то перед чтением этот объект будет заблокирован в транзакции  $T_1$  в режиме  $S$ , а перед изменением — в транзакции  $T_2$  в режиме  $X$ . Не-

совместимость этих блокировок позволит избежать конфликта транзакций  $T_1$  и  $T_2$  вида R/W, и для этого не требуется несовместимость блокировок IS и IX объекта о. Аналогично можно показать совместимость блокировок IS и SIX. Несовместимость блокировок IS и X очевидна, поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида R/W.

Блокировка объекта о в режиме SIX в транзакции  $T_1$  позволяет этой транзакции читать любой объект о', входящий в о, без его дополнительной блокировки и изменять любой объект о', входящий в о, с его предварительной блокировкой в режиме X. Эта блокировка совместима с блокировкой объекта о в режиме IS в транзакции  $T_2$ . Действительно, блокировка объекта о в режиме IS в транзакции  $T_2$  направлена на то, чтобы в транзакции  $T_2$  читать какой-либо объект о', входящий в о. Перед этим в транзакции  $T_2$  должна быть установлена блокировка объекта о' в режиме S. К этому моменту у объекта о' может отсутствовать явная блокировка, установленная в транзакции  $T_1$ , что, в соответствии с семантикой блокировки SIX, означает наличие неявной блокировки о' по чтению. Очевидно, что в этом случае конфликт транзакций  $T_1$  и  $T_2$  не возникает. К этому же моменту у объекта о' может иметься блокировка в режиме X, установленная в транзакции  $T_1$ . В этом случае запрос блокировки объекта о' в режиме S удовлетворен не будет и конфликт транзакций  $T_1$  и  $T_2$  вида W/R будет предотвращен без потребности в несовместимости блокировок SIX и IS. Блокировка объекта о в режиме SIX в транзакции  $T_1$  не совместима с блокировкой объекта о в режиме X в транзакции  $T_2$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида R/W. Блокировка объекта о в режиме SIX в транзакции  $T_1$  не совместима с блокировкой объекта о в режиме S или IS в транзакции  $T_2$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида W/R при доступе к некоторым объектам о', входящим в о. Наконец, блокировка объекта о в режиме SIX в транзакции  $T_1$  не совместима с блокировкой объекта о в режиме IX или SIX в транзакции  $T_2$ , поскольку иначе мог бы проявиться конфликт транзакций  $T_1$  и  $T_2$  вида R/W при доступе к некоторым объектам о', входящим в о.

**Предикатные синхронизационные блокировки.** Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием блокировок таблиц в режимах S и X). Давно известно, что для решения этой проблемы необходимо перейти от блокировок индивидуальных («физических») объектов базы данных,

к блокировке условий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для блокировок уровня таблиц именно потому, что таблица как логический объект представляет собой неявное условие для входящих в него кортежей. Блокировка таблицы — это простой и частный случай предикатной блокировки.

Поскольку любая операция над реляционной базой данных задается некоторым условием (т.е. в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационную блокировку в режиме S или X именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных блокировок. Ясно, что без этого использовать предикатные блокировки для сериализации транзакций невозможно, а в общей форме проблема неразрешима.

Один из компромиссных подходов предлагался участниками проекта System R. Подход основывался на том, что при открытии сканирования таблицы по индексу в RSS передается дополнительная информация (диапазон сканирования), которая ограничивает множество кортежей, среди которых не должны возникать фантомы.

Опираясь на наличие этой информации, предлагалось ввести в систему блокировок System R элементы предикатных блокировок. Заметим сначала, что в System R блокировки сегментов (файлов), таблиц и кортежей технически трактовались единообразно, как блокировки идентификаторов кортежей (tid'ов). При блокировке кортежа на самом деле блокировался его tid; при блокировке сегмента или таблицы — tid описателя соответствующего объекта во внутренних таблицах-каталогах сегментов или таблиц.

Предлагалось расширить систему синхронизации, разрешив применять блокировки к паре «идентификатор индекса, интервал значений ключа этого индекса». К такой паре можно было применять блокировки в любом из допустимых режимов, причем две такие блокировки считались совместимыми в том и только в том случае, если они были совместимы в соответствии с приведенной табл. 9.2 или указанные диапазоны значений ключей не пересекались.

При наличии такой возможности, если открывается сканирование таблицы через индекс, то таблица блокируется в режиме

IS, и в этом же режиме блокируется пара «идентификатор индекса, диапазон сканирования». При занесении (удалении) кортежа таблица блокируется в режиме IX, и в этом же режиме для каждого индекса, определенного на данной таблице отношении, блокируется пара «идентификатор индекса, значение ключа из затрагиваемого операцией кортежа». Это позволяет избежать конфликтов читающих транзакций с теми изменяющими транзакциями, которые затрагивают диапазоны сканирования читающих транзакций. При этом решается проблема фантомов, и параллельность транзакций ограничивается «по существу», т.е. только в тех случаях, когда их параллельное выполнение создает проблемы.

Заметим сразу, что описанное решение проблемы фантомов далеко от идеального. Во-первых, по-прежнему при сканировании таблиц без использования индексов отсутствие фантомов можно гарантировать только при блокировке всего отношения в режиме S. Во-вторых, даже при сканировании по индексу условие реальной выборки кортежа часто может быть гораздо строже простого указания диапазона сканирования, а это значит, что блокировка этого диапазона будет слишком сильной, т.е. затронет более широкое множество кортежей, чем то, которое будет реальным результатом сканирования.

Известно следующее более совершенное решение. Будем называть простым условием конъюнкцию простых предикатов сравнения, имеющих вид `имя_поля { = > < } значение`. В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистемы управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию оператора SQL со сложным условием в последовательность обращений к подсистеме управления памятью, в каждом из которых содержатся только простые условия.

Более точно, простое условие явно указывается в операции открытия сканирования таблицы (напрямую или через индекс; в последнем случае оно конъюнктивно соединяется с условием, задаваемым диапазоном сканирования). Кроме того, при открытии сканирования всегда можно указать, для какой цели оно будет использоваться: для выборки кортежей, их удаления или обновления (это известно компилятору SQL). Отметим также, что неявные условия задаются операциями вставки и удаления кортежей (конъюнктивное логическое выражение, состоящее из простых предикатов вида `имя_поля = значение` для всех полей

таблицы) и операциями обновления кортежей (комбинированное логическое выражение, состоящее из простых предикатов вида  $\text{имя\_поля} = \text{значение}$  для всех обновляемых полей таблицы). Поэтому в случае типовой организации SQL-ориентированной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных блокировок легко определяется на основе следующей геометрической интерпретации. Пусть  $\text{Tab}$  — таблица с полями  $a_1, a_2, \dots, a_n$ , а  $m_1, m_2, \dots, m_n$  — множества допустимых значений  $a_1, a_2, \dots, a_n$  соответственно (естественно, все эти множества — конечны). Тогда можно сопоставить  $\text{Tab}$  конечное  $n$ -мерное пространство возможных значений кортежей  $\text{Tab}$ . Легко видеть, что любое простое условие, представляющее собой конъюнкцию простых предикатов, «вырезает» в этом пространстве  $m$ -мерный прямоугольник ( $m \leq n$ ).

Достаточно очевидно следующее утверждение:

пусть имеются два простых условия  $\text{scond}_1$  и  $\text{scond}_2$ . Пусть транзакция  $T_1$  запрашивает блокировку  $\text{scond}_1$ , а транзакция  $T_2$  —  $\text{scond}_2$  в режимах, которые были бы несовместимы, если бы  $\text{scond}_1$  и  $\text{scond}_2$  являлись не условиями, а объектами базы данных (S-X, X-S, X-X). Эти блокировки совместимы в том и только том случае, когда прямоугольники, соответствующие  $\text{scond}_1$  и  $\text{scond}_2$ , не пересекаются.

Это утверждение действительно очевидно (каждому  $m$ -мерному прямоугольнику в  $n$ -мерном пространстве возможных значений кортежей  $\text{Tab}$  соответствует некоторое подмножество возможных значений кортежей, и отсутствие пересечения у двух прямоугольников гарантирует отсутствие конфликтов транзакций), но для наглядности на рис. 9.5 приводится иллюстрирующий пример, показывающий, что в каких бы режимах не требовала транзакция  $T_1$  блокировки условия  $(0 < a < 5) \& (b = 5)$ , а транзакция  $T_2$  — блокировки условия  $(0 < a < 6) \& (0 < b < 4)$ , эти блокировки всегда будут совместимы.

Интересно, что при поддержке такой системы блокировок простых условий можно обойтись без гранулированных блокировок. В частности, чтобы гарантированно заблокировать

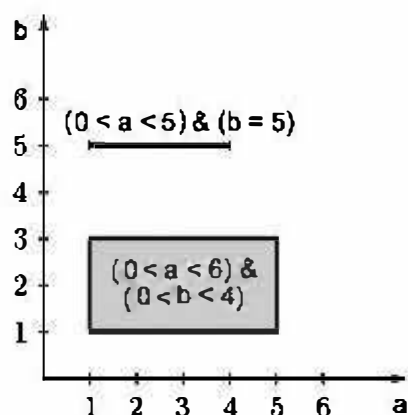


Рис. 9.5. Простые условия, блокировки которых совместимы



таблицу целиком, достаточно заблокировать условие  $\&_{1 \leq i \leq n} (\min(m_i) < \text{имя\_поля}_i < \min(m_i))$ . Чтобы заблокировать базу данных, достаточно заблокировать условие, являющееся конъюнкцией условий блокировки всех таблиц этой базы данных.

Заметим, что блокировки простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов с гранулированными блокировками. Поэтому введение в СУБД механизма предикатных блокировок не приводит к значительным усложнениям.

### 9.2.2. Синхронизационные тупики, их распознавание и разрушение

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных блокировок является возможность возникновения тупиков (deadlocks) между транзакциями. Синхронизационные тупики возможны при применении любого из рассмотренных ранее вариантов механизмов блокировок.

На рис. 9.6 показан простой сценарий возникновения синхронизационного тупика между транзакциями  $T_1$  и  $T_2$ :

- транзакции  $T_1$  и  $T_2$  устанавливают монопольные блокировки объектов  $o_1$  и  $o_2$  соответственно;
- после этого  $T_1$  требуется совместная блокировка объекта  $o_2$ , а  $T_2$  — совместная блокировка объекта  $o_1$ ;
- ни одно из этих требований блокировки не может быть удовлетворено, следовательно, ни одна из транзакций не может продолжаться; поэтому монопольные блокировки объектов никогда не будут сняты, а требования совместных блокировок не будут удовлетворены.

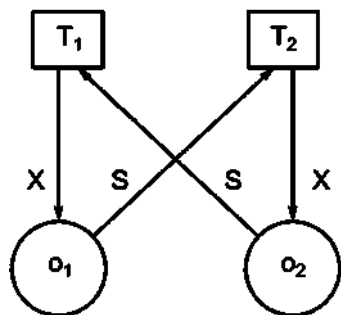


Рис. 9.6. Ситуация синхронизационного тупика между транзакциями  $T_1$  и  $T_2$

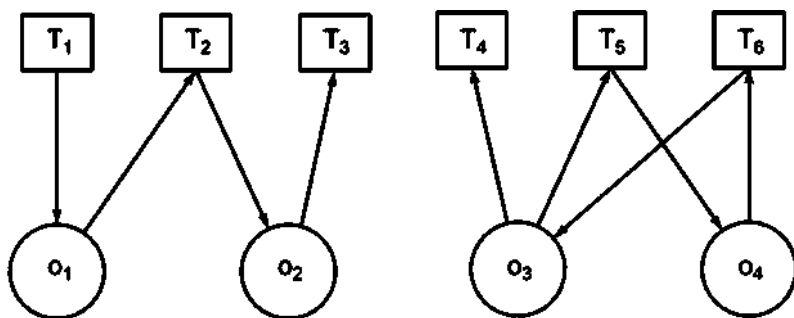
Поскольку тупики возможны и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

**Обнаружение тупиковых ситуаций.** Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций — это ориентированный двудольный граф, в котором существует два типа вершин — вершины, соответствующие

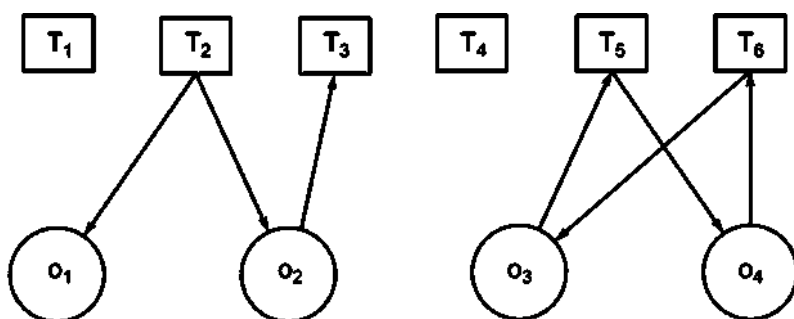
транзакциям (будем изображать их прямоугольниками), и вершины, соответствующие объектам блокировок (будем изображать их окружностями). В этом графе дуги соединяют только вершины-транзакции с вершинами-объектами. Дуга из вершины-транзакции к вершине-объекту существует в том и только том случае, если для этой транзакции имеется удовлетворенная блокировка данного объекта. Дуга из вершины-объекта к вершине-транзакции существует тогда и только тогда, когда эта транзакция ожидает удовлетворения запроса блокировки данного объекта. Легко показать, что в системе существует тупиковая ситуация в том и только том случае, когда в графе ожидания транзакций имеется хотя бы один цикл. Простейший пример графа ожидания транзакций с циклом показан на рис. 9.6.

Для распознавания тупиковых ситуаций периодически производится построение графа ожидания транзакций (как уже отмечалось, иногда граф ожидания поддерживается постоянно) и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения циклов в ориентированном графе является редукция графа.

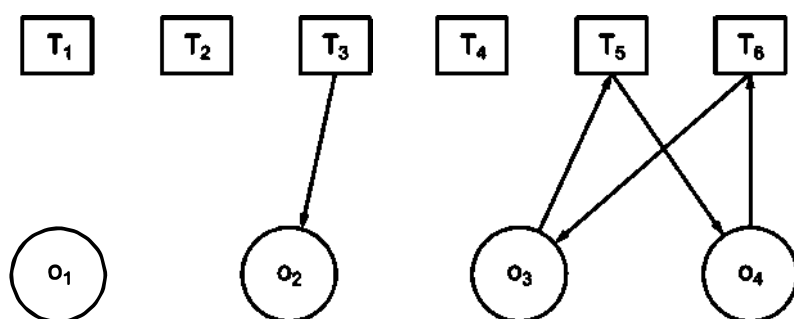
Пример применения алгоритма редукции к графу ожидания транзакций показан на рис. 9.7 (в целях упрощения примера предполагается, что все блокировки являются монопольными, т.е. для каждой вершины-объекта имеется не более одной входящей дуги). В этом случае редукция состоит в том, что, прежде всего, из графа ожидания (начальное состояние которого показано на рис. 9.7, а) удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это основывается на том разумном предположении, что транзакции, не ожидающие удовлетворения запроса блокировок, могут успешно завершиться и освободить блокировки). Кроме того, удаляются дуги, входящие в вершины-транзакции, из которых не исходят ведущие к вершинам-объектам (транзакции, ожидающие удовлетворения блокировок, но не удерживающие заблокированные объекты, не могут быть причиной тупика). Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация одной из исходящих дуг (выбираемой произвольным образом) изменяется на противоположную (это моделирует удовлетворение запроса блокировки). Состояние графа после выполнения первого шага редукции показано на рис. 9.7, б. После этого снова повторяются описанные действия (состояние графа после выполнения второго шага редукции показано на рис. 9.7, в), и так до тех пор, пока не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл (см. рис. 9.7, в).



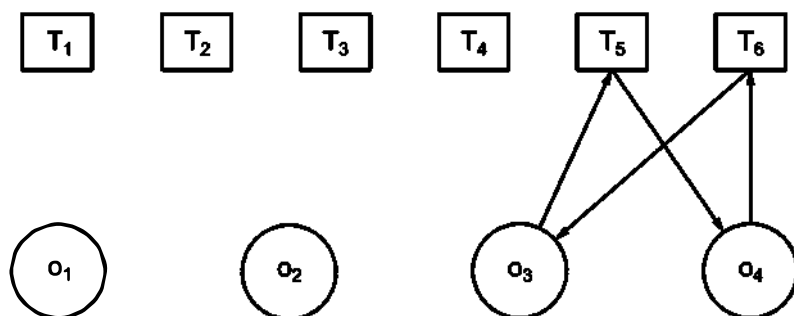
*a*



*б*



*в*



*г*

Рис. 9.7. Применение алгоритма редукции к графу ожидания транзакций:

*a* — исходное состояние графа ожидания; *б* — первый шаг редукции; *в* — второй шаг редукции; *г* — конечный вид графа ожидания

Предположим теперь, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь?

**Разрушение тупиков.** Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, т.е. транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Выбрать «жертву» не так уж легко, поскольку для этого могут использоваться различные, зачастую противоречивые критерии. С одной стороны, было бы разумно жертвовать наиболее «богатой» транзакцией, т.е. той транзакцией, которая удерживает наибольшее число блокировок объектов. В этом случае после принудительно завершения такой транзакции освободилось бы наибольшее число объектов, что с большой вероятностью привело бы к исчезновению тупиковой ситуации. Но, с другой стороны, «богатая» транзакция, скорее всего, выполнялась дольше других транзакций. На ее выполнение уже затрачено большое количество системных ресурсов и, вероятно, она скоро завершится самостоятельно. Поэтому этот выбор может оказаться в системном отношении не самым удачным.

Можно пожертвовать самой «молодой» транзакцией, которая существует в системе в течение наименьшего времени. Такую транзакцию менее всего жалко, поскольку она еще не успела израсходовать много системных ресурсов. Но, с другой стороны, такая транзакция не могла и накопить много блокировок, и поэтому ее насильственное завершение вряд ли поможет устранить тупиковую ситуацию. Так стоит ли ею жертвовать?

Можно выбрать транзакцию-жертву случайным образом из всех транзакций, попавших в тупик. Возможно, что в среднем этот подход привел бы к хорошим результатам. Но, к сожалению, в нем не учитывается возможная приоритетность транзакций. Было бы не слишком хорошо, например, жертвовать транзакцией, запущенной от имени руководителя организации.

Поэтому обычно при выборе транзакции-жертвы используется многофакторная оценка ее стоимости, в которую с разными весами входят время выполнения, число накопленных блокировок, приоритет и т.д. В качестве «жертвы» выбирают транзакцию, для которой эта оценка выдает наиболее подходящий результат.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный (до некоторой точки сохранения) характер. При этом, есте-

ственно, освобождаются блокировки, и может быть продолжено выполнение других транзакций.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать.

Заметим, что в централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

### 9.2.3. Метод временных меток

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редкого возникновения конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании временных меток. Основная идея метода временных меток (Timestamp Ordering, TO), у которого существует множество разновидностей, состоит в следующем: если транзакция  $T_1$  началась раньше транзакции  $T_2$ , то система обеспечивает такой сериальный план, как если бы транзакция  $T_1$  была целиком выполнена до начала  $T_2$ .

Для этого каждой транзакции  $T$  предписывается временная метка  $t(T)$ , соответствующая времени начала выполнения транзакции  $T$ . При выполнении операции над объектом  $o$  транзакция  $T$  помечает его своими идентификатором, временной меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом  $o$  транзакция  $T_1$  осуществляет следующие действия:

- проверяет, помечен ли объект  $o$  какой-либо транзакцией  $T$ . Если не помечен, то помечает этот объект своей временной меткой и типом операции и выполняет операцию. Конец действий;
- иначе транзакция  $T_2$  проверяет, не завершилась ли транзакция  $T_1$ , пометившая этот объект. Если транзакция  $T_1$  закончилась, то  $T_2$  помечает объект  $o$  и выполняет свою операцию. Конец действий;
- если транзакция  $T_1$  не завершилась, то  $T_2$  проверяет конфликтность операций. Если операции неконфликтны, то при объекте  $o$  запоминается идентификатор транзакции  $T_1$ , остается или проставляется временная метка с меньшим значением, и транзакция  $T_2$  выполняет свою операцию;

- если операции транзакций  $T_2$  и  $T_1$  конфликтуют, то если  $t(T_1) > t(T_2)$  (т. е. транзакция  $T_1$  является более «молодой», чем  $T_2$ ), то производится откат  $T_1$  и всех других транзакций, идентификаторы которых сохранены при объекте  $o$ , и  $T_2$  выполняет свою операцию;
- если же  $t(T_1) < t(T_2)$  ( $T_1$  «старше»  $T_2$ ), то производится откат  $T_2$ ;  $T_2$  получает новую временную метку и начинается заново.

К недостаткам метода ТО относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временные метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.

#### **9.2.4. Методы сериализации транзакций на основе поддержки версий объектов базы данных**

Основная идея алгоритмов сериализации транзакций, описываемых в этом подразделе, состоит в том, что в базе данных допускается существование нескольких «версий» одного и того же объекта. Эти алгоритмы, главным образом, направлены на преодоление конфликтов транзакций категорий R/W и W/R, позволяя выполнять операции чтения над некоторой предыдущей версией объекта базы данных. В результате операции чтения выполняются без задержек и тупиков, свойственных механизмам синхронизационных блокировок, а также без некоторых откатов, возможных при применении метода временных меток, описанного в подразд. 9.2.3.

Алгоритмы управления транзакциями, основанные на поддержке версий, достаточно широко распространены в области SQL-ориентированных СУБД. В частности, подобные алгоритмы используются в СУБД Oracle и PostgreSQL. В дальнейшем в этом подразделе будем называть алгоритмы этой категории *версионными* алгоритмами.

**Версионный вариант алгоритма временных меток.** Одним из наиболее старых и простых версионных алгоритмов является версионный вариант алгоритма временных меток

(Multiversion Timestamp Ordering, MVTO). Как и в простом методе временных меток, описанном в предыдущем подразделе, в алгоритме MVTO порядок выполнения операций одновременно выполняемых транзакций задается порядком временных меток, которые получают транзакции во время старта. Временные метки также используются для идентификации версий данных при чтении и модификации — каждая версия получает временную метку той транзакции, которая ее записала. Алгоритм MVTO не только следит за порядком выполнения операций транзакций, но также отвечает за трансформацию операций над объектами базы данных в операции над версиями этих объектов, т.е. каждая операция над объектом базы данных  $o$  преобразуется в соответствующую операцию над некоторой версией объекта  $o$ .

При описании алгоритма будем использовать следующие обозначения. Как и раньше, временную метку, полученную транзакцией  $T_i$  в начале ее работы, будем обозначать как  $t(T_i)$ . Операция чтения объекта базы данных  $o$ , выполняемая в транзакции  $T_i$ , будет обозначаться как  $R_i(o)$ . Для обозначения того, что транзакция  $T_i$  читает версию объекта базы данных  $o$ , созданную транзакцией  $T_k$ , будем использовать запись  $R_i(o_k)$ . Для обозначения того, что транзакция  $T_i$  записывает версию элемента данных  $o$ , будем использовать запись  $W_i(o_i)$ .

Алгоритм MVTO работает следующим образом.

- Любая операция  $R_i(o)$  преобразуется в операцию  $R_i(o_k)$ , где  $o_k$  — это версия объекта  $o$ , помеченная наибольшей временной меткой  $t(T_k)$ , такой что  $t(T_k) \leq t(T_i)$ . Другими словами, транзакции  $T_i$  для чтения дается версия объекта  $o$ , созданная транзакцией  $T_k$ , которая не моложе  $T_i$ , но старше любой другой транзакции  $T_n$ , создававшей свою версию объекта  $o$ .
- При обработке операции  $W_i(o)$  выполняются следующие действия:
  - если к этому времени некоторой незафиксированной транзакцией  $T_n$  уже выполнена некоторая операция  $R_n(o_k)$ , такая что  $t(T_k) \leq t(T_i) < t(T_n)$ , то операция  $W_i(o)$  не выполняется, а транзакция  $T_i$  откатывается;
  - в противном случае  $W_i(o)$  преобразуется в  $W_i(o_i)$ , т.е. образуется еще одна версия объекта  $o$ .
- При откате любой транзакции уничтожаются все созданные ею версии объектов базы данных и откатываются все транзакции, прочитавшие хотя бы одну из этих версий. Тем самым, откаты транзакций могут быть «каскадными».
- Выполнение операции фиксации транзакции  $T_i$  (COMMIT) откладывается до того момента, когда завершатся все

транзакции, записавшие версии данных, прочитанные  $T_i$ . Легко видеть, что без соблюдения этого требования не соблюдалось бы свойство долговечности (durability) транзакций, поскольку при откате некоторых транзакций потребовалось бы откатывать и ранее зафиксированные транзакции.

Преимущества алгоритма MVTO лучше всего иллюстрируются поведением транзакций  $T_1$  и  $T_2$ . При использовании блокировок между ними возник бы синхронизационный тупик, а при использовании обычного метода временных меток одна из транзакций подверглась бы откату. Однако при применении версий такие неприятности не возникают из-за того, что первая транзакция читает «старые» версии объектов  $o$  и  $\omega$ .

Транзакция  $T_3$  ожидает фиксации транзакции  $T_2$  перед своим собственным завершением (на рис. 9.8 это показано штриховой линией). Это происходит потому, что транзакция  $T_3$  прочитала версию  $o_2$  объекта  $o$ , образованную еще не зафиксированной транзакцией.

Транзакция  $T_4$  пытается создать версию  $\omega_4$  объекта  $\omega$  после того, как еще не зафиксированная транзакция  $T_5$  (начавшая-

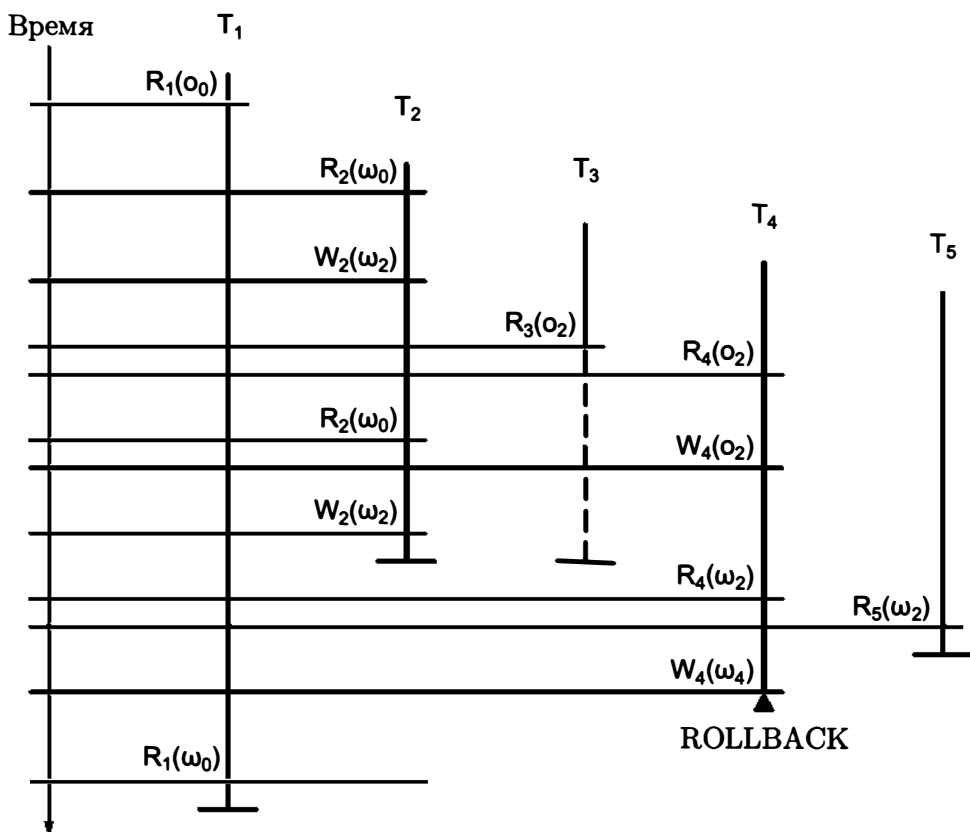


Рис. 9.8. Пример работы алгоритма MVTO



ся позже) уже прочитала более раннюю версию  $\omega_4$ . Поэтому транзакция  $T_5$  не сможет «увидеть» изменения объекта  $\omega$ , произведенные транзакцией  $T_4$ . Следовательно, сериализация транзакций в порядке получения ими временных меток становится невозможной и приходится произвести откат транзакции  $T_4$ .

Итак, основными преимуществами алгоритма MVTO является отсутствие задержек и откатов при выполнении операций чтения, а основным недостатком — возможность возникновения каскадных откатов транзакций при выполнении операций записи. Кроме того, в базе данных может накапливаться произвольное число версий одного и того же объекта, и определение того, какие версии больше не требуются, является серьезной технической проблемой.

**Версионный вариант двухфазного протокола синхронизационных блокировок.** При описании двухверсионного варианта протокола 2PL (Two-Version Two-Phase Locking Protocol, 2V2PL) будем называть *текущими* версиями объектов базы данных версии, созданные зафиксированными транзакциями с наиболее поздним временем фиксации; *незафиксированными* версиями — версии, созданные еще незавершившимися транзакциями. При следовании протоколу 2V2PL в каждый момент времени существует не более одной незафиксированной версии каждого объекта базы данных.

Операции любой транзакции  $T_i$  над объектом базы данных  $o$  обрабатываются следующим образом:

- операция  $R_i(o)$  немедленно выполняется над текущей версией объекта  $o$ ;
- операция  $W_i(o)$ , приводящая к созданию новой версии объекта  $o$ , выполняется только после завершения (фиксации или отката) транзакции, создавшей незафиксированную версию объекта  $o$ ;
- выполнение операции COMMIT откладывается до тех пор, пока не завершатся все транзакции  $T_k$ , прочитавшие текущие версии объектов базы данных, которые должны замениться незафиксированными версиями этих объектов, созданными транзакцией  $T_i$ .

Для реализации такого поведения используются три типа блокировок:

- RL (Read Lock) — в этом режиме блокируется любой объект базы данных  $o$  перед выполнением операции чтения его текущей версии; удержание этой блокировки до конца транзакции гарантирует, что при повторном чтении объекта  $o$  будет прочитана та же версия этого объекта;

- **WL (Write Lock)** — в этом режиме блокируется любой объект базы данных **о** перед выполнением операции, приводящей к созданию новой (незафиксированной) версии этого объекта; удержание этой блокировки до конца транзакции гарантирует, что в любой момент времени будет существовать не более одной незафиксированной версии любого объекта базы данных;
- **CL (Commit Lock)** — блокировка устанавливается во время выполнения операции **COMMIT** транзакции и затрагивает любой объект базы данных, новую версию которого создала данная транзакция; удовлетворение этой блокировки для данной транзакции гарантирует, что завершились все транзакции, читавшие текущие версии объектов, новые версии которых были созданы при выполнении данной транзакции, и, следовательно, их можно заменить.

В табл. 9.3 показаны правила совместимости этих блокировок.

Как видно, операция чтения может блокироваться только на время фиксации транзакции, заменяющей текущую версию требуемого объекта базы данных. Для выполнения операции записи требуется долговременная монополярная блокировка соответствующего объекта базы данных, которая, однако, в этом случае совместима с блокировкой этого же объекта по чтению (поскольку в действительности блокируются разные версии этого объекта). И, конечно, как и во всех схемах сериализации транзакций на основе блокировок, здесь возможны синхронизационные тупики.

**Версионно-блокировочный протокол сериализации транзакций для поддержки только читающих транзакций.** В заключение обсудим гибридный протокол, поддерживающий эффективное выполнение транзакций, не изменяющих состояние базы данных (Multiversion Protocol for Read-Only Transactions, ROMV). При применении этого протокола при об-

**Таблица 9.3. Таблица совместимости «версионных» блокировок**

|       | RL(o) | WL(o) | CL(o) |
|-------|-------|-------|-------|
| RL(o) | да    | да    | нет   |
| WL(o) | да    | нет   | нет   |
| CL(o) | нет   | нет   | нет   |

ровании каждой транзакции явно указывается ее тип — только читающая (read-only) или изменяющая (update) транзакция. В только читающих транзакциях допускается использование только операций чтения объектов базы данных, а в изменяющих транзакциях — операций и чтения, и записи.

Изменяющие транзакции выполняются в соответствии с обычным протоколом 2PL, т.е. перед выполнением операции чтения или записи объекта базы данных о этот объект должен быть заблокирован в режиме S или X соответственно, и блокировки объектов удерживаются до конца изменяющей транзакции. Каждая операция записи объекта о создает его новую версию, которая при завершении транзакции помечается временной меткой, соответствующей моменту фиксации этой транзакции.

Каждая только читающая транзакция при своем образовании получает соответствующую временную метку. При выполнении операции чтения объекта базы данных о транзакция получает доступ к версии объекта о, образованной изменяющей транзакцией, которая хронологически последней зафиксировалась к моменту образования данной читающей транзакции.

Основным плюсом протокола ROMV по сравнению с ранее описанным протоколом 2V2PL является принципиальное отсутствие синхронизационных задержек при выполнении операций чтения только читающих транзакций. Если сравнивать ROMV с MVTO, то он выигрывает в принципиальном отсутствии откатов только читающих транзакций. Конечно, при работе изменяющих транзакций возможно возникновение синхронизационных тупиков и откатов, и здесь требуется использовать обычные методы распознавания и разрушения тупиков.

Кроме того, при использовании протокола ROMV в базе данных может возникать произвольное число версий объектов. Требуется создание специального сборщика мусора, который должен удалять ненужные версии данных. Простейший сборщик мусора удаляет все неиспользуемые версии, значения временных меток которых меньше значения временной метки старейшей активной только читающей транзакции.

\* \* \*

В этой главе рассмотрены основные принципы управления транзакциями в системах управления базами данных, различные методы, алгоритмы и протоколы, способствующие достижению целей управления транзакциями. Следует заметить, что суще-

ствуется достаточно развитая теория управления транзакциями с собственными средствами формализации постановки задач и доказательства корректности алгоритмов. Для обеспечения более простого понимания сути материала в него не включены все эти формализмы.

В главе описаны два основных подхода к сериализации транзакций — на основе синхронизационных блокировок и временных меток. У каждого из этих подходов имеются свои достоинства и недостатки, но на практике существенно больше распространен метод синхронизационных блокировок. В заключение главы были рассмотрены расширения этих подходов с применением версий объектов базы данных. Соответствующие алгоритмы и протоколы позволяют уменьшить число потенциальных конфликтов транзакций, но для их поддержки требуются дополнительные расходы внешней памяти и усложнение общей архитектуры СУБД.

# СРЕДСТВА ЖУРНАЛИЗАЦИИ И ВОССТАНОВЛЕНИЯ БАЗ ДАННЫХ

Одним из основных требований к развитым СУБД является надежность хранения баз данных. Это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

- результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных (т. е. должно поддерживаться свойство *долговечности* (*durability*) транзакций);
- результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных (в противном случае состояние базы данных могло бы оказаться не целостным).

Это, собственно, и означает, что восстанавливается последнее по времени согласованное состояние базы данных.

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных:

- Индивидуальный откат транзакции. Тривиальной ситуацией отката транзакции является ее явное завершение оператором **ROLLBACK**. Возможны также ситуации, когда откат транзакции инициируется системой. Примерами могут быть возникновение исключительной ситуации в прикладной программе (например, деление на ноль) или

выбор транзакции в качестве жертвы при разрушении синхронизационного тупика. Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации базы данных, которые выполнялись в этой транзакции.

- Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть при аварийном выключении электрического питания, при возникновении неустранимого сбоя процессора (например, срабатывании контроля основной памяти) и т. д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти СУБД.
- Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но, тем не менее, СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и журнал изменений базы данных.

Во всех трех случаях основой восстановления является хранение избыточных данных. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении базы данных.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный журнал изменений базы данных этой транзакцией. Эти локальные журналы используются для индивидуальных откатов транзакций и могут поддерживаться в основной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

Данный подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант — поддержка только общего журнала изменений базы данных, который используется и при выполнении индивидуальных откатов. Здесь мы рассматриваем именно этот вариант.

В этой главе сначала проанализируем особенности подсистемы СУБД, управляющей буферами основной памяти, и связь механизмов буферизации и журнализации. Затем

на содержательном уровне без технических деталей обсудим общие принципы журнализации изменений и восстановления целостного состояния базы данных после сбоев, опираясь, в основном на методы, применявшиеся в System R и ее ранних предшественниках.

## **10.1. Буферизация блоков базы данных в основной памяти и ее связь с журнализацией**

Журнализация операций изменения базы данных\* тесно связана не только с управлением транзакциями, но и с буферизацией блоков базы данных в основной памяти. По причинам объективно существующей разницы в скорости работы процессоров и основной памяти и устройств внешней памяти (эта разница в скорости существовала, существует, и будет существовать всегда) буферизация блоков базы данных в основной памяти является единственным реальным способом достижения приемлемой эффективности СУБД. Без поддержки буферизации базы данных СУБД работала бы со скоростью магнитных дисков, т.е. на несколько порядков медленнее, чем если бы обработка данных происходила в основной памяти.

Если бы каждая запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции обновления базы данных, реально немедленно перемещалась бы во внешнюю память, это привело бы к существенному замедлению работы системы. Фактически, тогда каждая операция обновления базы данных выполнялась бы со скоростью магнитного диска. Поэтому записи в журнал тоже буферизуются: при нормальной работе буфер выталкивается во внешнюю память журнала только при полном заполнении записями. Более точно, для буферизации записей журнала обычно используются два буфера. После полного заполнения первый буфер выталкивается на магнитный диск, и пока совершается этот обмен, журнальные записи размещаются во втором буфере. К моменту конца обмена заполняется второй буфер, он выталкивается во внешнюю память, а журнальные записи снова размещаются в первом буфере и т.д.

Следует заметить, что здесь идет речь об использовании буферов (и базы данных, и журнала), располагающихся именно

---

\* Следует подчеркнуть, что здесь речь идет о логических операциях низкого уровня, т.е. уровня RSS, а не SQL.

в физической основной памяти, управляемой непосредственно СУБД, а не в виртуальной памяти СУБД, управляемой операционной системой. Использование буферов виртуальной памяти является практически бессмысленным делом, поскольку в этом случае операционная система, руководствуясь своими собственными стратегиями управления основной памяти, в любой момент может удалить буферную страницу СУБД из основной памяти и перенести ее копию во внешнюю память в область свопинга. Тогда при следующей попытке записи СУБД в эту страницу возникнет прерывание, при обработке которого операционная система подкачает страницу в основную память, выполнив совершенно не ожидаемый СУБД обмен с внешней памятью.

Нельзя надеяться на то, что операционная система настолько грамотно управляет основной памятью, что нужные страницы виртуальной памяти СУБД в нужное время будут находиться в основной памяти. Операционная система просто не обладает достаточной информацией, чтобы всегда принимать правильные решения. Правильно управлять своей буферной памятью может только сама СУБД, «отбирающая» у операционной системы часть физической основной памяти для размещения в ней буферов базы данных и журнала.

### **10.1.1. Управление буферным пулом базы данных**

В развитых (вернее сказать, правильно организованных) СУБД поддерживается собственная стратегия замещения страниц буферного пула. Задача, которую решает СУБД, очень похожа на задачу, которую решает операционная система при управлении виртуальной памятью.

В случае операционной системы, если некоторый процесс требует обеспечения доступа к странице виртуальной памяти, отсутствующей в основной памяти, и нет свободных страниц основной памяти, в соответствии с некоторым критерием выбирается некоторая занятая страница основной памяти, освобождается (т. е. изымается из виртуальной памяти какого-то процесса и, может быть, копируется на диск) и подключается к виртуальной памяти запросившего процесса с предварительным считыванием с диска нужных данных.

В случае СУБД, если при выполнении некоторой операции в некоторой транзакции требуется доступ к некоторому блоку базы данных и копия этого блока отсутствует в буферном пуле, СУБД должна выделить какую-либо страницу буферного пула, считать в нее с диска требуемый блок базы данных и предоста-



вить доступ к этой странице запросившей операции. Конечно, в буферном пуле может не оказаться свободных страниц, и тогда СУБД в соответствии с некоторым критерием находит некоторую занятую страницу, освобождает ее (возможно, выталкивает во внешнюю память).

Основная разница между этими случаями состоит в критерии выборки занятой страницы для «откачки». Не будем обсуждать здесь стратегии замещения страниц, используемые в операционных системах. Заметим лишь, что почти всегда операционная система стремится заменить страницу, к которой предположительно дольше всего не будет обращений, но, поскольку предвидение будущего невозможно, оно аппроксимируется прошлым. В частности, в одном из популярных алгоритмов замещения страниц LRU (Least Recently Used) принимается предположение, что дольше всего в будущем не потребуется та страница, к которой дольше всего не обращались в прошлом.

В стратегии замещения страниц буферного пула СУБД тоже чаще всего используется некоторая разновидность алгоритма LRU. Но, как уже отмечалось ранее, СУБД располагает большей информацией о страницах буферного пула, чем операционная система о страницах основной памяти.

Например, если в некоторой транзакции выполняется сканирование некоторой таблицы без использования индекса и при выполнении операции NEXT был затребован доступ к некоторому блоку базы данных (с соответствующим перемещением копии этого блока в некоторую страницу буферного пула), то подсистема управления буферным пулом «знает», что эта страница еще точно потребуется до тех пор, пока не будет прочитан последний кортеж сканируемой таблицы, располагающийся в данной странице. Более того, СУБД «знает», какой блок базы данных потребуется после завершения просмотра кортежей данного блока, и может заранее переместить его копию в некоторую страницу буферного пула.

Кроме того, некоторые блоки базы данных заведомо требуются чаще других блоков. Например, при любом просмотре таблицы на основе некоторого индекса гарантированно потребуется доступ к корневому блоку соответствующего В-дерева. При вставке кортежа в любую таблицу или удалении из нее кортежа будет необходимо должным образом изменить все определенные для нее индексы, и для этого тоже гарантированно потребуется доступ к корневым блокам всех соответствующих В-деревьев.

Поэтому в стратегии замещения страниц буферного пула базы данных обычно используется алгоритм LRU с приоритетами

страниц (грубо говоря, высокоприоритетные страницы стареют, т. е. становятся кандидатами на замещение, медленнее, чем низкоприоритетные страницы). В частности, страницы, содержащие копии корневых блоков индексов, являются настолько высокоприоритетными, что обычно никогда не замещаются. Кроме того, поддерживается предварительное считывание в буферную память копий блоков, доступ к которым вскоре понадобится.

### 10.1.2. Физическая синхронизация

Поскольку в СУБД может одновременно («параллельно») выполняться несколько транзакций, вполне реальна ситуация, когда в двух одновременно выполняемых операциях требуется доступ к одному и тому же блоку базы данных (т. е. к одной и той же буферной странице, содержащей копию этого блока). Понятно, что в одновременном доступе для чтения содержимого блока ничего плохого нет, но параллельное изменение блока может привести к непредсказуемым результатам.

Следует заметить, что, вообще говоря, координацию параллельного доступа к страницам буферного пула не обеспечивает логическая синхронизация, используемая для сериализации транзакций (см. гл. 9). Например, предположим, что в двух параллельно выполняемых транзакциях одновременно выполняются операции модификации кортежей, у одного из которых  $tid = (n, 1)$ , а у другого  $tid = (n, 2)$ . Если в СУБД используются блокировки на уровне кортежей, то система допустит параллельное выполнение этих двух операций и они будут одновременно изменять страницу, содержащую копию блока базы данных с номером  $n$ . При выполнении обеих операций может потребоваться перемещение кортежей внутри этого блока, и понятно, что в результате ничего хорошего, скорее всего, не получится. Аналогично, логическая синхронизация может легко допустить параллельное выполнение нескольких операций, требующих обновления одного и того же индекса. Некоординированное параллельное обновление В-дерева с большой вероятностью приводит к разрушению его структуры.

Поэтому при выполнении операций уровня RSS необходимо поддерживать дополнительную «физическую» синхронизацию, в которой единицами блокировки служат страницы буферного пула (или блоки) базы данных. В пределах операции перед чтением из страницы буферного пула (блока базы данных) требуется запросить у подсистемы управления буферным пулом блокировку соответствующей страницы (блока) в режиме S,

а перед записью в страницу (в блок) — ее блокировку в режиме X. Совместимость блокировок обычная, такая же, как в табл. 9.1.

Но блокировки страниц буферного пула нужны не только для координации параллельного доступа к страницам при параллельном выполнении транзакций. При выполнении операций уровня RSS могут возникать ошибки, обнаруживаемые в середине операции, уже после того, как одна или несколько страниц буферного пула (блоков базы данных) было изменено. Например, может выполняться операция вставки кортежа в некоторую таблицу, нарушающая уникальность некоторого индекса, определенного над этой таблицей. Нарушение уникальности этого индекса будет обнаружено при попытке вставить в него новый ключ, но до этого новый кортеж уже мог быть размещен в блоке данных и некоторые индексы уже могли быть успешно обновлены.

При обнаружении ошибки операции нужно ликвидировать все ее следы в базе данных и выдать соответствующий код ошибки на уровень RDS. Проще всего сделать это, произведя обратные изменения всех страниц (блоков базы данных), которые были изменены при прямом выполнении операции. Но для этого требуется, чтобы все страницы (блоки базы данных), заблокированные при выполнении операции, оставались заблокированными до конца этой операции.

Тем самым, для подсистемы управления буферным пулом операции уровня RSS являются (почти) тем же, чем являются транзакции для подсистемы управления транзакциями. Достаточным условием корректного выполнения операций является соблюдение двухфазного протокола синхронизационных блокировок над страницами буферного пула в пределах операций.

Заметим (хотя и без подробных объяснений), что это условие не является необходимым. Каждую операцию уровня RSS можно разбить на последовательность «микроопераций» и потребовать соблюдения двухфазного протокола синхронизационных блокировок в пределах микроопераций. Например, операцию INSERT уровня RSS можно разбить на следующие микрооперации:

- 1) нахождение блока данных для вставки;
- 2) вставка кортежа в найденный блок;
- 3) обновление индекса 1;

.....

- n) обновление индекса  $n$ ,

где  $n$  — число индексов, определенных для данной таблицы. Общий принцип состоит в том, что в пределах одной микроопе-

рации блокируются все блоки базы данных, которые обязаны быть изменены согласованным образом.

### **10.1.3. Протокол упреждающей записи в журнал и его связь с буферизацией**

Реальная ситуация является более сложной. Имеются два вида буферов — буфера журнала и буферный пул страниц основной памяти, — которые содержат связанную информацию. И те, и другие буфера могут выталкиваться во внешнюю память. Основной причиной выталкивания буфера журнала является его полное заполнение журнальными записями. Страницы буферного пула базы данных чаще всего выталкиваются во внешнюю память, когда требуется переместить в основную память некоторый блок базы данных, а свободных страниц в буферном пуле нет. Тогда срабатывает алгоритм замещения страниц, выбирается страница, содержимое которой, вероятно, дольше всего не потребуется, и эта страница (если ее содержимое изменялось) выталкивается в соответствующий блок внешней памяти базы данных. Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния базы данных после сбоя.

Заметим, что эта проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое основной памяти не утрачено, и при восстановлении можно пользоваться содержимым как буфера журнала, так и буферных страниц базы данных. Но если произошел мягкий сбой и содержимое буферов утрачено, то для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферных страниц базы данных является то, что запись об изменении объекта базы данных должна оказаться во внешней памяти журнала раньше, чем измененный объект окажется во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется WAL (Write Ahead Log, «пиши сначала в журнал») и состоит в том, что если требуется вытолкнуть во внешнюю память буферную страницу, содержащую измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта.

При следовании протоколу WAL, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. если во внешней памяти в журнале содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершенная транзакция должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние базы данных, содержащее результаты всех транзакций, зафиксированных до момента сбоя.

Самым простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц базы данных, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные расходы при выполнении операции фиксации транзакции.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная запись о конце транзакции.

Рассмотрим теперь, как можно выполнять операции восстановления базы данных в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей, поддерживаемый в соответствии с протоколом WAL.

## **10.2. Индивидуальный откат транзакции**

Для обеспечения возможности индивидуального отката транзакции по общему журналу все записи в журнале от данной транзакции связываются в обратный список. В начале списка для незавершенных транзакций находится запись о последнем изменении базы данных, произведенном данной транзакцией. Заметим, что в этом случае хронологически последние записи могут быть еще не вытолкнуты во внешнюю память журнала

и могут находиться в буфере основной памяти. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала, т. е. весь список находится во внешней памяти. Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, чтобы можно было восстановить прямой список записей об изменениях базы данных данной транзакцией.

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для незавершенных транзакций) выполняется следующим образом:

- выбирается очередная журнальная запись из списка данной транзакции;
- выполняется противоположная по смыслу операция: вместо операции **INSERT** выполняется соответствующая операция **DELETE**, вместо операции **DELETE** выполняется **INSERT**, и вместо прямой операции **UPDATE** — обратная операция **UPDATE**, восстанавливающая предыдущее состояние объекта базы данных;
- любая из этих обратных операций также журнализуется. Собственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуются откатить транзакции, для которых не полностью выполнен индивидуальный откат;
- при успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

### 10.3. Восстановление после мягкого сбоя

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например блок данных и несколько блоков индексов. Блоки базы данных буферизуются в оперативной памяти и выталкиваются независимо. После мягкого сбоя набор блоков внешней памяти базы данных может оказаться несогласованным, т. е. часть блоков внешней памяти соответствует объекту до изменения, часть — после изменения. Например, в результате

выполнения операции **UPDATE** соответствующий кортеж мог переместиться в другой блок. В этом случае (см. гл. 8) изменяются два блока: в описатель кортежа в его исходном блоке записывается его новый *tid*, а в новом блоке размещается сам модифицированный кортеж. Очевидно, что если хотя бы один из этих блоков не попал во внешнюю память базы данных к моменту мягкого сбоя, то при восстановлении не удастся вернуть кортеж на его прежнее место. Другими словами, к такому состоянию внешней памяти базы данных не применимы операции логического уровня.

Состояние внешней памяти базы данных называется *физически согласованным*, если наборы страниц всех объектов согласованы, т.е. соответствуют состоянию любого объекта либо после его изменения, либо до изменения.

### 10.3.1. Схема восстановления от точки физической согласованности

Будем считать, что в журнале отмечаются точки физической согласованности базы данных — моменты времени, в которые во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и отсутствуют результаты операций, которые не завершились, а буфер журнала вытолкнут во внешнюю память. Немного позже мы обсудим, как можно достичь физической согласованности. Назовем такие точки *ppc* (*point of physical consistency*).

Все возможные состояния транзакций к моменту мягкого сбоя показаны на рис. 10.1.

Предположим, что каким-то образом удалось восстановить внешнюю память базы данных к состоянию на момент времени *tppc* (как это можно сделать, обсудим в следующем подразделе). Тогда восстановление последнего по времени логически целостного состояния базы данных производится следующим образом:

- для транзакции  $T_1$  никаких действий производить не требуется. Она закончилась до момента *tppc*, и все ее результаты гарантированно отражены во внешней памяти базы данных;
- для транзакции  $T_2$  нужно повторно выполнить (*redo*) последовательность операций, которые выполнялись после установки точки физически согласованного состояния в момент *tppc*. Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись

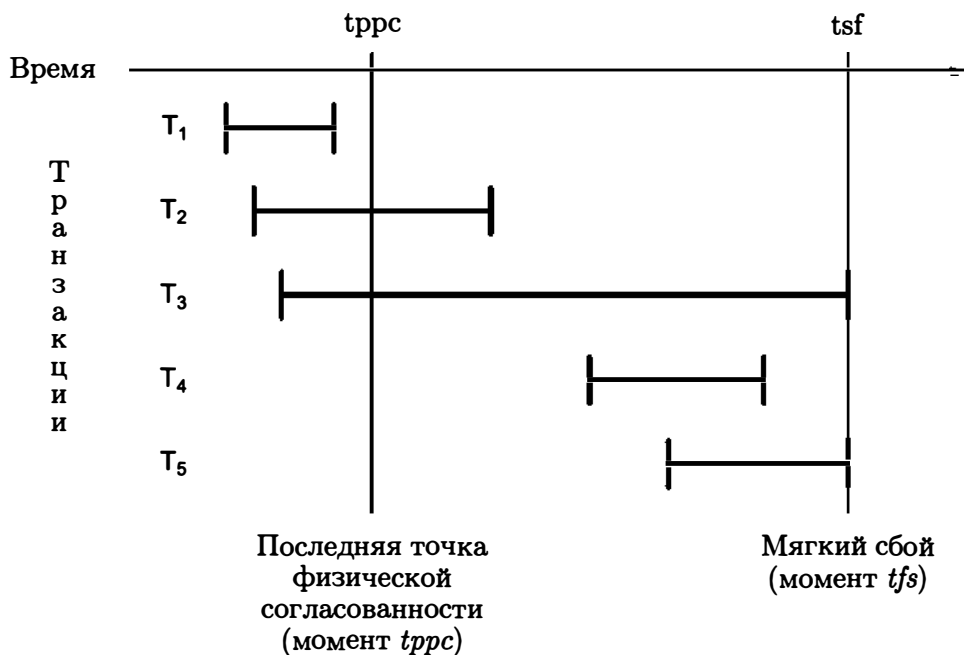


Рис. 10.1. Возможные состояния транзакций к моменту мягкого сбоя

в транзакции  $T_2$  после момента  $tppc$ . Следовательно, повторное прямое (по смыслу и хронологии) выполнение операций транзакции  $T_2$  корректно и приведет к логически согласованному состоянию базы данных (поскольку транзакция  $T_2$  успешно завершилась до момента мягкого сбоя  $tsf$ , в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией);

- для транзакции  $T_3$  нужно выполнить в обратном направлении (undo) ту часть операций, которую она успела выполнить до момента  $tppc$ . Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций  $T_3$ , которые были выполнены после момента  $tppc$ . С другой стороны, во внешней памяти гарантированно присутствуют результаты операций  $T_3$ , которые были выполнены до момента  $tppc$ . Следовательно, обратное выполнение (по смыслу и хронологии) операций  $T_3$  корректно и приведет к согласованному состоянию базы данных (поскольку транзакция  $T_3$  не завершилась к моменту мягкого сбоя  $tsf$ , при восстановлении необходимо устранить все последствия ее выполнения);
- для транзакции  $T_4$ , которая успела начаться после момента  $tppc$  и закончиться до момента мягкого сбоя  $tsf$ , нужно произвести полное повторное выполнение операций в прямом направлении (поскольку транзакция  $T_4$  успешно заверши-



лась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией);

- наконец, для транзакции  $T_5$ , начавшейся после момента *tppc* и не успевшей завершиться к моменту мягкого сбоя *tfs*, никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

### 10.3.2. Восстановление физической согласованности базы данных

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, т. е. как восстановить состояние базы данных в момент *tppc*? Для этого используются два основных подхода: подход, основанный на использовании теневого механизма, и подход, в котором применяется журнализация постраничных изменений базы данных.

**Теневой механизм.** Теневой механизм был изначально предложен для поддержания целостности файлов при аварийном отключении питания компьютера. Общая идея теневого механизма для файлов показана на рис. 10.2. Файл представляется как набор блоков внешней памяти, для доступа к которым поддерживается таблица отображения (см. гл. 1). При открытии файла таблица отображения номеров его логических блоков в адреса физических блоков внешней памяти считывается в оперативную память. При модификации любого блока файла во внешней памяти выделяется новый блок. При этом текущая таблица отображения (в основной памяти) изменяется, а теневая остается неизменной. Если во время работы с открытым файлом происходит сбой, во внешней памяти автоматически сохраняется состояние файла до его открытия. Для явного восстановления файла достаточно повторно считать в основную память теневую таблицу отображения.

В контексте базы данных теневой механизм используется следующим образом [37]. Периодически выполняются операции установки точки физической согласованности базы данных. При выполнении этой операции все логические операции завершаются, все страницы буферного пула базы данных, содержимое которых отличается от содержимого соответствующих блоков внешней памяти, выталкиваются. Теневая таблица отображения файлов (сегментов) базы данных заменяется текущей таблицей отображения (правильнее сказать, текущая таблица отображения записывается на место теневой).

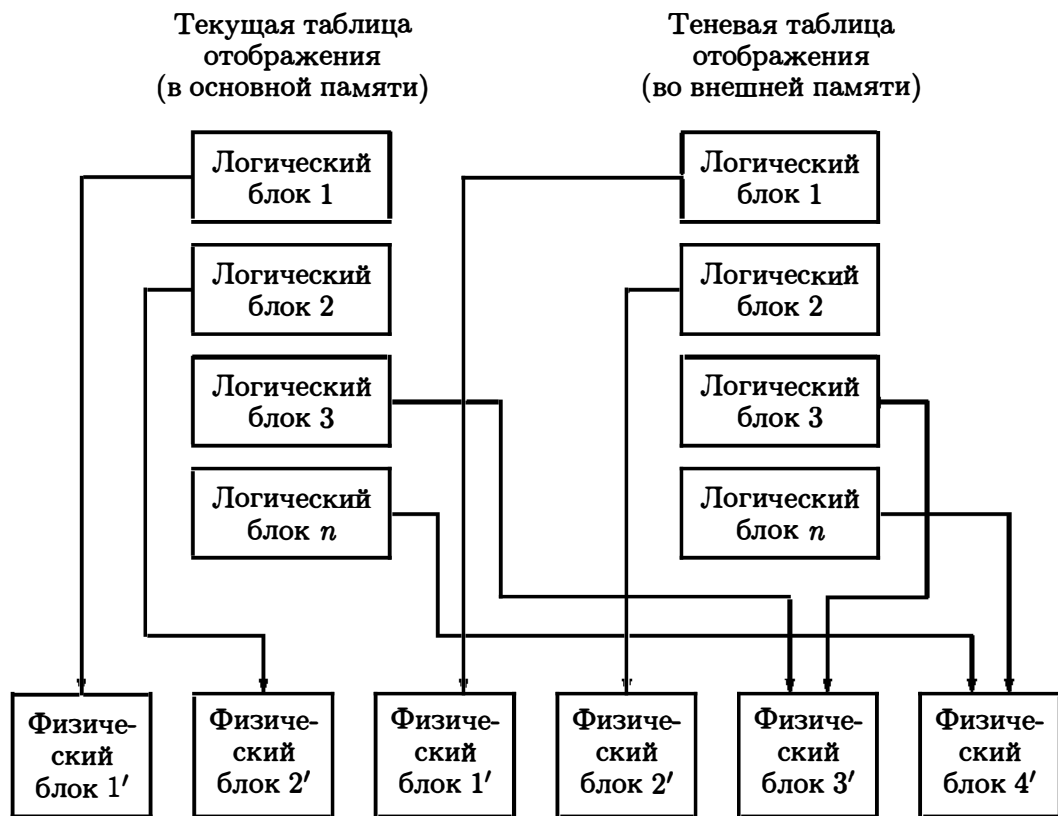


Рис. 10.2. Теневой механизм для файлов

Здесь имеется некоторая проблема, состоящая в том, что в любой момент времени теневая таблица отображения должна быть корректной, т. е. соответствовать некоторому ранее зафиксированному физически целостному состоянию базы данных. Для этого необходимо обеспечить атомарность операции замены теневой таблицы отображения. В общем случае таблица отображения может занимать несколько блоков внешней памяти и для записи текущей таблицы отображения на место теневой таблицы в этом случае потребуется несколько обменов с дисками. Если в промежутке между этими обменами возникнет мягкий сбой, то будет благополучно утрачена текущая таблица отображения и безнадежно испорчена теневая таблица, т. е. мы просто лишимся возможности восстанавливаться за счет использования последнего физически согласованного состояния базы данных.

Чтобы это не произошло, во внешней памяти поддерживаются две области хранения таблицы отображения файлов (будем называть их областями А и В). Кроме того, в отдельном блоке внешней памяти хранится флаг F, показывающий, какая из этих областей в данный момент содержит действующую теневую таблицу отображения (назовем соответствующие значения флага

$F_A$  и  $F_B$ ). Тогда, если сохраненным во внешней памяти значением флага является  $F_A$ , то текущая таблица отображения записывается в область В. Если эта операция выполняется успешно, то в блок флага записывается значение  $F_B$ . Считается, что операция записи одного блока на диск является атомарной. Если эта операция заканчивается успешно, это означает, что новая теньевая таблица отображения хранится в области В. Если же запись текущей таблицы отображения в область В не удалась или не выполнялась операция записи блока с флагом  $F$ , то продолжает действовать старая теньевая таблица отображения.

Восстановление хронологически последнего сохраненного физически согласованного состояния базы данных происходит мгновенно: текущая таблица отображения заменяет теньевой таблицей (при восстановлении просто считывается действующая теньевая таблица отображения). Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных.

**Журнализация постраничных изменений.** Возможен другой подход, при использовании которого наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате невыполненных логических операций. Подобно тому, как это делается с логическими записями по отношению к транзакциям, последней записью о постраничных изменениях от одной логической операции является запись о конце операции. Вообще, выполнение логических операций уровня RSS носит транзакционный характер. В частности, как уже отмечалось ранее, при выполнении логической операции обновления базы данных, вообще говоря, изменяется несколько блоков базы данных. Для обеспечения возможности отката отдельной операции (а это может потребоваться, например, если обнаруживается нарушение свойства уникальности какого-либо индекса) приходится до конца операции монополюно блокировать все страницы буферного пула базы данных, содержащие копии изменяемых этой операцией блоков базы данных.

Чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, при выталкивании любой страницы из буферного пула основной памяти в нее помещается номер последней записи о постраничном изменении этой страницы. Этот же номер запоминается в самой записи. Тогда, чтобы понять, нужно ли применить данную запись о постра-

ничном изменении соответствующего блока внешней памяти для восстановления состояния этого блока, требуется всего лишь сравнить номер, содержащийся в этом блоке, с номером, содержащимся в журнальной записи. Если в блоке содержится номер, меньший номера журнальной записи, то это означает, что буферная страница, в которой выполнялось соответствующее изменение, не была к моменту мягкого сбоя вытолкнута во внешнюю память, и применять данную запись для восстановления соответствующего блока внешней памяти не требуется.

Для иллюстрации на рис. 10.3 показано пять записей об изменении блока  $b$  с номерами  $n-2$ ,  $n-1$ ,  $n$ ,  $n+1$ ,  $n+2$ . В блоке  $b$  содержится номер  $n$ . Это означает, что в состоянии блока отражены результаты операций изменения блока, соответствующих журнальным записям  $LR(b)_n$ ,  $LR(b)_{n-1}$  и  $LR(b)_{n-2}$ . Изменения блока, произведенные операциями, которым соответствуют две хронологически последние журнальные записи  $LR(b)_{n+1}$  и  $LR(b)_{n+2}$ , в его состоянии во внешней памяти не отражены, поскольку не было выполнено выталкивание во внешнюю память страницы буферного пула, содержащей копию блока  $b$ . Поэтому при восстановлении состояния блока требуется выполнить обратные операции изменения блока  $b$ , соответствующие журнальным записям  $LR(b)_n$ ,  $LR(b)_{n-1}$  и  $LR(b)_{n-2}$ .

В этом подходе имеются два поднаправления. В первом поднаправлении поддерживается общий журнал логических и страничных операций. Естественно, наличие двух видов записей, интерпретируемых абсолютно по-разному, усложняет структуру журнала. Кроме того, записи о постраничных изменениях, ак-

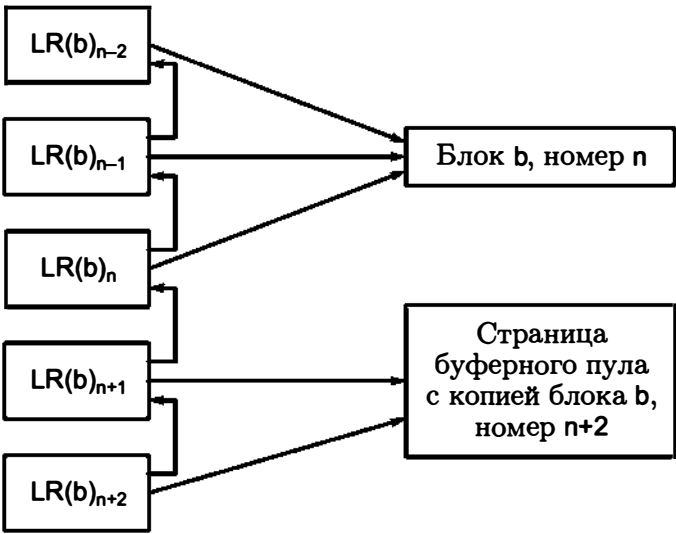


Рис. 10.3. Нумерация записей об изменении блока

туальность которых носит локальный характер, существенно (и не очень осмысленно) увеличивают журнал.

Поэтому распространено поддержание отдельного (короткого) журнала постраничных изменений. Такой журнал обычно называют *физическим* журналом, поскольку он содержит записи об изменении физических объектов — блоков внешней памяти. В отличие от этого, журнал логических операций принято называть *логическим* журналом, поскольку в нем содержатся записи об операциях над логическими объектами — кортежами.

Как уже отмечалось, логический и физический журналы имеют разную природу. Во-первых, логический журнал должен поддерживать как обратное выполнение журнализованных операций (*undo*), так и их повторное прямое выполнение (*redo*). В отличие от этого, от физического журнала требуется только поддержка обратного выполнения постраничных операций. Во-вторых, логический журнал обычно начинает заполняться заново только после выполнения операций резервного копирования базы данных или архивирования самого журнала (см. следующий подраздел). До этого времени он линейно растет. Понятно, что в любом случае для размещения журнала выделяется внешняя память ограниченного размера. Предельный размер журнала определяется администратором базы данных и должен согласовываться с размером интервала времени, через которое производится резервное копирование базы данных.

Потенциальное переполнение логического журнала регулируется следующим образом. На пути к достижению максимально возможного размера журнала устанавливаются «желтая» и «красная» зоны. Когда записи в журнал достигают «желтой» зоны, выдается предупреждение администратору базы данных и прекращается образование новых транзакций. Если все существующие транзакции завершаются до достижения «красной» зоны, автоматически выполняется архивация базы данных или логического журнала. Если какие-то транзакции не успевают завершиться до достижения «красной» зоны журнала, выполняется их аварийный откат, после чего производится архивация базы данных или журнала. Естественно, размер «желтой» и «красной» зон логического журнала должен устанавливаться администратором базы данных с учетом максимально допустимого числа одновременно существующих транзакций и их возможной протяженности.

В отличие от этого, физический журнал существует сравнительно недолгое время (интервал времени между соседними операциями установки точки физической согласованности базы данных) и, как правило, занимает существенно меньшее дисковое

пространство, чем логический журнал. При выполнении операции установки точки физической согласованности выполняются следующие действия:

- прекращают инициироваться новые логические операции;
- после завершения всех выполняемых логических операций происходит выталкивание во внешнюю память всех модифицированных страниц буферного пула;
- формируется и выталкивается во внешнюю память логического журнала специальная запись о точке физически согласованного состояния;
- в случае успешного предыдущего действия разрешается инициация новых логических операций и физический журнал пишется заново.

Предпоследняя операция является атомарной (это опять же запись одного блока на диск): если она успешно выполняется, то при следующем восстановлении после мягкого сбоя будет использоваться новая точка физически согласованного состояния, иначе ситуация воспринимается как мягкий сбой с восстановлением логически согласованного состояния базы данных от предыдущей точки физически согласованного состояния (с оповещением об этом администратора базы данных).

## **10.4. Восстановление базы данных после жесткого сбоя**

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя журнала изменений базы данных явно недостаточно. Самый простой способ восстановления основывается на использовании логического журнала и архивной копии базы данных.

Восстановление начинается с обратного копирования (на исправный носитель) базы данных из архивной копии. Затем для всех закончившихся транзакций выполняется *redo*, т. е. операции повторно выполняются в прямом смысле.

Более точно, происходит следующее:

- по журналу в прямом направлении выполняются все операции;
- для транзакций, которые не закончились к моменту сбоя, выполняется откат.

Очевидно, что после этого будет получено хронологически последнее до момента жесткого сбоя логически согласованное состояние базы данных.

Следует заметить, что при некоторой дисциплине выполнения логических операций над базой данных при восстановлении базы данных после жесткого сбоя можно просто последовательно повторно выполнять операции в соответствии с журнальными записями, не обращая внимание на то, в каких транзакциях они выполнялись до жесткого сбоя. В частности, если сериализация транзакций основывается на блокировках объектов, то эта дисциплина заключается в том, что при выполнении операции в штатном режиме нужно сначала дождаться удовлетворения блокировки изменяемого объекта, затем поместить запись в буфер логического журнала, и только после этого реально выполнять операцию.

На самом деле, поскольку жесткий сбой не сопровождается утратой буферов оперативной памяти, можно восстановить базу данных до такого уровня, чтобы можно было продолжить даже выполнение незавершенных транзакций. Но обычно это не делается, потому что восстановление после жесткого сбоя — это достаточно длительный процесс.

Хотя к ведению журнала предъявляются особые требования по части надежности, в принципе возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к архивной копии. Конечно, в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который здесь следует еще раз обсудить, касается производства архивных копий базы данных и/или журнала. Самый простой способ состоит в архивировании базы данных по явному указанию администратора или при переполнении журнала. Но можно выполнять архивацию базы данных реже, чем переполняется журнал. Вместо базы данных можно архивировать сам журнал. В пределе для полного восстановления базы данных после жесткого сбоя достаточно иметь исходную архивную копию базы данных, последовательность архивных копий журналов и последний логический журнал. Может показаться, что восстановление базы данных на основе таких архивных источников будет занимать недопустимо большое время, однако здесь возможна значительная оптимизация.

Во-первых, архивированный логический журнал можно сжимать. Для этого для каждого объекта базы данных нужно найти последовательность журнальных записей, относящихся к этому объекту, в хронологическом порядке и заменить их одной записью, соответствующей операции над объектом, результат которой эквивалентен результату последовательного

выполнения журнализованных операций из построенной последовательности. Например, на рис. 10.4 показан процесс сжатия последовательности журнальных записей, соответствующих последовательности операций над кортежем, у которого  $tid = k$  и имеются четыре целочисленных поля. Заметим, что если хронологически последней в последовательности является запись, соответствующая операции **DELETE**, то после сжатия этой последовательности она станет пустой.

Во-вторых, точно таким же образом можно совместно сжать два хронологически последовательных полных или сжатых журнала. Таким образом, для восстановления после жесткого сбоя можно воспользоваться исходной архивной копией, одним сжатым архивным журналом и последним логическим журна-

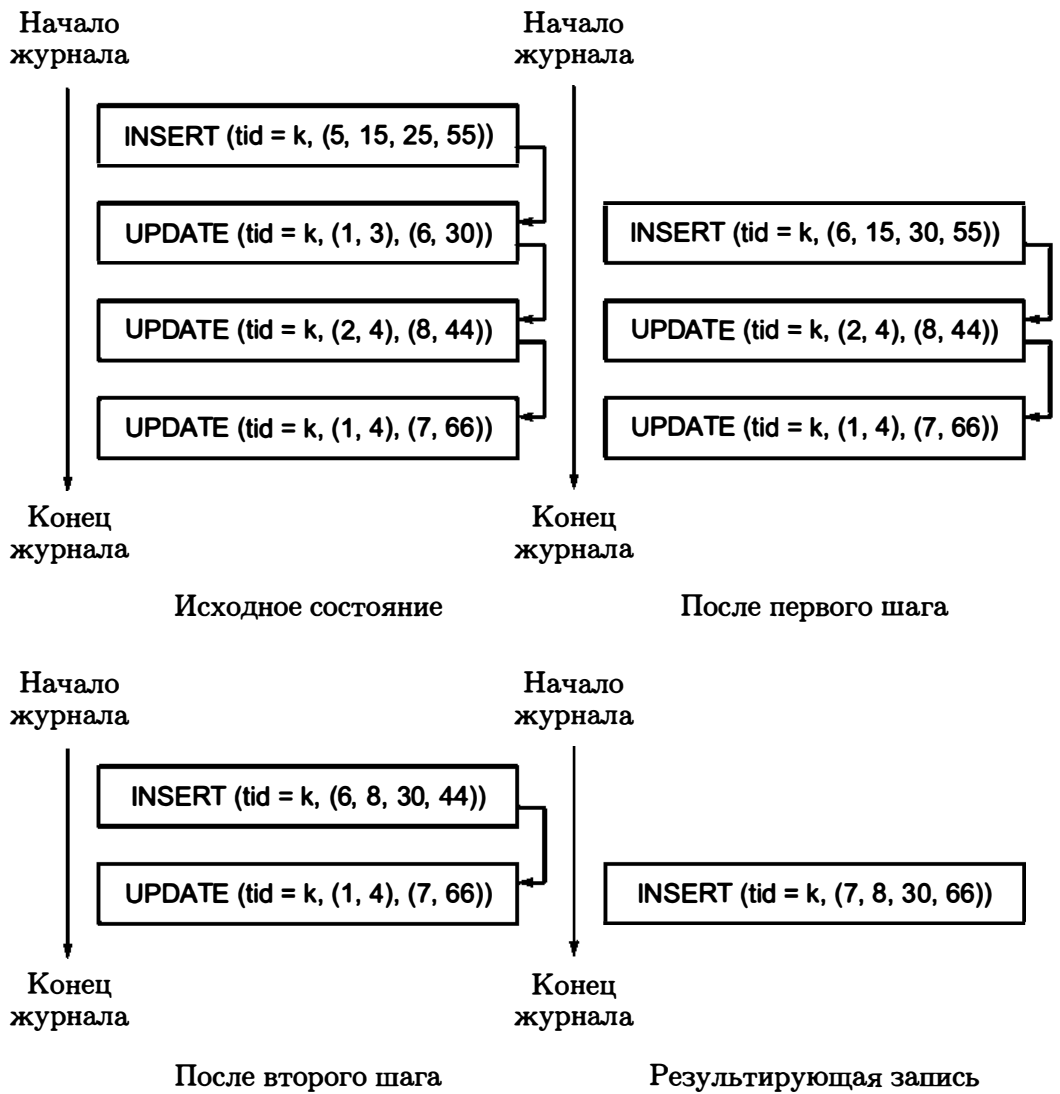


Рис. 10.4. Процесс сжатия последовательности журнальных записей



лом. Снова могут возникнуть сомнения относительно сложности и продолжительности процесса сжатия журнала. Но здесь следует заметить, что эта работа может выполняться на отдельном компьютере в режиме off-line. Кроме того, если имеется архивная копия базы данных, сжатый архивный журнал и набор еще не сжатых архивных журналов, то этого уже достаточно для восстановления, так что сроки завершения процесса полного сжатия не являются критическими.

\* \* \*

В этой главе рассматривались основные принципы и алгоритмы подсистем СУБД, предназначенных для управления буферами основной памяти, журнализации и восстановления базы данных после различных сбоев. Изложение велось без технических деталей, таких как возможные структуры данных журналов.

Заметим, что во многих современных производственных СУБД журнализация и восстановление основаны на применении семейства алгоритмов ARIES, разработанных в 1980-е гг. известным исследователем из компании IBM К. Моханом (С. Mohan). В этой главе не приводится описание алгоритмов семейства ARIES, поскольку, по мнению автора, это перегрузило бы ее подробностями, не способствующими пониманию основных идей. Тем не менее читателям, которых заинтересовала эта тема, полезно познакомиться с этими алгоритмами, для чего можно воспользоваться, например, [37] или оригинальными статьями Мохана и его коллег.

### Глава 11

#### **ИСТОРИЯ СТАНДАРТА ЯЗЫКА SQL. ТИПЫ ДАННЫХ. СРЕДСТВА ЯЗЫКА SQL ДЛЯ ОПРЕДЕЛЕНИЯ И ИЗМЕНЕНИЯ ДОМЕНОВ, БАЗОВЫХ ТАБЛИЦ И ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ**

Оставшиеся главы книги посвящены языку баз данных SQL. В книге о реляционных базах данных невозможно обойтись без материала, который относится к этому языку. Это связано совсем не с тем, что язык представляет собой особое достижение в области реляционных баз данных. Напротив, многие черты SQL, начиная с самых первых его вариантов, противоречили принципам реляционной модели данных, заложенным Э. Коддом. С другой стороны, как отмечалось в гл. 2, спецификация языка SQL, по своей сути, является завершенной спецификацией модели данных, которая сегодня играет роль суррогата реляционной модели.

В настоящее время SQL является общепринятым языком в мире баз данных. Интерфейсы, основанные на SQL, поддерживаются практически во всех используемых СУБД, далеко не все из которых первоначально разрабатывались как реляционные системы. И похоже, что эта ситуация при жизни нынешнего поколения радикальным образом не изменится. Кроме того, язык сам по себе достаточно интересен. В нем нашел отражение многолетний практический опыт многих людей, и он впитал в себя многие положительные (и отрицательные) черты других языков и подходов (не только языков баз данных и не только реляционного подхода).

В данной главе после небольшой исторической справки и краткого введения в структуру языка SQL будут рассмотрены типы данных, допустимые в языке SQL и в SQL-ориентированных базах данных, средства определения и изменения базовых, ограничения целостности, допустимые в SQL-ориентированных базах данных, и языковые средства их определения.

## 11.1. История стандарта SQL и структура языка

Как отмечалось в гл. 8, язык SQL возник при выполнении проекта System R компании IBM. Первую коммерческую систему, в которой поддерживался SQL, выпустила на рынок в 1979 г. компания Oracle. Только через два года, в 1981 г. компания IBM ответила на этот вызов выпуском СУБД SQL/DS, предназначенным для использования в среде операционных систем DOS/VSE и VM/CMS на своих мейнфреймах. Система SQL/DS была основана на кодах System R. В 1983 г. IBM выпустила первую версию полностью переписанной SQL-ориентированной СУБД DB2, которая работала в среде операционной системы MVS. До конца 1990-х г. СУБД SQL/DS поддерживалась как отдельный продукт IBM, но затем была переименована в DB2 для VM и VSE. В 1985 г. свой первый SQL-ориентированный продукт INFORMIX-SQL выпустила компания Informix. В 1987 г. появился SQL Server компании Sybase.

### 11.1.1. Этапы процесса стандартизации языка SQL

Деятельность по стандартизации языка SQL началась практически одновременно с появлением его первых коммерческих реализаций. В 1982 г. комитету по базам данных Американского национального института стандартов (ANSI) было поручено разработать спецификацию стандартного языка реляционных баз данных. Стандарт был принят ANSI в 1986 г., а в 1987 г. одобрен Международной организацией по стандартизации (ISO). Этот стандарт принято называть SQL/86.

В качестве основы стандарта нельзя было использовать SQL System R. Во-первых, этот вариант языка не был должным образом технически проработан. Во-вторых, его слишком сложно было бы реализовать (кто знает, как бы сложилась судьба SQL, если бы все идеи проекта System R были реализованы полностью). За основу был взят диалект языка SQL, сложившийся в IBM к началу 1980-х гг. В сущности, этот диалект представлял собой технически проработанное небольшое подмножество SQL System R.

К 1989 г. стандарт SQL/86 был несколько расширен и был подготовлен и принят следующий стандарт, получивший название ANSI/ISO SQL/89. Анализ доступных документов показывает, что процесс стандартизации SQL происходил очень сложно с использованием не только научных доводов. В результате SQL/89 во многих частях имел чрезвычайно общий характер

и допускал очень широкое толкование. В этом стандарте полностью отсутствовали такие важные разделы, как манипулирование схемой БД и динамический SQL. Многие важные аспекты языка в соответствии со стандартом определялись в реализации, что практически обесценивало стандарт с точки зрения программистов приложений баз данных, поскольку не давало возможности создавать приложения, не привязанные к особенностям конкретных СУБД.

Возможно, наиболее важными достижениями стандарта SQL/89 являлись четкая стандартизация синтаксиса и семантики операторов выборки данных и манипулирования данными и фиксация средств ограничения целостности БД. Были специфицированы средства определения первичного и внешних ключей отношений и так называемых проверочных ограничений целостности, которые представляют собой подмножество немедленно проверяемых ограничений целостности SQL System R. Средства определения внешних ключей позволяли легко формулировать требования так называемой ссылочной целостности БД. Это распространенное в реляционных БД требование можно было сформулировать и на основе общего механизма ограничений целостности SQL System R, но формулировка на основе понятия внешнего ключа более проста и понятна.

Осознавая неполноту стандарта SQL, на фоне завершения разработки этого стандарта специалисты различных компаний начали работу над стандартом SQL2. Эта работа также длилась несколько лет, было выпущено множество проектов стандарта, пока наконец в марте 1992 г. не был принят окончательный проект стандарта (SQL/92). Этот стандарт существенно полнее стандарта SQL/89 и охватывает практически все аспекты, необходимые для реализации приложений: манипулирование схемой БД, управление транзакциями (появились точки сохранения) и сессиями (сессия — это последовательность транзакций, в пределах которой сохраняются временные отношения), подключения к БД, динамический SQL. Наконец, были стандартизованы отношения-каталоги БД, что вообще-то не связано непосредственно с языком, но очень сильно влияет на реализацию.

Для пояснения смысла последнего предложения предыдущего абзаца заметим, что, начиная с System R, в базах данных, управляемых SQL-ориентированными СУБД, хранились как данные, так и метаданные — описатели таблиц и их столбцов, представлений, ограничения целостности и т. д. Для хранения метаданных использовались специальные служебные таблицы,

которые принято называть таблицами-каталогами. Из таблиц-каталогов можно выбрать данные с помощью обычных средств языка SQL. Конечно, организация служебных данных — это вопрос реализации SQL, но этот вопрос непосредственно касается потенциальных пользователей SQL-ориентированных СУБД, и поэтому стандартизация представления пользователю отношений-каталогов (в стандарте SQL — информационной схемы базы данных) является исключительно важным делом.

В 1995 г. стандарт был дополнен спецификацией интерфейса уровня вызова (Call-Level Interface — SQL/CLI). SQL/CLI представляет собой набор спецификаций интерфейсов процедур, вызовы которых позволяют выполнять динамически задаваемые операторы SQL. По сути дела, SQL/CLI представляет собой альтернативу динамическому SQL. Интерфейсы процедур определены для всех основных языков программирования: C, Ada, Pascal, PL/1 и т. д. Следует заметить, что стандарт SQL/CLI послужил основой для создания повсеместно распространенных сегодня интерфейсов ODBC (Open Database Connectivity) и JDBC (Java Database Connectivity).

В 1996 г. к стандарту SQL/92 был добавлен еще один компонент — SQL/PSM (Persistent Stored Modules). Основная цель этой спецификации состоит в том, чтобы стандартизировать способы определения и использования *хранимых процедур*, т. е. специальным образом оформленных программ, включающих операторы SQL, которые сохраняются в базе данных, могут вызываться приложениями и выполняются внутри СУБД.

Незадолго до завершения работ по определению стандарта SQL2 была начата разработка стандарта SQL3. Первоначально планировалось завершить проект в 1995 г. и включить в язык некоторые объектные возможности: определяемые пользователями типы данных, поддержку триггеров, поддержку темпоральных свойств данных и т. д. Реально работу над новым стандартом удалось частично завершить только в 1999 г., и по этой причине (а также в связи с проблемой 2000 г.) стандарт получил название SQL:1999.

Каждый новый вариант стандарта языка SQL был существенно объемнее предыдущих версий. Так, если стандарт SQL/89 занимал около 600 страниц, то объем SQL/92 составлял на 300 с лишним страниц больше. Самые первые проекты SQL3 занимали около 1500 страниц. Это вполне естественно, потому что язык усложняется, а его спецификации становятся более детальными и точными. Но разработчики SQL3 пришли к выводу, что при

таких объемах стандарта вероятность его принятия и последующей успешной поддержки заметно уменьшается. Поэтому было принято решение разбить стандарт на относительно независимые части, которые можно было бы разрабатывать и поддерживать по отдельности.

В 1999 г. были приняты пять первых частей стандарта SQL:1999. Первая часть (SQL/Framework) посвящена описанию концептуальной структуры стандарта. В этой части приводится развернутая аннотация следующих четырех частей и формулируются требования к реализациям, претендующим на соответствие стандарту.

Вторая часть SQL:1999 (SQL/Foundation) образует базис стандарта. Вводится система типов языка, формулируются правила определения функциональных зависимостей и возможных ключей, определяются синтаксис и семантика основных операторов SQL: операторов определения и манипулирования схемой базы данных; операторов манипулирования данными; операторов управления транзакциями; операторов управления подключениями к базе данных и т. д.

Третью часть занимает уточненная по сравнению с SQL/92 спецификация SQL/CLI. В четвертой части специфицируется SQL/PSM — синтаксис и семантика языка определения хранимых процедур. Наконец, в пятой части — SQL/Bindings — определяются правила связывания SQL для стандартных версий языков программирования FORTRAN, COBOL, PL/1, Pascal, Ada, C и MUMPS.

В конце 2003 г. был принят и опубликован новый вариант международного стандарта SQL:2003, который состоит из следующих частей: 9075-1, SQL/Framework; 9075-2, SQL/Foundation; 9075-3, SQL/CLI; 9075-4, SQL/PSM; 9075-9, SQL/MED; 9075-10, SQL/OLB; 9075-11, SQL/Schemata; 9075-13, SQL/JRT; 9075-14, SQL/XML.

Части 1 — 4 с необходимыми изменениями остались такими же, как и в SQL:1999. Часть 5 (SQL/Bindings) перестала существовать; соответствующие спецификации включены в часть 2. Раздел части 2 SQL:1999, посвященный информационной схеме, выделен в отдельную часть 11. Появились две новые части — 9, 10 и 13, 14. Часть 9 посвящена спецификации языковых средств взаимодействия с внешними данными, часть 10 — правилами связывания SQL с объектно-ориентированными языками программирования. Часть 13 полностью называется «SQL Routines and Types Using the Java Programming Language» («Использование подпрограмм и типов SQL в языке програм-

мирования Java»). Появление такой части стандарта оправдано повышенным вниманием к языку Java со стороны ведущих производителей SQL-ориентированных СУБД. Наконец, последняя часть SQL:2003 посвящена спецификациям языковых средств, позволяющих работать с XML-документами в среде SQL.

В 2006 г. была опубликована полностью переработанная 14-я часть стандарта SQL:2003, что позволило говорить о появлении стандарта SQL:2006, хотя остальные части SQL:2003 в ней практически не изменились.

В целом, текущее состояние процесса стандартизации языка SQL отражает текущее состояние технологии SQL-ориентированных баз данных. Ведущие поставщики соответствующих СУБД (сегодня это компании IBM, Oracle и Microsoft) стараются максимально быстро реагировать на потребности и конъюнктуру рынка и расширяют свои продукты все новыми и новыми возможностями. Очевидна потребность в стандартизации соответствующих языковых средств, но процесс стандартизации явно не поспевает за происходящими изменениями.

### 11.1.2. Структура языка SQL

В этой главе мы начинаем систематически обсуждать базовые механизмы языка SQL. Чтобы пояснить, о какой части языка пойдет речь в этой и следующих главах, обратимся к рис. 11.1.

Язык SQL, соответствующий последним стандартам SQL:2003, SQL:1999 (и даже SQL/92), — очень богатый и сложный язык, все возможности которого трудно сразу осознать и тем более понять. Поэтому приходится разбивать язык на уровни, или слои, такие, что каждый уровень языка включает все конструкции, входящие в более низкие уровни. В стандарте определяются несколько способов разбиения языка на уровни. В одной

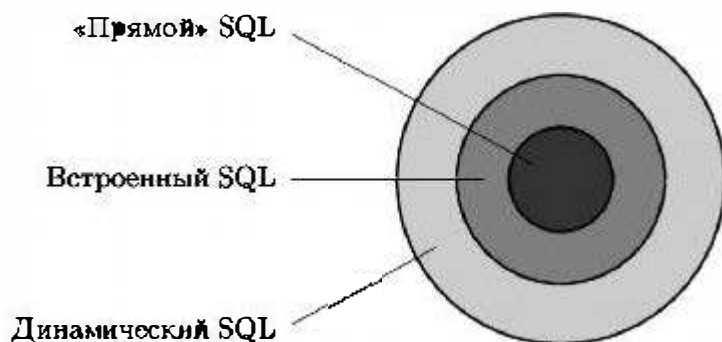


Рис. 11.1. Один из способов разбиения языка SQL на уровни

из классификаций язык разбивается на «базовый» (*entry*), «промежуточный» (*intermediate*) и «полный» (*full*) уровни.

Эта классификация ориентирована, прежде всего, на производителей SQL-ориентированных СУБД. Реализация базового уровня языка является обязательным условием хотя бы какого-то соответствия стандарту. Реализация промежуточного уровня желательна, и обычно именно такой уровень языка поддерживается ведущими компаниями-производителями SQL-ориентированных СУБД. Наконец, полный уровень языка является целью, к достижению которой следует стремиться. В данной классификации критерием отнесения той или иной возможности языка к некоторому уровню является оцениваемая создателями стандарта SQL (большая часть которых является сотрудниками ведущих компаний, производящих SQL-ориентированные СУБД) техническая сложность реализации этой возможности. Конечно, такая классификация важна и для программистов приложений баз данных, но только для того, чтобы оценить реальные возможности конкретной СУБД. Для понимания языка SQL это разбиение на уровни несущественно.

Другая классификация показана на рис. 11.1. Среди всех конструкций языка SQL можно выделить такие конструкции, которые можно было бы использовать при «прямом» (*direct*) взаимодействии конечного пользователя с СУБД (например, в интерактивном режиме). В некотором смысле этот уровень также является базовым, поскольку соответствующие средства языка в наибольшей степени отражают его ориентированность на работу с мультимножествами. На следующем уровне, уровне «встраиваемого» (*embedded*) SQL, язык расширяется конструкциями, позволяющими использовать возможности прямого SQL в программах, написанных на традиционных языках программирования. Наконец, на уровне «динамического» (*dynamic*) SQL во встраиваемый SQL добавляются конструкции, позволяющие приложениям обращаться к СУБД с конструкциями прямого SQL, которые динамически образуются во время выполнения программы.

Вторая классификация является более полезной для читателя, постигающего основы языка SQL. Дополнительные возможности, присутствующие во встраиваемом и в динамическом SQL, не слишком сильно влияют на модельное представление языка. Конечно, возможности встраиваемого и динамического SQL необходимо хорошо знать разработчикам приложений SQL-ориентированных баз данных. Но поскольку задачей этой книги не является обучение использованию языка SQL при



программировании приложений баз данных, в ней не будут затрагиваться эти темы. Обратимся к прямому SQL, причем далеко не в полном объеме стандартов SQL:2003 и SQL:1999. Обсудим только наиболее важные аспекты. (Заметим, что в этой и следующих главах используется терминология стандарта SQL: *таблицы, строки таблиц, столбцы* и т. д.)

## 11.2. Типы данных SQL

Данные, хранящиеся в столбцах таблиц (а также в других типизированных объектах языка SQL, которые мы не затрагиваем в этой книге) SQL-ориентированной базы данных, являются типизированными, т. е. представляют собой значения одного из типов данных, предопределенных в языке SQL или определяемых пользователями путем использования соответствующих средств языка. Для этого при определении таблицы каждому ее столбцу предписывается некоторый тип данных (или домен), и в дальнейшем СУБД должна следить, чтобы в каждом столбце каждой строки каждой таблицы присутствовали только допустимые значения. В этом подразделе обсудим систему типов языка SQL.

Все допустимые в SQL типы данных разбиваются на следующие категории:

- точные числовые типы (*exact numerics*);
- приближенные числовые типы (*approximate numerics*);
- типы символьных строк (*character strings*);
- типы битовых строк (*bit strings*);
- типы даты и времени (*datetimes*);
- типы временных интервалов (*intervals*);
- булевский тип (*Booleans*);
- типы коллекций (*collection types*);
- анонимные строчные типы (*anonymous row types*);
- типы, определяемые пользователем (*user-defined types*);
- ссылочные типы (*reference types*).

В столбцах таблиц, определенных на любых типах данных, наряду со значениями этих типов допускается сохранение неопределенного значения, которое обозначается ключевым словом NULL. В языке определено, что результатом выражений вида  $x \text{ a\_op NULL}$ ,  $\text{NULL a\_op } x$ ,  $\text{NULL a\_op NULL}$  является NULL для всех арифметических операций *a\_op* (+, – и т. д.), допустимых для типа данных выражения *x* (выражение  $\text{NULL a\_op NULL}$  является допустимым для любой арифметической операции *a\_op*).

Также по определению полагается, что значением выражений *x comp\_op NULL*, *NULL comp\_op x*, *NULL comp\_op NULL* для всех операций сравнения *comp\_op* (*=*, *≠*, *>*, *<* и т.д.), определенных для типа выражения *x*, является третье логическое значение *unknown* (выражение *NULL comp\_op NULL* является допустимым для любой операции сравнения *comp\_op*).

### 11.2.1. Точные числовые типы

К категории точных числовых типов в SQL относятся те типы, значения которых точно представляют числа. Типы данных этой категории распадаются на две части: истинно целые типы (**INTEGER** и **SMALLINT**) и типы, допускающие наличие дробной части (**NUMERIC** и **DECIMAL**).

**Истинно целые типы.** Тип **INTEGER** служит для представления целых чисел. Точность чисел (число сохраняемых бит) определяется в реализации. При определении столбца данного типа достаточно указать просто **INTEGER**.

Тип **SMALLINT** также служит для представления целых чисел. Точность определяется в реализации, но не должна быть больше точности типа **INTEGER**. При определении столбца указывается просто **SMALLINT**. Заметим, что в стандарте SQL не определяется число байт, занимаемых при хранении в памяти значений целых типов.

Литералы типов целых чисел представляются в виде строк символов, изображающих десятичные числа; в начале строки могут присутствовать символы «+» или «-» (если символ знака отсутствует, подразумевается знак «+»). Примеры литералов типов **INTEGER** и **SMALLINT**: 1826545, 876.

**Точные типы, допускающие наличие дробной части.** Тип **NUMERIC** — это не просто тип данных, а параметризуемый тип. При определении столбца можно указать спецификацию **NUMERIC (p, s)**, где *p* и *s* — литералы истинно целого типа, и *p* задает точность значений (число сохраняемых десятичных цифр), а *s* — шкалу (число десятичных цифр в дробной части). Задаваемая шкала не должна быть отрицательной и превышать значение точности. При определении столбца можно использовать сокращенные формы спецификации типа — **NUMERIC** и **NUMERIC (p)**. Первая форма предполагает использование точности, определяемое по умолчанию в реализации, и шкалы, равной нулю, а вторая — использование заданной точности и шкалы, равной нулю. Допустимые диапазоны значений *p* и *s* определяются в реализации.

Тип **DECIMAL** аналогичен типу **NUMERIC**. Отличие состоит в том, что если при определении столбца типа **DECIMAL** задается точность  $p$ , то на самом деле используется точность  $m$ , определяемая в реализации, такая что  $m \geq p$ . Шкала всегда устанавливается такой, как явно или неявно (по умолчанию) задается. При указании типа столбца можно использовать спецификации **DECIMAL**, **DECIMAL (p)** и **DECIMAL (p, s)**.

Литералы типов точных чисел, допускающих наличие дробной части, представляются в виде строк символов, изображающих десятичные числа, в начале которых могут присутствовать символы «+» или «-» (если символ знака отсутствует, подразумевается знак «+»), а внутри последовательности цифр может присутствовать символ «.». Примеры литералов типов **NUMERIC** и **DECIMAL**: 125, 26.36.

### 11.2.2. Приближенные числовые типы

К категории приближенных числовых типов в SQL относятся те типы, значения которых представляют числа приближенным образом. Приближенные числа представляются в виде пары *<мантисса, порядок>*, где *мантисса* состоит из значащих цифр числа, а *порядок* определяет реальный размер числа. В реализациях приближенным числовым типам SQL обычно соответствуют типы с плавающей точкой. В SQL поддерживаются три варианта приближенных числовых типов.

Значения типа **REAL** соответствуют числам с плавающей точкой одинарной точности. Точность определяется в реализации, но обычно совпадает с точностью одинарной плавающей арифметики, поддерживаемой на аппаратной платформе, которая используется реализацией. При определении столбца указывается просто **REAL**.

Тип типа **DOUBLE PRECISION** определяется в реализации, но она должна быть больше точности типа **REAL**. Обычно приближенным числам SQL с двойной точностью соответствуют поддерживаемые аппаратурой числа с плавающей точкой двойной точности. При определении столбца указывается просто **DOUBLE PRECISION**.

Тип **FLOAT** — это параметризуемый тип, значение параметра  $p$  которого задает требуемую точность значений. Требуется, чтобы реально обеспечиваемая реализацией точность значений была не меньше  $p$ . Допустимый диапазон значений параметра  $p$  определяется в реализации. При определении столбца можно указать либо **FLOAT (p)**, либо просто **FLOAT**. В последнем случае

подразумевается точность, определяемая реализацией по умолчанию.

Литералы приближенных числовых типов представляются в виде литерала точного числового типа, за которым могут следовать символ «Е» и литерал целого числового типа. Примеры литералов приближенных числовых типов: 123, 123.12, 123Е12, 123.12Е12. Литеральное выражение  $xЕy$  представляет значение  $x * (10^y)$ .

### 11.2.3. Типы символьных строк

В SQL определены три параметризуемых типа символьных строк: **CHARACTER** (или **CHAR**), **CHARACTER VARYING** (или **CHAR VARYING**, или **VARCHAR**) и **CHARACTER LARGE OBJECT** (или **CLOB**). В контексте локализации SQL-ориентированной СУБД (средства локализации входят в стандарт языка) можно определить еще три типа символьных строк — **NATIONAL CHARACTER**, **NATIONAL CHARACTER VARYING** и **NATIONAL CHARACTER LARGE OBJECT**. Аспекты интернационализации и локализации составляют отдельное измерение языка и здесь не обсуждаются.

Значениями типа **CHARACTER** являются символьные строки. Конкретный набор допустимых символов определяется в реализации, но, как правило, включает набор символов ASCII. При определении столбца допускается использование спецификаций **CHARACTER (x)** и просто **CHARACTER**. Последний вариант эквивалентен заданию **CHARACTER (1)**. После определения столбца типа **CHARACTER (x)** СУБД будет резервировать место для хранения  $x$  символов этого столбца во всех строках соответствующей таблицы. Если, например, определен столбец типа **CHARACTER (8)** и в некоторой строке таблицы в него заносится символьная строка длиной пять, то реально будут храниться восемь символов, последние три из которых будут пробелами.

При определении столбца допускается использование спецификаций **CHARACTER VARYING (x)** и просто **CHARACTER VARYING**. Последний вариант эквивалентен заданию **CHARACTER VARYING (1)**. Если в некоторой таблице определяется столбец типа **CHARACTER VARYING (x)**, то в каждой строке этой таблицы значение данного столбца будут занимать ровно столько места, сколько требуется для сохранения соответствующей символьной строки (но ни одна такая строка не может состоять более чем из  $x$  символов). Максимально допустимая длина строк постоянного и переменного размера (значение параметра  $x$ ) определяется в реализации.

Тип данных **CHARACTER LARGE OBJECT** предназначен для определения столбцов, хранящих большие и разные по размеру

группы символов. При определении столбца задается спецификация CLOB (z), где z задает максимальный размер соответствующей группы символов. Максимально возможное значение параметра z определяется в реализации, но, очевидно, что оно должно быть существенно больше максимально возможного значения параметра x, присутствующего в типах CHAR и CHAR VARYING. Поскольку значения z могут быть очень большими, допускается сокращенная форма их задания в виде nK, nM и nG, где n — положительное целое число, а K, M и G означают *кило*, *мега* и *гига* соответственно.

В стандарте SQL определен ряд операций, которые можно выполнять над символьными строками. Перечислим некоторые из них.

- Операция конкатенации (обозначается в виде «||») возвращает символьную строку, произведенную путем соединения строк-операндов в том порядке, как они заданы.
- Функция выделения подстроки (SUBSTRING) принимает три аргумента — строку, номер начальной позиции и длину — и возвращает строку, выделенную из строки-аргумента в соответствии со значениями двух последних параметров.
- Функция UPPER возвращает строку, в которой все малые буквы строки-аргумента заменяются прописными буквами. Функция LOWER, наоборот, заменяет в заданной строке все прописные буквы на малые буквы.
- Функция определения длины (CHARACTER\_LENGTH, OCTET\_LENGTH, BIT\_LENGTH) возвращает длину заданной символьной строки в символах, октетах или битах (в зависимости от вида вычисляющей функции) в виде целого числа.
- Функция определения позиции (POSITION) определяет первую позицию в строке S, с которой в нее входит заданная строка S1 (если не входит, то возвращается нуль).

Литералы типов символьных строк представляются в виде последовательностей символов, заключенных в одинарные или двойные кавычки. В первом случае среди набора символов литерала допускается наличие символов двойной кавычки, а во втором — символов одинарной кавычки. Примеры литералов символьных строк: 'ABCDEF', 'Ab"Ctd', "Fbcdef", "ab'cdtF".

#### 11.2.4. Типы битовых строк

В SQL определены три параметризуемых типа битовых строк: BIT, BIT VARYING и BINARY LARGE OBJECT (или BLOB).

Значениями типа **BIT** являются битовые строки. При определении столбца допускается использование спецификаций **BIT (x)** и просто **BIT**. Последний вариант эквивалентен заданию **BIT (1)**. После определения столбца типа **BIT (x)** СУБД будет резервировать место для хранения  $x$  бит этого столбца во всех строках соответствующей таблицы.

При определении столбца типа битовой строки переменного размера допускается использование только спецификации без умолчания вида **BIT VARYING (x)**, где значение  $x$  определяет максимальную длину битовой строки, которую можно хранить в данном столбце.

Тип данных предназначен для определения столбцов, хранящих большие и разные по размеру группы байтов. При определении столбца задается спецификация **BLOB (z)**, где  $z$  задает максимальный размер соответствующей группы байтов. С технической точки зрения типы **CLOB** и **BLOB** очень похожи. Их разделение требуется для того, чтобы подчеркнуть, что значения типа **CLOB** состоят из символов (в частности, в них может осмысленно производиться текстовый поиск), а значения типа **BLOB** состоят из произвольных байтов, не обязательно кодирующих символы.

Над битовыми строками определен ряд операций, аналогичных операциям над строками символов, обсуждавшимся в подразд. 11.2.3.

Литералы типов битовых строк представляются как заключенные в одинарные кавычки последовательности символов «0» и «1», предваряемые символом «B»; или предваряемые символом «X» последовательности символов, которые изображают шестнадцатеричные цифры (за цифрой «9» следуют «A», «B», «C», «D», «E» и «F»). Примеры литералов типов битовых строк: **B'01110011110001111111'**, **X'78FBCD0012FFFFA'**. Заметим, что в литерале **BLOB** всегда должно содержаться четное число шестнадцатеричных цифр.

### 11.2.5. Типы даты и времени

Возможность сохранения в базе данных информации о дате и времени очень важна с практической точки зрения. Достаточно вспомнить взбудоражившую весь мир «проблему 2000-го года», одним из основных источников которой было некорректное хранение дат в базах данных. В стандарте SQL поддержке средств работы с датой и временем уделяется большое внимание. В частности, поддерживаются специальные «темпоральные»

типы данных DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE и TIMESTAMP WITH TIME ZONE.

**Тип даты.** Значения типа DATE состоят из компонентов-значений года, месяца и дня некоторой даты. Значение года состоит ровно из четырех десятичных цифр и соответствует летоисчислению от Рождества Христова до 9999 г. Значение месяца состоит из двух десятичных цифр и варьируется от 01 до 12. Значение номера дня месяца состоит из двух десятичных цифр и варьируется от 01 до 31, хотя значение месяца даты может накладывать ограничения на возможность использования значений дня месяца 29, 30 и 31. В стандарте SQL не накладываются какие-либо ограничения на внутренний способ представления дат, используемый в реализации. При определении столбца типа DATE указывается просто DATE.

Литералы типа DATE представляются в виде строки DATE 'yyyy-mm-dd', где символы «y», «m» и «d» должны изображать десятичные числа. Например, литерал DATE '1949-04-08' представляет дату 8 апреля 1949 г.

**Типы времени.** Значения параметризованного типа TIME состоят из компонентов-значений часа, минуты и секунды некоторого времени суток. Значение часа состоит ровно из двух десятичных цифр и варьируется от 00 до 23. Значение минуты состоит из двух десятичных цифр и варьируется от 00 до 59. Основное значение секунды также состоит из двух цифр, но может включать дополнительные цифры, представляющие доли секунды. Так что в целом значение секунды варьируется от 00 до 61.999... В значении времени присутствуют две лишние секунды, поскольку Всемирная служба времени иногда добавляет 2 с к последней минуте года для синхронизации мирового времени с реальным временем. Решение о поддержке этих «високосных» секунд принимается на уровне реализации. Число цифр в доле секунды также определяется в реализации. В стандарте требуется только то, чтобы это число было не меньше шести. При определении столбца типа TIME может указываться TIME (p) (значение p задает точность долей секунды) или просто TIME (в этом случае доли секунды не учитываются).

Литералы типа TIME представляются в виде строки TIME 'hh:mm:ss:f', где символы «h», «m», «s» и «f» должны изображать десятичные числа. Например, литерал TIME '16:33-20:333' представляет время суток 16 ч 33 мин 20 целых и 333 тысячных секунды.

**Типы временной метки.** Значения параметризованного типа TIMESTAMP состоят из компонентов-значений года,

месяца и дня некоторой даты, а также компонентов-значений часа, минуты и секунды некоторого времени суток (т.е. каждое значение задает некоторую абсолютную временную метку — отсюда название типа **TIMESTAMP**). Число десятичных цифр в значениях-компонентах и ограничения этих значений такие же, как у значений типов **DATE** и **TIME**. При определении столбца типа **TIMESTAMP** может указываться **TIMESTAMP (p)** (значение **p** задает точность долей секунды) или просто **TIMESTAMP** (в этом случае, в отличие от типа данных **TIME**, по умолчанию принимается, что в доли секунды используются шесть десятичных цифр). Максимально допустимое значение **p** определяется в реализации.

Литералы типа **TIMESTAMP** представляются в виде строки **TIMESTAMP 'yyyy-mm-dd hh:mm:ss:f...f'**, где символы «y», «m», «d», «h», «m», «s» и «f» должны изображать десятичные числа. Например, литерал **TIMESTAMP '1949-04-08 16:33-20:333'** представляет временную метку 16 ч 33 мин 20 целых и 333 тысячных секунды 8 апреля 1949 г.

**Типы времени и временной метки с временной зоной.** Тип данных **TIME WITH TIME ZONE** является таким же, как и тип **TIME**, с тем отличием, что значения типа **TIME WITH TIME ZONE** включают дополнительный компонент-значение, характеризующее смещение соответствующего времени от гринвичского времени (теперь его называют **UTC** — *universal time coordinated*). Не будем касаться деталей представления этого дополнительного компонента.

Тип данных **TIMESTAMP WITH TIME ZONE** является таким же, как и тип **TIMESTAMP**, с тем отличием, что значения типа **TIMESTAMP WITH TIME ZONE** включают дополнительный компонент-значение, характеризующее смещение соответствующего времени от гринвичского времени.

### 11.2.6. Типы временных интервалов

Вообще говоря, *временным интервалом* называется разность между двумя значениями даты или времени. В **SQL** определены две категории типов временных интервалов: «год-месяц» и «день-время суток». Временные интервалы языка **SQL** не привязываются к начальному и/или конечному значению даты/времени, а описывают только протяженность во времени. В общем случае при определении столбца типа временного интервала указывается **INTERVAL start (p) [ TO end (q) ]**, где в качестве **start** и **end** могут задаваться **YEAR**, **MONTH**, **DAY**, **HOURL**, **MINUTE** и **SECOND**.



Параметр *p* задает требуемую точность лидирующего поля интервала (число десятичных цифр). Параметр *q* может задаваться только в том случае, когда в качестве *end* используется **SECOND**, и указывает точность долей секунды. Более точно, допускаются следующие вариации типов временных интервалов.

- Типы категории «год-месяц». Можно определить столбцы следующих типов: **INTERVAL YEAR**, **INTERVAL YEAR (p)** (значения этих типов — временные интервалы в годах), **INTERVAL MONTH**, **INTERVAL MONTH (p)** (значения этих типов — временные интервалы в месяцах), **INTERVAL YEAR TO MONTH**, **INTERVAL YEAR (p) TO MONTH** (значения этих типов — временные интервалы в годах и месяцах). Если значение параметра *p* не указывается явно, по умолчанию принимается его значение 2.
- Типы категории «день-время суток». При определении столбца можно использовать следующие комбинации (для полноты перечислим все возможности): **INTERVAL DAY (p)**, **INTERVAL DAY, INTERVAL DAY (p) TO HOUR**, **INTERVAL DAY TO HOUR**, **INTERVAL DAY (p) TO MINUTE**, **INTERVAL DAY TO MINUTE**, **INTERVAL DAY (p) TO SECOND (q)**, **INTERVAL DAY TO SECOND (q)**, **INTERVAL DAY (p) TO SECOND**, **INTERVAL DAY TO SECOND**, **INTERVAL HOUR (p)**, **INTERVAL HOUR, INTERVAL HOUR (p) TO MINUTE**, **INTERVAL HOUR TO MINUTE**, **INTERVAL HOUR (p) TO SECOND (q)**, **INTERVAL HOUR TO SECOND (q)**, **INTERVAL HOUR TO SECOND**, **INTERVAL MINUTE (p)**, **INTERVAL MINUTE, INTERVAL MINUTE (p) TO SECOND (q)**, **INTERVAL MINUTE TO SECOND (q)**, **INTERVAL MINUTE (p) TO SECOND**, **INTERVAL MINUTE TO SECOND**, **INTERVAL SECOND (p, q)**, **INTERVAL SECOND (p)**, **INTERVAL SECOND**. Если значение параметра *p* не указывается явно,

Таблица 11.1. Операции темпоральных типов данных

| Тип первого операнда | Операция | Тип второго операнда | Тип результата |
|----------------------|----------|----------------------|----------------|
| Datetime             | –        | Datetime             | Interval       |
| Datetime             | + или –  | Interval             | Datetime       |
| Interval             | +        | Datetime             | Datetime       |
| Interval             | + или –  | Interval             | Interval       |
| Interval             | * или /  | Numeric              | Interval       |
| Numeric              | *        | Interval             | Interval       |

по умолчанию принимается его значение 2. Значением по умолчанию параметра *q* является 6.

Приведем только один пример литерала одной из разновидностей типа **INTERVAL**: **INTERVAL '10:20' MINUTE TO SECOND** — временной интервал в 10 мин и 20 с.

Над значениями темпоральных типов могут выполняться арифметические операции, смысл которых определяется табл. 11.1.

11.2.7. Булевский тип

При определении столбца булевского типа указывается просто спецификация **BOOLEAN**. Булевский тип состоит из трех значений: *true*, *false* и *unknown* (соответствующие литералы

| AND     | TRUE    | FALSE | UNKNOWN |
|---------|---------|-------|---------|
| TRUE    | TRUE    | FALSE | UNKNOWN |
| FALSE   | FALSE   | FALSE | FALSE   |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

| OR      | TRUE | FALSE   | UNKNOWN |
|---------|------|---------|---------|
| TRUE    | TRUE | TRUE    | TRUE    |
| FALSE   | TRUE | FALSE   | UNKNOWN |
| UNKNOWN | TRUE | UNKNOWN | UNKNOWN |

| XOR     | TRUE  | FALSE | UNKNOWN |
|---------|-------|-------|---------|
| TRUE    | TRUE  | FALSE | FALSE   |
| FALSE   | FALSE | TRUE  | FALSE   |
| UNKNOWN | FALSE | FALSE | TRUE    |

| NOT     |         |
|---------|---------|
| TRUE    | FALSE   |
| FALSE   | TRUE    |
| UNKNOWN | UNKNOWN |

Рис. 11.2. Таблицы истинности основных логических операций в трехзначной логике SQL

обозначаются TRUE, FALSE и UNKNOWN). В стандарте SQL:2003 имеется следующее уточнение: «В этой спецификации не проводится различие между NULL-значением булевского типа данных и истинностным значением *unknown*, являющимся результатом вычисления предиката, условия поиска или булевского выражения; они могут использоваться взаимозаменяемо и означают в точности одно и то же». Такой подход во многом является некорректным, но здесь не будем на этом останавливаться.

Поддерживается возможность построения булевских выражений, которые вычисляются в трехзначной логике. Необычной особенностью булевских выражений SQL является то, что в них нельзя в качестве элементарных составляющих использовать литералы булевского типа. Вместо этого требуется использовать вызовы специальной двухместной булевой функции IS. Таблицы истинности основных логических операций показаны на рис. 11.2.

### 11.2.8. Типы коллекций

Начиная с SQL:1999, в языке поддерживается возможность использования типов данных, значения которых являются коллекциями значений некоторых других типов. Обычно под термином *коллекция* понимается одно из следующих образований: *массив*, *список*, *множество* и *мультимножество*. В варианте SQL:1999, принятом в 1999 г., были специфицированы только типы массивов. В стандарте SQL:2003 появилась спецификация типа мультимножества.

**Типы массивов.** Любой возможный тип массива получается путем применения *конструктора типов* ARRAY. При определении столбца, значения которого должны принадлежать некоторому типу массива, используется конструкция *dt* ARRAY [*mc*], где *dt* специфицирует некоторый допустимый в SQL тип данных, а *mc* является литералом некоторого точного числового типа с нулевой длиной шкалы и определяет максимальное число элементов в значении типа массива (в терминологии SQL:1999 это значение называется *максимальной кардинальностью* массива). В стандарте SQL:1999 не поддерживались многомерные массивы и массивы массивов. Однако в стандарте SQL:2003 это ограничение было снято, и теперь типом элементов любого типа коллекций теперь может быть любой допустимый в SQL тип данных, кроме самого конструируемого типа коллекции.

Элементам каждого значения типа массива соответствуют их порядковые номера, называемые *индексами*. Значение ин-

декса всегда должно принадлежать отрезку  $[1, mc]$ . Значениями типа массива `dt ARRAY [mc]` являются все те массивы, состоящие из элементов типа `dt`, максимальное значение индекса которых `cs` не превосходит значения `mc`. При сохранении в базе данных значение типа массива занимает столько памяти, сколько требуется для сохранения `cs` элементов. Обеспечивается доступ к элементам массива по их индексам. В частности, можно объявить столбец типа `INTEGER ARRAY [10]` и при вставке строки в соответствующую таблицу задать значение только пятого элемента массива. Тогда в строку будет занесен массив из пяти элементов, причем первые четыре элемента будут содержать неопределенное значение (`NULL`).

Основными операциями над массивами являются выборка значения элемента массива по его индексу, изменение некоторого элемента массива или массива целиком и конкатенация (сцепление) двух массивов. Кроме того, для любого значения типа массива можно узнать значение его `cs`.

**Типы мультимножеств.** При определении столбца таблицы типа мультимножеств используется конструкция `dt MULTiset`, где `dt` задает тип данных элементов конструируемого типа мультимножеств. Значениями типа мультимножеств являются мультимножества, т. е. неупорядоченные коллекции элементов одного и того же типа, среди которых допускаются дубликаты. Например, значениями типа `INTEGER MULTiset` являются мультимножества, элементами которых являются целые числа. Примером такого значения может быть мультимножество `{12, 34, 12, 45, -64}`.

В отличие от массива мультимножество является неограниченной коллекцией; при конструировании типа мультимножеств не указывается предельная кардинальность значений этого типа. Однако это не означает, что возможность вставки элементов в мультимножество действительно не ограничена; стандарт всего лишь не требует наличия границы. Ситуация аналогична той, которая возникает при работе с таблицами, для которых в SQL не объявляется максимально допустимое число строк. (Конечно, на практике такие ограничения устанавливаются в документации конкретной используемой СУБД либо даже администратором конкретной базы данных.)

Для типов мультимножеств поддерживаются операции для преобразования типа значения-мультимножества к типу массивов или другому типу мультимножеств с совместимым типом элементов (операция `CAST`), для удаления дубликатов из мультимножества (функция `SET`), для определения числа элементов

в заданном мультимножестве (функция **CARDINALITY**), для выборки элемента мультимножества, содержащего в точности один элемент (функция **ELEMENT**). Кроме того, для мультимножеств обеспечиваются операции объединения (**MULTISET UNION**), пересечения (**MULTISET INTERSECT**) и определения разности (**MULTISET EXCEPT**). Каждая из операций может выполняться в режиме с сохранением дубликатов (режим **ALL**) или с устранением дубликатов (режим **DISTINCT**).

Расширенные в SQL:2003 возможности работы с типами коллекций являются принципиально важными. Даже при наличии определяемых пользователями типов данных (см. далее) и типов массивов SQL:1999 не предоставлял полных возможностей для преодоления исторически присущего реляционной модели данных вообще и SQL в частности ограничения «плоских таблиц». После появления конструктора типов мультимножеств и устранения ограничений на тип данных элементов коллекции это историческое ограничение полностью ликвидировано. Мультимножество, типом элементов которого является анонимный строчный тип (см. далее), является полным аналогом таблицы. Тем самым, в базе данных допускается произвольная вложенность таблиц. Возможности выбора структуры базы данных безгранично расширяются.

### 11.2.9. Анонимные строчные типы

Анонимный строчный тип — это тип данных, получаемый путем применения конструктора типов **ROW**, позволяющего производить безымянные типы строк (кортежей). Любой возможный строчный тип получается путем использования конструктора **ROW**. При определении столбца, значения которого должны принадлежать некоторому строчному типу, используется конструкция **ROW (*fld*<sub>1</sub>, *fld*<sub>2</sub>, ..., *fld*<sub>*n*</sub>)**, где каждый элемент *fld*<sub>*i*</sub>, определяющий поле строчного типа, задается в виде тройки ***fldname*, *fldtype*, *fldoptions***. Подэлемент *fldname* задает имя соответствующего поля строчного типа. Подэлемент *fldtype* специфицирует тип данных этого поля. В качестве типа данных поля строчного типа можно использовать любой допустимый в SQL тип данных, включая типы коллекций, определяемые пользователями типы и другие строчные типы. Необязательный подэлемент *fldoptions* может задаваться для указания применяемого по умолчанию порядка сортировки, если соответствующий подэлемент *fldtype* указывает на тип символьных строк, а также должен задаваться, если *fldtype* указывает на *ссылочный тип*

(см. подразд. 11.2.11). *Степенью* строчного типа называется число его полей.

### 11.2.10. Типы, определяемые пользователем

Эта категория типов данных связана с объектными расширениями языка SQL. Синтаксические конструкции SQL, позволяющие определять пользовательские типы, довольно сложны, и их подробное рассмотрение выходит за границы материала этой книги. Ограничимся здесь кратким обзором (подробнее см. [38]).

**Структурные типы** (Structured Types). Соответствующие возможности SQL:1999 позволяют определять долговременно хранимые, именованные типы данных, включающие один или более атрибутов любого из допустимых в SQL типа данных, в том числе другие структурные типы, типы коллекций, строчные типы и т. д. Стандарт SQL не накладывает ограничений на сложность получаемой в результате структуры данных, однако не запрещает устанавливать такие ограничения в реализации. Дополнительные механизмы определяемых пользователями методов, функций и процедур позволяют определить поведенческие аспекты структурного типа. При определении нового структурного типа данных можно использовать механизм одиночного наследования от ранее определенного структурного типа.

**Индивидуальные типы** (Distinct Types). Можно определить долговременно хранимый, именованный тип данных, опираясь на единственный предопределенный тип. Например, можно определить индивидуальный тип данных **PRICE**, опираясь на тип **DECIMAL (5, 2)**. Тогда значения типа **PRICE** представляются точно так же, как и значения типа **DECIMAL (5, 2)**. Однако в SQL:1999 индивидуальный тип не наследует от своего опорного типа набор операций над значениями. Например, чтобы сложить два значения типа **PRICE**, требуется явно сообщить системе, что с этими значениями нужно обращаться как со значениями типа **DECIMAL (5, 2)**. Другая возможность состоит в явном определении методов, функций и процедур, связанных с данным индивидуальным типом. Похоже, что в будущих версиях стандарта появятся и другие, более удобные возможности.

### 11.2.11. Ссылочные типы

Эта категория типов данных связана с объектными расширениями языка SQL, и мы снова опустим подробное описание

этого механизма и обсудим его очень коротко. Обеспечивается механизм конструирования типов (*ссылочных типов*), которые могут использоваться в качестве типов столбцов некоторого вида таблиц (*типизированных таблиц*). Фактически, значениями ссылочного типа являются строки соответствующей типизированной таблицы. Более точно, каждой строке типизированной таблицы назначается уникальное значение (нечто вроде первичного ключа, назначаемого системой или приложением), которое может использоваться в методах, определенных для табличного типа, для уникальной идентификации строк соответствующей таблицы. Эти уникальные значения называются *ссылочными значениями*, а их тип — *ссылочным типом*. Ссылочный тип может содержать только те значения, которые действительно ссылаются на экземпляры указанного типа (т.е. на строки соответствующей типизированной таблицы).

### 11.3. Средства определения доменов

Как уже отмечалось, при определении столбцов таблицы требуется явно указывать тип данных каждого столбца. Для этого можно использовать описанные выше средства спецификации типа. Но в SQL поддерживается и другой механизм, механизм доменов, которые мало связаны с доменами, упоминающимися в реляционной модели данных. Домен является долговременно хранимым, именованным объектом схемы базы данных. Домены можно создавать (определять), изменять (изменять определения) и ликвидировать (отменять определение). Имена доменов можно использовать при определении столбцов таблиц. Можно считать, что в SQL определение домена представляет собой вынесенное за пределы определения индивидуальной таблицы «родовое» определение столбца, которое можно использовать для определения различных реальных столбцов базовых таблиц.

#### 11.3.1. Определение домена

Для определения домена в SQL используется оператор **CREATE DOMAIN**. Общий синтаксис этого оператора следующий:

```
domain_definition ::= CREATE DOMAIN domain_name
[AS] data_type
        [ default_definition ]
        [ domain_constraint_definition_list ]
```

Здесь **domain\_name** задает имя создаваемого домена\*, **data\_type** — спецификация определяющего типа данных. В необязательных разделах **default\_definition** и **domain\_constraint\_definition\_list** специфицируются значение домена по умолчанию, используемое в дальнейшем в качестве значения по умолчанию для любого столбца, определенного на данном домене, если для этого столбца не определено собственное значение по умолчанию, и набор ограничений целостности, которые будут применяться к любому столбцу, определенному на этом домене.

Имеются три возможности определения значения по умолчанию. Для этого можно использовать литерал базового типа данных домена, обозначение неопределенного значения **NULL** и вызов одной из предопределенных в SQL функций без параметров, например, **CURRENT\_USER**, возвращающей идентификатор текущего пользователя.

Если в операторе **CREATE DOMAIN** значение по умолчанию не специфицируется, считается, что такого значения нет. Однако позже к определению домена можно добавить раздел значения по умолчанию с помощью оператора **ALTER DOMAIN**. Кроме того, этот оператор позволяет удалить раздел значения по умолчанию из существующего определения домена.

Элемент списка **domain\_constraint\_definition\_list** имеет вид

```
[ CONSTRAINT constraint_name ] CHECK  
( conditional_expression )
```

Необязательный раздел **CONSTRAINT constraint\_name** позволяет определить имя нового ограничения целостности. Если явное указание имени отсутствует, ограничению назначается имя, автоматически генерируемое системой. Что касается вида условного выражения, служащего собственно ограничением целостности, то в стандарте запрещается лишь прямое или косвенное использование в нем домена, в определение которого входит данное условное выражение. Однако наиболее естественным (и наиболее распространенным) видом ограничения домена является следующий вид:

```
CHECK (VALUE IN (list_of_valid_values))
```

---

\* В общем случае имена объектов SQL-ориентированных баз данных имеют вид **имя\_каталога.имя\_схемы.имя\_объекта**. Этот подход к именованию объектов базы данных позволяет независимо создавать объекты в разных схемах, не заботясь о том, чтобы эти объекты имели разные простые имена. При использовании в операторе SQL простого имени объекта система должна автоматически уточнить это имя, исходя из идентификатора пользователя, от имени которого выполняется оператор.



Такое ограничение запрещает появление в любом столбце, определенном на данном домене, любого значения определяющего типа, не входящего в список допустимых значений.

В дальнейших примерах нам понадобятся определения нескольких доменов. Приведем их в этом подразделе. В примерах будем иметь дело с таблицами служащих (**EMP**), отделов (**DEPT**) и проектов (**PRO**). Каждый служащий обладает уникальным номером (**EMP\_NO**) и получает заработную плату (**SALARY**). Определим домены **EMP\_NO** и **SALARY**.

```
CREATE DOMAIN EMP_NO AS INTEGER  
CHECK (VALUE BETWEEN 1 AND 10000);
```

Номера служащих являются целыми числами, поэтому базовым типом домена **EMP\_NO** является тип **INTEGER**. Кроме того, на значения этого домена устанавливается то ограничение, что они должны быть больше нуля и не должны превосходить целое значение 10000.

Домен **SALARY** определим следующим образом:

```
CREATE DOMAIN SALARY AS NUMERIC (10, 2)  
DEFAULT 10000.00  
CHECK (VALUE BETWEEN 10000.00 AND 20000000.00)  
CONSTRAINT SAL_NOT_NULL CHECK (VALUE IS NOT NULL);
```

Размер заработной платы является значением точного числового типа **NUMERIC** из 10 десятичных цифр, две из которых составляют дробную часть. По умолчанию размер заработной платы составляет 10 000 руб. Установлен диапазон допустимого размера заработной платы от 10 000 руб. до 20 000 000 руб. Неопределенное значение заработной платы не допускается (на уровне определения домена).

### 11.3.2. Изменение определения домена

Для изменения характеристик ранее определенного домена используется оператор SQL **ALTER DOMAIN**. Синтаксис этого оператора выглядит следующим образом:

```
domain_alteration ::=  
  ALTER DOMAIN domain_default_alteration_action  
  | domain_constraint_alteration_action
```

Как видно, при изменении определения домена можно выполнить действие по изменению раздела значения по умолчанию либо изменить ограничение домена. В первом случае

можно установить новое значение по умолчанию или отменить существующее значение по умолчанию. При установке нового значения по умолчанию это значение автоматически применяется ко всем столбцам, определенным на данном домене. Более точно, во всех строках всех таблиц установленное по умолчанию значение всех столбцов, которые были определены на данном домене, меняется на новое значение по умолчанию (если только при определении столбца не было явно указано его собственное значение по умолчанию). При отмене значения по умолчанию упоминание о значении по умолчанию убирается из хранимых определений всех столбцов, которые были определены на данном домене (если только при определении столбца не было явно указано его собственное значение по умолчанию).

Действие по изменению ограничения домена может состоять в добавлении нового ограничения домена или в отмене существующего ограничения. Добавление нового определения ограничения домена приводит к тому, что новое условие добавляется через **AND** к существующему ограничению домена. Если к моменту выполнения соответствующего оператора **ALTER DOMAIN** существуют столбцы некоторых таблиц, текущие значения которых противоречат новому ограничению, то СУБД должна отвергнуть этот оператор **ALTER DOMAIN**. Действие по отмене ограничения домена (**DROP**) приводит к исчезновению соответствующей части общего ограничения соответствующего домена, что, естественно, не влияет на существующие значения существующих столбцов существующих таблиц.

Для примера немного поупражняемся с доменом **SALARY**. Для изменения значения заработной платы по умолчанию с 10 000 на 11 000 руб. нужно выполнить оператор

```
ALTER DOMAIN SALARY SET DEFAULT 11000.00;
```

Для отмены значения по умолчанию в домене **SALARY** нужно воспользоваться оператором

```
ALTER DOMAIN SALARY DROP DEFAULT;
```

Если к определению домена **SALARY** требуется добавить ограничение (например, запретить значение заработной платы, равное 15 000 руб.), то нужно выполнить оператор

```
ALTER DOMAIN SALARY ADD CHECK (VALUE <> 15000.00);
```

Наконец, если требуется отменить (именованное!) ограничение целостности, препятствующее наличию неопределенных

значений в столбцах, которые определены на домене SALARY, то нужно выполнить оператор

```
ALTER DOMAIN SALARY DROP CONSTRAINT SAL_NOT_NULL;
```

### 11.3.3. Отмена определения домена

Чтобы отменить ранее созданное определение домена, нужно воспользоваться оператором **DROP DOMAIN** в следующем синтаксисе:

```
DROP DOMAIN domain_name {RESTRICT | CASCADES}
```

Если в операторе указано ключевое слово **RESTRICT** и соответствующий домен использован в определении некоторого столбца, некоторого представления или ограничения целостности, то оператор **DROP DOMAIN** отвергается. В противном случае определение домена ликвидируется.

Если в операторе **DROP DOMAIN** указано **CASCADES**, то оператор выполняется всегда. При этом уничтожаются все представления и ограничения целостности, в определении которых использовалось имя данного домена. Столбцы, определенные на этом домене, автоматически переопределяются следующим образом: считается, что каждый такой столбец теперь относится к определяющему типу уничтожаемого домена; если у столбца не было определено собственное значение по умолчанию, то считается, что теперь у него имеется такое значение по умолчанию, совпадающее со значением по умолчанию уничтожаемого домена; каждый столбец наследует все ограничения уничтожаемого домена.

## 11.4. Средства определения, изменения и ликвидации базовых таблиц

Базовые (реально хранимые в базе данных) таблицы создаются (определяются) с использованием оператора **CREATE TABLE**. Для изменения определения базовой таблицы используется оператор **ALTER TABLE**. Уничтожить хранимую таблицу (отменить ее определение) можно с помощью оператора **DROP TABLE**.

*Замечание:* хотя внешне операторы **CREATE TABLE**, **ALTER TABLE** и **DROP TABLE** похожи на соответствующие операторы определения, изменения определения и отмены определения домена, между ними имеется принципиальное различие. Определение домена приводит всего лишь к созданию некоторых новых описателей, входящих в состав метаданных базы данных.

Создание базовой таблицы, кроме создания соответствующих описателей, приводит к выделению новой области внешней памяти, в которой будут храниться данные, порождаемые пользователями. Тем самым, базовая таблица SQL-ориентированной базы данных является прямым аналогом переменной отношения реляционной модели данных.

#### 11.4.1. Определение базовой таблицы

Оператор создания базовой таблицы **CREATE TABLE** имеет следующий синтаксис:

```
base_table_definition ::= CREATE TABLE
    base_table_name (base_table_element_commalist)*
base_table_element ::= column_definition
                    | base_table_constraint_definition
```

Здесь **base\_table\_name** задает имя новой (изначально пустой) базовой таблицы. Каждый элемент определения базовой таблицы является либо определением столбца, либо определением табличного ограничения целостности.

Элемент определения столбца специфицируется на основе следующих синтаксических правил:

```
column_definition ::= column_name { data_type |
domain_name }
                    [ default_definition ]
                    [ column_constraint_definition_list ]
```

В элементе определения столбца **column\_name** задает имя определяемого столбца. Тип столбца специфицируется путем явного указания типа данных (**data\_type**) или путем указания имени ранее определенного домена (**domain\_name**).

При определении значения столбца по умолчанию имеются те же возможности, что и при определении значения по умолчанию в операторах определения или изменения определения домена. Действующее значение по умолчанию для данного столбца определяется следующим образом:

- если в определении столбца явно присутствует раздел **DEFAULT**, то значением столбца по умолчанию является значение, указанное в этом разделе;

---

\* В круглых скобках указывается список определений элементов базовой таблицы (должно присутствовать определение хотя бы одного столбца), разделенных запятыми.

- иначе, если столбец определяется на домене и в определении этого домена явно присутствует раздел **DEFAULT**, то значением столбца по умолчанию является значение, указанное в этом разделе;
- иначе значением по умолчанию столбца является **NULL**\*.

Заметим, что если значением по умолчанию неявно объявлено неопределенное значение (**NULL**), но среди ограничений целостности столбца присутствует ограничение **NOT NULL**, то считается, что у столбца вообще отсутствует значение по умолчанию. Это означает, что при любой вставке новой строки в соответствующую базовую таблицу значение данного столбца должно быть явно задано.

Элемент необязательного списка ограничений целостности столбца определяется следующими синтаксическими правилами:

```
column_constraint_definition ::= [ CONSTRAINT
constraint_name ]
                                NOT NULL
                                | { PRIMARY KEY | UNIQUE }
                                | references_definition
                                | CHECK ( conditional_expression )
```

Как будет показано далее, любое ограничение целостности, включаемое в определение столбца, может быть эквивалентным образом выражено в виде табличного ограничения целостности. Единственный резон определения ограничений на уровне столбца состоит в том, что в этом случае в ограничении целостности не требуется явно указывать имя столбца. Тем не менее кратко обсудим ограничения целостности столбца отдельно.

Ограничение **NOT NULL** означает, что в определяемом столбце никогда не должны содержаться неопределенные значения. Если определяемый столбец имеет имя **C**, то это ограничение эквивалентно следующему табличному ограничению: **CHECK (C IS NOT NULL)**.

В определение столбца может входить одно из ограничений: ограничение первичного ключа (**PRIMARY KEY**) или ограничение возможного ключа (**UNIQUE**). Включение в определение столбца любого из этих ограничений означает требование уникальности

---

\* Заметим, что хотя столбец может получить значение **NULL** по умолчанию как явным, так и неявным образом, эти два случая не являются эквивалентными. Явное задание **NULL** в качестве значения по умолчанию запрещает наследование столбцом значения по умолчанию домена.

значений определяемого столбца (т.е. во все время существования определяемой таблицы во всех ее строках значения данного столбца должны быть различны<sup>\*</sup>). Ограничение **PRIMARY KEY**, в дополнение к этому, влечет ограничение **NOT NULL** для определяемого столбца. Эти ограничения столбца эквивалентны следующим табличным ограничениям: **PRIMARY KEY (C)** и **UNIQUE (C)**.

Ограничение **references\_definition** означает объявление определяемого столбца внешним ключом таблицы и обладает следующим синтаксисом:

```
references_definition ::=  
REFERENCES base_table_name [ (column_commalist) ]  
    [ MATCH { SIMPLE | FULL | PARTIAL } ]  
    [ ON DELETE referential_action ]  
    [ ON UPDATE referential_action ]
```

На самом деле, эта синтаксическая конструкция работает и в случае определения внешнего ключа на уровне таблицы (в одном из определений табличных ограничений целостности). Поэтому отложим обсуждение до рассмотрения этого общего случая. Пока заметим только, что при использовании конструкции на уровне определения столбца **column\_commalist** может содержать имя только одного столбца (потому что внешний ключ состоит из одного определяемого столбца). Ограничение эквивалентно следующему табличному ограничению: **FOREIGN KEY ( C ) references\_definition**.

Определение проверочного ограничения **CHECK (conditional\_expression)** приводит к тому, что в данном столбце могут находиться только те значения, для которых вычисление **conditional\_expression** приводит к результату *true*. В условном выражении проверочного ограничения столбца разрешается использовать имя только определяемого столбца. Заметим, что проверочное ограничение столбца может быть безо всяких изменений перенесено на уровень определения табличных ограничений.

**Определение табличного ограничения.** Элемент определения табличного ограничения целостности задается в следующем синтаксисе:

---

<sup>\*</sup> В этом случае SQL опирается на семантику неопределенных значений, отличную от семантики, используемой в большинстве других случаев. Здесь считается, что  $(NULL = NULL) = true$  и что  $(a = NULL) = (NULL = a) = false$  для любого значения *a*, отличного от **NULL**.

```

base_table_constraint_definition ::= [ CONSTRAINT
constraint_name ]
    { PRIMARY KEY | UNIQUE } ( column_commalist )
    | FOREIGN KEY ( column_commalist )
references_definition
    | CHECK ( conditional_expression )

```

Как видно, имеется три разновидности табличных ограничений: ограничение первичного или возможного ключа (**PRIMARY KEY** или **UNIQUE**), ограничение внешнего ключа (**FOREIGN KEY**) и проверочное ограничение (**CHECK**). Любому ограничению может явным образом назначаться имя, если перед определением ограничения поместить конструкцию **CONSTRAINT constraint\_name**.

Наличие табличного ограничения первичного или возможного ключа { **PRIMARY\_KEY** | **UNIQUE** } (**column\_commalist**) означает требование уникальности составных значений указанной группы столбцов (т. е. во все время существования определяемой таблицы во всех ее строках составные значения данной группы столбцов должны быть различны). Ограничение **PRIMARY KEY**, в дополнение к этому, влечет ограничение **NOT NULL** для всех столбцов, упоминаемых в определении ограничения. В определении таблицы допускается произвольное число определений возможного ключа (для разных комбинаций столбцов), но не более одного определения первичного ключа. Обратите особое внимание на последнюю часть предыдущего предложения. Нет, вы не ошиблись! В языке SQL действительно допускается определение таблиц, у которых отсутствуют возможные ключи. Эта особенность языка, среди прочего, очевидным образом противоречит базовым требованиям реляционной модели данных.

Определение табличного ограничения вида **CHECK (conditional\_expression)** приводит к тому, что указанное условное выражение будет вычисляться при каждой попытке обновления соответствующей таблицы (вставке новой строки, удалении или модификации существующей строки). Считается, что попытка обновления таблицы нарушает проверочное ограничение целостности, если после выполнения операции обновления вычисление условного выражения дает результат *false*. Другими словами, таблица находится в соответствии с данным проверочным табличным ограничением, если для всех строк таблицы результатом вычисления соответствующего условного выражения не является *false*.

Обсуждение допустимых разновидностей условных выражений приводится в следующей главе в контексте рассмотрения оператора **SELECT** языка SQL.

**Табличное ограничение внешнего ключа.** Синтаксис и семантика определения внешнего ключа в операторе SQL определения базовой таблицы являются довольно запутанными и сложными, поэтому этой языковой конструкции посвящается отдельный подраздел.

Табличное ограничение **FOREIGN KEY ( column\_commalist ) references\_definition** означает объявление внешним ключом группы столбцов, имена которых перечислены в списке **column\_commalist**. Обсудим теперь смысл ограничения внешнего ключа при разных вариантах формирования определения ссылок (**references\_definition**). Для удобства повторим синтаксическое правило.

```
references_definition ::=  
    REFERENCES base_table_name [ (column_commalist) ]  
    [ MATCH { SIMPLE | FULL | PARTIAL } ]  
    [ ON DELETE referential_action ]  
    [ ON UPDATE referential_action ]
```

В этом определении **base\_table\_name** должно представлять собой имя некоторой базовой таблицы (пусть, например, эта таблица имеет имя *T*). Если определение ссылок включает список столбцов (**column\_commalist**), то этот список должен совпадать (с точностью до порядка следования имен столбцов) со списком имен столбцов, использованных в некотором определении первичного или возможного ключа (**PRIMARY\_KEY** или **UNIQUE**) в определении таблицы *T*. Если в определении ссылок список столбцов явно не задан, то считается, что он совпадает со списком столбцов, использованных в определении первичного ключа (**PRIMARY\_KEY**) таблицы *T*.

Пусть определяемая таблица имеет имя *S*. Обсудим смысл необязательного раздела определения внешнего ключа **MATCH { SIMPLE | FULL | PARTIAL }**. Если этот раздел отсутствует или если присутствует и имеет вид **MATCH SIMPLE**, то ограничение внешнего ключа (ссылочное ограничение) удовлетворяется в том и только том случае, когда для каждой строки таблицы *S* либо *какой-либо* столбец, входящий в состав внешнего ключа, содержит **NULL**, либо таблица *T* содержит в *точности одну строку* такую, что значение внешнего ключа в данной строке таблицы *S* совпадает со значением соответствующего возможного ключа в этой строке таблицы *T*.

Если раздел **MATCH** присутствует в определении внешнего ключа и имеет вид **MATCH PARTIAL**, то ограничение внешнего ключа удовлетворяется в том и только том случае, когда для каждой строки таблицы *S* либо *каждый* столбец, входящий в со-



став внешнего ключа, содержит **NULL**, либо таблица *T* содержит, по крайней мере, одну такую строку, что для каждого столбца данной строки таблицы *S*, значение которого *отлично от NULL*, его значение совпадает со значением соответствующего столбца возможного ключа в этой строке таблицы *T*.

Если раздел **MATCH** имеет вид **MATCH FULL**, то ограничение внешнего ключа удовлетворяется в том и только том случае, когда для каждой строки таблицы *S* либо *каждый* столбец, входящий в состав внешнего ключа, содержит **NULL**, либо *ни один* столбец, входящий в состав внешнего ключа, не содержит **NULL**, и таблица *T* содержит *в точности одну строку* такую, что значение внешнего ключа в данной строке таблицы *S* совпадает со значением соответствующего возможного ключа в этой строке таблицы *T*.

Легко видеть, что только при наличии спецификации **MATCH FULL** ссылочное ограничение соответствует требованиям реляционной модели. Тем не менее в определении ограничения внешнего ключа базовых таблиц в SQL по умолчанию предполагается наличие спецификации **MATCH SIMPLE**.

В связи с определением ограничения внешнего ключа нам осталось рассмотреть еще два необязательных раздела — **ON DELETE referential\_action** и **ON UPDATE referential\_action**. Прежде всего приведем синтаксическое правило:

```
referential_action ::=  
    { NO ACTION | RESTRICT | CASCADE |  
      SET DEFAULT | SET NULL }
```

Чтобы объяснить, в каких случаях, и каким образом выполняются эти действия, требуется сначала определить понятие *ссылающейся строки* (*referencing row*). Если в определении ограничения внешнего ключа отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** или **MATCH FULL**, то для данной строки *t* таблицы *T* строкой таблицы *S*, ссылающейся на строку *t*, называется каждая строка таблицы *S*, значение внешнего ключа которой совпадает со значением соответствующего возможного ключа строки *t*. Если в определении ограничения внешнего ключа присутствует спецификация **MATCH PARTIAL**, то для данной строки *t* таблицы *T* строкой таблицы *S*, ссылающейся на строку *t*, называется каждая строка таблицы *S*, отличные от **NULL** значения столбцов внешнего ключа которой совпадают со значениями соответствующих столбцов соответствующего возможного ключа строки *t*. В случае **MATCH PARTIAL** строка таблицы *S* называется *ссылающейся исключительно*

на строку  $t$  таблицы  $T$ , если эта строка таблицы  $S$  является ссылающейся на строку  $t$  и не является ссылающейся на какую-либо другую строку таблицы  $T$ .

Теперь приступим к обсуждению ссылочных действий. Пусть определение ограничения внешнего ключа содержит раздел **ON DELETE referential\_action**. Предположим, что предпринимается попытка удалить строку  $t$  из таблицы  $T$ . Тогда, если в качестве требуемого ссылочного действия указано **NO ACTION** или **RESTRICT**, то операция удаления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа. Если в качестве требуемого ссылочного действия указано **CASCADE**, то строка  $t$  удаляется, и если в определении ограничения внешнего ключа отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** или **MATCH FULL**, то удаляются все строки, ссылающиеся на  $t$ . Если же в определении ограничения внешнего ключа присутствует спецификация **MATCH PARTIAL**, то удаляются только те строки, которые ссылаются исключительно на строку  $t$ . Если в качестве требуемого ссылочного действия указано **SET DEFAULT**, то строка  $t$  удаляется и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку  $t$ , проставляется заданное при их определении значение по умолчанию. Если в определении внешнего ключа содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы  $S$ , которые ссылаются исключительно на строку  $t$ . Если в качестве требуемого ссылочного действия указано **SET NULL**, то строка  $t$  удаляется и во всех столбцах, которые входят в состав внешнего ключа, всех строк, ссылающихся на строку  $t$ , проставляется **NULL**. Если в определении внешнего ключа содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы  $S$ , которые ссылаются исключительно на строку  $t$ .

Пусть определение ограничения внешнего ключа содержит раздел **ON UPDATE referential\_action**. Предположим, что предпринимается попытка обновить столбцы соответствующего возможного ключа в строке  $t$  из таблицы  $T$ . Тогда если в качестве требуемого ссылочного действия указано **NO ACTION** или **RESTRICT**, то операция обновления отвергается, если ее выполнение вызвало бы нарушение ограничения внешнего ключа. Если в качестве требуемого ссылочного действия указано **CASCADE**, то строка  $t$  обновляется, и если в определении ограничения внешнего ключа отсутствует раздел **MATCH** или присутствуют спецификации **MATCH SIMPLE** или **MATCH FULL**, то соответствующим образом

обновляются все строки, ссылающиеся на  $t$  (в них должным образом изменяются значения столбцов, входящих в состав внешнего ключа). Если же в определении ограничения внешнего ключа присутствует спецификация **MATCH PARTIAL**, то обновляются только те строки, которые ссылаются исключительно на строку  $t$ . Если в качестве требуемого ссылочного действия указано **SET DEFAULT**, то строка  $t$  обновляется и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы  $T$ , всех строк, ссылающихся на строку  $t$ , проставляется заданное при их определении значение по умолчанию. Если в определении внешнего ключа содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы  $S$ , которые ссылаются исключительно на строку  $t$ , причем в них изменяются значения только тех столбцов, которые не содержали **NULL**. Если в качестве требуемого ссылочного действия указано **SET NULL**, то строка  $t$  обновляется и во всех столбцах, которые входят в состав внешнего ключа и соответствуют изменяемым столбцам таблицы  $T$ , всех строк, ссылающихся на строку  $t$ , проставляется **NULL**. Если в определении внешнего ключа содержится спецификация **MATCH PARTIAL**, то подобному воздействию подвергаются только те строки таблицы  $S$ , которые ссылаются исключительно на строку  $t$ .

**Примеры определений базовых таблиц.** Определим таблицы служащих (**EMP**), отделов (**DEPT**) и проектов (**PRO**). Эти таблицы имеют заголовки, показанные на рис. 11.3.

Столбцы **EMP\_NO**, **EMP\_SAL**, **DEPT\_NO**, **PRO\_NO**, **DEPT\_TOTAL\_SAL**, **DEPT\_MNG** и **PRO\_MNG** определяются на ранее определенных доменах (определения доменов **EMP\_NO** и **SALARY** приведены в предыдущем подразделе). Первичными ключами отношений **EMP**, **DEPT** и проектов **PRO** являются столбцы **EMP\_NO**, **DEPT\_NO** и **PRO\_NO** соответственно. В таблице **EMP** столбцы **DEPT\_NO** и **PRO\_NO** являются внешними ключами, указывающими на отдел, в котором работает служащий, и на выполняемый им проект соответственно. В таблице **DEPT** внешним ключом является столбец **DEPT\_NO**, указывающий на служащего, являющегося руководителем соответствующего отдела, а в таблице **PRO** внешним ключом является столбец **PRO\_MNG**, указывающий на служащего, являющегося менеджером соответствующего проекта. Другие ограничения целостности будут приведены далее.

Определим таблицу **EMP**:

- (1) `CREATE TABLE EMP (`
- (2) `EMP_NO EMP_NO PRIMARY KEY,`

**EMP:**

|                    |
|--------------------|
| EMP_NO : EMP_NO    |
| EMP_NAME : VARCHAR |
| EMP_BDATE : DATE   |
| EMP_SAL : SALARY   |
| DEPT_NO : DEPT_NO  |
| PRO_NO : PRO_NO    |

**DEPT:**

|                         |
|-------------------------|
| DEPT_NO : DEPT_NO       |
| DEPT_NAME : VARCHAR     |
| DEPT_EMP_NO : INTEGER   |
| DEPT_TOTAL_SAL : SALARY |
| DEPT_MNG : EMP_NO       |

**PRO:**

|                      |
|----------------------|
| PRO_NO : PRO_NO      |
| PRO_TITLE : VARCHAR  |
| PRO_SDATE : DATE     |
| PRO_DURAT : INTERVAL |
| PRO_MNG : EMP_NO     |
| PRO_DESC : CLOB      |

Рис. 11.3. Заголовки таблиц EMP, DEPT и PRO

- (3) EMP\_NAME VARCHAR(20) DEFAULT 'Incognito'  
NOT NULL,
- (4) EMP\_BDATE DATE DEFAULT NULL CHECK  
(VALUE >= DATE '1917-10-24'),
- (5) EMP\_SAL SALARY,
- (6) DEPT\_NO DEPT\_NO DEFAULT NULL REFERENCES  
DEPT ON DELETE SET NULL,
- (7) PRO\_NO PRO\_NO DEFAULT NULL,
- (8) FOREIGN KEY PRO\_NO REFERENCES PRO (PRO\_NO)

```
ON DELETE SET NULL,  
(9) CONSTRAINT PRO_EMP_NO CHECK  
    ((SELECT COUNT (*) FROM EMP E  
    WHERE E.PRO_NO = PRO_NO) <= 50));
```

Последовательно обсудим части этого определения. В части (1) указывается, что создается таблица с именем **EMP**. В части (2) определяется столбец **EMP\_NO** на домене **EMP\_NO**. У этого столбца не определено значение по умолчанию, и он объявлен первичным ключом таблицы (это ограничение целостности добавляется через **AND** к ограничениям, унаследованным столбцом от определения домена). Помимо прочего, это означает неявное указание запрета для этого столбца неопределенных значений. В части (3) определен столбец **EMP\_NAME** на базовом типе данных символьных строк переменной длины с максимальной длиной 20. Для столбца указано значение по умолчанию — строка 'Incognito', и в качестве ограничения целостности запрещены неопределенные значения. В части (4) определяется столбец **EMP\_BDATE** (дата рождения служащего). Он имеет тип данных **DATE**, значением по умолчанию является **NULL** (даты рождения некоторых служащих неизвестны). Кроме того, ограничение столбца запрещает принимать на работу лиц, про которых известно, что они родились до Октябрьского переворота. В части (5) определен столбец **EMP\_SAL** на домене **SALARY**. Значение по умолчанию и ограничения целостности наследуются из определения домена. В части (6) столбец **DEPT\_NO** определяется на одноименном домене (для наших целей его определение несущественно), но явно объявляется, что значением по умолчанию этого столбца будет **NULL** (некоторые служащие не приписаны ни к какому отделу). Кроме того, добавляется ограничение внешнего ключа: столбец **DEPT\_NO** ссылается на первичный ключ таблицы **DEPT**. Определено ссылочное действие: при удалении строки из таблицы **DEPT** во всех строках таблицы **EMP**, ссылавшихся на эту строку, столбцу **DEPT\_NO** должно быть присвоено неопределенное значение. В части (7) определяется столбец **PRO\_NO**. Его определение аналогично определению столбца **DEPT\_NO**, но ограничение внешнего ключа вынесено в часть (8), где оно определяется в полной форме как табличное ограничение. Наконец, в части (9) определяется табличное проверочное ограничение с именем **PRO\_EMP\_NO**, которое требует, чтобы ни в одном проекте не работало больше 50 служащих (правила построения соответствующего условного выражения поясняются в гл. 12).

Определим таблицу DEPT:

```
(1) CREATE TABLE DEPT (  
(2) DEPT_NO DEPT_NO PRIMARY KEY,  
(3) DEPT_EMP_NO INTEGER NO NULL CHECK  
    (VALUE BETWEEN 1 AND 100),  
(4) DEPT_NAME VARCHAR(200) DEFAULT 'Nameless'  
    NOT NULL,  
(5) DEPT_TOTAL_SAL SALARY DEFAULT 1000000.00  
    NO NULL CHECK (VALUE > = 100000.00),  
(6) DEPT_MNG EMP_NO DEFAULT NULL  
    REFERENCES EMP ON DELETE SET NULL  
    CHECK (IF (VALUE IS NOT NULL) THEN  
        ((SELECT COUNT(*) FROM DEPT  
            WHERE DEPT.DEPT_MNG = VALUE) = 1)),  
(7) CHECK (DEPT_EMP_NO =  
    (SELECT COUNT(*) FROM EMP  
        WHERE DEPT_NO = EMP.DEPT_NO)),  
(8) CHECK (DEPT_TOTAL_SAL >=  
    (SELECT SUM(EMP_SAL) FROM EMP  
        WHERE DEPT_NO = EMP.DEPT_NO));
```

Это определение обсудим в менее систематической манере, чем предыдущее. Отметим только наиболее интересные моменты. В части (3) столбец **DEPT\_EMP\_NO** (число служащих в отделе) определен на базовом типе **INTEGER** без значения по умолчанию, с запретом неопределенного значения и с проверочным ограничением, устанавливающим допустимый диапазон значений числа служащих в отделе. Еще одно проверочное ограничение этого столбца — (7) — вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы **DEPT** значение столбца **DEPT\_EMP\_NO** должно равняться общему числу строк таблицы **EMP**, в которых значение столбца **DEPT\_NO** равно значению одноименного столбца данной строки таблицы **DEPT**. В части (5) для определения столбца **DEPT\_TOTAL\_SAL** (объем фонда заработной платы отдела) используется домен **SALARY\***. Но при этом явно установлено значение столбца по умолчанию (отличное от значения по умолчанию домена), запрещено наличие неопределенных значений и установлено дополнительное проверочное ограничение, устанавливающее нижний порог объема фонда заработной платы отдела. Еще одно

---

\* По этой причине мы ввели в предыдущем подразделе такую большую верхнюю границу 20000000.00 значений домена **SALARY**.

проверочное ограничение — (8) — вынесено на уровень определения табличного ограничения. Это ограничение устанавливает, что в каждой строке таблицы **DEPT** значение столбца **DEPT\_TOTAL\_SAL** должно быть не меньше суммы значений столбца **EMP\_SAL** во всех строках таблицы **EMP**, в которых значение столбца **DEPT\_NO** равно значению одноименного столбца данной строки таблицы **DEPT**. Обратите внимание на определение столбца **DEPT\_MNG** — часть (6). Этот столбец объявляется внешним ключом таблицы **DEPT**. Но мы хотим сказать больше. У отдела могут временно отсутствовать руководители, поэтому в столбце допускаются неопределенные значения. Но если у отдела имеется руководитель, то он должен являться руководителем только этого отдела. На первый взгляд, можно было бы воспользоваться ограничением столбца **UNIQUE**. Но такое ограничение допустило бы наличие неопределенного столбца **DEPT\_MNG** только в одной строке таблицы **DEPT**, а мы хотим допустить отсутствие руководителя у нескольких отделов. Поэтому потребовалось ввести более громоздкое проверочное ограничение столбца.

Наконец, определим таблицу **PRO**.

```
(1) CREATE TABLE PRO (  
(2) PRO_NO PRO_NO PRIMARY KEY,  
(3) PRO_TITLE VARCHAR(200) DEFAULT 'No title'  
    NOT NULL,  
(4) PRO_SDATE DATE DEFAULT CURRENT_DATE NOT NULL,  
(5) PRO_DURAT INTERVAL YEAR DEFAULT INTERVAL '1'  
    YEAR NOT NULL,  
(6) PRO_MNG EMP_NO UNIQUE NOT NULL REFERENCES  
    EMP ON DELETE NO ACTION,  
(7) PRO_DESC CLOB(10M));
```

Столбец **PRO\_SDATE** содержит дату начала проекта, а столбец **PRO\_DURAT** — продолжительность проекта в годах. В этом определении имеет смысл прокомментировать часть (6). Мы считаем, что если отдел, по крайней мере, временно может существовать без руководителя, то у ведущегося проекта всегда обязан быть менеджер. Поэтому определение столбца **PRO\_MNG** является гораздо более строгим, чем определение столбца **DEPT\_MNG** в таблице **DEPT**. Сочетание ограничений **UNIQUE** и **NOT NULL** при отсутствии значений по умолчанию приводит к абсолютной уникальности значений столбца **PRO\_MNG**. Другими словами, этот столбец обладает всеми характеристиками первичного ключа, хотя объявлен только как возможный ключ. Кроме того, он объявлен как внешний ключ с действием при удалении

строки таблицы EMP с соответствующим значением первичного ключа NO ACTION, запрещающим такие удаления. В совокупности это гарантирует, что у любого проекта будет существовать менеджер, являющийся служащим предприятия. В части (5) столбец PRO\_DESC (описание проекта) определен как большой символьный объект с максимальным размером 10 Мб.

#### 11.4.2. Изменение определения базовой таблицы

Оператор изменения определения базовой таблицы ALTER TABLE имеет следующий синтаксис:

```
base_table_alteration ::= ALTER TABLE base_table_
name
    column_alteration_action
    | base_table_constraint_alteration_action
```

Как видно из этого синтаксического правила, при выполнении одного оператора ALTER TABLE может быть выполнено либо действие по изменению определения столбца, либо действие по изменению определения табличного ограничения целостности.

**Добавление, изменение или удаление определения столбца.** Действие по изменению определения столбца специфицируется в следующем синтаксисе:

```
column_alteration_action ::=
    ADD [ COLUMN ] column_definition
    | ALTER [ COLUMN ] column_name
        { SET default_definition | DROP DEFAULT }
    | DROP [ COLUMN ] column_name
        { RESTRICT | CASCADE }
```

Итак, с использованием оператора ALTER TABLE можно добавлять к определению таблицы определение нового столбца (действие ADD) и изменять или отменять определение существующего столбца (действия ALTER и DROP соответственно).

Смысл действия ADD COLUMN почти полностью совпадает со смыслом раздела определения столбца в операторе CREATE TABLE. Указывается имя нового столбца, его тип данных или домен. Могут определяться значение по умолчанию и ограничения целостности. Однако имеется одно существенное отличие: столбец, определяемый в действии ADD оператора ALTER TABLE, добавляется к уже существующей таблице, которая, скорее всего, содержит некоторый набор строк. В каждой из существующих строк новый столбец должен содержать некоторое значение,



и считается, что сразу после выполнения действия **ADD** этим значением является значение столбца по умолчанию. Поэтому столбец, определяемый в действии **ADD**, обязательно должен иметь значение по умолчанию, т. е. для него недопустима ситуация, когда значением по умолчанию явно или неявно объявлено неопределенное значение (**NULL**), но среди ограничений целостности столбца присутствует ограничение **NOT NULL**.

В действии **ALTER COLUMN** можно изменить (**SET default\_definition**) или отменить определение значения по умолчанию для существующего столбца. Правила определения нового действующего значения столбца по умолчанию совпадают с соответствующими правилами, действующими для подраздела определения столбца в операторе **CREATE TABLE**. Заметим, что изменение значения столбца по умолчанию не оказывает влияния на состояние существующих строк таблицы (даже если в некоторых из них хранится предыдущее значение столбца по умолчанию). Если столбец определен на домене, у которого существует значение по умолчанию, то после отмены определения значения столбца по умолчанию для этого столбца начинает действовать значение по умолчанию домена.

Действие **DROP COLUMN** отменяет определение существующего столбца (удаляет его из таблицы). Действие **DROP COLUMN** отвергается, если указанный столбец является единственным столбцом таблицы или в этом действии присутствует спецификация **RESTRICT**, и данный столбец используется в определении каких-либо представлений или ограничений целостности, не считая тех табличных ограничений целостности, которые определены в составе определения базовой таблицы, содержащей данный столбец, и не содержат ссылок на какие-либо другие столбцы. Если в действии присутствует спецификация **CASCADE**, то его выполнение порождает неявное выполнение оператора **DROP** для всех представлений и ограничений целостности, в определении которых используется данный столбец.

Приведем несколько примеров добавления и изменения определения столбца. Предположим, что на предприятии ввели систему премирования служащих. Каждый служащий может дополнительно к заработной плате получать ежемесячную премию, не превышающую размер его заработной платы. Тогда разумно добавить к таблице **EMP** новый столбец **EMP\_BONUS**, используя оператор **ALTER TABLE**:

```
ALTER TABLE EMP
ADD EMP_BONUS SALARY DEFAULT NULL
CONSTRAINT BONSAL CHECK (VALUE < EMP_SAL);
```

При определении столбца **EMP\_SAL** таблицы **EMP** для него явно не определялось значение по умолчанию (оно наследовалось из определения домена). Если в какой-то момент это стало неправильным (например, повысился размер минимальной заработной платы), можно установить новое значение по умолчанию:

```
ALTER TABLE EMP ALTER EMP_SAL SET DEFAULT  
15000.00;
```

При определении столбца **DEPT\_TOTAL\_SAL** таблицы **DEPT** для него было установлено значение по умолчанию 1 000 000. Главный бухгалтер предприятия может быть недоволен тем, что такие важные данные, как объем фонда заработной платы отделов, могут устанавливаться по умолчанию. Тогда можно отменить это значение по умолчанию:

```
ALTER TABLE DEPT ALTER DEPT_TOTAL_SAL DROP DEFAULT;
```

Обратите внимание, что после выполнения этого оператора при вставке новой строки в таблицу **DEPT** всегда потребуется явно указывать значение столбца **DEPT\_TOTAL\_SAL**. Хотя формально у столбца будет существовать значение по умолчанию, наследуемое от домена **SALARY** (10000.00), оно не сможет быть занесено в таблицу **DEPT**, поскольку противоречит ограничению столбца **DEPT\_TOTAL\_SAL CHECK (VALUE >= 100000.00)**.

Можно задуматься, действительно ли требуется поддерживать в таблице **DEPT** столбец **DEPT\_EMP\_NO**. Как мы видели, для его поддержки требуется проверять громоздкое ограничение целостности, а число служащих в любом отделе можно получить динамически с помощью простого запроса к таблице **EMP** (собственно, этот запрос входит в ограничение целостности). Поэтому может оказаться разумным отменить определение столбца **DEPT\_EMP\_NO**, выполнив следующий оператор **ALTER TABLE**:

```
ALTER TABLE DEPT DROP DEPT_EMP_NO CASCADE;
```

Напомним, что спецификация **CASCADE** ведет к тому, что при выполнении оператора будет уничтожено не только определение указанного столбца, но и определения всех ограничений целостности и представлений, в которых используется уничтожаемый столбец. В нашем случае единственное связанное с этим столбцом ограничение целостности, определенное вне определения столбца, было бы отменено, даже если бы в операторе отмены определения столбца **DEPT\_EMP\_NO** содержалась спецификация **RESTRICT**, поскольку это единственное внешнее

определение ограничения является ограничением только столбца DEPT\_EMP\_NO.

**Изменение набора табличных ограничений.** Действие по изменению набора табличных ограничений специфицируется в следующем синтаксисе:

```
base_table_constraint_alteration_action ::=  
    ADD [ CONSTRAINT ] base_table_constraint_  
        definition  
    | DROP CONSTRAINT constraint_name  
        { RESTRICT | CASCADE }
```

Действие **ADD [ CONSTRAINT ]** позволяет добавить к набору существующих ограничений существующей таблицы новое ограничение целостности. Можно считать, что новое ограничение добавляется через **AND** к конъюнкции существующих ограничений, как если бы оно определялось в составе оператора **CREATE TABLE**. Но здесь имеется одно существенное отличие. Если внимательно посмотреть на все возможные виды табличных ограничений, можно убедиться, что любое из них удовлетворяется на пустой таблице. Поэтому, какой бы набор табличных ограничений ни был определен при создании таблицы, это определение является допустимым и не препятствует выполнению оператора **CREATE TABLE**. При добавлении нового табличного ограничения с использованием действия **ADD [ CONSTRAINT ]** имеем другую ситуацию, поскольку таблица, скорее всего, уже содержит некоторый набор строк, для которого условное выражение нового ограничения может принять значение *false*. В этом случае выполнение оператора **ALTER TABLE**, включающего действие **ADD [ CONSTRAINT ]**, отвергается.

Выполнение действия **DROP CONSTRAINT** приводит к отмене определения существующего табличного ограничения. Можно отменить определение только именованных табличных ограничений. Спецификации **RESTRICT** и **CASCADE** осмысленны только в том случае, если отменяемое ограничение является ограничением возможного ключа (**UNIQUE** или **PRIMARY KEY**)\*. При указании **RESTRICT** действие отвергается, если на данный возможный ключ ссылается хотя бы один внешний ключ. При указании **CASCADE** действие **DROP CONSTRAINT** выполняется в любом случае, и все определения таких внешних ключей также отменяются.

---

\* Хотя формально требуется указывать одно из этих ключевых слов в любом действии **DROP CONSTRAINT**.

Приведем пару примеров изменения набора табличных ограничений. Напомним, что мы добавили к таблице **EMP** столбец **EMP\_BONUS**, в котором сохраняются размеры ежемесячных премий служащих. Предположим, что премии выплачиваются из фонда заработной платы отдела, в котором работает служащий. Тогда проверочное ограничение столбца **DEPT\_TOTAL\_SAL**, устанавливающее, что объем фонда заработной платы отдела не должен быть меньше суммарной заработной платы служащих этого отдела, становится недостаточным и требуется добавить к набору ограничений таблицы **DEPT** новое ограничение:

```
ALTER TABLE DEPT ADD CONSTRAINT TOTAL_INCOME
CHECK (DEPT_TOTAL_SAL >=
      (SELECT SUM(EMP_SAL + COALESCE(EMP_BONUS, 0))
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT_NO));
```

Несмотря на громоздкость данного ограничения, смысл его очень прост: суммарный доход служащих отдела не должен превышать объем заработной платы отдела. В арифметическом выражении под знаком агрегатной операции **SUM** используется операция **COALESCE**. Эта двуместная операция определяется следующим образом:

**COALESCE (x, y)  $\equiv$  IF x IS NOT NULL THEN x ELSE y,**

т.е. значением операции является значение первого операнда, если оно не равно **NULL**, и значение второго операнда в противном случае. Нам пришлось воспользоваться этой операцией, поскольку в столбце **EMP\_BONUS** допускается наличие неопределенных значений.

При определении таблицы **EMP** было установлено проверочное табличное ограничение **PRO\_EMP\_NO**, устанавливающее, что над одним проектом не должно работать более 50 служащих. Как уже отмечалось, это ограничение носит чисто административный характер и может быть отменено без нарушения логики базы данных. Для отмены ограничения следует выполнить следующий оператор:

```
ALTER TABLE EMP DROP CONSTRAINT PRO_EMP_NO;
```

### **11.4.3. Отмена определения (уничтожение) базовой таблицы**

Для отмены определения (уничтожения) базовой таблицы служит оператор **DROP TABLE**, задаваемый в следующем синтаксисе:

```
DROP TABLE base_table_name { RESTRICT | CASCADE }
```

Успешное выполнение оператора приводит к тому, что указанная базовая таблица перестает существовать. Уничтожаются все ее строки, определения столбцов и табличные определения целостности. При наличии спецификации **RESTRICT** выполнение оператора **DROP TABLE** отвергается, если имя таблицы используется в каком-либо определении представления или ограничения целостности, не считая тех табличных ограничений целостности, которые указаны в составе определения базовой таблицы, содержащей данный столбец, и не содержат ссылок на какие-либо другие столбцы. При наличии спецификации **CASCADE** оператор выполняется в любом случае, и все определения представлений и ограничений целостности, содержащие ссылки на данную таблицу, также отменяются.

## 11.5. Средства определения и отмены общих ограничений целостности

Виды ограничений целостности, которые были рассмотрены в предыдущих подразделах этой главы, образуют иерархию. Ограничения целостности, входящие в определение домена, наследуются всеми столбцами, определенными на этих доменах, и являются ограничениями этих столбцов. Кроме того, в определение столбца могут входить определения дополнительных ограничений. Ограничения целостности, входящие в определение столбца (включая ограничения, унаследованные из определения домена), являются ограничениями таблицы, в состав определения которой входит определение данного столбца. Кроме того, в определение таблицы могут входить определения дополнительных ограничений.

Но иерархия видов ограничений целостности этим не исчерпывается. Ограничения целостности, входящие в определение таблицы (включая явные и унаследованные от определения доменов ограничения столбцов), являются ограничениями базы данных, частью которой является данная таблица. Кроме того, могут определяться дополнительные ограничения базы данных. В стандарте SQL такие дополнительные ограничения базы данных называются **ASSERTION**, здесь будем их называть *общими ограничениями целостности*.

### 11.5.1. Определение общих ограничений целостности

Для определения общего ограничения целостности служит оператор **CREATE ASSERTION**, задаваемый в следующем синтаксисе:

```
CREATE ASSERTION constraint_name CHECK  
(conditional_expression)
```

Заметим, что при создании общего ограничения целостности его имя всегда должно указываться явно. Хотя синтаксис определения общего ограничения совпадает с синтаксисом определений ограничений столбца и таблицы, в данном случае допускается только специальные виды условных выражений. В данной главе точно сформулировать свойства этих видов условий мы не сможем, поскольку разновидности условных выражений будут подробно рассмотрены в гл. 12. Если говорить неформально, то особые свойства условий связаны с тем, что при определении общих ограничений целостности контекстом, в котором вычисляется условное выражение, является весь набор таблиц базы данных, а не набор строк таблицы, как это было при определении табличных ограничений. Продемонстрируем и прокомментируем несколько примеров определений общих ограничений целостности.

В определении таблицы EMP содержалось ограничение столбца EMP\_BDATE

```
CHECK (EMP_BDATE >= '1917-10-24')
```

(к работе на предприятии допускаются только те лица, которые родились после Октябрьского переворота). Вот каким образом можно определить такое же ограничение на уровне общих ограничений целостности:

```
CREATE ASSERTION MIN_EMP_BDATE CHECK  
((SELECT MIN(EMP_BDATE, CURRENT_DATE)  
FROM EMP) >= '1917-10-24');
```

В логическом условии этого общего ограничения выбирается минимальное значение столбца EMP\_BDATE (дата рождения самого старого служащего). Значением условного выражения будет *false* в том и только том случае, если среди служащих имеется хотя бы один, родившийся до запретной даты.

Покажем, как можно сформулировать в виде общего ограничения целостности ограничение внешнего ключа. Например, приведем такую эквивалентную формулировку для определения внешнего ключа PRO\_NO, входящего в состав определения таблицы EMP:

```
FOREIGN KEY PRO_NO REFERENCES PRO (PRO_NO)
```

В виде общего ограничения целостности это может выглядеть следующим образом:

```

(1) CREATE ASSERTION FK_PRO_NO CHECK
(2) ( NOT EXISTS (SELECT * FROM EMP
                WHERE PRO_NO IS NOT NULL AND
(3)         NOT EXISTS (SELECT * FROM PRO
(4)         WHERE PRO.PRO_NO = EMP.PRO_NO) ) );

```

Логическое выражение этого ограничения выглядит достаточно сложным и нуждается в пояснении. Условие выборки оператора **SELECT** в строке (2) состоит из двух частей, связанных через **AND**. Первая часть отфильтровывает те строки таблицы **EMP**, у которых в столбце **PRO\_NO** содержится **NULL**. Если этот столбец содержит **NULL** во всех строках таблицы, то результирующая таблица оператора выборки в строке (2) будет пустой и значением предиката **NOT EXISTS** будет *true*, т. е. ограничение удовлетворяется.

Теперь предположим, что в таблице **EMP** нашлась строка *emp*, в столбце **PRO\_NO** которой содержится значение, отличное от **NULL**. Назовем это значение *cand\_pro\_no*. Для него вычисляется вторая часть условия выборки оператора **SELECT** в строке (2). Оператор выборки в строке (3) выбирает все строки таблицы **PRO**, значение столбца **PRO\_NO** которых равняется *cand\_pro\_no*. Если для данного значения *cand\_pro\_no* нашлась хотя бы одна такая строка, то результирующая таблица оператора выборки на строке (3) будет непустой и значением предиката **NOT EXISTS** на строке (3) будет *false*. Соответственно, все условие выборки первого оператора **SELECT** примет значение *false* и строка со значением *cand\_pro\_no* в столбце **PRO\_NO** будет отфильтрована, т. е. *cand\_pro\_no* является допустимым значением внешнего ключа.

Если же найдется хотя бы одна строка таблицы **EMP** с таким значением *cand\_pro\_no* столбца **PRO\_NO**, что в таблице **PRO** не найдется ни одной строки, значение столбца **PRO\_NO** которой равнялось бы этому *cand\_pro\_no*, то результирующая таблица оператора выборки на строке (3) будет пустой и значением предиката **NOT EXISTS** на строке (3) будет *true*. Тогда все условие выборки первого оператора **SELECT** примет значение *true* и эта строка таблицы **EMP** будет пропущена в результирующую таблицу. Значением предиката **NOT EXISTS** будет *false*, т. е. ограничение не удовлетворяется.

Мы сознательно привели такое подробное пояснение не только для того, чтобы прояснить смысл условного выражения общего ограничения целостности **FK\_PRO\_NO**, но и для того, чтобы дать понять, во что реально вырождается простая синтаксическая конструкция определения внешнего ключа. Как показывает опыт, многие начинающие проектировщики SQL-ориентирован-

ных баз данных думают, что ссылочные ограничения так же легко поддерживать, как и определять.

### 11.5.2. Отмена определения общего ограничения целостности

Чтобы отменить ранее определенное общее ограничение целостности, нужно воспользоваться оператором **DROP ASSERTION**, задаваемым в следующем синтаксисе:

```
DROP ASSERTION constraint_name
```

Вот пример оператора, отменяющего определение дискриминационного общего ограничения целостности **MIN\_EMP\_BDATE**:

```
DROP ASSERTION MIN_EMP_BDATE;
```

### 11.5.3. Немедленная и откладываемая проверка ограничений

На первый взгляд кажется, что ограничения целостности (всех видов) должны немедленно проверяться в случае выполнения любого действия, изменяющего содержимое базы данных (вставка в любую таблицу новой строки, изменение или удаление существующих строк). Однако можно определить такие ограничения целостности, логическое выражение которых будет принимать значение *false* при любой немедленной проверке. Одним из примеров такого ограничения является ограничение

```
CHECK (DEPT_EMP_NO =  
        (SELECT COUNT(*) FROM EMP  
         WHERE DEPT_NO = EMP.DEPT_NO))
```

из определения таблицы **DEPT**. Предположим, например, что в отдел зачисляется новый служащий. Тогда нужно выполнить две операции: вставить новую строку в таблицу **EMP** и изменить соответствующую строку таблицы **DEPT** (прибавить единицу к значению столбца **DEPT\_EMP\_NO**). Очевидно, что в каком бы порядке не выполнялись эти операции, сразу после выполнения первой из них ограничение целостности будет нарушено, соответствующее действие будет отвергнуто, и мы никогда не сможем принять на работу нового служащего.

Поскольку ограничения целостности, немедленная проверка которых бессмысленна, являются нужными и полезными, в язык SQL включены средства, позволяющие регулировать время проверки ограничений. Более точно, в контексте каждой выполняе-



мой транзакции каждое ограничение целостности должно находиться в одном из двух режимов: режиме *немедленной проверки* (*immediate*) или режиме *отложенной проверки* (*deferred*). Все ограничения целостности, находящиеся в режиме немедленной проверки, проверяются при выполнении в транзакции любой операции, изменяющей состояние базы данных. Если действие операции нарушает какое-либо немедленно проверяемое ограничение целостности, то это действие отвергается\*. Ограничения целостности, находящиеся в режиме отложенной проверки, проверяются при завершении транзакции (выполнении операции COMMIT). Если действия этой транзакции нарушают какое-либо отложено проверяемое ограничение целостности, то транзакция откатывается (операция COMMIT трактуется как операция ROLLBACK)\*\*.

Для этого в качестве заключительной синтаксической конструкции к любому определению ограничения целостности (любого вида) может быть добавлена спецификация INITIALLY в следующей синтаксической форме:

```
INITIALLY { DEFERRED | IMMEDIATE } [ [ NOT ]  
DEFERRABLE ]
```

Эта спецификация указывает, в каком режиме должно находиться данное ограничение целостности при начале выполнения любой транзакции (INITIALLY IMMEDIATE означает, что в начале выполнения транзакции данное ограничение будет находиться в режиме немедленной проверки, а INITIALLY DEFERRED — что в начале любой транзакции ограничение будет находиться в режиме отложенной проверки), а также в возможности смены режима этого ограничения при выполнении транзакции (DEFERRABLE означает, что для данного ограничения может быть установлен режим отложенной проверки, а NOT DEFERRABLE — что не может\*\*\*).

---

\* Конечно, в грамотных реализациях SQL при выполнении операции проверяются не все немедленно проверяемые ограничения целостности, а только те, которые в принципе могут быть нарушены данной операцией.

\*\* Конечно, в грамотных реализациях SQL при завершении транзакции проверяются не все отложено проверяемые ограничения целостности, а только те, которые в принципе могут быть нарушены данной транзакцией.

\*\*\* Для некоторых ограничений целостности режим отложенной проверки не имеет смысла. К таким ограничениям относятся, например, ограничения домена, ограничения NOT NULL и ограничения возможного ключа (хотя при их определении допускается указание DEFFERABLE). Если же возможный ключ используется в некотором определении внешнего ключа, то в стандарте SQL требуется, чтобы ограничение этого возможного ключа было NOT DEFFERABLE.

Комбинация **INITIALLY DEFERRED NOT DEFERRABLE** является недопустимой. Если в определении ограничения спецификация начального режима проверки отсутствует, то подразумевается наличие спецификации **INITIALLY IMMEDIATE**. При наличии явной или неявной спецификации **INITIALLY IMMEDIATE** и отсутствии явного указания возможности смены режима подразумевается наличие спецификации **NOT DEFERRABLE**. При наличии спецификации **INITIALLY DEFERRED** и отсутствии явного указания возможности смены режима подразумевается наличие спецификации **DEFERRABLE**.

При выполнении транзакции можно изменить режим проверки некоторых или всех ограничений целостности для данной транзакции. Для этого служит оператор **SET CONSTRAINTS**, задаваемый в следующем синтаксисе:

```
SET CONSTRAINTS { constraint_name_commalist | ALL }  
                { DEFERRED | IMMEDIATE }
```

Если в операторе указывается список имен ограничений целостности, то все они должны быть **DEFERRABLE**; если хотя бы для одного ограничения из списка это требование не выполняется, то операция **SET CONSTRAINTS** отвергается. При указании ключевого слова **ALL** режим устанавливается для всех ограничений, в определении которых явно или неявно было указано **DEFERRABLE**. Если в качестве желаемого режима проверки ограничений задано **DEFERRED**, то все указанные ограничения переводятся в режим отложенной проверки. Если в качестве желаемого режима проверки ограничений задано **IMMEDIATE**, то все указанные ограничения переводятся в режим немедленной проверки. При этом, если хотя бы одно из этих ограничений не удовлетворяется, то операция **SET CONSTRAINTS** отвергается и все указанные ограничения остаются в предыдущем режиме.

При выполнении операции **COMMIT** неявно выполняется операция **SET CONSTRAINTS ALL IMMEDIATE**. Если эта операция отвергается, то **COMMIT** срабатывает как **ROLLBACK**.

\* \* \*

В этой главе были обсуждены наиболее важные аспекты языка SQL, связанные с определением схемы базы данных — типы данных SQL, средства определения доменов, базовых таблиц и ограничений целостности. Кроме того, были рассмотрены средства SQL, позволяющие динамически изменять и удалять

определения этих объектов. Чтобы не перегружать материал, некоторые не слишком существенные детали были опущены.

Язык SQL устроен таким образом, что практически невозможно изложить какую-либо его часть независимо от других частей. И хотя эта глава по смыслу должна быть первой среди глав, посвященных SQL (было бы странно обсуждать операторы выборки строк из таблиц, вставки, изменения и удаления строк до обсуждения средств создания таблиц и ограничений целостности), нам пришлось забежать вперед и воспользоваться материалом следующей главы для объяснения средств определения ограничений целостности. Автор надеется, что это не создало слишком больших неудобств для читателей.

# БАЗОВЫЕ ВОЗМОЖНОСТИ ВЫБОРКИ ДАННЫХ В ЯЗЫКЕ SQL

Несмотря на то, что язык SQL является полным языком баз данных, включающим множество разнообразных средств определения схемы, ограничения и поддержки целостности базы данных, поддержки администрирования, заполнения и модификации таблиц базы данных, поддержки разработки приложений, для подавляющего большинства пользователей этот язык остается *языком запросов*, т. е. языком, позволяющим формулировать произвольно сложные и точные декларативные запросы к базе данных.

Как отмечалось в заключении к предыдущей главе, структура стандарта языка SQL фактически не позволяет описать одну часть языка (в частности, средства запросов) в отрыве от других частей. Тем не менее, полагая, что средства выборки данных составляют наиболее интересную и практически значимую часть языка, автор этой книги выделил для их рассмотрения отдельную главу, материал которой полезен для более полного понимания предыдущей и последующих глав.

Напомним, что в этой книге автор ограничивается базовым подмножеством SQL:1999 и, частично, SQL:2003 (*прямым SQL*), и даже это подмножество описывает далеко не в полном объеме стандарта. Начнем эту главу с некоторой общей картины, дающей представление об операторе выборки, а затем будем постепенно уточнять и расширять эту общую картину.

## 12.1. Общая структура оператора выборки в языке SQL

Для выборки данных в прямом SQL используется оператор **SELECT**, возвращающий набор из одной или нескольких строк одинаковой структуры и задаваемый в следующем синтаксисе:

```

SELECT [ ALL | DISTINCT ] select_item_commalist
FROM table_reference_commalist
[ WHERE conditional_expression ]
[ GROUP BY column_name_commalist ]
[ HAVING conditional_expression ]
[ ORDER BY order_item_commalist ]

```

### 12.1.1. Семантика оператора выборки

Для начала опишем общую схему выполнения оператора **SELECT** в соответствии с предписаниями стандарта\*. Выполнение запроса состоит из нескольких шагов, соответствующих разделам оператора выборки. На первом шаге выполняется раздел **FROM**. Если список ссылок на таблицы (*table\_reference\_commalist*) этого раздела соответствует таблицам  $T_1, T_2, \dots, T_n$ \*\*, то в результате выполнения раздела **FROM** образуется таблица (назовем ее  $T$ ), являющаяся расширенным декартовым произведением таблиц  $T_1, T_2, \dots, T_n$ . Если в разделе **FROM** указана только одна таблица, то она же и является результатом выполнения этого раздела. Напомним из гл. 4, что в реляционной алгебре Кодда для корректного выполнения операции взятия расширенного декартова произведения отношений в общем случае требуется применение операции переименования атрибутов. Соответствующие возможности переименования столбцов таблиц, указанных в списке раздела **FROM**, поддерживаются и в SQL. Альтернативный способ именования столбцов результирующей таблицы  $T$  основывается на использовании квалифицированных имен столбцов. Идея этого подхода (более раннего в истории SQL) заключается в том, что с любой таблицей, ссылка на которую содержится в списке раздела **FROM** можно связать некоторое имя-псевдоним (в стандарте оно называется *correlation name*). Тогда, если с такой таблицей  $T_i$  связан псевдоним  $Z_i$ , то в пределах оператора выборки можно ссылаться на любой столбец  $A$  таблицы по квалифицированному имени  $Z_i.A$ . Обсудим это

---

\* Не следует полагать, что запросы к SQL-ориентированной базе данных действительно должны выполняться в точности так, как в описываемой схеме. Более того, такой схемы не придерживается ни одна реализация SQL. Но как бы реально ни выполнялся оператор выборки, результат должен быть таким же, как если бы он получался при точном следовании данной схеме выполнения.

\*\*  $T_1, T_2, \dots, T_n$  не обязаны являться базовыми таблицами. См. следующий подраздел.

подробнее в следующем подразделе, а пока будем считать, что имена всех столбцов таблицы *T* определены и различны.

На втором шаге выполняется раздел **WHERE**. Условное выражение (*conditional\_expression*) этого раздела применяется к каждой строке таблицы *T*, и результатом является таблица *T1*, содержащая те и только те строки таблицы *T*, для которых результатом вычисления условного выражения является *true*. (Заголовки таблиц *T* и *T1* совпадают.) Если раздел **WHERE** в операторе выборки отсутствует, то это трактуется как наличие раздела **WHERE true**, т. е. *T1* содержит те и только те строки, которые содержатся в таблице *T*. Обратите внимание на разницу в трактовке логических выражений в операторах выборки и в табличных ограничениях целостности. Логическое выражение раздела **WHERE** (и раздела **HAVING**) оператора выборки *разрешает* выборку строки в том и только том случае, когда результатом вычисления логического выражения на данной строке является *true* (значения *false* и *unknown* не являются разрешающими). Логическое выражение табличного ограничения целостности *запрещает* наличие строки в таблице в том и только том случае, когда результатом вычисления логического выражения на данной строке является *false* (значения *true* и *unknown* не являются запрещающими).

Если в операторе выборки присутствует раздел **GROUP BY**, то он выполняется на третьем шаге. Каждый элемент списка имен столбцов (*column\_name\_commalist*), указываемого в этом разделе, должен быть одним из имен столбцов таблицы *T1*. В результате выполнения раздела **GROUP BY** образуется *сгруппированная таблица T2*, в которой строки таблицы *T1* расставлены в минимальное число групп, таких что во всех строках одной группы значения столбцов, указанных в списке имен столбцов раздела **GROUP BY** (*столбцов группировки*), одинаковы\*. Заметим, что сгруппированные таблицы не могут являться окончательным результатом оператора выборки. Они существуют только на концептуальном уровне на стадии выполнения запроса, содержащего раздел **GROUP BY**.

Если в операторе выборки присутствует раздел **HAVING**, то он выполняется на следующем шаге. Условное выражение этого раздела применяется к каждой группе строк таблицы *T2*, и результатом является сгруппированная таблица *T3*, содержащая те и только те группы строк таблицы *T2*, для которых резуль-

---

\* Если говорить более точно, то в одной группе все строки, составленные из значений столбцов группировки, являются дубликатами.

татом вычисления условного выражения является *true*. Условное выражение раздела **HAVING** строится по синтаксическим правилам, общим для всех условных выражений, но обладает той спецификой, что применяется к группам строк, а не к отдельным строкам. Поэтому предикаты, из которых строится это условное выражение, должны быть предикатами на группу в целом. В них могут использоваться имена столбцов группировки (*инварианты группы*) и так называемые *агрегатные функции* (**COUNT**, **SUM**, **MIN**, **MAX**, **AVG**) от других столбцов.

При наличии в запросе раздела **HAVING**, которому не предшествует раздел **GROUP BY**, таблица *T1* рассматривается как сгруппированная таблица, состоящая из одной группы строк, без столбцов группирования. В этом случае логическое выражение раздела **HAVING** может состоять только из предикатов с агрегатными функциями, а результат вычисления этого раздела *T3* либо совпадает с таблицей *T1*, либо является пустым.

Если в операторе выборки присутствует раздел **GROUP BY**, но отсутствует раздел **HAVING**, то это трактуется как наличие раздела **HAVING true**, т. е. *T3* содержит те и только те группы строк, которые содержатся в таблице *T2*.

После выполнения раздела **WHERE** (если в запросе отсутствуют разделы **GROUP BY** и **HAVING**, случай (а)) или выполнения явно или неявно заданного раздела **HAVING** (случай (б)) выполняется раздел **SELECT**. При выполнении этого раздела на основе таблицы *T1* в случае (а) или на основе сгруппированной таблицы *T3* в случае (б) строится таблица *T4*, содержащая столько строк, сколько строк или групп строк содержится в таблицах *T1* или *T3* соответственно. Число столбцов в таблице *T4* зависит от числа элементов в списке элементов выборки (*select\_item\_commalist*) и от вида элементов.

Рассмотрим, каким образом формируются значения столбцов в таблице *T4*. Элемент списка выборки может задаваться одним из двух способов:

```
select_item ::= value_expression [ [ AS ] column_name ]  
              | [ correlation_name . ] *
```

Сначала обсудим первый вариант. В этом случае каждый элемент списка элементов выборки соответствует столбцу таблицы *T4*. Столбцу может быть явным образом приписано имя (когда и для чего могут быть использованы имена таблицы *T4*, обсудим позже). Порядок формирования значения этого столбца для выделенных выше случаев (а) и (б) различается, и мы рассмотрим эти случаи по отдельности.

В случае (а) выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций со значениями соответствующих типов (в том числе, функций без параметров типа **CURRENT USER**). Кроме того, в выражении могут использоваться имена столбцов таблицы *T1*. Выражение вычисляется для каждой строки таблицы *T1*, и именам столбцов соответствуют значения этих столбцов в данной строке таблицы *T1*.

В случае (b), как и в случае (а), выражение, содержащееся в элементе выборки, может содержать литеральные константы и вызовы функций. Но, в отличие от случая (а), в выражение могут входить непосредственно имена только тех столбцов таблицы *T3*, которые входили в список столбцов группировки раздела **GROUP BY** оператора выборки. (Если сгруппированная таблица *T3* была образована за счет наличия раздела **HAVING** без присутствия раздела **GROUP BY**, то в выражении элемента выборки вообще нельзя непосредственно использовать имена столбцов таблицы *T3*.) Имена других столбцов таблицы *T3* могут использоваться только в конструкциях вызова агрегатных функций **COUNT**, **SUM**, **MIN**, **MAX**, **AVG**. Выражение вычисляется для каждой группы строк таблицы *T3*. Именам столбцов, входящих в выражение непосредственно, сопоставляются значения этих столбцов, которые соответствуют данной группе строк таблицы *T3*.

Во втором варианте спецификация элемента списка выборки вида  $[Z_i]^*$  является сокращенной формой записи списка  $Z_i.c_1, Z_i.c_2, \dots, Z_i.c_n$ , где  $c_1, c_2, \dots, c_n$  представляет собой полный список имен столбцов таблицы, псевдонимом которой является  $Z_i$ . Следует сделать три замечания. Во-первых, для именованной таблицы, входящей в список раздела **FROM** только один раз, можно использовать имя таблицы вместо псевдонима. Во-вторых, во втором варианте спецификации элемента списка выборки можно опустить псевдоним только в том случае, если в разделе **FROM** указана только одна таблица. В третьих, в случае (b) второй вариант спецификации элемента выборки допустим только в том случае, когда все столбцы таблицы с псевдонимом  $Z_i$  входят в список столбцов группировки раздела **GROUP BY**.

---

\* Заметим, что любой элемент  $Z_i.c_j$  этого неявно заданного списка может быть явно включен в список элементов выборки. Кстати, если в списке выборки присутствует явно или неявно заданный элемент вида  $Z_i.c$ , то в пределах запроса соответствующий столбец таблицы *T4* получает то же имя.



Итак, мы получили таблицу *T4*. Если в спецификации раздела **SELECT** отсутствует ключевое слово **DISTINCT**, или присутствует ключевое слово **ALL**, или отсутствуют и **ALL**, и **DISTINCT**, то *T4* является результатом выполнения раздела **SELECT**. В противном случае на завершающей стадии выполнения раздела **SELECT** в таблице *T4* удаляются строки-дубликаты.

Если в операторе выборки не содержится раздел **ORDER BY**, то таблица *T4* является результирующей таблицей запроса. Иначе на завершающей стадии выполнения запроса производится сортировка строк таблицы *T4* в соответствии со списком элементов сортировки (**order\_item\_commalist**) раздела **ORDER BY**. В стандарте SQL:1999 элемент списка элементов сортировки имеет следующую синтаксическую форму:

```
order_item ::= value_expression  
              [ collate_clause ] [ { ASC | DESC } ]
```

Выполнение раздела **ORDER BY** производится следующим образом\*. Выбирается первый элемент списка сортировки, и строки таблицы *T4* расставляются в порядке возрастания (если в элементе присутствует спецификация **ASC**; при отсутствии спецификации **ASC/DESC** предполагается наличие **ASC**) или в порядке убывания (при наличии спецификации **DESC**) в соответствии со значениями выражения, содержащегося в данном элементе, которое вычисляются для каждой строки таблицы *T4*. Далее выбирается второй элемент списка сортировки и в соответствии со значениями заданных в нем выражения и порядка сортировки расставляются строки, которые после первого шага сортировки образовали группы с одинаковым значением выражения первого элемента списка сортировки. Операция продолжается до исчерпания списка элементов сортировки. Результирующий отсортированный список строк является окончательным результатом запроса.

В общем случае выражение, входящее в элемент списка сортировки, основывается на именах столбцов таблицы *T4* и именах столбцов таблицы, над которой вычислялся раздел **SELECT** (*T1* или *T3*). Идея состоит в том, что если некоторое выражение могло бы быть использовано в элементе списка выборки, то его можно использовать в элементе списка сортировки. В стандарте SQL:1999 присутствует ряд чисто технических ограничений на вид выражений, допустимых в элементах списка сортировки,

---

\* Здесь снова игнорируется спецификация раздела **collate**, связанная с использованием национальных наборов символов.

если в запросе присутствуют разделы **GROUP BY** и/или **HAVING** и если в разделе **SELECT** присутствует спецификация **DISTINCT**. Но в любом случае это выражение может иметь вид *a*, где *a* — имя столбца таблицы *T4*.

### 12.1.2. Ссылки на таблицы раздела **FROM**

Напомним, что раздел **FROM** оператора выборки определяется синтаксическим правилом

```
FROM table_reference commalist
```

Рассмотрим более подробно, какой вид могут иметь элементы этого списка. Для начала приведем полный набор синтаксических правил SQL:1999, определяющий *table\_reference*\*.

```
table_reference ::= table_primary | joined_table
table_primary ::= table_or_query_name [ [ AS ]
correlation_name
    [ (derived_column_list) ] ]
    | derived_table [ [ AS ] correlation_name
    [ (derived_column_list) ] ]
    | lateral_derived_table [ [ AS ]
    correlation_name
    [ (derived_column_list) ] ]
    | collection_derived_table [ [ AS ]
    correlation_name
    [ (derived_column_list) ] ]
    | ONLY (table_or_query_name)[ [ AS ]
    correlation_name
    [ (derived_column_list) ] ]
    | (joined_table)
table_or_query_name ::= { table_name |
    query_name }
derived_table ::= (query_expression)
lateral_derived_table ::= LATERAL
    (query_expression)
collection_derived_table ::= UNNEST
    (collection_value_exression)
    [ WITH ORDINALITY ]
```

---

\* В связи с введением в стандарте SQL:2003 конструктора типов мультимножеств в качестве элемента списка ссылок на таблицы раздела **FROM** теперь можно использовать и выражение со значением-мультимножеством. Однако здесь мы не будем рассматривать эту возможность.

Громоздкость материала и ограниченность объема книги не позволяют описать все возможные конструкции, поэтому ограничимся только первыми двумя вариантами (`table_or_query_name` и `derived_table`), которые наиболее часто используются в реальных запросах. Обсуждение порождаемых таблиц с горизонтальной связью (`lateral_derived_table`), «соединенных таблиц» (`joined_table`) и конструкции ONLY (`table_or_query_name`) можно найти в [38]. Но даже при этих самоограничениях для дальнейшего продвижения требуется определить несколько дополнительных синтаксических конструкций языка SQL.

*Табличным выражением* (`table_expression`) называется конструкция

```
table_expression ::= FROM table_reference_commalist
                  [ WHERE conditional_expression ]
                  [ GROUP BY column_name_commalist ]
                  [ HAVING conditional_expression ]
```

*Спецификацией запроса* (`query_specification`) называется конструкция

```
query_specification ::= SELECT [ ALL | DISTINCT ]
                        select_item_commalist table_expression
```

Наконец, *выражением запросов* (`query_expression`) называется конструкция

```
query_expression ::= [ with_clause ]
                    query_expression_body
query_expression_body ::=
    { non_join_query_expression | joined_table }
non_join_query_expression ::= non_join_query_term
                             | query_expression_body
                             { UNION | EXCEPT } [ ALL | DISTINCT ]
                             [ corresponding_spec ] query_term
query_term ::= non_join_query_term | joined_table
non_join_query_term ::= non_join_query_primary
                     | query_term INTERSECT [ ALL | DISTINCT ]
                     [ corresponding_spec ] query_primary
query_primary ::= non_join_query_primary |
                  joined_table
non_join_query_primary ::= simple_table |
                           (non_join_query_expression)
simple_table ::= query_specification
               | table_value_constructor
               | TABLE table_name
```

```
corresponding_spec ::= CORRESPONDING  
    [ BY column_name_commalist ]
```

Если не обращать внимания на оставленную в стороне конструкцию `joined_table` и пока не обсуждавшуюся конструкцию `table_value_constructor`, синтаксические правила показывают, что выражение запросов строится из выражений, значениями которых являются таблицы, с использованием «теоретико-мультимножественных» операций **UNIOIN** (объединение), **EXCEPT** (вычитание) и **INTERSECT** (пересечение). Операция пересечения является «мультипликативной» и обладает приоритетом, большим приоритета «аддитивных» операций объединения и вычитания. Вычисление выражения производится слева направо с учетом приоритетов операций и круглых скобок. При этом действуют следующие правила.

Если выражение запросов не включает ни одной теоретико-множественной операции, то результатом вычисления выражения запросов является результат вычисления простой или соединенной таблицы.

Если в терме (`non_join_query_term`) или выражении запросов без соединения (`non_join_query_expression`) присутствует теоретико-множественная операция, то пусть *T1*, *T2* и *TR* обозначают первый операнд, второй операнд и результат терма или выражения соответственно, а **OP** — используемую теоретико-множественную операцию.

Если в операции присутствует спецификация **CORRESPONDING**, то:

- если присутствует конструкция `BY column_name_commalist`, то все имена в этом списке должны быть различны и каждое имя должно являться именем некоторого столбца таблицы *T1* и именем некоторого столбца таблицы *T2*, причем типы этих столбцов должны быть совместимыми; обозначим этот список имен через *SL*;
- если список соответствия столбцов не задан, то пусть *SL* обозначает список имен столбцов, являющихся именами столбцов и в *T1*, и в *T2*, в том порядке, в котором эти имена фигурируют в *T1*;
- вычисляемые терм или выражение запросов без соединения эквивалентны выражению `(SELECT SL FROM T1) OP (SELECT SL FROM T2)`, не включающему спецификации **CORRESPONDING**.

При отсутствии в операции спецификации **CORRESPONDING** эта операция выполняется таким образом, как если бы эта специфи-

кация присутствовала и включала конструкцию **BY column\_name\_comma\_list**, в которой были бы перечислены все столбцы таблицы *T1*. Другими словами, при отсутствии спецификации **CORRESPONDING** требуется, чтобы заголовки таблиц-операндов совпадали за исключением, возможно, порядка следования столбцов.

Здесь и далее до конца подраздела будем считать, что заголовки таблиц *T1* и *T2* совпадают. При выполнении операции **OP** две строки *s1* и *s2* с именами столбцов *c<sub>1</sub>, c<sub>2</sub>, ..., c<sub>n</sub>* считаются строками-дубликатами, если для каждого *i* (*i* = 1, 2, ..., *n*) либо (*ci* = *di*) = *true*, либо и *ci*, и *di* содержат **NULL**.

Если в операции **OP** не задана спецификация **ALL**, то из *T1* и *T2* удаляются строки дубликаты, и **OP** выполняется как обычная теоретико-множественная операция.

Если спецификация **ALL** задана, то пусть *s* — некоторая строка, соответствующая заголовку *T1* и *T2*; пусть *m* — число дубликатов *s* в *T1*, а *n* — число дубликатов *s* в *T2*. Тогда (по определению), если **OP**  $\equiv$  **UNION**, то число дубликатов *s* в *TR* равно *m* + *n*; если **OP**  $\equiv$  **EXCEPT**, то число дубликатов *s* в *TR* равно *max* ((*m-n*), 0); если **OP**  $\equiv$  **INTERSECT**, то число дубликатов *s* в *TR* равно *min* (*m, n*).

Как видно из синтаксиса выражения запросов, в этом выражении может присутствовать раздел **WITH**. Он задается в следующем синтаксисе:

```
with_clause ::= WITH [ RECURSIVE ]
               with_element_comma_list
with_element ::= query_name [ (column_name_list) ]
               AS (query_expression)
               [ search_or_cycle_clause ]
```

Общая форма раздела **WITH** применяется для формулировки рекурсивных запросов, которые в этой книге не рассматриваются (подробнее см. [38]). Ограничимся случаем, когда в разделе **WITH** отсутствуют спецификация **RECURSIVE** и **search\_or\_cycle\_clause**. Тогда конструкция

```
WITH query_name (c1, c2, ... cn) AS (query_exp_1)
query_exp_2
```

означает, что в любом месте выражения запросов **query\_exp\_2**, где допускается появление ссылки на таблицу, можно использовать имя **query\_name**. Можно считать, что перед выполнением **query\_exp\_2** происходит выполнение **query\_exp\_1** и результирующая таблица с именами столбцов *c1, c2, ..., cn* сохраняется под именем **query\_name**. Как будет показано далее, в этом случае

раздел **WITH** фактически служит для локального определения представляемой таблицы (**VIEW**).

Чтобы завершить обсуждение выражений запросов (с учетом введенных самоограничений), рассмотрим конструкции **table\_value\_constructor** и **TABLE table\_name**.

В определении конструктора значения-таблицы используется конструктор значения-строки, который строит упорядоченный набор скалярных значений, представляющий строку (возможно и использование подзапроса\*):

```
row_value_constructor ::=
    row_value_constructor_element
    | [ ROW ]
    (row_value_constructor_element_comma_list)
    | row_subquery
row_value_constructor_element ::= value_expression
| NULL | DEFAULT
```

Заметим, что значение элемента по умолчанию можно использовать только в том случае, когда конструктор значения-строки используется в операторе **INSERT** (тогда этим значением будет значение по умолчанию соответствующего столбца).

Конструктор значения-таблицы производит таблицу на основе заданного набора конструкторов значений-строк:

```
table_value_constructor ::= VALUES
    row_value_constructor_comma_list
```

Конечно, для корректного построения таблицы требуется, чтобы строки, производимые всеми конструкторами строк, были одной и той же степени и чтобы типы (или домены) соответствующих столбцов являлись приводимыми.

Наконец, конструкция **TABLE table\_name** является сокращенной формой записи выражения **SELECT \* FROM table\_name**.

**Ссылки на базовые, представляемые и порождаемые таблицы.** Теперь можно завершить обсуждение разновидностей ссылок на таблицу в разделе **FROM**. Для удобства повторим синтаксические правила (опустив конструкции, рассмотрение которых выходит за пределы материала данной книги):

```
table_reference ::= table_primary
table_primary ::= table_or_query_name [ [ AS ]
```

---

\* Подзапросы более подробно обсуждаются в подразд. 12.2. Пока заметим, что **row\_subquery** — это запрос, результирующая таблица которого состоит из одной строки.

```

correlation_name
    [ (derived_column_list) ] ]
| derived_table [ AS ] correlation_name
    [ (derived_column_list) ]
table_or_query_name ::= { table_name | query_name }
derived_table ::= (query_expression)

```

Итак, в самом простом случае в качестве ссылки на таблицу используется имя таблицы (базовой или представляемой) или имя запроса, присоединенного к данному запросу с помощью раздела **WITH**. Во втором случае (**derived\_table**) порождаемая таблица задается выражением запроса, заключенным в круглые скобки. Явное указание имен столбцов результата запроса из раздела **WITH** или порождаемой таблицы требуется в том случае, когда эти имена не выводятся явно из соответствующего выражения запроса. Обратите внимание, что в этих случаях в соответствующем элементе списка раздела **FROM** должен указываться псевдоним (**correlation\_name**), потому что иначе таблица была бы вообще лишена имени. Можно считать, что выражение запроса вычисляется и сохраняется во временной таблице при обработке раздела **FROM**.

Возможно, некоторых читателей смутила рекурсивная природа синтаксических определений, приведенных в этом подразделе. Чтобы определить понятие ссылки на таблицу в разделе **FROM** оператора выборки, который опирается на спецификацию запроса, нам пришлось ввести более общее понятие выражения запросов, в определении которого используется спецификация запроса. Да, действительно, многие синтаксические конструкции SQL определяются рекурсивно. Но эта рекурсия никогда не приводит к заикливанию. В частности, раскрутка рекурсии операторов выборки основывается на базовой, не выделяемой отдельными синтаксическими правилами форме, в которой в разделе **FROM** указываются только имена базовых таблиц.

## 12.2. Представляемые таблицы, или представления (VIEW)

Еще одним примером рекурсивности спецификаций языка SQL является то, что в середине этой главы автор вынужден прервать обсуждение оператора выборки и ввести понятие *представляемой таблицы*, или *представления*, которое можно использовать в операторе выборки наряду с базовыми таблицами.

Только после этого можно будет считать обсуждение ссылок на таблицы в разделе **FROM** сравнительно завершенным. Итак, в общем случае оператор создания представления определяется следующими синтаксическими правилами:

```
create_view ::= CREATE [ RECURSIVE ] VIEW
table_name
    [ column_name_comma_list ]
    AS query_expression
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Рекурсивные представления (такие, в определении которых присутствует ключевое слово **RECURSIVE**) здесь не обсуждаются, а необязательный раздел **WITH CHECK OPTION** будет рассматриваться в гл. 13 (пока лишь заметим, что этот раздел связан с особенностями выполнения операций обновления базы данных через представления). Здесь мы кратко обсудим только простую форму представлений, определяемых по следующим правилам:

```
create_view ::= CREATE VIEW table_name
    [ column_name_comma_list ]
    AS query_expression
```

Имя таблицы, задаваемое в определении представления, существует в том же пространстве имен, что и имена базовых таблиц, и, следовательно, должно отличаться от всех имен таблиц (базовых и представляемых), созданных тем же пользователем. Если имя представления встречается в разделе **FROM** какого-либо оператора выборки, то вычисляется выражение запроса, указанное в разделе **AS**, и оператор выборки работает с результирующей таблицей этого выражения запроса\*. Явное указание имен столбцов представляемой таблицы требуется в том случае, когда эти имена не выводятся из соответствующего выражения запроса.

---

\* По крайней мере, так это следует понимать в соответствии с семантикой представлений в языке SQL. При реальной обработке запросов над представлениями такая явная «материализация» представления выполняется крайне редко. Вместо этого используется техника подстановки тела представления в тело запроса с гарантией того, что результат модифицированного запроса будет в точности таким же, что и результат исходного запроса над материализованным представлением. Но это уже относится к тематике оптимизации SQL-запросов, выходящей за пределы этого курса.



Как и для всех других вариантов оператора **CREATE**, для **CREATE VIEW** имеется обратный оператор **DROP VIEW table\_name**, выполнение которой приводит к отмене определения представления (реально это выражается в удалении данных о представлении из таблиц-каталогов базы данных). После выполнения операции пользоваться представлением с данным именем становится невозможно.

## 12.3. Логические выражения раздела **WHERE**

Синтаксически логическое выражение раздела **WHERE** определяется как булевское выражение, которое строится на основе предикатов с использованием операций **AND**, **OR** и **NOT**, а также круглых скобок. Предикат позволяет специфицировать условие, результатом вычисления которого может быть *true*, *false* или *unknown*. В языке SQL:1999 допустимы следующие предикаты\*:

```
predicate ::= comparison_predicate
           | between_predicate
           | null_predicate
           | in_predicate
           | like_predicate
           | similar_predicate
           | exists_predicate
           | unique_predicate
           | overlaps_predicate
           | quantified_comparison_predicate
           | match_predicate
           | distinct_predicate
```

Далее будем последовательно обсуждать разные виды предикатов и приводить примеры запросов с использованием базы данных **СЛУЖАЩИЕ-ОТДЕЛЫ-ПРОЕКТЫ**, определения таблиц которой на языке SQL были приведены в гл. 11.

### 12.3.1. Предикат сравнения

Предикат сравнения предназначен для спецификации сравнения двух строчных значений. Синтаксис предиката следующий:

---

\* В этой книге не обсуждаются предикаты, которые основаны на использовании выражений типа мультимножества, введенных в стандарте SQL:2003.

```

comparison_predicate ::=
    row_value_constructor comp_op
    row_value_constructor
comp_op ::= = | <> | < | > | <= | >=

```

Строки, являющиеся операндами операции сравнения, должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы. Пусть  $R_x$  и  $R_y$  обозначают первую и вторую строки-операнды соответственно, пусть  $N$  — степень операндов и пусть  $R_{x_i}$  и  $R_{y_i}$  обозначают  $i$ -е поля  $R_x$  и  $R_y$  соответственно ( $1 \leq i \leq N$ ). Тогда результат  $R_x \text{ comp\_op } R_y$  определяется следующим образом:

- $R_x = R_y$  есть *true* тогда и только тогда, когда  $R_{x_i} = R_{y_i}$  есть *true* для всех  $i$ ;
- $R_x < R_y$  есть *true* тогда и только тогда, когда  $R_{x_i} = R_{y_i}$  есть *true* для всех  $i < n$ , и  $R_{x_n} < R_{y_n}$  есть *true* для некоторого  $n$ ;
- $R_x = R_y$  есть *false* тогда и только тогда, когда  $\text{NOT } (R_{x_i} = R_{y_i})$  есть *true* для некоторого  $i$ ;
- $R_x < R_y$  есть *false* тогда и только тогда, когда  $R_x = R_y$  есть *true* или  $R_y < R_x$  есть *true*;
- $R_x <> R_y$  эквивалентно выражению  $\text{NOT } (R_x = R_y)$ ;
- $R_x > R_y$  эквивалентно выражению  $R_y < R_x$ ;
- $R_x <= R_y$  эквивалентно выражению  $(R_x = R_y \text{ OR } R_x < R_y)$ ;
- $R_x >= R_y$  эквивалентно выражению  $(R_x = R_y \text{ OR } R_x > R_y)$ ;
- $R_x \text{ comp\_op } R_y$  есть *unknown* тогда и только тогда, когда  $R_x \text{ comp\_op } R_y$  не есть *true* и не есть *false*.

**Пример 12.1.** Найти номера отделов, в которых работают служащие с фамилией 'Smith'.

```

SELECT DISTINCT EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_NAME = 'Smith';

```

В разделе **SELECT** присутствует спецификация **DISTINCT**, потому что в одном отделе могут работать несколько служащих с фамилией 'Smith', а их число нас в данном случае не интересует. Если бы нас интересовало число служащих с фамилией 'Smith', работающих в каждом отделе, то можно было бы написать такой запрос (**пример 12.1, а**):

```

SELECT EMP.DEPT_NO, COUNT(*)
FROM EMP
WHERE EMP.NAME = 'Smith'
GROUP BY EMP.DEPT_NO;

```

**Пример 12.2.** Найти номера, имена и номера отделов служащих, родившихся после 15 апреля 1965 г.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_BDATE > DATE '1965-04-15';
```

В результате этого запроса дубликатов быть не может, поскольку в список выборки включен столбец, являющийся первичным ключом таблицы EMP.

**Пример 12.3.** Найти номера, имена и номера отделов служащих, размер заработной платы которых составляет больше 1/10 объема фонда заработной платы их отделов.

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP
WHERE EMP.EMP_SAL > 0.1*
      (SELECT DEPT_TOTAL_SAL
       FROM DEPT
       WHERE DEPT.DEPT_NO = EMP.DEPT_NO);
```

В этом SQL-запросе имеются две интересные особенности, которые мы до сих пор не обсуждали. Во-первых, второй операнд операции сравнения содержит подзапрос, возвращающий единственное значение, поскольку логическое выражение раздела **WHERE** этого подзапроса состоит из условия, однозначно определяющего значение первичного ключа таблицы DEPT. Во-вторых, в условии раздела **WHERE** подзапроса используется ссылка на столбец таблицы EMP, указанной в разделе **FROM** «внешнего» запроса. Подобные подзапросы в терминологии SQL традиционно называются *корреляционными*, и их следует понимать следующим образом\*.

При выполнении внешнего запроса последовательно, строка за строкой, в некотором порядке, определяемом системой, производится проверка соответствия строк результирующей таблицы раздела **FROM** условию раздела **WHERE**. Если это условие включает *корреляционные* подзапросы, то внутри каждого из этих подзапросов ссылка на столбец внешней таблицы трактуется как ссылка

---

\* Здесь снова идет речь о *семантике* выполнения оператора **SELECT**. В стандарте, естественно, не требуется, чтобы в реализации языка запросы с корреляционными подзапросами выполнялись в точности так, как описывается ниже, но, какой бы реальный алгоритм выполнения такого запроса не использовался, результат выполнения должен быть точно таким же, как если бы запрос выполнялся по описываемой схеме.

на столбец текущей строки этой таблицы во внешнем цикле. Естественно, условие **WHERE** любого подзапроса может включать более глубоко вложенные подзапросы, на которые распространяется то же правило корреляции с внешними таблицами.

Эквивалентной формулировкой на языке SQL примера 12.3 является следующая формулировка (*пример 12.3, а*):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO
FROM EMP, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO AND
      EMP.EMP_SAL > 0.1 * DEPT.TOTAL_SAL;
```

Легко видеть, что в терминах реляционной алгебры этот запрос представляет собой ограничение (по условию **EMP.EMP\_SAL > 0.1 \* DEPT.TOTAL\_SAL**) *эквисоединения* таблиц **EMP** и **DEPT** (по условию **EMP.DEPT\_NO = DEPT.DEPT\_NO**). Подобную операцию часто называют *полусоединением* (*semijoin*), поскольку в результирующей таблице используются столбцы только одного из операндов операции эквисоединения. Мы привели вторую формулировку запроса, преследуя две цели:

- продемонстрировать, каким образом предикат сравнения может использовать для задания условия соединения;
- показать, что запросы, содержащие вложенные запросы, часто могут быть переформулированы в запросы с соединениями.

**Пример 12.4.** Найти номера, имена, номера отделов и имена руководителей отделов служащих, размер заработной платы которых меньше 15 000 руб.

```
SELECT EMP1.EMP_NO, EMP1.EMP_NAME, EMP1.DEPT_NO,
EMP2.EMP_NAME
FROM EMP AS EMP1, EMP AS EMP2, DEPT
WHERE EMP1.EMP_SAL < 15000.00 AND
      EMP1.DEPT_NO = DEPT.DEPT_NO AND
      DEPT.DEPT_MNG = EMP2.EMP_NO;
```

Этот запрос представляет собой эквисоединение ограничения таблицы **EMP** (по условию **EMP\_SAL < 15000.00**) с таблицами **DEPT** и **EMP** (по условиям **EMP.DEPT\_NO = DEPT.DEPT\_NO** и **DEPT.DEPT\_MNG = EMP2.EMP\_NO** соответственно). Таблица **EMP** участвует в качестве операнда операции эквисоединения два раза. Поэтому в разделе **FROM** ей присвоены два псевдонима — **EMP1** и **EMP2**. Следуя предписанному стандартом порядку выполнения запроса, можно считать, что введение этих псевдонимов

обеспечивает переименование столбцов таблицы EMP, требуемое для выполнения раздела FROM с образованием расширенного декартова произведения таблиц-операндов. Заметим также, что в данном случае мы имеем дело с *полным* эквисоединением трех таблиц (а не с *полусоединением*), поскольку в списке выборки присутствуют имена столбцов каждой из них.

Покажем способ формулировки этого запроса с использованием вложенного подзапроса в качестве элемента списка выборки (*пример 12.4, а*):

```
SELECT EMP.EMP_NO, EMP.EMP_NAME, EMP.DEPT_NO,  
       (SELECT EMP_NAME  
        FROM EMP  
        WHERE EMP_NO = DEPT_MNG)  
FROM EMP, DEPT  
WHERE EMP.EMP_SAL < 15000.00 AND  
       EMP.DEPT_NO = DEPT.DEPT_NO;
```

Как показывает последний пример, в условии выборки подзапроса, участвующего в списке выборки, можно использовать имена столбцов таблиц внешнего запроса. Из этой возможности языка SQL видно, что в подразд. 12.1.1 для облегчения понимания материала автор немного исказил семантику оператора выборки. Там было сказано следующее: «После выполнения раздела WHERE (если в запросе отсутствуют разделы GROUP BY и HAVING, случай (а)) или выполнения явно или неявно заданного раздела HAVING (случай (б)) выполняется раздел SELECT. При выполнении этого раздела на основе таблицы T1 в случае (а) или на основе сгруппированной таблицы T3 в случае (б) строится таблица T4, содержащая столько строк, сколько строк или групп строк содержится в таблицах T1 или T3 соответственно.» В действительности, в общем случае очередная строка таблицы T4 должна строиться в тот момент, когда очередная строка или группа строк заносится в таблицу T1 или T3 соответственно.

### 12.3.2. Предикат between

Предикат **between** позволяет специфицировать условие вхождения в диапазон значений. Операндами являются строки:

```
between_predicate ::=  
    row_value_constructor [ NOT ] BETWEEN  
    row_value_constructor AND  
    row_value_constructor
```

Все три строки-операнды должны иметь одну и ту же степень. Типы данных соответствующих значений строк-операндов должны быть совместимыми.

Пусть  $R_x$ ,  $R_y$  и  $R_z$  обозначают первый, второй и третий операнды соответственно. Тогда по определению выражение  $X \text{ NOT BETWEEN } Y \text{ AND } Z$  эквивалентно выражению  $\text{NOT } (X \text{ BETWEEN } Y \text{ AND } Z)$ . Выражение  $X \text{ BETWEEN } Y \text{ AND } Z$  по определению эквивалентно булевскому выражению  $X \geq Y \text{ AND } X \leq Z$ .

**Пример 12.5.** Найти номера, имена и размер заработной платы служащих, получающих заработную плату, размер которой не меньше средней заработной платы своего отдела и не больше заработной платы руководителя отдела.

```
SELECT EMP_NO, EMP_NAME, EMP_SAL
FROM EMP
WHERE EMP_SAL BETWEEN
      (SELECT AVG(EMP1.EMP_SAL)
       FROM EMP EMP1
       WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
      AND
      (SELECT EMP1.EMP_SAL
       FROM EMP EMP1
       WHERE EMP1.EMP_NO =
         (SELECT DEPT.DEPT_MNG
          FROM DEPT
          WHERE DEPT.DEPT_NO = EMP.DEPT_NO));
```

В этом запросе можно выделить три интересных момента. Во-первых, диапазон значений предиката **BETWEEN** задан двумя подзапросами, результатом каждого из которых является единственное значение. Первый подзапрос выдает единственное значение, поскольку в списке выборки содержится агрегатная функция (**AVG**) и отсутствует раздел **GROUP BY**, а второй — потому что в его разделе **WHERE** присутствует условие, задающее единственное значение первичного ключа. Во-вторых, в обоих подзапросах таблица **EMP** получает псевдоним **EMP1** (в формулировке этого запроса мы старались использовать как можно меньше вспомогательных идентификаторов). Поскольку подзапросы выполняются независимо один от другого, использование общего имени не вызывает проблем. Наконец, в условии второго подзапроса присутствует более вложенный подзапрос и в условии его раздела **WHERE** используется ссылка на столбец таблицы из самого внешнего раздела **FROM**.

12.3.3. Предикат null

Предикат `null` позволяет проверить, являются ли неопределенными значения всех элементов строки-операнда:

```
null_predicate ::= row_value_constructor IS
[ NOT ] NULL
```

Пусть *R<sub>x</sub>* обозначает строку-операнд. Если значения всех полей *R<sub>x</sub>* являются неопределенными, то значением условия *R<sub>x</sub> IS NULL* является *true*; иначе — *false*. Если ни у одного поля *R<sub>x</sub>* значение не является неопределенным, то значением условия *R<sub>x</sub> IS NOT NULL* является *true*; иначе — *false*.

*Замечание:* условие *R<sub>x</sub> IS NOT NULL* имеет то же значение, что условие *NOT R<sub>x</sub> IS NULL* для любого *R<sub>x</sub>* в том и только том случае, когда степень *X* равна 1. Полная семантика предиката `null` приведена в табл. 12.1.

**Пример 12.6.** Найти номера и имена служащих, номер отдела которых неизвестен.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO IS NULL;
```

Таблица 12.1. Семантика предиката null

| Вид операнда                                                     | Вид условия |               |               |                   |
|------------------------------------------------------------------|-------------|---------------|---------------|-------------------|
|                                                                  | X IS NULL   | X IS NOT NULL | NOT X IS NULL | NOT X IS NOT NULL |
| Степень 1: значение NULL                                         | true        | false         | false         | true              |
| Степень 1: значение от-<br>лично от NULL                         | false       | true          | true          | false             |
| Степень > 1: у всех<br>полей значение NULL                       | true        | false         | false         | true              |
| Степень > 1: у некото-<br>рых (не у всех) полей<br>значение NULL | false       | false         | true          | true              |
| Степень > 1: ни у одно-<br>го поля нет значения<br>NULL          | false       | true          | true          | false             |

#### 12.3.4. Предикат in

Предикат `in` позволяет специфицировать условие вхождения строчного значения в указанное множество значений. Синтаксические правила следующие:

```
in_predicate ::= row_value_constructor [ NOT ] IN
in_predicate_value
in_predicate_value ::= table_subquery
                    | (value_expression_comma_list)
```

Строка, являющаяся первым операндом, и таблица-второй операнд должны быть одинаковой степени. В частности, если второй операнд представляет собой список значений, то первый операнд должен иметь степень 1. Типы данных соответствующих столбцов операндов должны быть совместимы.

Пусть  $R_x$  обозначает строку-первый операнд, а  $S$  — множество строк второго операнда. Обозначим через  $R_s$  строку-элемент этого множества. Тогда условие  $R_x \text{ IN } S$  эквивалентно булевскому выражению  $\text{OR}_{R_s \in S} (R_x = R_s)$ . Другими словами,  $R_x \text{ IN } S$  принимает значение *true* в том и только том случае, когда во множестве  $S$  существует хотя бы один элемент  $R_s$ , такой, что значением предиката  $R_x = R_s$  является *true*.  $R_x \text{ IN } S$  принимает значение *false* в том и только том случае, когда для всех элементов  $R_s$  множества  $S$  значением операции сравнения  $R_x = R_s$  является *false*. Иначе значением условия  $R_x \text{ IN } S$  является *unknown*. Заметим, что для пустого множества  $S$  значением  $R_x \text{ IN } S$  является *false*.

По определению условие  $R_x \text{ NOT IN } S$  эквивалентно  $\text{NOT } (R_x \text{ IN } S)$ .

**Пример 12.7.** Найти номера, имена и номера отделов служащих, работающих в отделах 15, 17 и 19.

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO IN (15, 17, 19);
```

Конечно, эта формулировка запроса эквивалентна следующей формулировке (*пример 12.7, а*):

```
SELECT EMP_NO, EMP_NAME, DEPT_NO
FROM EMP
WHERE DEPT_NO = 15
      OR DEPT_NO = 17
      OR DEPT_NO = 19;
```



**Пример 12.8.** Найти номера сотрудников, не являющихся руководителями отделов и получающих заработную плату, размер которой равен размеру заработной платы какого-либо руководителя отдела.

```
SELECT EMP_NO
FROM EMP
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
  AND EMP_SAL IN (SELECT EMP_SAL FROM EMP, DEPT
  WHERE EMP_NO = DEPT_MNG);
```

Запросы, содержащие предикат IN с подзапросом, легко переформулировать в запросы с соединениями. Например, запрос из примера 12.3 эквивалентен следующему запросу с соединениями (**пример 12.8, а**):

```
SELECT DISTINCT EMP_NO
FROM EMP, EMP EMP1, DEPT
WHERE EMP_NO NOT IN (SELECT DEPT_MNG FROM DEPT)
  AND EMP_SAL = EMP1_SAL
  AND EMP1.EMP_NO = DEPT.DEPT_MNG;
```

По поводу второй формулировки следует сделать два замечания. Во-первых, как видно, была изменена только та часть условия, в которой использовался предикат IN, а предикат NOT IN остался. Запросы с предикатами NOT IN так просто не заменяются запросами с соединениями. Во-вторых, в разделе SELECT было добавлено ключевое слово DISTINCT, потому что в результате запроса во второй формулировке для каждого сотрудника будет содержаться столько строк, сколько существует руководителей отделов, получающих такую же заработную плату, что и данный сотрудник.

### 12.3.5. Предикат like

Формально предикат like определяется следующими синтаксическими правилами:

```
like_predicate ::= source_value [ NOT ] LIKE
pattern_value
[ ESCAPE escape_value ]
```

Все три операнда (source\_value, pattern\_value и escape\_value) должны быть одного типа, либо типа символьных строк, либо типа битовых строк. В первом случае значением последнего операнда должна быть строка из одного символа, во втором — стро-

ка из 8 бит. Второй операнд, как правило, задается литералом соответствующего типа. В обоих случаях значение предиката равняется *true* в том и только том случае, когда исходная строка (*source\_value*) может быть сопоставлена с заданным шаблоном (*pattern\_value*).

Если обрабатываются символьные строки и если раздел **ESCAPE** условия отсутствует, то при сопоставлении шаблона со строкой производится специальная интерпретация двух символов шаблона: символ подчеркивания ('\_') обозначает любой одиночный символ; символ процента ('%') обозначает последовательность произвольной длины произвольных символов (длина последовательности может быть нулевой). Если же раздел **ESCAPE** присутствует и специфицирует некоторый одиночный символ *x*, то пары символов '*x*\_' и '*x*%' представляют одиночные символы '\_' и '%' соответственно.

Значение предиката *like* есть *unknown*, если значение первого или второго операндов является неопределенным. Условие *x NOT LIKE y ESCAPE z* эквивалентно условию *NOT x LIKE y ESCAPE z*.

**Пример 12.9.** Найти номера отделов, сотрудники которых являются менеджерами проектов, и название каждого из этих проектов начинается с названия отдела.

```
SELECT DISTINCT DEPT.DEPT_NO
FROM EMP, DEPT, PRO
WHERE EMP.EMP_NO = PRO.PRO_MNG
      AND EMP.DEPT_NO = DEPT.DEPT_NO
      AND PRO.PRO_TITLE LIKE DEPT.DEPT_NAME || '%';
```

### 12.3.6. Предикат *similar*

Формально предикат *similar* определяется следующими синтаксическими правилами:

```
similar_predicate ::= source_value [ NOT ]
SIMILAR TO pattern_value
[ ESCAPE escape_value ]
```

Основное отличие предиката *similar* от рассмотренного ранее предиката *like* состоит в существенно расширенных возможностях задания шаблона, основанных на использовании правил построения *регулярных выражений*. Регулярные выражения предиката *similar* определяются следующими синтаксическими правилами:

```
regular_expression ::= regular_term
| regular_expression vertical_bar regular_term
```

```

regular_term ::= regular_factor | regular_term
regular_factor
regular_factor ::= regular_primary
                  | regular_primary *
                  | regular_primary +
regular_primary ::= character_specifier
                  | %
                  | regular_character_set
                  | ( regular_expression )
character_specifier ::= non_escape_character |
escape_character
regular_character_set ::=
    | left_bracket character_enumeration_list
    right_bracket
    | left_bracket ^ character_enumeration_list
    right_bracket
    | left_bracket : regular_charset_id :
    right_bracket
character_enumeration ::= character_specifier
                      | character_specifier -
                      character_specifier
regular_charset_id ::= ALPHA | UPPER | LOWER |
DIGIT | ALNUM

```

Поскольку в синтаксических правилах регулярных выражений символы '|', '[' и ']', используемые в качестве метасимволов в BNF, являются терминальными символами, они изображены как **vertical\_bar**, **left\_bracket** и **right\_bracket** соответственно.

Создаваемое по приведенным правилам регулярное выражение представляет собой символьную строку, содержащую все символы, которые требуется явно сопоставлять с символами строки-источника. В строке могут находиться специальные символы, представляющие собой заменители обычных символов ('%' и '\_'), обозначения операций ('|'), показатели числа возможных повторений ('\*' и '+') и т.д. При вычислении регулярного выражения образуются все возможные символьные строки, не содержащие специальных символов и соответствующие исходному шаблону. Тем самым, значением предиката **similar** является *true* в том и только том случае, когда среди всех символьных строк, генерируемых по регулярному выражению **pattern\_value**, найдется символьная строка, совпадающая с **source\_value**.

Рассмотрим несколько примеров регулярных выражений. Выражение '(This is string1)|(This is string2)' производит в точности

две символьные строки: '(This is string1)' и '(This is string2)'. В общем случае в круглых скобках могут находиться произвольные регулярные выражения `rexp1` и `rexp2`. Результатом вычисления '(rexp1)|(rexp2)' является множество символьных строк, генерируемых выражением `rexp1`, объединенное с множеством символьных строк, генерируемых выражением `rexp2`.

Выражение 'This is string [12]\*' генерирует символьные строки 'This is string', 'This is string 1', 'This is string 2', 'This is string 11', 'This is string 22', 'This is string 12', 'This is string 22', 'This is string 111' и т.д. Конструкция в квадратных скобках представляет собой один из вариантов определения набора символов (`regular_character_set`). В данном случае символы, входящие в определяемый набор, просто перечисляются. При вычислении регулярного выражения в каждой из генерируемых символьных строк конструкция в квадратных скобках заменяется одним из символов соответствующего набора.

Специальный символ '\*', стоящий после закрывающей квадратной скобки, является показателем числа повторений. «Звездочка» означает, что в генерируемых символьных строках элемент регулярного выражения, непосредственно предшествующий «звездочке», может появляться ноль или более раз. Использование в такой же ситуации специального символа '+' означает, что в генерируемых символьных строках элемент регулярного выражения, непосредственно предшествующий символу «плюс», может появляться один или более раз.

Другая форма определения набора символов иллюстрируется регулярным выражением 'This is string [:DIGIT:]'. В этом случае конструкция в квадратных скобках представляет собой любой одиночный символ, изображающий десятичную цифру. Другими допустимыми в SQL идентификаторами наборов символов (`regular_charset_id`) являются `ALPHA` (любой символ алфавита), `UPPER` (любой символ верхнего регистра), `LOWER` (любой символ нижнего регистра) и `ALNUM` (любой алфавитно-цифровой символ).

Определяемый набор символов может задаваться нижней и верхней границами диапазона значений кодов допустимых символов. Например, в регулярном выражении 'This is string [3-8]' конструкция в квадратных скобках представляет собой любой одиночный символ, изображающий цифры от 3 до 8 включительно. Заметим, что при задании диапазона можно использовать любые символы, но требуется, чтобы значение кода символа левой границы диапазона было не больше значения кода символа правой границы.

Наконец, имеется еще одна возможность определения набора символов. Соответствующая конструкция позволяет указать, какие символы из общего набора символов SQL не входят в определяемый набор символов. Например, регулярное выражение `'_S[^t]*ing%'` генерирует все символьные строки, у которых вторым символом является 'S', за которым (не обязательно непосредственно) следует подстрока 'ing', но между 'S' и 'ing' отсутствуют вхождения символа 't'.

Как и в предикате `like`, символ, определенный в разделе `ESCAPE`, поставленный перед любым специальным символом, отменяет специальную интерпретацию этого символа.

**Пример 12.10.** Найти номера и названия отделов, название которых начинается со слов 'Hardware' или 'Software', а за ними (не обязательно непосредственно) следует последовательность десятичных цифр, предваряемых символом подчеркивания.

```
SELECT DEPT_NAME, DEPT_NO
FROM DEPT
WHERE DEPT_NAME SIMILAR TO
'(HARD|SOFT)WARE%\_[:DIGIT:]+' ESCAPE '\\';
```

### 12.3.7. Предикат `exists`

Предикат `exists` определяется следующим синтаксическим правилом:

```
exists_predicate ::= EXISTS (query_expression)
```

Значением условия `EXISTS (query_expression)` является *true* в том и только том случае, когда мощность таблицы-результата выражения запросов больше нуля, иначе значением условия является *false*.

**Пример 12.11.** Найти номера отделов, среди служащих которых имеются менеджеры проектов.

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE EXISTS
    (SELECT EMP.EMP_NO
     FROM EMP
     WHERE EMP.DEPT_NO = DEPT.DEPT_NO
  AND EXISTS
      (SELECT PRO.PRO_MNG
       FROM PRO
       WHERE PRO.PRO_MNG = EMP.EMP_NO));
```

Запросы с предикатом **EXISTS** можно переформулировать в виде запросов с предикатом сравнения (*пример 12.11, а*):

```
SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT COUNT(*)
      FROM EMP, DEPT
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO
      AND PRO.PRO_MNG = EMP.EMP_NO ) >= 1;
```

### 12.3.8. Предикат **unique**

Этот предикат позволяет сформулировать условие отсутствия дубликатов в результате запроса:

`unique_predicate ::= UNIQUE (query_expression)`

Результатом вычисления условия **UNIQUE (query\_expression)** является *true* в том и только том случае, когда в таблице-результате выражения запросов отсутствуют какие-либо две строки, одна из которых является дубликатом другой. В противном случае значение условия есть *false*.

**Пример 12.12.** Найти номера отделов, служащих которых можно различить по имени и дате рождения.

```
SELECT DEPT_NO
FROM DEPT
WHERE UNIQUE
      (SELECT EMP_NAME, EMP_BDATE
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO);
```

### 12.3.9. Предикат **overlaps**

Предикат **overlaps** служит для проверки перекрытия во времени двух событий. Условие определяется следующим синтаксисом:

`overlaps_predicate ::= row_value_constructor  
OVERLAPS row_value_constructor`

Степень каждой из строк-операндов должна быть равна двум. Тип данных первого столбца каждого из операндов должен быть типом даты-времени, и типы данных первых столбцов должны быть совместимы. Тип данных второго столбца каждого из операндов должен быть типом даты-времени или интервала. Если это тип интервала, то точность типа должна быть такой, чтобы

интервал можно было прибавить к значению типа дата-время первого столбца. Если это тип дата-время, то он должен быть совместим с типом данных дата-время первого столбца.

Пусть **D1** и **D2** обозначают значения первого столбца первого и второго операндов соответственно. Если второй столбец первого операнда имеет тип дата-время, то пусть **E1** обозначает его значение. Если второй столбец первого операнда имеет тип **INTERVAL**, то пусть **I1** обозначает его значение, а  $E1 = D1 + I1$ . Если **D1** является неопределенным значением, или если  $E1 < D1$ , то пусть  $S1 = E1$  и  $T1 = D1$ . В противном случае, пусть  $S1 = D1$  и  $T1 = E1$ . Аналогично определяются **S2** и **T2** применительно ко второму операнду. Результат условия совпадает с результатом вычисления следующего булевского выражения:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))  
OR  
(S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))  
OR  
(S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

**Пример 12.13.** Найти номера проектов, которые выполнялись в период с 15 января 2000 г. по 31 декабря 2002 г.

```
SELECT PRO_NO  
FROM PRO  
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS  
      (DATE '2000-01-15', DATE '2002-12-31');
```

**Пример 12.14.** Найти названия проектов, которые будут выполняться в течение следующего года.

```
SELECT PRO_TITLE  
FROM PRO  
WHERE (PRO_SDATE, PRO_DURAT) OVERLAPS  
      (CURRENT_DATE, INTERVAL '1' YEAR);
```

### 12.3.10. Предикат сравнения с квантором

Этот предикат позволяет специфицировать квантифицированное сравнение строчного значения и определяется следующим синтаксическим правилом:

```
quantified_comparison_predicate ::= row_value_  
constructor  
                                comp_op { ALL | SOME | ANY }  
                                query_expression
```

Степень первого операнда должна быть такой же, как и степень таблицы-результата выражения запросов. Типы данных значений строки-операнда должны быть совместимы с типами данных соответствующих столбцов выражения запроса. Сравнение строк производится по тем же правилам, что и для предиката сравнения.

$Rx = Rs$

Обозначим через  $Rx$  строку-первый операнд, а через  $S$  — результат вычисления выражения запроса. Пусть  $Rs$  обозначает произвольную строку таблицы  $S$ .

Условие  $Rx \text{ comp\_op ALL } S$  имеет значение *true* в том и только том случае, когда  $S$  пусто, или значение условия  $Rx \text{ comp\_op } Rs$  равно *true* для каждой строки  $Rs$ . Условие  $Rx \text{ comp\_op ALL } S$  имеет значение *false* в том и только том случае, когда значение предиката  $Rx \text{ comp\_op } Rs$  равно *false* хотя бы для одной строки  $Rs$ . В остальных случаях значение условия  $Rx \text{ comp\_op ALL } S$  равно *unknown*.

Условие  $Rx \text{ comp\_op SOME } S$  имеет значение *false* в том и только том случае, когда  $S$  пусто, или значение условия  $Rx \text{ comp\_op } Rs$  равно *false* для каждой строки  $Rs$ . Условие  $Rx \text{ comp\_op SOME } S$  имеет значение *true* в том и только том случае, когда значение предиката  $Rx \text{ comp\_op } Rs$  равно *true* хотя бы для одной строки  $Rs$ . В остальных случаях значение условия  $x \text{ comp\_op SOME } S$  равно *unknown*.

Условие  $Rx \text{ comp\_op ANY } S$  эквивалентно условию  $Rx \text{ comp\_op SOME } S$ .

**Пример 12.15.** Найти номера сотрудников отдела номер 65, заработная плата которых в этом отделе не является минимальной.

```
SELECT EMP_NO
FROM EMP
WHERE DEPT_NO = 65
      AND EMP_SAL > SOME (SELECT EMP1.EMP_SAL
                          FROM EMP EMP1
                          WHERE EMP.DEPT_NO = EMP1.DEPT_NO);
```

**Пример 12.16.** Найти номера и имена сотрудников отдела 65, однофамильцы которых работают в этом же отделе.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65 AND
```



```
EMP_NAME = SOME (SELECT EMP1.EMP_NAME
                     FROM EMP EMP1
                     WHERE EMP.DEPT_NO = EMP1.DEPT_NO
                     AND EMP.EMP_NO <> EMP1.EMP_NO);
```

Возможно множество эквивалентных формулировок этого запроса, но, по-видимому, наиболее лаконичным образом этот запрос можно сформулировать с использованием соединения (*пример 12.16, а*):

```
SELECT DISTINCT EMP.EMP_NO, EMP.EMP_NAME
FROM EMP, EMP EMP1
WHERE EMP.DEPT_NO = 65
      AND EMP.EMP_NAME = EMP1.EMP_NAME
      AND EMP.DEPT_NO = EMP1.DEPT_NO
      AND EMP.EMP_NO <> EMP1.EMP_NO;
```

### 12.3.11. Предикат *match*

Предикат *match* позволяет сформулировать условие соответствия строчного значения результату табличного подзапроса. Синтаксис определяется следующим правилом:

```
match_predicate ::= row_value_constructor
MATCH [ UNIQUE ] [ SIMPLE | PARTIAL | FULL ]
query_expression
```

Степень первого операнда должна совпадать со степенью таблицы-результата выражения запроса. Типы данных полей первого операнда должны быть совместимы с типами соответствующих столбцов результата выражения запроса. Пусть *R<sub>x</sub>* обозначает строку-первый операнд.

Если отсутствует спецификация вида сопоставления или специфицирован тип сопоставления **SIMPLE**, то, если значение некоторого поля *R<sub>x</sub>* является неопределенным, значением условия является *true*. Если в *R<sub>x</sub>* нет неопределенных значений, то:

- если не указано **UNIQUE** и в результате выражения запроса существует (возможно, не уникальная) строка *R<sub>s</sub>* такая, что *R<sub>x</sub> = R<sub>s</sub>*, то значением условия является *true*;
- если указано **UNIQUE** и в результате выражения запроса существует уникальная строка *R<sub>s</sub>* такая, что *R<sub>x</sub> = R<sub>s</sub>*, то значением условия является *true*;
- в противном случае значением условия является *false*.

Если в условии присутствует спецификация **PARTIAL**, то, если значения всех полей *R<sub>x</sub>* являются неопределенными, значение условия есть *true*. В противном случае:

- если не указано **UNIQUE** и в результате выражения запроса существует (возможно, не уникальная) строка *Rs* такая, что каждое отличное от неопределенного значение *Rx* равно соответствующему значению *Rs*, то значение условия есть *true*;
- если указано **UNIQUE** и в результате выражения запроса существует уникальная строка *Rs* такая, что каждое отличное от неопределенного значение *Rx* равно соответствующему значению *Rs*, то значение условия есть *true*;
- в противном случае значение условия есть *false*.

Если в условии присутствует спецификация **FULL**, то, если значения всех полей *Rx* неопределенные, значение условия есть *true*. Если некоторые, но не все поля *Rx* содержат **NULL**, то значение условия есть *false*. Если ни одно значение в *Rx* не является неопределенным, то:

- если не указано **UNIQUE** и в результате выражения запроса существует (возможно, не уникальная) строка *Rs* такая, что  $Rx = Rs$ , то значение условия есть *true*;
- если указано **UNIQUE** и в результате выражения запроса существует уникальная строка *Rs* такая, что  $Rx = Rs$ , то значение условия есть *true*;
- в противном случае значение условия есть *false*.

**Пример 12.17.** Найти номера служащих и номера их отделов для служащих, для которых либо неизвестны номер отдела или дата рождения (или и то, и другое), либо в отделе данного служащего работает, по крайней мере, еще один человек с той же датой рождения.

```
SELECT EMP_NO, DEPT_NO
FROM EMP
WHERE (DEPT_NO, EMP_BDATE) MATCH SIMPLE
      (SELECT EMP1.DEPT_NO, EMP1.EMP_BDATE
       FROM EMP EMP1
       WHERE EMP1.EMP_NO <> EMP.EMP_NO);
```

### 12.3.12. Предикат **distinct**

Предикат **distinct** позволяет проверить, являются ли две строки дубликатами. Условие определяется следующим синтаксическим правилом:

```
distinct_predicate ::= row_value_constructor
IS DISTINCT FROM
                    row_value_constructor
```

Строки-операнды должны быть одинаковой степени. Типы данных соответствующих значений строк-операндов должны быть совместимы. Значением условия  $R_x$  IS DISTINCT FROM  $R_y$  является *true* в том и только том случае, когда строки  $R_x$  и  $R_y$  не являются дубликатами. В противном случае значением условия является *false*.

Заметим, что отрицательная форма условия — IS NOT DISTINCT FROM — в стандарте SQL не поддерживается. Вместо этого можно воспользоваться выражением NOT  $s_1$  IS DISTINCT FROM  $s_2$ .

**Пример 12.18.** Найти номера и имена служащих отдела 65, которых можно отличить по данным об имени и дате рождения от руководителя отдела 65.

```
SELECT EMP_NO, EMP_NAME
FROM EMP
WHERE DEPT_NO = 65
      AND (EMP_NAME, EMP_BDATE) DISTINCT FROM
          (SELECT EMP1.EMP_NAME, EMP1.EMP_BDATE
           FROM EMP EMP1, DEPT
           WHERE EMP1.DEPT_NO = EMP.DEPT_NO
                AND DEPT.DEPT_MNG = EMP1.EMP_NO);
```

## 12.4. Логические выражения раздела HAVING

Приведем примеры использования в логических выражениях раздела HAVING некоторых предикатов, обсуждавшихся в предыдущем подразделе. Теоретически в этих логических выражениях можно использовать все предикаты, но применение тех предикатов, которые мы проиллюстрируем, является более естественным.

### 12.4.1. Предикаты сравнения

**Пример 12.19.** Найти номера всех отделов, в которых средний размер заработной платы сотрудников превосходит 12 000 руб.

```
SELECT DEPT_NO
FROM EMP
WHERE DEPT_NO IS NOT NULL
GROUP BY DEPT_NO
HAVING AVG(EMP_SAL) > 12000.00;
```

Возможна формулировка этого запроса без использования разделов **GROUP BY** и **HAVING** (*пример 12.19, а*):

```
SELECT DISTINCT DEPT_NO
FROM EMP
WHERE (SELECT AVG(EMP1.EMP_SAL)
      FROM EMP EMP1
      WHERE EMP1.DEPT_NO = EMP.DEPT_NO) > 12000.00;
```

Обсудим, чем вторая формулировка отличается от первой. В соответствии с семантикой оператора **SELECT** при выполнении запроса 12.19, *а* для каждой строки таблицы **EMP** в цикле просмотра внешнего запроса будет выполняться подзапрос, который выберет из таблицы **EMP** (**EMP1**) все строки со значением столбца **DEPT\_NO**, равным значению этого столбца в текущей строке внешнего цикла. Другими словами, для каждой строки внешнего цикла образуется группа, для нее проверяется условие выборки, и в списке выборки используется имя столбца этой неявной группировки. Из-за того, что группа образуется и оценивается для каждой строки таблицы **EMP**, приходится включать в раздел **SELECT** спецификацию **DISTINCT**.

Формулировка, приведенная в примере 12.19, обеспечивает более четкие указания для выполнения запроса. Нужно сразу сгруппировать таблицу **EMP** в соответствии со значениями столбца **DEPT\_NO**, отобрать нужные группы и для каждой отобранной группы вычислить значения выражений списка выборки. В этом случае семантика выполнения запроса не предписывает выполнения лишних действий. Конечно, в развитой реализации SQL компилятор должен суметь понять, что формулировки 12.19 и 12.19, *а* эквиваленты, и избежать выполнения лишних действий.

**Пример 12.20.** Для каждого отдела найти его номер, имя руководителя, число сотрудников, минимальный, максимальный и средний размеры заработной платы сотрудников.

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*),
MIN(EMP1.EMP_SAL),
      MAX(EMP1.EMP_SAL), AVG(EMP1.EMP_SAL)
FROM DEPT, EMP, EMP EMP1
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO
GROUP BY DEPT.DEPT_NO, DEPT.DEPT_MNG, EMP.EMP_NO,
EMP.EMP_NAME
HAVING DEPT.DEPT_MNG = EMP.EMP_NO;
```

Этот запрос иллюстрирует несколько интересных черт языка SQL. Во-первых, это первый пример запроса с соединениями, в котором присутствуют разделы **GROUP BY** и **HAVING**. Во-вторых, одно условие соединения находится в разделе **WHERE**, а другое — в разделе **HAVING**. На самом деле можно было бы перенести в раздел **WHERE** и второе условие соединения, и, скорее всего, на практике использовалась бы следующая формулировка (*пример 12.20, а*):

```
SELECT DEPT.DEPT_NO, EMP.EMP_NAME, COUNT(*) ,  
MIN(EMP1.EMP_SAL) ,  
MAX(EMP1.EMP_SAL) , AVG(EMP1.EMP_SAL)  
FROM DEPT, EMP, EMP EMP1  
WHERE DEPT.DEPT_NO = EMP1.DEPT_NO  
AND DEPT.DEPT_MNG = EMP.EMP_NO  
GROUP BY DEPT.DEPT_NO, EMP.EMP_NAME;
```

Но первая формулировка тоже верна, поскольку второе условие соединения определено на столбцах группировки.

Наконец, легко видеть, что по существу группировка производится по значениям столбца **DEPT.DEPT\_NO**. Остальные столбцы, указанные в списке столбцов группировки, функционально определяются столбцом **DEPT.DEPT\_NO**. Тем не менее в первой формулировке мы включили в этот список столбцы **DEPT.DEPT\_MNG** и **EMP.EMP\_NO**, чтобы их имена можно было использовать в условии раздела **HAVING**, и столбец **EMP.EMP\_NAME**, чтобы можно было использовать его имя в списке выборки раздела **SELECT**. Другими словами, мы вынуждены расширять запрос избыточными данными, чтобы выполнить формальные синтаксические требования языка. Как видно, во второй формулировке мы смогли удалить из списка группировки два столбца. Кстати, не следует думать, что многословие первой формулировки мешает СУБД выполнить запрос настолько же эффективно, как запрос во второй формулировке. Грамотно построенный оптимизатор SQL сам приведет первую формулировку ко второй.

#### 12.4.2. Предикат in

*Пример 12.21.* Найти номера отделов, в которых средний размер заработной платы сотрудников равен максимальному размеру заработной платы сотрудников какого-либо другого отдела.

```
SELECT DEPT.DEPT_NO  
FROM DEPT, EMP
```

```

WHERE DEPT.DEPT_NO = EMP.DEPT_NO
GROUP BY DEPT.DEPT_NO
HAVING AVG(EMP.EMP_SAL) IN
      (SELECT MAX(EMP1.EMP_SAL)
       FROM EMP, DEPT DEPT1
       WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
       AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
       GROUP BY DEPT.DEPT_NO);

```

Этот запрос, помимо прочего, демонстрирует наличие в условии раздела **HAVING** вложенного подзапроса с корреляцией. Как и раньше, можно избавиться от разделов **GROUP BY** и **HAVING** во внешнем запросе (*пример 12.21, а*):

```

SELECT DEPT.DEPT_NO
FROM DEPT
WHERE (SELECT AVG(EMP_SAL)
      FROM EMP
      WHERE EMP.DEPT_NO = DEPT.DEPT_NO) IN
      (SELECT MAX(EMP1.EMP_SAL)
       FROM EMP, DEPT DEPT1
       WHERE EMP.DEPT_NO = DEPT1.DEPT_NO
       AND DEPT1.DEPT_NO <> DEPT.DEPT_NO
       GROUP BY DEPT.DEPT_NO);

```

Но в данном случае мы не можем отказаться от раздела **GROUP BY** во втором вложенном запросе, поскольку без этого невозможно получить множество значений результатов вызова агрегатной функции.

\* \* \*

В этой главе очень коротко описаны основные черты и возможности оператора выборки языка SQL. Скорее всего, описанных и продемонстрированных на примерах возможностей оператора **SELECT** хватит для практической работы с SQL-ориентированными базами данных, но, чтобы чувствовать себя совсем уверенно, лучше познакомиться с этим оператором в полном объеме. Частично в этом может помочь [38], но полную картину может предоставить только стандарт SQL. Проекты стандартов SQL, очень близкие к официально принятым документам, можно найти на сайте [39].

## БАЗОВЫЕ ВОЗМОЖНОСТИ МОДИФИКАЦИИ БАЗ ДАННЫХ В ЯЗЫКЕ SQL

Базы данных, по крайней мере в приложениях категории OLTP, являются высоко динамичными объектами. В таких приложениях на две операции выборки данных в среднем приходится одна операция обновления содержимого базы данных (добавления новых данных, удаления или модификации существующих данных). Поэтому для пользователей и разработчиков OLTP-приложений средства *манипулирования данными* по важности находятся на втором месте после средств выборки данных.

В этой главе обсуждаются средства манипулирования данными, входящие в *прямой SQL*. Заметим, что с точки зрения практики более важными являются средства манипулирования данными, выходящие за пределы прямого SQL и присутствующие во *встраиваемом* и *динамическом SQL*. Но, как уже неоднократно отмечалось, в этой книге не обсуждаются возможности использования SQL для создания приложений. По мнению автора, материал данной главы полезен для общего понимания специфики операторов манипулирования данными, а расширения этих операторов, присутствующие во *встраиваемом* и *динамическом SQL*, в любом случае нужно изучать совместно с другими аспектами этих уровней языка.

Глава состоит из трех подразделов. В первом подразделе обсуждаются синтаксис и семантика операторов манипулирования данными в предположении, что они действуют над базовыми таблицами. Во втором подразделе будет продемонстрировано, что в ряде случаев, специфицированных в стандарте языка SQL, операторы манипулирования данными можно применять к порождаемым таблицами и представлениям с однозначным отображением результатов действия этих операторов на соответствующие базовые таблицы. Третий подраздел посвящен механизму триггеров, которые, по существу, представляют собой «хранимые процедуры», автоматически вызываемые при возникновении соответствующих условий. Триггеры не обязательно связываются с действиями, производимыми при манипулирова-

нии данными, но поскольку одно из основных функций этого механизма состоит в поддержании целостности баз данных, то, как правило, такая связь имеется. Поэтому обсуждение механизма триггеров в соответствии со стандартом SQL включено именно в эту главу.

## 13.1. Базовые средства манипулирования данными

К базовым средствам манипулирования данными языка SQL относятся «поисковые» варианты операторов `UPDATE` и `DELETE`. Эти варианты называются поисковыми, потому что при задании соответствующей операции задается логическое условие, налагаемое на строки адресуемой оператором таблицы, которые должны быть подвергнуты модификации или удалению. Кроме того, к этой категории языковых средств относится оператор `INSERT`, позволяющий добавлять строки в существующие таблицы. Логично начать изложение именно с оператора `INSERT`, поскольку, для того чтобы можно было что-либо модифицировать в таблицах или удалять из таблиц, нужно, чтобы в таблицах содержались какие-то строки.

### 13.1.1. Оператор `INSERT` для вставки строк в существующие таблицы

Общий синтаксис оператора `INSERT` выглядит следующим образом:

```
INSERT INTO table_name
      { [ (column_commalist) ] query_expression
      | DEFAULT VALUES }
```

На вид синтаксические правила кажутся очень простыми, пока не вспомнишь, что обозначает синтаксическая категория `query_expression` (см. подразд. 12.1.2). Даже если ограничиться простейшей составляющей этой конструкции (`simple_table`), имеются следующие возможности:

```
simple_table ::= query_specification
              | table_value_constructor
              | TABLE table_name
```

**Вставка всех строк указанной таблицы.** Тем самым, стандарт допускает вставку в указанную таблицу всех строк некоторой другой таблицы (вариант `table_name`). Эта «дру-



гая» таблица может быть как базовой, так и представляемой. Естественно, в последнем случае в определении представления не должны присутствовать ссылки на таблицу, в которую производится вставка. При использовании этого варианта оператора вставки число столбцов вставляемой таблицы должно совпадать с числом столбцов таблицы, в которую производится вставка, или с числом столбцов, указанных в списке `column_commalist`, если этот список задан. Типы данных соответствующих столбцов вставляемой таблицы и таблицы, в которую производится вставка, должны быть совместимыми. Если в операции задан список `column_commalist` и в нем содержатся не все имена столбцов таблицы, в которую производится вставка, то в оставшиеся столбцы во всех строках заносятся значения столбцов по умолчанию. Если для какого-либо из оставшихся столбцов значение по умолчанию не определено, при выполнении операции вставки фиксируется ошибка.

**Пример 13.1.** Чтобы привести пример этого варианта операции `INSERT`, предположим, что в базе данных `EMP-DEPT-PRO` имеется еще одна промежуточная таблица `EMP_TEMP`, в которой временно хранятся данные о служащих, проходящих испытательный срок. Пусть эта таблица имеет заголовок, показанный на рис. 13.1.

В таблице `EMP_TEMP` хранятся неполные сведения о служащих, а именно те, которые требуются на время испытательного срока. Если выполнить операцию

```
INSERT INTO EMP (EMP_NO, EMP_NAME, EMP_BDATE)
TABLE EMP_TEMP;
```

то в основной таблице `EMP` появятся строки, соответствующие служащим, проходившим испытательный срок. При этом в столбцах `EMP_NO`, `EMP_NAME`, `EMP_BDATE` этих строк будут содержаться данные, взятые из таблицы `EMP_TEMP`, а в столбцах `EMP_SAL`, `DEPT_NO`, `PRO_NO` будут находиться значения, определенные для этих столбцов по умолчанию. Конечно, по-

|                    |
|--------------------|
| EMP_NO : EMP_NO    |
| EMP_NAME : VARCHAR |
| EMP_BDATE : DATE   |

Рис. 13.1. Заголовок таблицы `EMP_TEMP`

скольку столбец `EMP_NO` является первичным ключом таблицы `EMP` (по всей видимости, и таблицы `EMP_TEMP`), операция вставки будет успешно выполнена только в том случае, когда ограничение первичного ключа таблицы `EMP` не будет нарушено (конечно же,

требуется выполнение и всех других ограничений целостности, определенных для таблицы EMP).

**Вставка явно заданного набора строк.** Теперь обратимся к варианту оператора INSERT, в котором набор вставляемых строк задается явно с использованием синтаксической конструкции `table_value_constructor`. Напомним синтаксические правила, определяющие эту конструкцию:

```
table_value_constructor ::= VALUES
row_value_constructor_commalist
row_value_constructor ::= row_value_constructor_
element
| [ ROW ]
(row_value_constructor_element_commalist)
| row_subquery
row_value_constructor_element ::= value_expression
| NULL | DEFAULT
```

**Пример 13.2.** Самый простой пример использования этого варианта оператора вставки состоит в занесении в таблицу EMP явно задаваемых данных о новом служащем:

```
INSERT INTO EMP
      ROW (244543, 'Brown', '1985-04-08',
          16500.00, 630, 772);
```

В этом примере явно заданы значения всех столбцов заносимой строки (как показывают синтаксические правила, ключевое слово ROW можно опустить).

Возможен и такой вариант (*пример 13.2, а*):

```
INSERT INTO EMP
      ROW (244543, DEFAULT, NULL, DEFAULT, NULL,
          NULL);
```

В этом случае мы знаем очень мало о новом служащем, но уверены в том, что его имя и размер заработной платы должны быть назначены по умолчанию, а про дату рождения, номер отдела и номер проекта ничего не известно. Обратите внимание, что выполнение этой операции не нарушает ограничения целостности таблицы EMP.

Если обладать полной информацией об определении таблицы EMP, то формулировку операции примера 13.2, а можно переписать короче следующим эквивалентным образом (*пример 13.2, б*):

```
INSERT INTO EMP (EMP_NO) 244543;
```

**Пример 13.3.** В список элементов конструктора строки могут входить скалярные запросы, т. е. запросы, результат выполнения которых состоит из единственной строки, включающей единственный столбец. Поэтому допустима, например, такая операция вставки:

```
INSERT INTO EMP VALUES
  ROW (244543, (SELECT EMP_NAME
                FROM EMP
                WHERE EMP_NO = 255555),
      '1985-04-08',
      (SELECT EMP_SAL
        FROM EMP
        WHERE EMP_NO = 255555),
      NULL, NULL ),
  ROW (244544, (SELECT EMP_NAME
                FROM EMP
                WHERE EMP_NO = 255556),
      '1978-05-09',
      (SELECT EMP_SAL
        FROM EMP
        WHERE EMP_NO = 255556),
      NULL, NULL );
```

После выполнения этой операции в таблице **EMP** появятся две новые строки для служащих с уникальными идентификаторами **244543** и **244544**, причем первому из них будет присвоено имя и размер заработной платы сотрудника с уникальным идентификатором **255555**, а второму — аналогичные данные о служащем с уникальным идентификатором **255556**.

**Вставка строк результата запроса.** Наконец, обсудим вариант оператора вставки, когда набор вставляемых строк

|                         |
|-------------------------|
| DEPT_NO : DEPT_NO       |
| DEPT_EMP_NO : INTEGER   |
| DEPT_MAX_SAL : SALARY   |
| DEPT_MIN_SAL : SALARY   |
| DEPT_TOTAL_SAL : SALARY |

Рис. 13.2. Заголовок таблицы DEPT\_SUMMARY

определяется через спецификацию запроса. Предположим, например, что требуется сохранить в отдельной таблице **DEPT\_SUMMARY** сведения о числе сотрудников каждого отдела, их максимальной, минимальной и суммарной заработной плате. Пусть таблица **DEPT\_SUMMARY** уже создана и имеет заголовок, показанный на рис. 13.2 (не бу-

дем приводить полное определение таблицы, включающее требуемые ограничения целостности).

**Пример 13.4.** Тогда заполнить таблицу можно с помощью следующей операции вставки:

```
INSERT INTO DEPT_SUMMARY
    (SELECT DEPT_NO, COUNT(*), MAX (EMP_SAL),
        MIN (EMP_SAL), SUM (EMP_SAL)
     FROM EMP
     GROUP BY DEPT_NO);
```

### 13.1.2. Оператор UPDATE для модификации существующих строк в существующих таблицах

Общий синтаксис оператора UPDATE выглядит следующим образом:

```
UPDATE table_name SET update_assignment_commalist
    WHERE conditional_expression
update_assignment ::= column_name =
    { value_expression | DEFAULT | NULL }
```

Семантика оператора модификации существующих строк определяется следующим образом:

- для всех строк таблицы с именем **table\_name** вычисляется булевское выражение **conditional\_expression**. Строки, для которых значением этого булевского выражения является *true*, считаются подлежащими модификации (обозначим множество таких строк через  $S_m$ );
- каждая строка  $s$  ( $s \in S_m$ ) подвергается модификации таким образом, что значение каждого столбца этой строки, указанного в списке **update\_assignment\_commalist**, заменяется значением, указанным в правой части соответствующего элемента списка модификации\*. Значения столбцов строки  $s$ , не указанные в списке модификации, остаются неизменными.

Приведем пару примеров операций модификации таблиц.

**Пример 13.5.** Перевести всех служащих, выполняющих проект с номером 772, в отдел 632 и повысить им заработную плату на 1 000 руб.

---

\* Если в правой части элемента модификации присутствует скалярное выражение (**value\_expression**), в котором содержится запрос, то в случае использования в этом запросе имен столбцов модифицируемой таблицы под значениями этих столбцов понимается значение до модификации.

```
UPDATE EMP SET DEPT_NO = 632, EMP_SAL = EMP_SAL
+ 1000.00
WHERE PRO_NO = 772;
```

При выполнении этой операции на первом шаге в таблице **EMP** будут найдены все строки, относящиеся к служащим, которые участвуют в проекте с номером 772. На втором шаге во всех этих строках значение столбца **DEPT\_NO** будет изменено на 632, а к значению столбца **EMP\_SAL** будет прибавлено 1000.00.

**Пример 13.6.** Для всех служащих, работающих в отделах, заработная плата менеджеров которых превышает 30 000 руб., установить размер заработной платы, на 1 000 руб. превышающий средний размер заработной платы соответствующего отдела, а номера проектов, в которых участвуют эти служащие, сделать неопределенными.

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
+ 1000.00, PRO_NO = NULL
WHERE (SELECT EMP1.EMP_SAL
FROM EMP EMP1, DEPT
WHERE EMP.DEPT_NO = DEPT.DEPT_NO
AND DEPT_MNG = EMP1.EMP_NO AND) >
30000.00;
```

Конечно, запрос, фигурирующий в разделе **WHERE**, можно переформулировать с использованием вложенного подзапроса (*пример 13.6, а*).

```
UPDATE EMP SET EMP_SAL = (SELECT AVG (EMP1_SAL)
FROM EMP EMP1
WHERE EMP.DEPT_NO = EMP1.DEPT_NO)
+ 1000.00, PRO_NO = NULL
WHERE DEPT.NO IN (SELECT DEPT.DEPT_NO
FROM EMP, DEPT
WHERE DEPT_MNG = EMP_NO
AND EMP_SAL > 30000.00);
```

Эти примеры дают возможность понять, насколько сильны возможности оператора **UPDATE**. В разделе **WHERE** может содержаться любое условие, допускаемое в операторе выборки, а в элементах списка раздела **SET** может присутствовать любой вид скалярного выражения, в том числе любой запрос, вырабатывающий одиночное значение (скалярный подзапрос).

### 13.1.3. Оператор DELETE для удаления строк в существующих таблицах

Общий синтаксис оператора DELETE выглядит следующим образом:

```
DELETE FROM table_name
WHERE conditional_expression
```

В некотором смысле оператор DELETE является частным случаем оператора UPDATE (или, наоборот, действие оператор UPDATE представляет собой комбинацию действий операторов DELETE и INSERT).

Семантика оператора модификации существующих строк определяется следующим образом:

- 1) для всех строк таблицы с именем `table_name` вычисляется булевское выражение `conditional_expression`. Строки, для которых значением этого булевского выражения является *true*, считаются подлежащими удалению (обозначим множество таких строк через  $S_d$ );
- 2) каждая строка  $s$  ( $s \in S_d$ ) удаляется из указанной таблицы.

В целях иллюстрации изложенного приведем два примера операции удаления строк.

**Пример 13.7.** Удалить из таблицы EMP все строки, относящиеся к служащим, которые участвуют в проекте с номером 772.

```
DELETE FROM EMP WHERE PRO_NO = 772;
```

**Пример 13.8.** Удалить из таблицы EMP все строки, относящиеся к служащим, размер заработной платы которых превышает размер заработной платы менеджеров их отделов.

```
DELETE FROM EMP WHERE EMP_SAL >
    (SELECT EMP1.EMP_SAL
     FROM EMP EMP1, DEPT
     WHERE EMP.DEPT_NO = DEPT.DEPT_NO
     AND DEPT.DEPT_MNG = EMP1.EMP_NO);
```

Как и в операторе UPDATE, в разделе WHERE оператора DELETE можно использовать любой вид булевского выражения, допустимого в операторе выборки. Поэтому возможности оператора удаления строк ограничены лишь фантазией пользователя.

## 13.2. Представления, над которыми возможны операции обновления

В подразд. 12.2 было введено понятие представления (VIEW). Кратко повторим, что представление — это сохраняемое в каталоге базы данных выражение запросов, обладающее собственным именем и, возможно, собственными именами столбцов. Для удобства повторим синтаксическое правило определения представления:

```
create_view ::= CREATE [ RECURSIVE ] VIEW
table_name
    [ column_name_comma_list ]
AS query_expression
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

В операциях выборки к любому представлению можно адресоваться таким же образом, как и к любой базовой таблице. Естественно, возникает вопрос: а можно ли использовать имена представлений и в операциях обновления базы данных, и если такая возможность допускается, то как это следует понимать?

Напомним, что в соответствии с семантикой языка SQL при выполнении запроса, в разделе **FROM** которого прямо или косвенно присутствует имя представления, прежде всего производится *материализация* представления, т. е. вычисляется результат соответствующего выражения запросов, сохраняется во временной базовой таблице, и далее запрос выполняется по отношению к этой базовой таблице. Хотя в реализациях SQL обычно стремятся всячески избегать материализации представлений, любая реализация обязана обеспечить такое выполнение запроса над представлением, которое было бы эквивалентно выполнению запроса с явной материализацией представления.

Если допустить выполнение над представлениями операций обновления (сразу заметим, что, вообще говоря, в языке SQL это всегда разрешалось), то в этом случае семантика материализации явно не подходит. На первое место выходит то требование, что операция обновления над представлением должна однозначно отображаться в одну или несколько операций обновления над теми постоянно хранимыми базовыми таблицами, над которыми прямо или косвенно определено данное представление.

### 13.2.1. Представления, допускающие применение операций обновления, в стандарте SQL/92

Поскольку базовым элементом выражения запросов является спецификация запроса, прежде всего нужно понять, какой

класс спецификаций запросов является *допускающим операции обновления* (термин *updatable* — обновляемый, используемый в стандарте SQL, кажется не слишком удачным в русском варианте). В стандарте SQL/92 спецификация запроса считалась допускающей операции обновления в том и только том случае, когда выполнялись следующие условия:

- в разделе **SELECT** спецификации запроса отсутствует ключевое слово **DISTINCT** (т.е. не требуется удаление строк-дубликатов из результата запроса);
- все элементы списка выборки раздела **SELECT** являются именами столбцов, и ни одно имя столбца не встречается в этом списке более одного раза;
- в разделе **FROM** присутствует только одна ссылка на таблицу, и она указывает либо на базовую таблицу, либо на порождаемую таблицу, допускающую операции обновления;
- прямые или косвенные ссылки на базовую таблицу, прямо или косвенно идентифицируемую ссылкой на таблицу в разделе **FROM**, не встречаются в разделе **FROM** ни одного подзапроса, присутствующего в разделе **WHERE** спецификации данного запроса;
- в спецификации запроса отсутствуют разделы **GROUP BY** и **HAVING**.

Нетрудно убедиться в том, что эти требования являются достаточными для однозначной интерпретации операций обновления над представлениями. Например, пусть имеется следующая спецификация запроса (*пример 13.9*):

```
SELECT EMP_SAL
FROM (SELECT EMP_SAL, DEPT_NO
      FROM EMP
      WHERE EMP_NAME = (SELECT EMP_NAME
                        FROM EMP
                        WHERE EMP_NO = 4425))
      WHERE DEPT_NO <> 630;
```

Эту спецификацию можно упростить до эквивалентной формулировки\*

---

\* Обратите внимание, что формально эта формулировка не отвечает требованиям SQL/92 для спецификаций запросов, допускающих применение операций обновления. Но в действительности здесь вложенный подзапрос вычисляется в единственное значение при отсутствии какой-либо корреляции с внешним вхождением таблицы **EMP**.



```

SELECT EMP_SAL
FROM EMP
WHERE EMP_NAME = (SELECT EMP_NAME
                   FROM EMP
                   WHERE EMP_NO = 4425 )
                   AND DEPT_NO <> 630;

```

Предположим, что с данной спецификацией запроса связано представление с именем **EMPSAL**. Тогда операция

```
UPDATE EMPSAL SET EMP_SAL = EMP_SAL - 1000.00;
```

эквивалентна операции

```

UPDATE EMP SET EMP_SAL = EMP_SAL - 1000.00
WHERE EMP_NAME = (SELECT EMP_NAME
                  FROM EMP
                  WHERE EMP_NO = 4425 )
                  AND DEPT_NO <> 630;

```

Операция

```
DELETE FROM EMPSAL WHERE EMP_SAL > 20000.00;
```

эквивалентна операции

```

DELETE EMPSAL
WHERE EMP_SAL > 20000.00 AND
      EMP_NAME = (SELECT EMP_NAME
                  FROM EMP
                  WHERE EMP_NO = 4425 )
                  AND DEPT_NO <> 630;

```

Операция вставки над представлением **EMPSAL**

```
INSERT INTO EMPSAL 25000.00;
```

трактруется так:

```

INSERT INTO EMP
ROW (DEFAULT, DEFAULT, DEFAULT, 25000.00,
DEFAULT, DEFAULT);

```

Понятно, что такая операция будет отвергнута системой, потому что для столбца **EMP\_NO** таблицы **EMP** значения по умолчанию не определены (это первичный ключ таблицы, значения которого должны явно задаваться в любой операции вставки).

С другой стороны, условия допустимости операций обновления, специфицированные в SQL/92, не являются необходимыми.

Например, над представлением EMPMNG, определенным над спецификацией запроса («выбрать данные о служащих, являющихся руководителями отделов»)

```
SELECT *  
FROM EMP  
WHERE EXISTS (SELECT *  
FROM DEPT  
WHERE DEPT_MNG = EMP_NO);
```

можно было бы совершенно корректно выполнять операции обновления (с некоторыми оговорками насчет операции вставки; см. далее в этом подразделе).

### 13.2.2. Представления, допускающие применение операций обновления, в стандарте SQL:1999

В стандарте SQL:1999 правила применимости операций обновления к спецификации запроса существенно уточнены.

Введены понятия *потенциальной применимости операций обновления*, *применимости операций обновления*, *простой применимости операций обновления* и *применимости операции вставки*. К спецификации запроса потенциально применимы операции обновления в том и только том случае, когда выполняются следующие условия:

- в разделе **SELECT** спецификации запроса отсутствует ключевое слово **DISTINCT**;
- элемент списка выборки раздела **SELECT**, состоящий из ссылки на некоторый столбец, не может присутствовать в этом списке более одного раза;
- в спецификации запроса отсутствуют разделы **GROUP BY** и **HAVING**.

Если выражение запросов отвечает условиям потенциальной применимости операций обновления и в его разделе **FROM** присутствует в точности одна ссылка на таблицу, то к каждому столбцу выражения запроса, соответствующему одному столбцу таблицы из раздела **FROM**, применимы операции обновления. Если выражение запроса отвечает условиям потенциальной применимости операций обновления, но в его разделе **FROM** присутствуют две или более ссылки на таблицы, то операции обновления применимы к столбцу выражения запросов только при выполнении следующих условий:

- столбец порождается из столбца только одной таблицы из раздела **FROM**;

- эта таблица используется в выражении запросов таким образом, что сохраняются свойства ее первичного и всех возможных ключей.

Другими словами, к столбцу таблицы, которая отвечает условиям потенциальной применимости операций обновления, применимы операции обновления только в том случае, когда этот столбец может быть однозначно сопоставлен с единственным столбцом единственной таблицы, участвующей в выражении запроса, и каждая строка выражения запроса может быть однозначно сопоставлена с единственной строкой этой таблицы.

Выражение запросов удовлетворяет условию применимости операций обновления, если, по крайней мере, к одному столбцу выражения запросов применимы операции обновления. Выражение запросов удовлетворяет условию простой применимости операций обновления, если в разделе **FROM** выражения запросов содержится ссылка только на одну таблицу и все столбцы выражения запросов удовлетворяют условию применимости операций обновления.

Выражение запросов удовлетворяет условию применимости операций вставки, если оно удовлетворяет условию применимости операций обновления; каждая из таблиц, от которых зависит это выражение (т.е. таблиц, на которые имеются ссылки в разделе **FROM**), удовлетворяет условию применимости операций вставки; выражение запросов не содержит операций **UNION**, **INTERSECT** и **EXCEPT**. Конечно, это определение базируется на том факте, что для любой базовой таблицы условие применимости операции вставки удовлетворяется.

### 13.2.3. Раздел **WITH CHECK OPTION** определения представления

Пусть в базе данных содержится (упрощенная) таблица **EMP**, содержащая следующее множество строк (рис. 13.3).

Предположим, что в базе данных имеется представление **RICH\_EMP**, определенное следующим образом:

```
CREATE VIEW RICH_EMP AS
SELECT *
  FROM EMP
 WHERE EMP_SAL > 18000.00;
```

Понятно, что в соответствии с правилами SQL (и здравым смыслом) над этим представлением можно выполнять операции обновления. Как видно, в таблице **EMP** содержится строка, ко-

| EMP_NO | DEPT_NO | EMP_BDATE | EMP_SAL  |
|--------|---------|-----------|----------|
| 2440   | 1       | 1950      | 15000.00 |
| 2441   | 1       | 1950      | 16000.00 |
| 2442   | 1       | 1960      | 14000.00 |
| 2443   | 1       | 1960      | 19000.00 |
| 2444   | 2       | 1950      | 17000.00 |
| 2445   | 2       | 1950      | 16000.00 |
| 2446   | 2       | 1960      | 14000.00 |
| 2447   | 2       | 1960      | 20000.00 |
| 2448   | 3       | 1950      | 18000.00 |
| 2449   | 3       | 1950      | 13000.00 |
| 2450   | 3       | 1960      | 21000.00 |
| 2451   | 3       | 1960      | 22000.00 |

Рис. 13.3. Содержимое упрощенной таблицы EMP

торая соответствует служащему с номером 2447, получающему заработную плату в размере 20 000 руб. Естественно, эта строка будет присутствовать в виртуальной таблице RICH\_EMP. Поэтому можно было бы выполнить, например, операцию

```
UPDATE RICH_EMP
SET EMP_SAL = EMP_SAL - 3000
WHERE EMP_NO = 4452;
```

Но если выполнение такой операции действительно допускается, то в результате строка, соответствующая служащему с номером 2447, исчезнет из виртуальной таблицы RICH\_EMP. Аналогичный эффект возникнет при выполнении операции вставки

```
INSERT INTO RICH_EMP (EMP_NO) 2452;
```

В базовой таблице EMP появится строка, в которой значением столбца EMP\_NO будет 2452, а значения остальных столбцов будут установлены по умолчанию. В частности, значением столбца EMP\_SAL будет 10000.00. Тем самым, если подобная операция вставки действительно допустима, то мы вставили в виртуальную таблицу RICH\_EMP строку, которую в этой виртуальной таблице невозможно увидеть.

Чтобы избежать такого противоречивого поведения представляемых таблиц, нужно включать в определение представления раздел **WITH CHECK OPTION**. При наличии этого раздела до реального выполнения операций модификации или вставки строк через представление для каждой строки будет проверяться, что она соответствует условиям представления. Если это условие не выполняется хотя бы для одной модифицируемой или вставляемой строки, то операция полностью отвергается. В некотором смысле (при наличии раздела **WITH CHECK OPTION**) условие выборки, содержащееся в выражении запросов представления, можно считать *ограничением целостности* этого представления.

**Режимы проверки **CASCADED** и **LOCAL**.** Вспомним теперь, что в полном виде синтаксис раздела **WITH CHECK OPTION** может включать ключевые слова **CASCADED** или **LOCAL**. Обсудим их смысл. Предположим, что представление **V2** определяется над представлением **V1** следующим образом:

```
CREATE VIEW V2 AS
SELECT ...
FROM V1
WHERE ...
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Пусть над **V2** выполняется некоторая операция **O** обновления базы данных. Тогда:

- если представление **V2** определялось без раздела **WITH CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие ограничения целостности **V1** (если в определении **V1** присутствовал раздел **WITH CHECK OPTION**), но никаким образом не будут учитываться условия выборки, содержащееся в выражении запросов представления **V2**;
- если в определении представления **V2** содержался раздел **WITH LOCAL CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие ограничения целостности **V1**, и все условия, содержащееся в выражении запросов представления **V2**;
- если в определении представления **V2** содержался раздел **WITH CASCADED CHECK OPTION**, то при выполнении операции **O** будут проверяться все условия, определяющие ограничения целостности **V1** (так, как если бы в определении **V1** присутствовал раздел **WITH CASCADED CHECK OPTION**). Тем самым будут проверяться все ограничения целостности,

установленные для всех базовых таблиц, на которых основывается определение *V1*; все условия всех представлений, определенных над этими базовыми таблицами; все условия, содержащиеся в выражении запросов представления *V2*.

**Примеры результатов действия раздела WITH CHECK OPTION.** Чтобы пояснить результаты действия раздела WITH CHECK OPTION, допустим, что в базе данных присутствуют определения двух представлений MIDDLE\_RICH\_EMP и MORE\_RICH\_EMP:

```
CREATE VIEW MIDDLE_RICH_EMP AS
  SELECT *
    FROM EMP
   WHERE EMP_SAL < 20000.00
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
CREATE VIEW MORE_RICH_EMP AS
  SELECT *
    FROM MIDDLE_RICH_EMP
   WHERE EMP_SAL > 18000.00
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
```

Очевидно, что в тело (материализованного) представления MIDDLE\_RICH\_EMP будут входить строки базовой таблицы EMP, показанные на рис. 13.4, а в тело (материализованного) представления MORE\_RICH\_EMP — строки представляемой таблицы MIDDLE\_RICH\_EMP, показанной на рис. 13.5.

| EMP_NO | DEPT_NO | EMP_BDATE | EMP_SAL  |
|--------|---------|-----------|----------|
| 2440   | 1       | 1950      | 15000.00 |
| 2441   | 1       | 1950      | 16000.00 |
| 2442   | 1       | 1960      | 14000.00 |
| 2443   | 1       | 1960      | 19000.00 |
| 2444   | 2       | 1950      | 17000.00 |
| 2445   | 2       | 1950      | 16000.00 |
| 2446   | 2       | 1960      | 14000.00 |
| 2448   | 3       | 1950      | 18000.00 |
| 2449   | 3       | 1950      | 13000.00 |

Рис. 13.4. Тело материализованного представления MIDDLE\_RICH\_EMP

| EMP_NO | DEPT_NO | EMP_BDATE | EMP_SAL  |
|--------|---------|-----------|----------|
| 2443   | 1       | 1960      | 19000.00 |

Рис. 13.5. Тело материализованного представления **MORE\_RICH\_EMP**

Таблица 13.1. Варианты действия раздела **WITH CHECK OPTION**

| MORE_RICH_EMP | MIDDLE_RICH_EMP |          |          |
|---------------|-----------------|----------|----------|
|               | none            | LOCAL    | CASCADED |
| none          | Случай 1        | Случай 2 | Случай 3 |
| LOCAL         | Случай 4        | Случай 5 | Случай 6 |
| CASCADED      | Случай 7        | Случай 8 | Случай 9 |

В каждом из представлений **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** может отсутствовать или присутствовать (в одном из двух видов) раздел **WITH CHECK OPTION**. В совокупности возможен один из девяти случаев, указанных в табл. 13.1.

Чтобы рассмотреть каждый из возможных случаев по отдельности, обсудим, что будет происходить в каждом случае при выполнении следующих двух операций модификации строк (будем называть эти операции *U1* и *U2* соответственно)\*:

```
UPDATE MORE_RICH_EMP
SET EMP_SAL = EMP_SAL + 7000.00;
UPDATE MORE_RICH_EMP
SET EMP_SAL = EMP_SAL - 7000.00;
```

*Случай 1.* Ни в одном из представлений не содержится раздел **WITH CHECK OPTION**.

Первый неожиданный результат состоит в том, что после выполнения операции *U1* тело представления **MORE\_RICH\_EMP** становится пустым. Действительно, у единственной строки таблицы **EMP** (со значением **EMP\_NO**, равным 2443), одновременно удовлетворяющей условиям обоих представлений, столбец **EMP\_SAL** принимает значение 26000.00. После этого строка перестает удовлетворять условию представления **MIDDLE\_RICH\_EMP** и исчезает из результирующей таблицы **MORE\_RICH\_EMP**. Этот

---

\* Будем считать, что тем, кто пользуется представлением **MORE\_RICH\_EMP**, неизвестно ограничение **EMP\_SAL < 20000.00**, на котором основывается представление **MIDDLE\_RICH\_EMP**.

результат может быть особенно неожиданным для пользователей базы данных, которым известно, что условие представления **MORE\_RICH\_EMP** имеет вид **EMP\_SAL > 18000.00** и соблюдение этого условия должно сохраняться при увеличении размера заработной платы.

Выполнение операции **U2** также приведет к опустошению тела **MORE\_RICH\_EMP** (в базовой таблице **EMP** не останется ни одной строки, удовлетворяющей условию этого представления). Возможно, это будет достаточно естественно для пользователей представления **MORE\_RICH\_EMP**, которым известно условие представления, но люди, работающие с представлением **MIDDLE\_RICH\_EMP**, с удивлением обнаружат в теле результирующей таблицы новые строки.

*Случай 2.* В определении представления **MIDDLE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**, а в определении **MORE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует.

В этом случае, в соответствии с первыми двумя правилами проверки корректности выполнения операций обновления над представлениями, операция **U1** должна быть отвергнута системой (поскольку ее выполнение нарушает условие представления **MIDDLE\_RICH\_EMP**). Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех пользователей базы данных, которым известно только определение «верхнего» представления **MORE\_RICH\_EMP**, поскольку операция **U1** явно не может нарушить видимое ими ограничение.

С другой стороны, операция **U2** будет успешно выполнена и по-прежнему приведет к опустошению тела результирующей таблицы представления **MORE\_RICH\_EMP**.

*Случай 3.* В определении представления **MIDDLE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**, а в определении **MORE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует.

В этом варианте будут проверяться условия, содержащиеся в определении представления **MIDDLE\_RICH\_EMP**, а также все ограничения целостности таблицы **EMP** и всех других представлений, определенных над этой базовой таблицей. В результате операция **U1** будет отвергнута системой, а операция **U2** будет «успешно» выполнена. Другими словами, будет повторен *случай 2*.

*Случай 4.* В определении представления **MIDDLE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.



Понятно, что в этом варианте не сработает операция *U2* (ее выполнение не будет допущено условием «ограничения целостности» представления **MORE\_RICH\_EMP**). Но операция *U1* (увеличение размера заработной платы служащих) будет успешно выполнена, поскольку она не противоречит локальным ограничениям представления **MORE\_RICH\_EMP**.

*Случай 5.* В определениях представлений **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.

Выполнение обеих операций *U1* и *U2* будет справедливо отвергнуто. На первый взгляд, все в порядке. Но если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление *V*, то мы можем попасть в ситуацию *случая 2*, где *V* будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** — роль **MORE\_RICH\_EMP**.

*Случай 6.* В определении представления **MIDDLE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**.

Снова, если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление *V*, то мы можем попасть в ситуацию *случая 2*, где *V* будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** — роль **MORE\_RICH\_EMP**.

*Случай 7.* В определении представления **MIDDLE\_RICH\_EMP** раздел **WITH CHECK OPTION** отсутствует, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**.

Если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление *V*, то мы можем попасть в ситуацию *случая 3*, где *V* будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** — роль **MORE\_RICH\_EMP**.

*Случай 8.* В определении представления **MIDDLE\_RICH\_EMP** содержится раздел **WITH LOCAL CHECK OPTION**, а в определении **MORE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**.

Если над представлением **MORE\_RICH\_EMP** будет определено еще одно представление *V*, то мы можем попасть в ситуацию *случая 3*, где *V* будет играть роль **MORE\_RICH\_EMP**, а **MIDDLE\_RICH\_EMP** — роль **MORE\_RICH\_EMP**.

*Случай 9.* В определениях представлений **MIDDLE\_RICH\_EMP** и **MORE\_RICH\_EMP** содержится раздел **WITH CASCADED CHECK OPTION**.

Только в этом случае операции обновления будут выполняться корректно независимо от того, имеются ли в базе данных представления, определенные над **MORE\_RICH\_EMP** или между **MORE\_RICH\_EMP**, **MIDDLE\_RICH\_EMP** и **EMP**.

Очевидным выводом из приведенного анализа является то, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления) является включение в определение каждого представления раздела **WITH CASCADED CHECK OPTION**. В этом случае поведение системы будет оставаться корректным при введении дополнительных представлений над представлением **MORE\_RICH\_EMP**, между представлениями **MORE\_RICH\_EMP** и **MIDDLE\_RICH\_EMP** или между представлением **MIDDLE\_RICH\_EMP** и базовой таблицей **EMP**, если в определениях всех этих представлений присутствует раздел **WITH CASCADED CHECK OPTION**.

### 13.2.4. Исторический очерк

Завершим обсуждение возможностей применения операций обновления к виртуальным таблицам небольшим экскурсом в историю. На протяжении более чем 30-летней истории реляционных баз данных вопрос о возможности однозначной интерпретации операций обновления баз данных через виртуальные таблицы интересовал большое число исследователей. Причины этого интереса, по мнению автора, состоят в следующем.

Во-первых, как отмечалось в гл. 4, одной из наиболее привлекательных черт реляционной алгебры является замкнутость относительно понятия отношения. В любой алгебраической операции, операндом которой является отношение, в качестве операнда можно использовать алгебраическое выражение. С другой стороны, имеется явное неравноправие по отношению к операциям обновления: можно вставлять, модифицировать и удалять кортежи в базовых отношениях, но нельзя (в общем случае) применять эти операции к алгебраическим выражениям. Хотелось максимальным образом устранить это неравноправие.

Во-вторых, на первый взгляд, задача не является очень трудной (по крайней мере, если оставаться в пределах реляционной алгебры). Действительно, базовых операций совсем немного и каждая базовая операция очень проста.

К сожалению, это ощущение простоты проблемы оказалось обманчивым. Было выполнено множество исследований, опубликовано множество статей, но так и не удалось обнаружить

полное множество алгебраических выражений, для которых возможна однозначная интерпретация операций обновления. По мнению автора, эта ситуация оказала серьезное влияние на подход к решению проблемы применимости операций обновления к виртуальным таблицам, которым руководствуются разработчики языка SQL.

В двух первых международных стандартах (SQL/89 и SQL/92) к виду таких виртуальных таблиц предъявлялись чрезмерно строгие ограничения. Это показывают даже те простые примеры, которые приводились в начале этого подраздела. И конечно, наличие таких ограничений в стандарте языка приводило к тому, что в реализациях SQL появлялось много расширений, которые поддерживались только отдельными компаниями-производителями СУБД. Создается впечатление, что когда 10 лет назад был инициирован проект нового стандарта SQL-3 (который, в конце концов, привел к появлению SQL:1999), разработчики находились в состоянии растерянности.

Кстати, одна из идей, включавшихся в ранние варианты проекта SQL-3, состояла в том, чтобы расширить определение представляемой таблицы средствами, позволяющими явно специфицировать действия, которые нужно предпринимать при выполнении над представлением операций INSERT, UPDATE и DELETE. Другими словами, предлагалось переложить решение проблемы на плечи пользователей СУБД. Конечно, такой подход является абсолютно радикальным, но, с другой стороны, он мог бы привести к полной анархии.

Как вы могли видеть, в официально принятом стандарте SQL:1999 используется некоторый компромиссный подход. В стандарте не фиксируются жесткие правила, ограничивающие вид виртуальных таблиц, к которым применимы операции обновления. Вместо этого сформулирован ряд рекомендаций, которыми следует руководствоваться производителям СУБД. Нельзя утверждать, что такое решение является идеальным, но более удачного решения найти не удалось.

### **13.3. Операции обновления баз данных и механизм триггеров**

Термин *триггер* в контексте реляционных баз данных был введен в обиход участниками проекта System R (см. подразд. 8.1). В терминологии этого проекта триггером называлась хранимая в базе данных процедура, автоматически вызываемая СУБД

при возникновении соответствующих условий. При определении триггера указывались два условия его применимости — общее условие (имя отношения и тип операции манипулирования данными) и конкретное условие (логическое выражение, построенное по правилам, близким к правилам для ограничений целостности), а также действие, которое должно быть выполнено над БД при наличии условий применимости.

Конечно, термин *trigger* в данном контексте является жаргонным. Но, с другой стороны, он достаточно образно соответствует ситуации: для применения процедуры должны быть произведены «возбуждающие» ее действия. После завершения проекта System R на протяжении более 10 лет триггеры не поддерживались ни в одной коммерческой SQL-ориентированной СУБД. Но затем практически во всех ведущих СУБД механизм триггеров в том или ином виде был реализован.

В стандарте же языка SQL спецификации триггеров отсутствовали до принятия стандарта SQL:1999. По словам главного редактора стандартов SQL/92, SQL:1999 и SQL:2003 Джима Мелтона (Jim Melton), эта спецификация была уже полностью готова к моменту принятия SQL/92 и не была включена в текст стандарта только по причине ограниченности его объема. Однако, как кажется автору данного курса, этому препятствовали и расхождения в подходах, существовавшие между основными компаниями-производителями СУБД.

Заметим, что альтернативным термином по отношению к базам данным, содержащим триггерные процедуры, является термин *активная база данных*. Наверное, этот термин является более точным, поскольку действительно речь идет о базах данных, содержащих процедуры, которые автоматически вызываются при срабатывании связанных с ними правилами. Однако в обиходе пользователей SQL-ориентированных СУБД по-прежнему более распространен термин *триггер*.

### 13.3.1. Понятие триггера в SQL:1999

В языке обеспечиваются возможности определения триггеров, которые вызываются («срабатывают») при вставке одной или нескольких строк в указанную таблицу, при модификации одной или нескольких строк в указанной таблице или при удалении одной или нескольких строк из указанной таблицы. Вообще говоря, триггер может производить любое действие, требуемое для соответствующего приложения. Можно определить триггеры, срабатывающие по одному разу для операций INSERT,

UPDATE или DELETE, но существует и возможность определения триггеров, вызываемых при вставке, модификации или удалении каждой отдельной строки. Таблица, с которой связывается определение триггера, называется *предметной таблицей* (*subject table*), а оператор SQL, выполнение которого приводит к срабатыванию триггера, мы будем называть *инициирующим оператором* (*triggering SQL statement*).

Триггеры могут срабатывать *после* и *до* реального выполнения инициирующего оператора SQL. В теле триггера допускается доступ к значениям вставляемых, модифицируемых и удаляемых строк. В случае операции модификации возможен доступ к значениям строк до модификации и к значениям после модификации. В соответствии со стандартом SQL:1999 любой триггер ассоциируется в точности с одной базовой таблицей. Не допускается определение триггеров над представлениями\*.

Можно придумать различные способы полезного применения механизма триггеров, но принято считать, что основными областями использования этого механизма являются следующие.

- *Журнализация и аудит.* С помощью триггеров можно отслеживать изменения таблиц, для которых требуется поддержка повышенного уровня безопасности. Данные об изменении таблиц могут сохраняться в других таблицах и включать, например, идентификатор пользователя, от имени которого выполнялась операция обновления; временную метку операции обновления; сами обновляемые данные и т. д.
- *Согласование и очистка данных.* С любым простым оператором SQL, обновляющим некоторую таблицу, можно связать триггеры, производящие соответствующие обновления других таблиц. Например, с операцией вставки новой строки в таблицу EMP (прием на работу нового служащего) можно было связать триггер, модифицирующий значения столбцов DEPT\_EMP\_NO и DEPT\_TOTAL\_SAL\*\* строки таблицы DEPT со значением столбца DEPT\_NO, которое соответствует номеру отдела нового сотрудника.
- *Операции, не связанные с изменением базы данных.* В триггерах могут выполняться не только операции обновления

---

\* Непонятно, откуда происходит это ограничение. Скорее всего, в будущих версиях стандарта оно будет снято.

\*\* В примерах этого подраздела будем считать, что в столбце DEPT\_TOTAL\_SAL таблицы DEPT хранится суммарное значение заработной платы служащих соответствующего отдела.

базы данных. Стандарт SQL позволяет определять хранимые процедуры (которые могут вызываться из триггеров), посылающие электронную почту, печатающие документы и т. д.

### 13.3.2. Синтаксис определения триггеров и типы триггеров

Для более подробного обсуждения механизма триггеров в SQL:1999 необходимо ввести набор синтаксических правил.

```
trigger_definition ::=
    CREATE TRIGGER trigger_name
    { BEFORE | AFTER }
    { INSERT | DELETE | UPDATE
      [ OF column_commalist ] }
    ON table_name [ REFERENCING
      old_or_new_values_alias_list ]
    triggered_action
triggered_action ::=
    [ FOR EACH { ROW | STATEMENT } ]
    [ WHEN left_paren conditional_expression
      right_paren ]
    triggered_SQL_statement
triggered_SQL_statement ::= SQL_procedure_statement
    | BEGIN ATOMIC
    SQL_procedure_statement_semicolonlist
    END
old_or_new_values_alias ::= OLD [ ROW ] [ AS ]
correlation_name
    | NEW [ ROW ] [ AS ] correlation_name
    | OLD TABLE [ AS ] identifier
    | NEW TABLE [ AS ] identifier
```

Естественно, в языке имеется и конструкция, отменяющая определение триггера:

```
DROP TRIGGER trigger_name.
```

Как легко видеть, синтаксические правила допускают несколько разновидностей определения триггера. Кратко обсудим эти разновидности.

**Триггеры BEFORE и AFTER.** Если в определении триггера указано ключевое слово **BEFORE**, то триггер будет срабатывать непосредственно до выполнения операции обновления базовой таблицы соответствующим инициирующим оператором SQL.

При задании ключевого слова **AFTER** триггер будет вызываться немедленно после выполнения инициирующего оператора.

**Триггеры INSERT, UPDATE и DELETE.** Выбор одного из этих ключевых слов при определении триггера указывает природу события, которое должно приводить к срабатыванию триггера. При задании ключевого слова **INSERT** к срабатыванию триггера может привести только выполнение операции вставки строк в предметную таблицу. Если указываются ключевые слова **UPDATE** или **DELETE**, то число возможных событий, приводящих к срабатыванию триггера, возрастает. Кроме явных операций модификации строк предметной таблицы или удаления из нее строк к срабатыванию триггера могут привести ссылочные действия (см. подразд. 11.4.1).

Заметим, что в стандарте SQL:1999 отсутствует возможность определения триггеров, для которых событием было бы выполнение операции выборки из предметной таблицы. Разработчики стандарта сочли, что область применения триггеров такого рода весьма узка (трудно придумать какое-либо применение, кроме журнализации и аудита).

**Триггеры ROW и STATEMENT.** Если в определении триггера присутствует конструкция **FOR EACH ROW**, то триггер будет вызываться для каждой строки предметной таблицы, обновляемой инициирующим SQL-оператором. Если же задано **FOR EACH STATEMENT** (или явная спецификация **FOR EACH** отсутствует), то триггер сработает один раз для всего выполнения инициирующего SQL-оператора.

**Раздел WHEN.** Включение в определение триггера раздела **WHEN** с соответствующим условным выражением позволяет более точно специфицировать условие применимости триггера. Вычисление условного выражения производится над строками предметной таблицы, и триггер срабатывает только в том случае, когда значением условного выражения является *true*. Понятно, что виды и интерпретация логических выражений, допускаемых в разделе **WHEN**, различаются у триггеров с **FOR EACH ROW** и у триггеров с **FOR EACH STATEMENT**. В первом случае условное выражение вычисляется для одной строки, которая должна быть обновлена инициирующим SQL-оператором. Во втором случае условное выражение вычисляется для всей предметной таблицы целиком и, по всей видимости, должно базироваться на «кванторных» предикатах. Следует также понимать, что вычисление условия раздела **WHEN** данного триггера производится только в том случае, если произошло событие срабатывания триггера.

**Тело триггера.** Операции, которые должны быть выполнены при срабатывании триггера, специфицируются в синтаксической конструкции `triggered_SQL_statement` (будем называть ее *иницилируемым SQL-оператором*). Как видно из синтаксических правил, возможны два вида построения этой конструкции: в виде одиночного оператора SQL и в виде списка операторов со скобками `BEGIN ATOMIC` и `END`.

Недоумение читателей может вызвать неуточненная конструкция `SQL_procedure_statement`. Постараемся объяснить ее происхождение и смысл. Дело в том, что в стандарте SQL:1999 определено процедурное расширение SQL, называемое SQL/PSM (от *Persistent Stored Modules*). Это достаточно большой язык, который подробно в этой книге не рассматривается\*. Тем не менее для понимания синтаксиса определения триггеров необходимо отметить, что SQL/PSM включает основные операторы SQL, связанные с обновлением данных; язык является вычислительно полным, т. е. включает развитые средства вычислений; в языке содержатся средства определения и вызова функций и процедур; SQL/PSM содержит стандартный комплект управляющих конструкций — циклы, ветвления разных типов и т. д. Тем самым, `SQL_procedure_statement` — это любая процедура, определенная на языке SQL/PSM\*\*. В частности, эта процедура может представлять собой оператор SQL обновления базы данных.

Обсудим теперь, откуда возникает потребность в составном инициируемом SQL-операторе. Дело в том, что на практике при определении триггеров в качестве `SQL_procedure_statement` чаще всего используются операторы SQL обновления базы данных. Иногда (и мы покажем это на примере) для корректного определения функциональности триггера одного оператора не хватает, а в SQL отсутствует возможность определения составных операторов. Поэтому допускается использование средств определения составных операторов, присутствующих в SQL/PSM (`BEGIN ATOMIC` и `END`).

**Пример 13.10.** Для иллюстрации случая, когда при определении триггера достаточно специфицировать один оператор

---

\* Для читателей, которые имеют хотя бы минимальный опыт работы с продуктами компании Oracle, заметим, что во многих своих чертах SQL/PSM напоминает PL/SQL.

\*\* На самом деле для написания процедур, функций и методов допускается использование не только языка SQL/PSM, но и традиционных языков программирования, для которых в стандарте определены правила связывания с SQL.



|                    |
|--------------------|
| EMP_NO : EMP_NO    |
| EMP_NAME : VARCHAR |
| DEPT_NO : DEPT_NO  |

Рис. 13.6. Заголовок таблицы EMP\_DISMISSED

SQL, приведем пример определения триггера, условием срабатывания которого является выполнение операции вставки новой строки в таблицу EMP (прием на работу нового служащего). Если значение столбца DEPT\_NO в очередной вставляемой строке отлично от NULL, то триггер

должным образом модифицирует значения столбцов DEPT\_EMP\_NO и DEPT\_TOTAL\_SAL строки таблицы DEPT со значением столбца DEPT\_NO, которое соответствует номеру отдела нового сотрудника.

```
CREATE TRIGGER DEPT_CORRECTION AFTER INSERT ON EMP
FOR EACH ROW
WHEN (EMP.DEPT_NO IS NOT NULL)
UPDATE DEPT SET
DEPT_EMP_NO = DEPT_EMP_NO + 1,
DEPT_TOTAL_SAL = DEPT_TOTAL_SAL + EMP_SAL
WHERE DEPT.DEPT_NO = EMP.DEPT_NO;
```

Теперь предположим, что при увольнении служащего (удалении строки из таблицы EMP) мы хотим не только должным образом модифицировать таблицу DEPT, но и сохранять (в целях аудита) данные об уволенном сотруднике в таблице EMP\_DISMISSED\*, заголовок которой показан на рис. 13.6.

**Пример 13.11.** Определение соответствующего триггера могло бы выглядеть следующим образом:

```
CREATE TRIGGER EMP_DISMISSION AFTER DELETE ON EMP
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO EMP_DISMISSED
ROW (EMP.EMP_NO, EMP.EMP_NAME,
EMP.DEPT_NO);
UPDATE DEPT SET
DEPT_EMP_NO = DEPT_EMP_NO - 1,
DEPT_TOTAL_SAL = DEPT_TOTAL_SAL - EMP_SAL
WHERE DEPT.DEPT_NO = EMP.DEPT_NO
END;
```

---

\* Для упрощения будем считать, что идентификаторы уволенных служащих не используются повторно.

### 13.3.3. Выполнение триггеров

При выполнении каждого триггера система устанавливает *контекст выполнения триггера*. Выполнение любого оператора SQL, обновляющего базовую таблицу базы данных, может привести к срабатыванию одного или нескольких триггеров, а выполнение операторов SQL, содержащихся в триггерах, может привести к обновлению других базовых таблиц. Эти «внутритриггерные» (инициируемые) операторы выполняются в контексте текущего триггера, но их выполнение может привести к срабатыванию других триггеров. Для каждого из «вторичных» триггеров образуется собственный контекст выполнения, позволяющий определить их действия точно и независимо от действий первого набора триггеров. Выполнение вторичных триггеров может привести к срабатыванию «третичных» триггеров и так далее — допускается произвольная глубина вложенности. Для каждого триггера на каждом уровне образуется собственный контекст.

Контекст выполнения триггера всегда является атомарным, т.е. иницилируемый SQL-оператор либо успешно завершается, либо результаты его действия гарантированно отсутствуют в базе данных.

Обсудим понятие контекста триггера более подробно. Предположим, что в базе данных EMP-DEPT-PRO должно поддерживаться то правило, что каждый служащий, становящийся руководителем проекта, автоматически получает прибавку к заработной плате в 10 000 руб. (Для простоты будет считать, что снятие служащего с позиции руководителя проекта не приводит к автоматическому изменению его заработной платы и что для каждого служащего, являющегося руководителем проекта, определен номер отдела, в котором он работает.) Тогда мы могли бы определить триггер CHANGE\_MNG\_NO следующим образом:

```
CREATE TRIGGER CHANGE_MNG_NO AFTER UPDATE OF  
PRO_MNG ON PRO  
FOR EACH ROW  
    UPDATE EMP SET EMP_SAL = EMP_SAL + 10000.00  
    WHERE EMP_NO = PRO_MNG;
```

Но очевидно, что для поддержания корректности данных в таблице DEPT нам требуется триггер, условием срабатывания которого было бы изменение значений столбца EMP\_SAL в таблице EMP. Определим соответствующий триггер DEPT\_CORRECTION\_1:

```

CREATE TRIGGER DEPT_CORRECTION_1 AFTER UPDATE OF
EMP_SAL ON EMP
  REFERENCING OLD ROW AS OLD_EMP NEW ROW AS
  NEW_EMP
  FOR EACH ROW
  UPDATE DEPT SET
    DEPT_TOTAL_SAL =
      DEPT_TOTAL_SAL + NEW_EMP.EMP_SAL -
      OLD_EMP.EMP_SAL
  WHERE EMP.DEPT_NO = DEPT.DEPT_NO;

```

Пусть теперь выполняется операция

```

UPDATE PRO SET PRO_MNG = 4455
WHERE PRO_NO = 554;

```

Сразу после выполнения этой операции сработает триггер **CHANGE\_MNG\_NO**. Этот триггер будет выполняться в контексте, который мы для удобства назовем *контекстом CMN*. Заметим, что исходный оператор модификации в действительности изменяет только одну строку таблицы **PRO**, но триггеру **CHANGE\_MNG\_NO** это неизвестно, и он будет работать так, как если изменялось произвольное число строк таблицы **PRO**.

Выполнение операции модификации таблицы **EMP** приведет к срабатыванию триггера **DEPT\_CORRECTION\_1**. В этот момент контекст **CMN** будет «упрятан в стек», и образуется и станет активным контекст следующего триггера — *контекст DR1*. После завершения выполнения этого триггера контекст **DR1** больше не требуется и он ликвидируется, а из стека восстанавливается контекст **CMN**, в котором и будет завершено выполнение триггера **CHANGE\_MNG\_NO**.

Контекст выполнения триггера служит для того, чтобы обеспечить СУБД данными, необходимыми для корректного выполнения иницируемого оператора SQL. Эти данные представляют собой набор *изменений состояния*, где каждое изменение состояния описывает изменение данных в целевой таблице триггера. Изменение состояния включает следующие данные:

- триггерное событие — **INSERT**, **UPDATE** или **DELETE**;
- имя предметной таблицы триггера;
- имена столбцов предметной таблицы, специфицированных в определении триггера (только для триггеров по **UPDATE**);
- набор *переходов* (представление всех строк, вставляемых в предметную таблицу, модифицируемых в ней или удаляемых из нее), а также список всех триггеров уровня

**STATEMENT**, уже выполненных в некотором (не обязательно активном) контексте выполнения, и список всех триггеров уровня **ROW**, уже выполненных в некотором (не обязательно активном) контексте выполнения, и строк, над которыми эти триггеры выполнялись.

Отслеживание уже выполненных триггеров ведется для предотвращения многократного выполнения одного и того же триггера в результате возникновения одного события, что могло бы потенциально привести к заикливанию выполнения системы триггеров.

При создании контекста выполнения триггера его набор изменений состояния изначально пуст. В набор изменений состояния добавляется каждое встречающееся «новое» изменение состояния, в котором не дублируются триггерное событие существующего изменения состояния, имя предметной таблицы и имени столбцов предметной таблицы. Набор переходов каждого изменения состояния изначально пуст, и переходы добавляются при каждом обновлении предметной таблицы, ассоциированной с изменением состояния (включая обновления, производимые ссылочными действиями).

**Возможности использования старых и новых значений.** Мы уже продемонстрировали использование старых и новых значений в определении триггера **DEPT\_CORRECTION\_1**. Поскольку эта возможность является важной особенностью языка SQL, обсудим ее более подробно.

Сначала немного поговорим о синтаксисе. Итак, в определении триггера может присутствовать раздел **REFERENCING old\_or\_new\_values\_alias\_list**, причем список определений псевдонимов может включать следующие элементы:

```
OLD [ ROW ] [ AS ] correlation_name
NEW [ ROW ] [ AS ] correlation_name
OLD TABLE [ AS ] identifier
NEW TABLE [ AS ] identifier
```

Каждая из этих конструкций может входить в список определений псевдонимов не более одного раза, и спецификации **OLD ROW** и **NEW ROW** могут присутствовать только в определении триггеров уровня **ROW**. Определяемые корреляционные имена и псевдонимы можно использовать внутри триггера для ссылок на значения предметной таблицы. Если определяются корреляционное имя для новых значений (**NEW ROW**) или псевдоним для нового содержимого таблицы (**NEW TABLE**), то эти имена можно использовать для ссылок на значения, которые будут существо-

вать в предметной таблице *после* выполнения операций **INSERT** или **UPDATE**. Если же определяется корреляционное имя для старых значений (**OLD ROW**) или псевдоним для старого содержимого таблицы (**OLD TABLE**), то эти имена можно использовать для ссылок на значения, которые существовали в предметной таблице *до* выполнения операций **UPDATE** или **DELETE**. Конечно, нельзя использовать **NEW ROW** или **NEW TABLE** в триггерах **DELETE**, поскольку никакие новые значения не создаются. Аналогично, нельзя использовать **OLD ROW** или **OLD TABLE** в триггерах **INSERT**, поскольку никакие старые значения не существовали.

Таблицы, на которые указывают корреляционные имена или псевдонимы, называются *переходными*. Эти таблицы не сохраняются в базе данных долговременно; они создаются и уничтожаются динамически, по мере своей надобности в контексте выполнения триггера. В триггерах уровня **ROW** можно использовать корреляционное имя, определенное в конструкции **OLD ROW**, для ссылки на значения строки, удаляемой или модифицируемой иницилирующим оператором, в том виде, в котором эта строка существовала в предметной таблице до того, как была удалена или модифицирована при выполнении иницилирующего оператора. В триггерах этого уровня можно также использовать псевдоним, определенный в конструкции **OLD TABLE**, для ссылки на любое значение переходной таблицы в том виде, в котором она находилась до удаления или модификации очередной строки при выполнении иницилирующего оператора. В триггерах уровня **ROW** можно использовать псевдоним, определенный в конструкции **OLD TABLE**, для ссылки на любое значение предметной таблицы в том виде, в котором она находилась до выполнения иницилирующего оператора. Аналогично обстоят дела с использованием корреляционных имен и псевдонимов, определенных в конструкциях **NEW ROW** и **NEW TABLE**.

Для триггеров категории **BEFORE** имеется существенное ограничение: в них не разрешается использовать конструкции **OLD TABLE** и **NEW TABLE**, а внутритриггерный SQL-оператор не может производить какие-либо изменения в базе данных. Основанием для такого ограничения является то, что на переходные таблицы, порождаемые **OLD TABLE** и **NEW TABLE**, могут существенно влиять ссылочные действия, которые активизируются в результате изменений базы данных при выполнении внутритриггерного SQL-оператора. Поэтому значения строк в таких таблицах могут оказаться нестабильными и недостаточно предсказуемыми, если триггер срабатывает раньше действия триггерного оператора SQL.

**Обработка нескольких триггеров, связанных с одной предметной таблицей.** В SQL:1999 не запрещается определе-

ние нескольких триггеров, ассоциированных с одной предметной таблицей, относящихся к одной и той же категории (**BEFORE** или **AFTER**) и срабатывающих по одному и тому же событию. Понятно, что при возникновении условия срабатывания всех таких триггеров система должна выбрать порядок, в котором они будут выполняться.

Решение, принятое в SQL, является предельно простым, хотя и несколько странным. При определении каждого триггера фиксируется временная метка выполнения оператора **CREATE TRIGGER**, и все триггеры, ассоциированные с одной предметной таблицей, относящиеся к одной и той же категории (**BEFORE** или **AFTER**) и срабатывающие по одному и тому же событию, упорядочиваются в соответствии со своими временными метками. Тогда при возникновении условия срабатывания всех триггеров одной группы сначала выполняется первый триггер, затем второй и т.д. В стандарте не специфицируется точность временной метки, связываемой с триггером, и если в одной группе обнаруживаются два или более триггеров с неразличимыми временными метками, то порядок их выполнения должен определяться в реализации.

Подход к установлению порядка выполнения триггеров в соответствии с их временными метками может вызвать чисто практические трудности у пользователей SQL-ориентированных СУБД. Например, если в ходе разработки приложения выяснится потребность в определении нового триггера, который должен выполняться раньше некоторого существующего триггера той же группы, то стандарт не может предложить ничего лучшего, кроме как уничтожить определения всех триггеров этой группы, а затем заново определить их в нужном порядке.

И еще одно интересное свойство триггеров в SQL:1999. Как уже отмечалось, каждый иницируемый SQL-оператор должен являться атомарным, т.е. если его выполнение завершается неуспешно, то в базе данных не должно остаться никаких следов этого выполнения. Но в стандарте указывается больше: неуспешное выполнение хотя бы одного триггера из группы с одинаковым условием срабатывания должно приводить к отмене результатов выполнения иницируемых SQL-операторов всех триггеров этой группы, а также к отмене результатов выполнения самого иницирующего SQL-оператора\*.

---

\* Помимо прочего, этот факт означает, что определение в базе данных нового триггера может привести к неработоспособности существующих приложений, разработчики которых, вообще говоря, могут даже и не знать о появлении нового триггера.

### 13.3.4. Триггеры и ссылочные действия

В подразд. 11.4.1 достаточно подробно обсуждался механизм определения ссылочных действий, служащий для автоматической поддержки ссылочной целостности. Кратко напомним, что ссылочные действия автоматически модифицируют значения внешнего ключа соответствующей таблицы при удалении или модификации строк таблицы, на которую указывают ссылки.

Конечно, ссылочные действия весьма напоминают триггеры, и в некоторых SQL-ориентированных СУБД ссылочные действия реализуются на основе общего механизма триггеров. Разработчики стандарта SQL:1999 считают этот подход неудачным, поскольку процедурная природа триггеров входит в противоречие с тщательно разработанной декларативной природой ссылочных ограничений целостности. Другими словами, спецификация ссылочной целостности, содержащаяся в стандарте, препятствует возможности встраивания в триггер упрощенного процедурного кода.

Однако даже в тех СУБД, где не смешиваются механизмы ссылочных действий и триггеров, неминуемо возникает взаимосвязь между ссылочными действиями, изменяющими некоторую таблицу, и триггерами, которые определены на этой таблице или также изменяют ее. В SQL:1999 эта взаимосвязь немного упрощается за счет того, что контролирование всех ограничений целостности (включая ссылочные ограничения) и выполнение всех ссылочных действий должны производиться до срабатывания триггеров категории **AFTER**. Если выполняется некоторая операция обновления таблицы *T*, то после ее выполнения и срабатывания всех ссылочных действий инициируются все триггеры, ассоциированные с таблицей *T* и видом произведенной операции, а также соответствующие триггеры, ассоциированные с любой таблицей, которая затрагивалась ссылочным действием, если в этой таблице была изменена хотя бы одна строка. Конечно, срабатывание триггера может привести к новым ссылочным действиям, которые повлекут срабатывание других триггеров и т. д.

В заключение этого подраздела, посвященного механизму триггеров, заметим, что многие спецификации стандарта SQL:1999 выглядят недостаточно убедительными. По всей видимости, полезные на практике триггеры слишком сложны с точки зрения теории. Создается впечатление, что за годы, прошедшие после завершения проекта System R, с этими трудностями так и не удалось справиться. Отсюда практический совет: если вам

действительно требуется использование триггеров, обращайтесь к документации используемой вами СУБД, а если и документация не содержит абсолютно понятных рекомендаций, прибегайте к осмысленным экспериментам.

\* \* \*

В этой главе были рассмотрены важные аспекты языка SQL, относящиеся к механизмам обновления данных. В первом подразделе были рассмотрены операторы прямого SQL, предназначенные для вставки, модификации и удаления данных из существующих таблиц. Операторы UPDATE и DELETE этой категории иногда называют поисковыми, поскольку в них включаются условия на строки таблицы, которые должны быть модифицированы или удалены. В языке SQL определены также *позиционные* операторы модификации и удаления строк, а также *динамические позиционные* варианты этих операторов, но для их обсуждения требуется общее рассмотрение встраиваемого и динамического SQL, что выходит за рамки данной книги. По мнению автора, поисковые версии операторов модификации и удаления хорошо характеризуют соответствующие возможности языка SQL.

Второй подраздел посвящен обсуждению возможностей языка SQL, связанных с применимостью операций обновления базы данных через виртуальные таблицы, в том числе, через представления. Мы рассмотрели ограничения языка SQL/92, накладываемые на виртуальные таблицы, к которым применимы операции обновления. Отмечалось, что эти ограничения являются достаточными, но не необходимыми для применения операций обновления. Был продемонстрирован подход стандарта SQL:1999, где предлагаются рекомендации, но не требования, которым следует придерживаться реализациям SQL, чтобы соответствовать стандарту.

Наконец, в третьем подразделе главы рассматривался механизм триггеров: основные понятия триггеров, которые были введены при выполнении проекта System R; основные синтаксические конструкции, предназначенные для определения триггеров, а также их базовая семантика, затем обсуждались принципы выполнения триггеров, заложенные в стандарт SQL:1999, и, наконец, имеющиеся взаимосвязи между ссылочными действиями и триггерами.



# МЕХАНИЗМЫ АВТОРИЗАЦИИ ДОСТУПА И УПРАВЛЕНИЯ ПОДКЛЮЧЕНИЯМИ, СЕССИЯМИ И ТРАНЗАКЦИЯМИ В ЯЗЫКЕ SQL

В этой главе будут обсуждаться средства языка, которые касаются скорее администраторов баз данных, а не конечных пользователей или программистов приложений. Но, по мнению автора, любой квалифицированный пользователь SQL-ориентированной базы данных должен иметь представление об административных средствах SQL (тем более, что средства управления транзакциями в большой степени затрагивают и интересы конечного пользователя).

Данная глава включает материал, в меньшей степени концептуально связанный, чем это было в предыдущих главах, посвященных языку SQL. В первом подразделе главы рассматриваются базовые идеи авторизации доступа к данным, заложенные в основу языка SQL. Метод авторизации доступа, используемый в SQL относится к *мандатным (mandatory)* видам защиты данных. При этом подходе с каждым зарегистрированным в системе пользователем (*субъектом*) и каждым защищаемым объектом системы связывается *мандат*, определяющий действия, которые может выполнять данный субъект над данным объектом. В отличие от этого, при применении *дискреционного (discretionary)* метода ограничения доступа с каждым из объектов системы связываются одна или несколько *категорий пользователей*, каждой из которых позволяются или запрещаются некоторые действия над объектом.

Следующий подраздел посвящается фундаментальному в области баз данных (и не только в этой области) понятию *транзакции* — последовательности операций над базой данных (в общем случае включающей операции обновления базы данных), которая воспринимается системой как одна единая операция. При классическом подходе к управлению транзакциями следуют принципу *ACID (Atomicity, Concurrency, Isolation, Durability)*. Этому принципу следовали и разработчики языка SQL.

В последнем, основном, подразделе данной главы обсуждаются средства языка SQL, предназначенные для управления *сессиями и подключениями* пользовательских приложений. По-

давляющее большинство реализаций языка SQL основывается на архитектурной модели *клиент/сервер*. Приложения обычно выполняются на *клиентской* аппаратуре, отделенной (по крайней мере, логически) от *серверной* аппаратуры, на которой работает собственно СУБД. Чтобы получить доступ к базе данных, приложение должно *подключиться* к серверу и образовать *сессию* в этом подключении. У приложения может одновременно существовать несколько подключений к разным серверам баз данных, но не более одной сессии в каждом подключении.

## 14.1. Поддержка авторизации доступа к данным в языке SQL

В общем случае база данных является слишком дорогостоящим предметом, чтобы можно было использовать ее в автономном режиме. Обычно с достаточно большой базой данных (параллельно или последовательно) работает много приложений и пользователей, и не для всех них было бы разумно обеспечивать равноправный доступ к хранящимся данным.

В языке SQL (SQL:1999) имеются возможности контролировать доступ к разным объектам базы данных, в том числе, к следующим объектам:

- таблицам;
- столбцам таблиц;
- представлениям;
- доменам;
- наборам символов\*;
- порядкам сортировки символов (*collation*);
- преобразованиям (*translation*);
- триггерам;
- подпрограммам, вызываемым из SQL;
- определенным пользователями типам.

В совокупности в SQL:1999 может поддерживаться девять видов защиты разных объектов в соответствии со следующими возможными действиями (табл. 14.1).

При разработке средств контроля доступа к объектам баз данных создатели SQL придерживались принципа сокрытия информации об объектах, содержащихся в схеме базы данных, от пользователей, которые лишены доступа к этим объектам. Другими словами,

---

\* Напомним, что в этой книге не затрагиваются вопросы интернационализации и локализации языка SQL.

Таблица 14.1. Объекты и привилегии

| Вид защиты и соответствующее действие | Название привилегии | Применимо к следующим объектам                                                                         |
|---------------------------------------|---------------------|--------------------------------------------------------------------------------------------------------|
| Просмотр                              | SELECT              | Таблицы, столбцы, подпрограммы, вызываемые из SQL                                                      |
| Вставка                               | INSERT              | Таблицы, столбцы                                                                                       |
| Модификация                           | UPDATE              | Таблицы, столбцы                                                                                       |
| Удаление                              | DELETE              | Таблицы                                                                                                |
| Ссылка                                | REFERENCES          | Таблицы, столбцы                                                                                       |
| Использование                         | USAGE               | Домены, определенные пользователями типы, наборы символов, порядки сортировки символов, преобразования |
| Инициирование                         | TRIGGER             | Таблицы                                                                                                |
| Выполнение                            | EXECUTE             | Подпрограммы, вызываемые из SQL                                                                        |
| Подтипизация                          | UNDER               | Структурные типы                                                                                       |

если некоторый пользователь не обладает, например, привилегией на просмотр таблицы **PRO**, то при выполнении операции **SELECT FROM PRO** он получит такое же диагностическое сообщение, как если бы таблица **PRO** не существовала. Если бы в случае отсутствия этой таблицы и в случае отсутствия привилегии доступа выдавались разные диагностические сообщения, то непривилегированный пользователь получил бы данные о том, что интересующая его таблица существует, но он лишен к ней доступа.

В SQL-ориентированной системе баз данных каждому зарегистрированному пользователю соответствует его уникальный идентификатор (в стандарте используется термин *идентификатор авторизации, authorization identifier* — *authID*). В стандарте SQL:1999 не зафиксированы точные правила представления идентификатора пользователя, хотя обычно в реализациях SQL функция **CURRENT USER** выдает текстовую строку, содержащую регистрационное имя пользователя, как оно сохраняется в файлах соответствующей операционной системы. Привилегии доступа

к объектам базы данных могут предоставляться пользователям, представляемых своими идентификаторами, а также *ролям*, выполнение которых, в свою очередь, может предоставляться пользователям. Кроме того, в SQL поддерживается концепция псевдоидентификатора (или идентификатора псевдо-) пользователя **PUBLIC**, который соответствует любому приложению или пользователю, зарегистрированному в системе баз данных. «Пользователю» **PUBLIC** могут предоставляться привилегии доступа к объектам базы данных, как и любому другому пользователю.

В модели контроля доступа SQL создатель любого объекта базы данных автоматически становится *владельцем* этого объекта. При этом владелец объекта может идентифицироваться либо своим идентификатором пользователя, либо именем своей роли. Вообще говоря, владелец объекта обладает полным набором привилегий для выполнения действий над объектом (с одним исключением, которое мы обсудим в этом подразделе позже). Владелец объекта, помимо прочего, обладает привилегией на передачу всех (или части) своих привилегий другим пользователям или ролям. В частности, владелец объекта может передать другим пользователям или ролям привилегию на передачу привилегий дальнейшим пользователям или ролям (эти действия с передачей привилегии на передачу привилегий могут продолжаться рекурсивно).

Во многих реализациях поддерживаются привилегии уровня **DBA (DataBase Administrator)** для возможности выполнения операций **CREATE**, **ALTER** и **DROP** над объектами, входящими в схему базы данных. В стандарте SQL требуется лишь соблюдение следующих правил:

- любой пользователь или его роль может выполнять любые подобные операции внутри схемы, которой владеет\*;
- не допускается выполнение каких-либо таких операций внутри схемы, которой не владеет пользователь или роль, пытающиеся выполнить соответствующую операцию;
- эти правила не допускают исключений.

#### 14.1.1. Пользователи и роли

Как отмечалось в начале этого подраздела, любой пользователь характеризуется своим *идентификатором авторизации*.

---

\* В соответствии со стандартом любые зарегистрированные в системе пользователи или роли автоматически являются владельцами части схемы базы данных, имена объектов которой начинаются с соответствующего идентификатора, за которым следует символ '!'.

В стандарте ничего не говорится о том, что **authID** должен быть идентичен *регистрационному имени пользователя* в смысле операционной системы. Согласно стандарту SQL:1999 **authID** строится по тем же правилам, как и любой другой идентификатор, и может включать до 128 символов. Тем не менее во многих реализациях SQL, выполненных в среде операционных систем семейства UNIX, длина **authID** составляет не более восьми символов, как это свойственно ограничениям на длину регистрационного имени в этих операционных системах.

В стандарте языка SQL не специфицированы средства создания идентификаторов авторизации. Более точно, в стандарте не определяется какой-либо явный способ создания допустимых идентификаторов пользователей. Идентификатор авторизации может являться либо идентификатором пользователя, либо именем роли, а для создания ролей в SQL поддерживаются соответствующие средства (см. далее). Но в соответствии с правилами стандарта SQL все **authID** должны отслеживаться СУБД (имеются в виду все **authID**, для которых существует хотя бы одна привилегия). И в стандарте поддерживаются точные правила порождения и распространения привилегий. Привилегии по отношению к объекту базы данных предоставляются системой владельцу схемы при создании объекта в этой схеме, и привилегии могут явно передаваться от имени одного **authID** другому **authID** при наличии у первого **authID** привилегии на передачу привилегий.

Итак, **authID** может являться либо идентификатором пользователя, либо идентификатором роли. Попробуем разобраться в сути термина *роль*. При работе с большими базами данных в больших организациях часто сотни служащих производят одни и те же операции над базой данных. Конечно, для этого каждый из этих служащих должен быть зарегистрированным пользователем соответствующей системы баз данных и, тем самым, обладать собственными **authID**. Используя базовые средства авторизации доступа (зафиксированные в стандарте SQL/92), можно предоставить каждому пользователю группы одни и те же привилегии доступа к требуемым объектам базы данных. Но схема авторизации доступа при этом становится очень сложной\*. В некотором

---

\* Для каждого объекта базы данных и для каждого пользователя, обладающего какими-либо привилегиями доступа к этому объекту, требуется хранить список его привилегий. Если учесть еще и возможность передачи привилегий от одного пользователя к другому, то образуется произвольно сложный граф, за которым трудно следить администраторам базы данных.

смысле имя *роли* идентифицирует динамически образуемую группу пользователей системы баз данных, каждый из которых обладает, во-первых, привилегией на исполнение данной роли и, во-вторых, всеми привилегиями данной роли для доступа к объектам базы данных. Другими словами, наличие ролей упрощает построение и администрирование системы авторизации доступа. Проиллюстрируем это на рис. 14.1.

Каждая стрелка на рис. 14.1 соответствует мандату доступа (паре <authID, набор\_привилегий\_доступа\_к\_объекту\_БД>), который требуется сохранять в каталоге базы данных и проверять при попытке доступа от имени authID. Как видно, в случае (а) требуется сохранение и проверка  $n \cdot m$  мандатов, где  $n$  — число пользователей в группе, а  $m$  — число объектов базы данных, для которых пользователи группы должны иметь одни и те же привилегии. В случае (б) число требуемых для корректной работы мандатов равно лишь  $n + m$ , и схема авторизации резко упрощается.

Группы пользователей, объединенных одной ролью, являются динамическими, поскольку в SQL поддерживаются возможности предоставления пользователю привилегии на исполнение данной

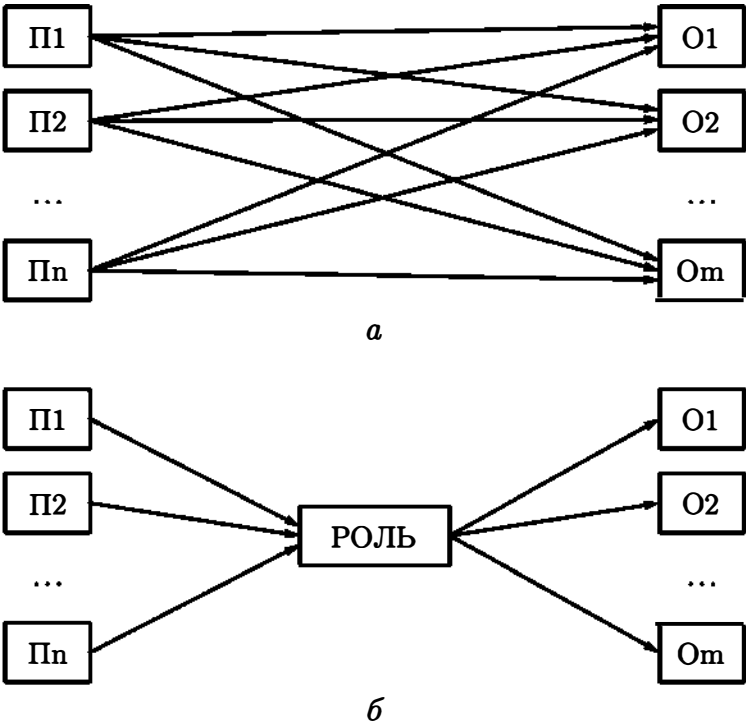


Рис. 14.1. Привилегии, пользователи и роли:

а — определение привилегий пользователей без использования роли; б — то же, с использованием роли

роли и лишения пользователя этой привилегии (см. далее в этом подразделе). Более того, имеются возможности предоставления заданной роли *A* всех или части привилегий другой роли *B*. Естественно, что при этом привилегии изменяются у всех пользователей, которые могут исполнять роль *A*.

#### 14.1.2. Использование идентификаторов пользователей и имен ролей

В этом подразделе мы вынуждены использовать понятие *SQL-сессии* (подробнее см. подразд. 14.3). Как и в предыдущих главах, посвященных языку SQL, можно оправдать эту нелогичность только рекурсивной природой стандарта SQL.

Итак, в любой момент каждая существующая SQL-сессия ассоциируется с идентификатором пользователя, называемым *идентификатором пользователя SQL-сессии*, и с именем роли, называемым *именем роли SQL-сессии*. Почти всегда привилегии, связанные с этими идентификатором и именем, используются для определения допустимости выполнения различных операций во время данной сессии.

В стандарте не специфицированы все способы ассоциирования *authID* с SQL-сессией. Определено лишь то, что если сессия образуется с помощью использования оператора **CONNECT** (см. подразд. 14.3), то *authID* указывается в качестве параметра соответствующей операции. Для реализаций SQL допускается, чтобы пользовательский идентификатор SQL-сессии совпадал с регистрационным именем пользователя с точки зрения операционной системы или являлся идентификатором, специально устанавливаемым специалистами организации, ответственными за обеспечение безопасности. Кроме того, допускается наличие в реализации оператора **SET SESSION AUTHORIZATION**, применение которого приводит к смене пользовательского идентификатора SQL-сессии. В начале SQL-сессии значение *текущего идентификатора пользователя SQL-сессии* совпадает со значением пользовательского идентификатора SQL-сессии, и это сохраняется до тех пор, пока пользовательский идентификатор SQL-сессии не будет каким-либо образом изменен. Значение текущего пользовательского идентификатора SQL-сессии возвращается вызовом функцией **SESSION\_USER**.

Для каждой SQL-сессии существует также *текущее имя роли* (это имя можно получить путем вызова функции **CURRENT\_ROLE**). Сразу после образования сессии текущему имени роли соответствует неопределенное значение, что трактуется в смысле

«роль для сессии не назначена». Имеется несколько способов подмены пользовательского идентификатора и/или имени роли SQL-сессии. При этом, если задается идентификатор пользователя, то одновременно полагается, что неявно указывается имя роли, имеющее неопределенное значение. Если же задается имя роли, то, за несколькими исключениями, полагается, что неявно указывается идентификатор пользователя, имеющий неопределенное значение (подробнее см. далее).

Если либо текущий идентификатор пользователя, либо текущее имя роли содержат неопределенное значение, то тот идентификатор или то имя, у которого значение не является неопределенным, используются в качестве *текущего* `authID` SQL-сессии. Если ни текущий идентификатор, ни текущее имя не содержат `NULL`, то текущим `authID` сессии служит текущий идентификатор пользователя.

Как и везде в этой книге, опустим детали, относящиеся к особенностям авторизации при использовании встраиваемого и динамического SQL.

### 14.1.3. Создание и ликвидация ролей

Для создания новой роли используется оператор `CREATE ROLE`, определяемый следующим синтаксическим правилом:

```
CREATE ROLE role_name  
[ WITH ADMIN { CURRENT_USER | CURRENT_ROLE } ]
```

Имя создаваемой роли должно отличаться от любого идентификатора авторизации, уже определенного и сохраненного в базе данных. В случае успешного создания роли некоторый `authID` получает привилегию на исполнение данной роли. Если в операторе `CREATE ROLE` не содержится раздел `WITH ADMIN`, то привилегию на исполнение роли получает текущий идентификатор пользователя SQL-сессии, если значение этого идентификатора отлично от `NULL`; иначе привилегия на исполнение роли дается текущему имени роли сессии.

Если в состав оператора включается раздел `WITH ADMIN`, то можно выбрать, будет ли являться владельцем роли `authID`, соответствующий текущему идентификатору пользователя SQL-сессии, или `authID`, соответствующий текущему имени роли (при условии, что соответствующие текущий идентификатор или текущее имя не содержат `NULL`). Кроме того, включение этого раздела означает, что `authID`-владелец роли получает право на передачу привилегии исполнения данной роли другим `authID`.



В соответствии со стандартом SQL:1999, привилегии, требуемые для выполнения оператора **CREATE ROLE**, определяются в реализациях SQL. Например, в некоторых реализациях выполнение этой операции разрешается только администратору базы данных.

Существующую роль можно ликвидировать с помощью оператора

```
DROP ROLE role_name
```

Для выполнения этой операции требуется, чтобы текущий **authID** SQL-сессии прямо или косвенно (через цепочку ролей) являлся владельцем ликвидируемой роли. При ликвидации роли прежде всего изымается привилегия на ее исполнение у всех **authID**, которым эта привилегия была ранее передана.

#### 14.1.4. Передача привилегий и ролей

Для передачи привилегий и ролей от одних **authID** другим поддерживается оператор **GRANT**, который мы обсудим в отдельности для случаев передачи привилегий и передачи ролей.

**Передача привилегий.** В случае передачи привилегий используется следующий синтаксис оператора **GRANT**:

```
GRANT { ALL PRIVILEGES | privilege_commalist }
      ON privilege_object
TO { PUBLIC | authID_commalist } [ WITH GRANT
OPTION ]
    [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
privilege ::= SELECT [ column_name_commalist ]
            | DELETE
            | INSERT [ column_name_commalist ]
            | UPDATE [ column_name_commalist ]
            | REFERENCES [ column_name_commalist ]
            | USAGE
            | TRIGGER
            | EXECUTE
privilege_object ::= [ TABLE ] table_name
                   | DOMAIN domain_name
                   | CHARACTER SET character_set_name
                   | COLLATION collation_name
                   | TRANSLATION translation_name
```

Поскольку **authID** может являться идентификатором пользователя или именем роли, привилегии могут передаваться

от пользователей пользователям, от пользователей ролям, от ролей ролям и от ролей пользователям.

В списке привилегий можно использовать **SELECT**, **DELETE**, **INSERT**, **UPDATE**, **REFERENCES** и **TRIGGER** только в том случае, когда в качестве объекта привилегий указывается таблица. Соответственно, список привилегий может состоять из единственной привилегии **USAGE** только в том случае, когда объектом является домен, набор символов, порядок сортировки или трансляция. Если в списке привилегий указывается более одной привилегии, то они все передаются указанным **authID**, но для этого текущий **authID** SQL-сессии должен обладать привилегией на передачу привилегий.

Использование ключевого слова **ALL PRIVILEGES** вместо явного задания списка привилегий означает, что передаются все привилегии доступа к соответствующему объекту базы данных, которыми обладает текущий **authID** SQL-сессии.

Как показывает синтаксис, один оператор **GRANT** позволяет передавать привилегии доступа только к одному объекту, но в том случае, когда объектом является таблица, разные привилегии могут передаваться по отношению к одному и тому же набору столбцов или к разным наборам. Если при указании привилегий **SELECT**, **DELETE**, **UPDATE** и **REFERENCES** список имен столбцов не задается, передаются привилегии по отношению ко всем столбцам таблицы. Заметим, что эти привилегии касаются всех существующих столбцов данной таблицы, а также всех столбцов, которые когда-либо будут к ней добавлены.

Включение в оператор необязательного раздела **WITH GRANT OPTION** означает, что получателям передаваемых привилегий дается также привилегия на дальнейшую передачу полученных привилегий, включая привилегию на передачу привилегий. Включение в оператор раздела **GRANTED BY** позволяет явно указать, передаются ли привилегии от имени текущего идентификатора пользователя или же от имени текущего имени роли.

При проверке возможности выполнения операции в SQL-сессии учитываются привилегии текущего **authID** SQL-сессии, а также привилегии всех ролей, которые переданы данному **authID**. Поскольку этим ролям могли быть переданы другие роли, обладающие собственными привилегиями, анализ возможности выполнения операции является рекурсивной процедурой.

Если одна и та же привилегия передается более одного раза одному и тому же **authID2** от имени одного и того же **authID1**, то возникает ситуация, называемая *избыточной*

*дублирующей привилегией.* Эта ситуация не вызывает дополнительных проблем, поскольку избыточная передача привилегии игнорируется. Для аннулирования этой привилегии у **authID2** от имени **authID2** требуется выполнение всего лишь одной операции **REVOKE** (см. далее в этом разделе). Если привилегия была один раз передана **authID2** от имени **authID1** вместе с привилегией на передачу этой привилегии (**WITH GRANT OPTION**), а в другой раз — без этой опции (порядок действий не является существенным), то **authID2** обладает данной привилегией и привилегией на ее передачу.

Если производится попытка передачи нескольких привилегий, но соответствующий **authID** не обладает ни одной из этих привилегий, то фиксируется ошибка. Аналогично, если производится попытка передачи нескольких привилегий с передачей привилегии на передачу привилегий, но соответствующий **authID** не обладает привилегией **WITH GRANT OPTION** ни для одной из передаваемых привилегий, то фиксируется ошибка. Наконец, если производится попытка передачи нескольких привилегий с передачей привилегии на передачу привилегий, и соответствующий **authID** обладает привилегией на передачу только части этих привилегий, то в результате выполнения операции вырабатывается предупреждение, но соответствующая часть привилегий передается с привилегией **WITH GRANT OPTION**.

**Привилегии и представления.** При определении представлений действуют специальные правила определения привилегий над этими представлениями. Если при создании обычных объектов базы данных, таких как таблица или домен, текущий **authID** автоматически получает все возможные привилегии доступа к соответствующему объекту, включая привилегию на передачу привилегий, то для представлений дела обстоят по-другому. Поскольку создаваемое представление всегда основывается на одной или нескольких базовых таблицах (или представлениях), привилегии, которые получает создатель представления, должны основываться на привилегиях, которыми располагает текущий **authID** по отношению к этим базовым таблицам или представлениям.

Например, чтобы операция создания представления была выполнена успешно, текущий **authID** должен обладать привилегией **SELECT** по отношению ко всем базовым таблицам и представлениям, на которых основывается новое представление. Тогда текущий **authID** автоматически получит привилегию **SELECT** для нового представления. Но текущий **authID** сможет

передавать эту привилегию другим **authID** только тогда, когда обладает соответствующей привилегией для всех базовых таблиц и представлений, на которых основывается новое представление. Аналогичным образом на представление распространяются привилегии **DELETE**, **INSERT**, **UPDATE** и **REFERENCES**. Поскольку триггеры над представлениями создавать не разрешается, представлениям не передается привилегия **TRIGGER**.

Наконец, обсудим, что происходит при смене привилегий владельца представления по отношению к таблицам, на которых основано это представление. Для простоты предположим, что представление **V** основано на базовой таблице **T**. Если во время создания **V** текущий **authID** (будущий владелец представления) обладал по отношению к **T** привилегиями **SELECT** и **INSERT**, то он будет обладать этими привилегиями и по отношению к **V**<sup>\*</sup>. Если впоследствии владелец представления получит по отношению к **T** дополнительные привилегии, то он (и все **authID**, которым передавались все привилегии — **ALL PRIVILEGES** для **V**) получают те же привилегии для **V**. Должно быть понятно, каким образом обобщается этот подход на случай, когда представление определяется над несколькими таблицами или представлениями.

**Передача ролей.** Для передачи ролей используется следующий вариант оператора **GRANT**:

```
GRANT role_name_commalist
  TO { PUBLIC | authID_commalist }
  [ WITH ADMIN OPTION ]
  [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
```

Как показывает синтаксис, оператор позволяет передавать произвольное число ролей произвольному числу **authID** (которые могут представлять собой идентификаторы пользователей или имена ролей). Как и в случае передачи привилегий, от данного **authID** можно передавать только те роли, которые были получены этим **authID** с привилегией на дальнейшую передачу (**WITH ADMIN OPTION**). При включении в состав оператора **GRANT** раздела **GRANTED BY** можно явно указать, передаются ли роли от имени текущего идентификатора пользователя или же от имени текущего имени роли.

---

<sup>\*</sup> Это один из тех случаев, когда *иметь право* не означает автоматически *иметь возможность реализации своего права*. SQL допускает, например, наличие привилегии **INSERT** для представления, к которому операция **INSERT** не применима.

#### 14.1.5. Изменение текущих идентификаторов пользователей и имен ролей

Как отмечалось ранее в этом подразделе, в SQL:1999 специфицированы некоторые операторы, позволяющие изменять текущий идентификатор пользователя и текущее имя роли SQL-сессии.

**Оператор SET SESSION AUTHORIZATION.** Для изменения текущего идентификатора пользователя SQL-сессии может быть использован оператор

```
SET SESSION AUTHORIZATION value_specification
```

Спецификация значения (*value\_specification*) может быть либо литералом (в данном случае литералом типа символьных строк), либо вызовом функции без параметров, такой как **CURRENT\_USER**, **SESSION\_USER** и т.д. Если указанная спецификация значения не соответствует требованиям, предъявляемым в реализации к представлению идентификатора пользователя, операция изменения текущего идентификатора пользователя аварийно завершается.

В стандарте также говорится, что если спецификация значения, заданная в операции, формально соответствует требованиям, предъявляемым к формату идентификатора пользователя конкретной системы, но в действительности не представляет *известный системе* идентификатор пользователя, то опять же фиксируется ошибка и операция не выполняется. Допускается, чтобы в реализации принималось решение о смене идентификатора пользователя сессии одновременно с регистрацией нового идентификатора пользователя. Ограничения на регистрацию таким способом нового пользователя тоже находятся в ответственности реализации. После успешного выполнения оператора **SET SESSION AUTHORIZATION** текущее имя роли соответствующей сессии принимает значение **NULL**, так что текущим **authID** этой сессии становится заданное значение идентификатора пользователя.

Снова забегаая вперед, заметим, что операцию смены текущего идентификатора пользователя SQL-сессии не разрешается выполнять внутри какой-либо транзакции этой сессии. Иначе терялся бы смысл привилегий доступа, которыми руководствуется система при выполнении операций внутри транзакции.

**Оператор SET ROLE.** Для смены текущего имени роли SQL-сессии можно использовать оператор

```
SET ROLE { value_specification | NONE }
```

Ограничения на выполнения операции **SET ROLE** почти совпадают с определенными в стандарте ограничениями на выполнение операции **SET SESSION AUTHORIZATION**. Наиболее важные отличия состоят в том, что эту операцию от имени текущего **authID** сессии всегда разрешается выполнять для ролей, которые переданы «пользователю» **PUBLIC** или данному текущему **authID**, а также в том, что всегда разрешается использование конструкции **SET ROLE NONE**. Выполнение последней конструкции приводит к тому, что значение текущего имени роли сессии становится неопределенным.

Заметим, что при смене текущего имени роли SQL-сессии значение текущего пользовательского идентификатора сессии не меняется, так что вполне вероятно, что после выполнения операции и текущий идентификатор, и текущее имя роли будут иметь значения, отличные от неопределенного значения. И конечно, операция **SET ROLE NONE** будет выполнена успешно только в том случае, когда значение текущего пользовательского идентификатора не является неопределенным.

#### 14.1.6. Аннулирование привилегий и ролей

Если от имени некоторого **authID** некоторые привилегия или роли были переданы одному или нескольким другим **authID**, то впоследствии первый **authID** (в сессии, где этот **authID** является текущим) можно изъять или *аннулировать* переданные привилегии или роли путем применения оператора **REVOKE**. Как и в случае передачи привилегий и ролей, способы аннулирования привилегий и ролей похожи, но между ними имеются важные отличия. Поэтому вновь обсудим эти способы в отдельности.

**Аннулирование привилегий.** Для аннулирования привилегий используется оператор **REVOKE**, определяемый следующим синтаксическим правилом:

```
REVOKE [ GRANT OPTION FOR] privilege_commalist
      ON privilege_object
FROM { PUBLIC | authID_commalist }
     [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
     { RESTRICT | CASCADE }
```

Синтаксис конструкций **privilege** и **privilege\_object** такой же, как для оператора **GRANT**. Общий смысл операции должен быть понятен из синтаксиса: у указанных **authID** аннулируются указанные привилегии доступа к указанному объекту базы данных.

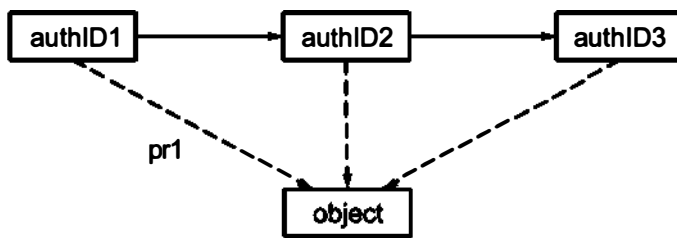


Рис. 14.2. Передача полученной привилегии

Первой важной особенностью оператора аннулирования привилегий является обязательность указания одного из ключевых слов **RESTRICT** или **CASCADE**. Если в операторе содержится **RESTRICT**, то при выполнении операции система проверит, не передавалась ли какая-либо из указанных привилегий каким-либо **authID** от того **authID**, у которого привилегия должны быть аннулирована (это вполне возможно, если ранее привилегия была передана с правом передачи). Если это действительно так, операция не выполняется; иначе указанные привилегии у указанных **authID** аннулируются. Иначе говоря, при наличии ключевого слова **RESTRICT** не допускается, например, ситуация, показанная на рис. 14.2.

На этом рисунке **authID1** является владельцем объекта базы данных с именем **object** и, следовательно, обладает всеми привилегиями над этим объектом. Штриховой стрелкой обозначена одна из этих привилегий **pr1**. От имени **authID1** привилегия **pr1** была передана **authID2** вместе с привилегией на ее дальнейшую передачу. Наконец, от имени **authID2** привилегия **pr1** была передана **authID3**. Тогда операция аннулирования этой привилегии от имени **authID1** у **authID2** при наличии ключевого слова **RESTRICT** не будет выполнена успешно.

В той же ситуации привилегия была бы аннулирована для **authID2** (и для **authID3**), если бы в операторе **GRANT** присутствовало ключевое слово **CASCADE**. В общем случае, если выполняется операция **REVOKE ... CASCADE**, то указанные привилегии аннулируются у всех **authID**, прямо или косвенно (через промежуточные **authID**) получивших привилегии от текущего **authID** SQL-сессии, в которой выполняется данная операция.

Если в операторе содержится раздел **GRANT OPTION FOR**, но имеется ключевое слово **RESTRICT**, то указанные привилегии для указанных **authID** не аннулируются, но у указанных **authID** аннулируется привилегия передачи данных привилегий (операция должна успешно выполняться только при соблюдении условий, обсуждавшихся ранее). Однако если в операторе одновременно

содержатся и **GRANT OPTION FOR**, и **CASCADE**, то указанные привилегии аннулируются у всех **authID**, которые прямо или косвенно (через промежуточные **authID**) получили привилегии от текущего **authID** SQL-сессии, в которой выполняется данная операция.

Задание в операторе необязательного раздела **GRANTED BY** позволяет явно указать, что должно использоваться в качестве текущего **authID** — текущий пользовательский идентификатор или текущее имя роли SQL-сессии. Если раздел **GRANTED BY** в операторе **REVOKE** не содержится, то действия производятся от имени текущего **authID** SQL-сессии.

Если текущий (или указанный) **authID** не обладает ни одной из указанных в операторе **REVOKE** привилегий, то выполнение операции не производится (фиксируется ошибка). Если **authID** обладает некоторыми, но не всеми, привилегиями из числа указанных привилегий, то операция выполняется по отношению к этим некоторым привилегиям, но выдается предупреждение.

Возможны ситуации, когда у некоторого **authID** остается некоторая привилегия после выполнения операции аннулирования у этого **authID** этой привилегии. Одна из таких ситуаций проиллюстрирована на рис. 14.3.

Здесь привилегия **pr1** передана от **authID1** к **authID2** вместе с правом на дальнейшую передачу этой привилегии. Далее привилегия **pr1** передается от **authID2** к **authID3**, затем выполняется прямая передача привилегии от **authID1** к **authID3** (в действительности, порядок этих действий не является существенным). Теперь предположим, что от имени **authID1** выполняется операция

`REVOKE pr1 ON object FROM authID2 CASCADE`

В соответствии с правилами SQL:1999 после выполнения этой операции **authID3** будет продолжать владеть привилегией **pr1** по отношению к объекту **object**, поскольку получил эту привилегию *двумя разными способами*. Грубо говоря, операция **REVOKE**, выполняемая от имени **authID1**, выполняется только

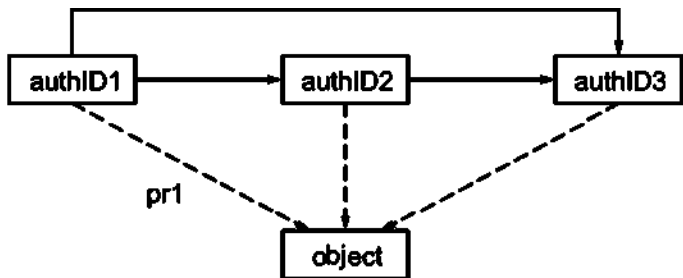


Рис.14.3. Косвенная и прямая передача привилегий



по тем путям графа идентификаторов авторизации и объектов базы данных, которые начинаются с узлов, соответствующих `authID`, указанных в разделе `FROM` этой операции.

Напомним, что если при передаче от `authID1` к `authID2` привилегии на выполнение некоторых действий над некоторой таблицей *T* (например, `UPDATE`) явно не указывается список имен столбцов этой таблицы, то привилегия распространяется на все столбцы этой таблицы (включая столбцы, которые, возможно, будут созданы в будущем). Если действительно использовался такой способ передачи привилегий, то в дальнейшем можно аннулировать привилегию `authID2` на модификацию отдельных (уже определенных) столбцов таблицы *T*, оставив привилегию на модификацию всех остальных столбцов (включая те, которые еще не созданы).

И последнее замечание. Если некоторая привилегия была передана `псевдоauthID PUBLIC`, то, конечно, этой привилегией обладают все `authID`. Но нет возможности аннулировать такую привилегию у отдельно указываемого `authID`. Привилегия была передана всем и аннулировать ее можно только сразу у всех.

**Аннулирование ролей.** Вариант оператора `REVOKE`, используемый для аннулирования ролей, выглядит следующим образом:

```
REVOKE [ ADMIN OPTION FOR ] role_name_commalist
FROM { PUBLIC | authID_commalist }
    [ GRANTED BY { CURRENT_USER | CURRENT_ROLE } ]
    { RESTRICT | CASCADE }
```

Действие операции аннулирования ролей очень похоже на действие операции аннулирования привилегий. Отличия состоят в том, что аннулируются не привилегии, а роли, а также в том, что для аннулирования привилегии на передачу роли используется раздел `ADMIN OPTION FOR`.

## 14.2. Управление транзакциями в SQL

В SQL используется классическое понимание транзакции, характеризующееся аббревиатурой `ACID` (см. подразд. 9.1).

### 14.2.1. Порождение транзакций в SQL

В соответствии со стандартом языка SQL:1999 транзакции могут образовываться *явным* образом с использованием опе-

ратора **START TRANSACTION** либо *неявно*, когда выполняется оператор, для которого требуется контекст транзакции, а этого контекста не существует. Например, операторы **SELECT**, **UPDATE** или **CREATE TABLE** могут выполняться только в контексте транзакции, а для выполнения оператора **CONNECT** (см. подразд. 14.3) такой контекст не требуется, и выполнение оператора **CONNECT** не приводит к неявному образованию транзакции. Для завершения транзакции должен быть явно использован один из двух операторов — **COMMIT** (требование завершить транзакцию с фиксацией ее результатов) или **ROLLBACK** (требование завершить транзакцию с удалением результатов всех выполненных операций из состояния базы данных).

**Установка характеристик транзакции.** У каждой выполняемой транзакции имеются три характеристики, значения которых существенно влияют на действия системы при управлении транзакцией: *уровень изоляции (isolation level)*, *режим доступа (access mode)* и *размер области диагностики*. При неявном образовании транзакции эти характеристики устанавливаются по умолчанию: транзакция получает максимальный уровень изоляции от одновременно выполняемых транзакций; режим доступа, позволяющий выполнять и операции выборки, и операции обновления базы данных; назначаемый по умолчанию размер области диагностики.

Если значения характеристик транзакции, устанавливаемые по умолчанию, в некотором случае не являются пригодными, то до выполнения оператора, неявно иницилирующего транзакцию, можно явно установить характеристики этой транзакции с использованием оператора **SET TRANSACTION**. Этот оператор определяется следующими синтаксическими правилами:

```
SET TRANSACTION mode_commlist
mode ::= isolation_level
      | access_mode
      | diagnostics_size
isolation_level ::= READ UNCOMMITTED
                | READ COMMITTED
                | REPEATABLE READ
                | SERIALIZABLE
access_mode ::= READ ONLY
             | READ WRITE
diagnostics_size ::= DIAGNOSTIC SIZE
value_specification
```

Операцию установки характеристик транзакции нельзя выполнять в контексте какой-либо активной транзакции. Выполнение операции допустимо только до образования первой транзакции SQL-сессии или между последовательно выполняемыми транзакциями этой сессии. В одном операторе **SET TRANSACTION** можно задать только по одному значению каждой из трех характеристик, но допускается последовательное выполнение нескольких таких операций с разными операндами.

Как видно из синтаксических правил, у характеристики *режим доступа* может быть указано одно из двух значений — **READ ONLY** или **READ WRITE**. Если устанавливается режим **READ ONLY**, то в транзакции нельзя будет выполнять никакие операции, изменяющие базу данных, в том числе операции обновления таблиц и определения новых объектов базы данных. Если режим доступа явно не указывается, по умолчанию принимается характеристика **READ WRITE**, если только в качестве значения характеристики *уровень изоляции* не указывается **READ UNCOMMITTED** (в этом случае устанавливается режим доступа **READ ONLY**).

Если указывается размер области диагностики, то после ключевых слов **DIAGNOSTIC SIZE** должен следовать целочисленный литерал, определяющий число диагностических элементов, которые должны разместиться в области диагностики (число исключительных ситуаций, предупреждений, сообщений об отсутствии данных и об успешном выполнении, которые будут вырабатываться при выполнении операторов внутри будущей транзакции). Если размер области диагностики явно не указывается, то решение о размере этой области принимается в реализации.

Уровни изоляции более подробно обсуждаются далее, но здесь заметим, что если значение уровня изоляции явно не задано, то по умолчанию принимается уровень изоляции **SERIALIZABLE**. Кроме того, обратим внимание читателей, что одновременное задание уровня изоляции **READ UNCOMMITTED** и режима доступа **READ WRITE** не допускается.

Еще одной интересной деталью оператора установки характеристик транзакции является то, что выполнение каждого следующего оператора **SET TRANSACTION** полностью перекрывает эффект выполнения предыдущего такого оператора. В частности, если в предыдущем операторе явно задавалось значение некоторой характеристики, а в следующем это значение принимается по умолчанию, то именно значение по умолчанию будет являться значением характеристики транзакции.

**Явная инициация транзакции.** Для явного образования транзакции поддерживается оператор **START TRANSACTION**, определяемый следующими синтаксическими правилами:

```
START TRANSACTION mode_commalist
```

Этот оператор очень похож на **SET TRANSACTION**. Единственное (хотя и очень существенное) отличие состоит в том, что выполнение оператора **START TRANSACTION** приводит не только к установке характеристик транзакции, но и к реальной инициации транзакции.

**14.2.2. Уровни изоляции SQL-транзакции**

В стандарте SQL:1999 уровни изоляции определяются на основе нескольких *феноменов*, которые могут возникать при выполнении транзакций (см. подразд. 9.1.3).

Феномен грязного чтения может наблюдаться у транзакций, выполняемых на уровне изоляции **READ UNCOMMITTED**. Рекомендуется использовать этот уровень изоляции только в тех транзакциях, для выполнения функций которых точные данные не обязательны (например, в транзакциях, производящих статистическую обработку). Феномен неповторяемого чтения может наблюдаться у транзакций, выполняемых на уровне изоляции **READ COMMITTED** (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена грязного чтения). Феномен фантомов может наблюдаться у транзакций, выполняемых на уровне изоляции **REPEATABLE READ** (этот уровень изоляции, как показывает его название, гарантирует отсутствие феномена неповторяемого чтения). Наконец, для транзакций, выполняемых на уровне изоляции **SERIALIZABLE**, невозможно и проявление феномена фантомов. Этот уровень гарантирует предельную изолированность транзакций. Общая картина взаимосвязи уровней изоляции и феноменов транзакций показана на табл. 14.2.

Таблица 14.2. Уровни изоляции и феномены

| Уровень          | Грязное чтение | Неповторяющееся чтение | Фантомы    |
|------------------|----------------|------------------------|------------|
| READ UNCOMMITTED | Возможно       | Возможно               | Возможны   |
| READ COMMITTED   | Невозможно     | Возможно               | Возможны   |
| REPEATABLE READ  | Невозможно     | Невозможно             | Возможны   |
| SERIALIZABLE     | Невозможно     | Невозможно             | Невозможны |

### 14.2.3. Завершение транзакций

Как отмечалось в начале этого подраздела, транзакции могут инициироваться как явным способом (с помощью оператора **START TRANSACTION**), так и неявно, при выполнении первого оператора, требующего наличия контекста транзакции. Для завершения транзакции *всегда*\* требуется выполнение одного из двух операторов **COMMIT** (фиксация транзакции) или **ROLLBACK** (откат транзакции), которые имеют следующий синтаксис:

```
COMMIT [ WORK ] [ AND [ NO ] CHAIN ]  
ROLLBACK [ WORK ] [ AND [ NO ] CHAIN ]  
[ TO SAVEPOINT savepoint_name ]
```

При желании завершить транзакцию таким образом, чтобы все произведенные ей изменения были навсегда сохранены в базе данных, следует завершать транзакцию оператором **COMMIT** (как видно из синтаксиса, допускается эквивалентный вид **COMMIT WORK**). Если требуется завершить транзакцию с аннулированием всех произведенных изменений, то нужно использовать оператор **ROLLBACK** (**ROLLBACK WORK**).

Заметим, что и операция фиксации транзакции, и операция отката являются достаточно сложными и выполняются не мгновенно. Поэтому в ходе выполнения этих операций, вообще говоря, может произойти аварийный отказ системы. Естественно (см. гл. 10), база данных будет восстановлена в свое последнее согласованное состояние, но ситуации вокруг прерванного выполнения операции фиксации или операции отката коренным образом различаются. Оператор **COMMIT** считается безусловно выполненным только в том случае, когда сервер баз данных подтвердил это после выполнения всех действий, требуемых для фиксации транзакции. Аварийная ситуация во время выполнения операции **ROLLBACK** ничем не отличается от аварийной ситуации, возникшей в процессе выполнения транзакции. В этом случае (при восстановлении базы данных) прерванная транзакция считается незафиксированной (что так и есть) и все ее изменения автоматически удаляются из состояния базы данных. Поэтому окончательный результат выполнения операции фиксации транзакции, прерванного аварийным отказом системы, эквивалентен успешному выполнению операции отката транзакции.

---

\* Правильнее было бы сказать *почти всегда*, поскольку в SQL предусматривается особый способ терминации транзакций, инициированных программными агентами. Но в данной книге мы этого не касаемся.

Синтаксис обоих операторов показывает, что в каждом из них может содержаться раздел **AND [ NO ] CHAIN**. Постараемся кратко пояснить смысл этого раздела (не вдаваясь в детали, поскольку стандарт SQL:1999 оставляет окончательное решение за реализацией).

Операции образования и завершения транзакции являются достаточно дорогостоящими. В особенности это касается операции завершения транзакции, при выполнении которой необходимо выполнить обмены с внешней памятью. С другой стороны, использование долговременных транзакций чревато снижением уровня параллелизма в системе, а также повышает риск утраты результатов транзакции в результате системного отказа.

Поэтому часто используется компромиссный вариант, при котором действия операторов **COMMIT** или **ROLLBACK** приводят не только к завершению текущей транзакции, но и к образованию новой транзакции. Именно эту возможность поддерживает раздел **AND [ NO ] CHAIN** операторов **COMMIT** и **ROLLBACK**. Если этот раздел отсутствует в операторе завершения транзакции, то подразумевается наличие раздела **AND NO CHAIN** и новая транзакция не образуется. Если же раздел **AND CHAIN** присутствует, то немедленно после завершения выполнения **COMMIT** или **ROLLBACK** текущей транзакции образуется новая транзакция, наследующая все характеристики завершенной транзакции.

Семантика раздела **TO SAVEPOINT** будет пояснена далее.

#### 14.2.4. Транзакции и ограничения целостности

Частично материал этого подраздела уже излагался в подразд. 11.5.3, но там это делалось в контексте определений ограничений целостности. Для полноты картины воспроизведем часть этого материала в контексте управления транзакциями.

Итак, любое ограничение целостности обладает атрибутом, определяющим время проверки данного ограничения. Этот атрибут может иметь значения **DEFERRABLE** (отложенная проверка) или **NOT DEFERRABLE** (немедленная проверка). Чтобы данное ограничение целостности могло когда-либо обладать свойством отложенной проверки, нужно, чтобы в определении этого ограничения присутствовали ключевые слова **INITIALLY DEFERRED** или **INITIALLY IMMEDIATE**. В любом случае в каждый момент времени выполнения транзакции любое ограничение целостности находится в одном из двух состояний — *отложенная проверка* или *немедленная проверка*. Если начальным состоянием ограничения является **INITIALLY DEFERRED**, то в начале

любой транзакции его текущим состоянием будет *отложенная проверка*. Аналогично для ограничений с начальным состоянием **INITIALLY IMMEDIATE**.

Любое ограничение, находящееся в состоянии *немедленной проверки*, всегда проверяется в конце выполнения любого оператора **SQL**\*. Немедленно проверяются и те ограничения, которые были определены как **NOT DEFERRABLE**, но для которых впоследствии был установлен режим *немедленной проверки*. Однако если текущим состоянием ограничения является *отложенная проверка*, оно будет проверяться только тогда, когда перейдет в состояние *немедленной проверки*. Это делается неявно при выполнении оператора **COMMIT** или явно при выполнении оператора **SET CONSTRAINTS**. Этот оператор имеет следующий синтаксис:

```
SET CONSTRAINTS { ALL | constraint_name_commalist }  
                { DEFERRED | IMMEDIATE }
```

Ключевое слово **ALL** является сокращенной формой задания списка имен всех ограничений целостности, определенных в базе данных, которые специфицированы с указанием ключевого слова **DEFERRABLE**. Если список имен ограничений задается явно, то все входящие в него имена должны соответствовать ограничениям, определенным с указанием ключевого слова **DEFERRABLE**.

При попытке фиксации транзакции, для которой имеются одно или нескольких ограничений целостности, текущим режимом которых является *отложенная проверка*, система (ненадолго, поскольку транзакция скоро тем или иным способом завершится) устанавливает для всех этих ограничений режим *немедленной проверки* и проверяет эти ограничения. Если какое-либо из ограничений нарушается, то операция **COMMIT** трактуется как операция **ROLLBACK** и пользователю (или приложению) сообщается, что у него возникла ошибка. Избежать этой неприятной ситуации можно явным выполнением оператора **SET CONSTRAINTS ALL IMMEDIATE** до фиксации транзакции, для которой имеются **DEFERRABLE** ограничения, текущим режимом которых является *отложенная проверка*.

#### 14.2.5. Точки сохранения

Как уже отмечалось, использование долговременных транзакций повышает риск полного аннулирования результатов

---

\* Естественно, на практике проверяются только те ограничения, которые могут быть потенциально нарушены в результате выполнения соответствующего оператора.

транзакции по причине нарушения ограничений с отложенной проверкой при выполнении каких-либо экспериментальных (недостаточно проверенных) операций. Конечно, теоретически можно было бы оформлять выполнение каждой такой подозрительной операции в виде отдельной транзакции, но это часто противоречит общей логике приложения, когда последовательность действий должна быть атомарной.

Частичное решение этой проблемы предоставляет механизм *точек сохранения (savepoint)* SQL:1999. Точка сохранения представляет собой своего рода пометку в последовательности операций транзакции, которую в дальнейшем можно использовать для частичного отката транзакции с сохранением жизнеспособности транзакции и результатов операций, выполненных в транзакции до точки сохранения. Пример использования точки сохранения приведен на рис. 14.4.

На этом рисунке показано, что после выполнения последовательности проверенных «безопасных» операций, которые, по мнению пользователя, не могут нарушить ограничения целостности с отложенной проверкой, устанавливается точка сохранения. За этой точкой следует серия «рискованных» операций. Если по каким-то причинам (например, путем немедленной проверки отложенных ограничений) затем принимается решение о нецелесообразности фиксации результатов этих операций, то выполняется частичный откат транзакции к точке сохранения, а затем фиксируются результаты безопасных операций.

Допускается установка в одной транзакции нескольких последовательных точек сохранения. При установке каждой точки сохранения ей назначается некоторое (локальное в пределах

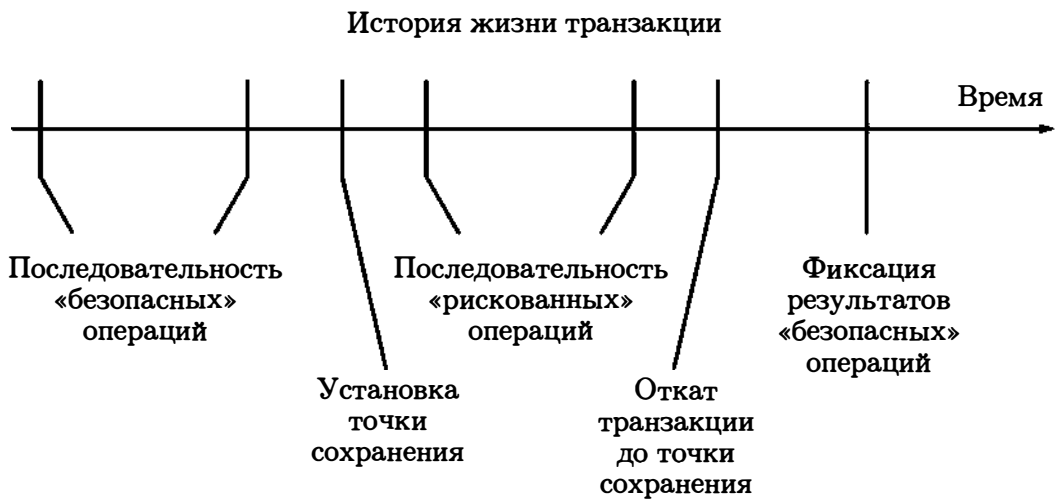


Рис. 14.4. Пример транзакции с точкой сохранения



транзакции) имя, которое в дальнейшем может использоваться в операции **ROLLBACK** для задания точки частичного отката транзакции. Если последовательно устанавливаются две точки сохранения **SP1** и **SP2** и затем выполняется операция **ROLLBACK TO SAVEPOINT SP1**, то восстановление производится до **SP1** (через **SP2**) и точка сохранения **SP2** забывается.

Обратите внимание, что оператор **ROLLBACK TO SAVEPOINT savepoint\_name**, хотя и синтаксически схож с «обычным» оператором **ROLLBACK**, принципиально отличается по своей семантике. Откат транзакции до указанной точки сохранения *не означает* завершения транзакции. Видимо, из соображений здравого смысла, следовало бы ввести в язык SQL операцию **ABORT TRANSACTION** для аварийного завершения транзакции с ликвидацией всех последствий ее операций в базы данных и, отдельно, операцию **ROLLBACK TO**, в которой всегда явно указывалось бы, до какого уровня требуется откат. Кстати, заметим, что комбинация

```
ROLLBACK AND CHAIN TO SAVEPOINT savepoint_name
```

является недопустимой (поскольку текущая транзакция не завершается).

Для установления точки сохранения используется оператор **SAVEPOINT** с очевидным синтаксисом

```
SAVEPOINT savepoint_name
```

Можно также отказаться от ранее установленной точки сохранения, удалив ее из контекста транзакции. Для этого имеется оператор **RELEASE**, синтаксис которого также очевиден:

```
RELEASE SAVEPOINT savepoint_name
```

После выполнения этой операции в данной транзакции невозможно выполнять какие-либо другие операции над точкой сохранения с данным именем, пока не будет образована другая одноименная точка сохранения.

### 14.3. Подключения и сессии

Может показаться странным, что в конце этой главы рассматривается материал, который, казалось бы, необходимо знать, чтобы иметь возможность начать работать с какой-либо из современных систем баз данных. Объяснение очень простое. Чем ниже уровень средств языка SQL, чем ближе эти средства

соприкасаются с индивидуальными особенностями реализаций, тем менее точен и конкретен стандарт SQL. А в этом подразделе речь идет о средствах, реализация которых в СУБД разных поставщиков обладает очень большой спецификой.

Сильно упрощая текущую ситуацию, можно сказать, что практически все современные продукты управления SQL-ориентированными базами данных основаны на архитектуре «клиент-сервер». Принципиальная схема клиент-серверной организации показана на рис. 14.5.

Конечно, это рисунок весьма условен. Под термином *пользователь* здесь, конечно, понимается некоторое приложение, с которым реально работает конечный пользователь (например, в этом приложении может быть реализован монитор прямого SQL). *Клиентская часть* СУБД — это тот системный компонент, с которым непосредственно взаимодействует *пользователь*. Этот компонент скрывает специфику реальных взаимодействий с серверной частью СУБД (например, используемые сетевые протоколы, если клиентская и серверная части СУБД разнесены по разным компьютерам сети). Наконец, *сервер баз данных* представляет собой основную часть СУБД, где, собственно, и происходит выполнение операторов SQL и осуществляется доступ к базе данных.

Важно обратить внимание, что программные компоненты, представляющие *пользователя* и *клиентскую часть* СУБД, обычно выполняются на одном компьютере, а *сервер баз данных* работает на другом (серверном) компьютере. Но вполне может

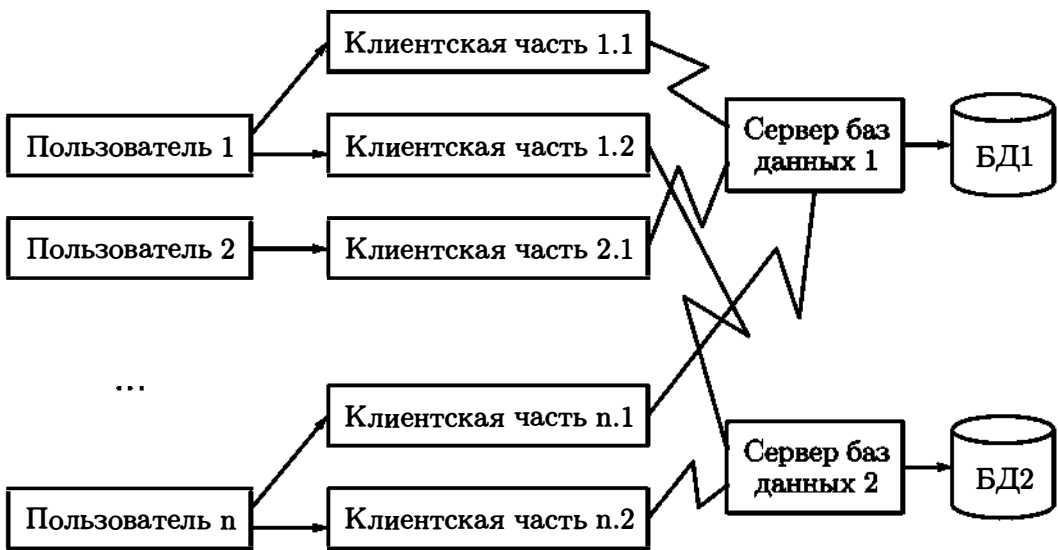


Рис. 14.5. Клиент-серверная архитектура СУБД

быть, что все три перечисленных программных компонента в действительности размещены на одном компьютере.

### 14.3.1. Установление соединений

Начиная со стандарта SQL/92 при разработке языковых средств стала приниматься во внимание клиент-серверная организация СУБД. Более точно, стал очевиден тот факт, что во всех существующих СУБД до начала работы приложения со средствами управления базой данных требуется выполнить некоторые предварительные иницилирующие действия. В частности, необходимо создать контекст, в котором будет работать система баз данных. В некоторых реализациях этот контекст создается автоматически при запуске приложения, поскольку клиентская часть СУБД компонуется к приложению. В других случаях прикладная программа связывается с СУБД за счет наличия специализированных реализационно-зависимых средств подключения к СУБД. Иногда контекст формируется на основе состояния системных переменных.

Очевидно, что для выработки языковых средств, которые не противоречили бы существующим реализациям, требовался значительный компромисс. Этот компромисс выразился в том, что в SQL:1999 допускается установление связи приложения с СУБД по умолчанию, а также обеспечиваются средства явного управления соединениями. Общий подход состоит в следующем.

- Почти все операторы SQL (с небольшим числом исключений) могут выполняться только при наличии подключения клиентской части СУБД к серверу базы данных.
- Если соединение с сервером установлено и приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), то его выполняет та СУБД, с которой установлено соединение.
- Если приложение пытается выполнить один из операторов SQL (для выполнения которых требуется соединение), а соединение не установлено, то прежде всего требуется установить соединение. В SQL:1999 указывается, что такое соединение является соединением с СУБД по умолчанию. Что собой представляет это умолчание, определяется в реализации. После установления соединения упомянутый оператор SQL выполняется той СУБД, с которой установлено соединение.
- Если первым (до установки соединения) выполняемым оператором SQL является оператор **CONNECT** (это одно

из исключений), то соединение по умолчанию не устанавливается, а происходит обращение к запрашиваемому серверу, и соединение устанавливается именно с ним.

- Можно выполнять оператор **CONNECT** для установления соединений со вторым, третьим и далее серверами, не разрывая ранее установленные соединения. Каждой вновь установленное соединение называется *текущим соединением* (*current connection*), а все ранее установленные соединения — *отложенными соединениями* (*dormant connection*).
- С каждым соединением ассоциирована сессия. Сессия, ассоциированная с текущим соединением, называется *текущей сессией* (*current session*), а сессии, ассоциированные с отложенными соединениями, называются *отложенными сессиями* (*dormant session*)\*.
- Если у приложения имеется несколько соединений, можно переключать их с помощью оператора **SET CONNECTION**.
- Для поддержания установленных соединений могут расходоваться значительные системные ресурсы. Поэтому может возникнуть потребность в ликвидации соединения. Это можно сделать с помощью оператора **DISCONNECT**. Все соединения, не ликвидированные явно до завершения работы приложения, ликвидируются системой автоматически. Попытка ликвидировать текущее соединение, в котором выполняется транзакция, расценивается как ошибка.
- В реализации определяется, можно ли переключать соединения во время выполнения транзакции. Однако, если реализация это допускает, то в соответствии со стандартом все операторы, выполняемые в одной транзакции, но в разных соединениях, являются частью одной общей транзакции.

### 14.3.2. Операторы SQL для управления соединениями

Как отмечалось ранее, в эту группу входят операторы **CONNECT**, **SET CONNECTION** и **DISCONNECT**.

---

\* Каждому соединению соответствует одна и только одна сессия. В сообществе SQL эти термины часто используются попеременно для обозначения одного и того же. Более строгие блюстители терминологии утверждают, что термин *подключение* относится к сетевому пути между клиентом и сервером, а *сессия* — это контекст, в котором работает SQL-сервер.

**Оператор CONNECT.** Оператор определяется следующими синтаксическими правилами:

```
CONNECT TO connection_target
connection_target ::= SQL_server_name
                    [ AS connection_name ]
                    [ USER connection_user_name ]
                    | DEFAULT
```

Здесь **SQL\_server\_name** — это литерально заданная символьная строка, идентифицирующая сервер, к которому требуется подключиться. Смысл (и формат) этого имени определяется в реализации.

В необязательном разделе **AS** указывается имя (**connection\_name**) выступает в роли временного имени соединения, которое впоследствии может быть использовано в операторах **SET CONNECTION** и **DISCONNECT**. Если в операторе **CONNECT** раздел **AS** не содержится, то по умолчанию **connection\_name** совпадает с **SQL\_server\_name**.

В необязательном разделе **USER** указывается имя (**connection\_user\_name**) идентифицирует пользователя, от имени которого устанавливается соединение. При отсутствии раздела **USER** в качестве **connection\_user\_name** по умолчанию принимается текущий **authID**. В стандарте допускается, что реализация может ограничить возможные значения **connection\_user\_name** (например, потребовать, чтобы это имя всегда совпадало с текущим **authID**).

Эффект использования оператора в форме **CONNECT TO DEFAULT** почти не отличается от результата действия системы при отсутствии какого-либо явного требования соединения. (Напомним, что соединение по умолчанию неявно устанавливается при попытке выполнения первого оператора SQL, требующего наличия соединения.) Однако имеется одно важное отличие. Если соединение по умолчанию устанавливается неявно, а затем вдруг прерывается по причине какой-то ошибки, то оно автоматически переустанавливается при выполнении следующего оператора SQL. Если же соединение по умолчанию устанавливается явным образом, то автоматическое повторное установление соединения после его разрыва не производится.

**Оператор SET CONNECTION.** Оператор определяется следующими синтаксическими правилами:

```
SET CONNECTION connection_object
connection_object ::= { connection_name | DEFAULT }
```

Условием успешного выполнения операции является наличие отложенного установленного соединения с именем `connection_name` или отложенного установленного соединения по умолчанию. В этом случае текущее соединение становится отложенным, а указанное отложенное соединение — текущим.

**Оператор DISCONNECT.** Оператор имеет следующий синтаксис:

```
DISCONNECT { connection_object | ALL | CURRENT }
```

Необходимым условием для возможности ликвидации соединения является отсутствие активной транзакции в этом соединении.

Если в операторе указывается `connection_object`, то соответствующее имя должно соответствовать установленному (текущему или отложенному) соединению. Если указывается `CURRENT`, то должно иметься текущее соединение.

Если оператор применяется к текущему соединению, то это соединение ликвидируется и ни одно соединение не является текущим. В этом случае для продолжения работы необходимо установить текущее соединение при помощи операторов `CONNECT` или `SET CONNECTION`.

Если в операторе указывается `ALL`, то ликвидируются все соединения, включая текущее.

\* \* \*

Несмотря на некоторую критику в адрес языка SQL, высказанную в начале гл. 11, мы потратили на обсуждение этого языка почти половину курса и больше его практически не критиковали. Не означает ли это, что язык все-таки очень хорош или что автор курса питает к нему особую привязанность? Конечно же, нет! В языке SQL имеется множество слабых мест, неточностей и даже прямых ошибок. Если бы мы задались целью продемонстрировать все промахи языка SQL, то эта книга никогда бы не закончилась, а его читатели так и не узнали бы, что представляет собой язык в целом.

При всех недостатках у SQL имеются два неоспоримых преимущества. Во-первых, за 30 лет существования языка к нему привыкли тысячи профессионалов в области баз данных. Как говорится, лучше плохое, да свое. Во-вторых, и это проверено многолетней практикой, язык SQL допускает эффективную и масштабируемую реализацию, и даже объектные расширения

языка не вызывают какую-либо деградацию производительности систем. Одним словом, нам предстоит еще долгая совместная работа с SQL, и чтобы она была удачной, нужно хорошо знать и достоинства, и недостатки этого языка.

1. *Codd E.F.* A Relational Model of Data for Large Shared Data Banks // *CACM* 13, 1970 (June). — No. 6. Republished in “Milestones of Research”, *CACM* 26, 1982 (January). — No. 1. В пер. : *Кодд Е. А.* Реляционная модель для больших совместно используемых банков данных // СУБД, 1995. — No 1 [<http://citforum.ru/database/classics/codd/>]
2. *Дейт К.* Введение в системы баз данных. — М.-СПб. : Вильямс, 2005.
3. *Цикритзис Д.* Модели данных / Д. Цикритзис, Ф. Лоховски. — М. : Финансы и статистика, 1985.
4. *Darwen H.* The Third Manifesto / H. Darwen, C. J. Date // *ACM SIGMOD Record* 24, 1995 (March). — No. 1. В пер.: *Дарвен Х.* Третий манифест / Х. Дарвен, К. Дейт // СУБД. 1996. — No 1 [[http://citforum.ru/database/classics/third\\_manifesto/](http://citforum.ru/database/classics/third_manifesto/)].
5. *Дейт К.* Основы будущих систем баз данных. Третий манифест / К. Дейт, Х. Дарвен. — М. : Янус-К, 2004.
6. *Atkinson M.* The Object-Oriented Database System Manifesto / [M. Atkinson, F. Bancilhon, D. DeWitt et al.] — Proc. 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989), N. Y.: Elsevier Science (1990). В пер. : *Аткинсон М.* Манифест систем объектно-ориентированных баз данных / [М. Аткинсон и др.] // СУБД, 1995. — No 4 [[http://citforum.ru/database/classics/oo\\_manifesto/](http://citforum.ru/database/classics/oo_manifesto/)].
7. The Object Data Standard: ODMG 3.0 / Edited by R. G. G. Cattell, Douglas K. Barry. Morgan Kauffmann Publishers, 2000.
8. *Stonebraker M.* Third-Generation Data Base System Manifesto / [M. Stonebraker, L. Rowe, B. Lindsay et al.]. Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases, 1990 (July), *ACM SIGMOD Record* 19, 1990 (September). — No. 3. В пер.: *Стоунбрейкер М.* Системы баз данных третьего поколения: Манифест / [М. Стоунбрейкер и др.] // СУБД, 1996. — No 2 [<http://citforum.ru/database/classics/manifest/>].
9. *Date C. J.* Foundation for Object/Relational Databases: The Third Manifesto / C. J. Date, H. Darwen // Addison-Wesley Pub Co, 1998 (June).
10. *Date C. J.* Foundation for Future Database Systems: The Third Manifesto / C. J. Date, H. Darwen // Addison-Wesley Pub Co; 2nd edition, 2000.



11. *Date C. J.* Databases, types and the relational model : The Third Manifesto / C. J. Date, H. Darwen // Addison Wesley; 3th edition, 2006.

12. *Date C. J.* The Birth of the Relational Model (Part 3 of 3). — Intelligent Enterprise, 1998 (December). — Vol. 1. — No 3 [[http://www.citforum.ru/database/digest/dig\\_0212.shtml](http://www.citforum.ru/database/digest/dig_0212.shtml)].

13. *Codd E. F.* Extending the database relational model to capture more meaning. ACM transactions on database systems, 1979 (December). — Vol. 4. — No. 4 [[http://citforum.ru/database/classics/codd\\_2/](http://citforum.ru/database/classics/codd_2/)].

14. *Zloof M. M.* Query-by-example : A data base language. IBM Systems Journal 16 (4): 324—343 (1977). В пер.: *Злуф М. М.* Query-by-Example: язык баз данных // СУБД, 1996. — No 3 [[http://citforum.ru/database/classics/query\\_by\\_example/](http://citforum.ru/database/classics/query_by_example/)].

15. *Ульман Дж.* Основы систем баз данных. — М.: Финансы и статистика, 1983.

16. *Гарсия-Молина Г.* Системы баз данных. Полный курс / Г. Гарсия-Молина, Дж. Ульман, Дж. Уидом. — М.-СПб.: Вильямс, 2004.

17. *Мейер Д.* Теория реляционных баз данных. — М.: Мир, 1987.

15. *Pin-Shan Chen P.* The entity-relationship model—toward a unified view of data. ACM Transactions on Database Systems 1 (1), 9—36 (1976). В пер.: *П. Пин-Шен Чен.* Модель «сущность-связь» — шаг к единому представлению о данных // СУБД, 1995. — No 3 [<http://citforum.ru/database/classics/chen/>].

16. *Chamberlin D. D.* SEQUEL : A Structured English Query Language / D. D. Chamberlin, R. F. Boyce // ACM SIGMOD Workshop Data Descr., Acc. Contr., Proc., Ann Arbor, Mich., 1974 (May). New-York, 1974. — P. 249—264 [<http://www.almaden.ibm.com/cs/people/chamberlin/sequel-1974.pdf>].

17. *Chamberlin D. D.* View, authorization, and locking in a relational database system / D. D. Chamberlin, J. N. Gray, I. L. Traiger // AFIPS Conf. Proc. : Nat. Comput. Conf., Chicago, Ill., May 4—7, 1975. Reston, Virg., 1975. — P. 425—430 [<http://research.microsoft.com/~Gray/papers/Views%20Authorization%20Locking%20in%20a%20Relational%20Data%20Base%20System%20RJ%201486.pdf>].

18. *Gray J. N.* Granularity of Locks in a Large Shared Data Base / [J. N. Gray, R. A. Lorie, G. R. Putzolu et al.] // 1st Int. Conf. Very Large Data Bases, Framingham, Mass., Sept. 1975. New-York, 1976. — P. 428—451 [<http://research.microsoft.com/~Gray/papers/Granularity%20of%20Locks%20In%20a%20Large%20Shared%20Database%20RJ%201606.pdf>].

19. *Chamberlin D. D.* SEQUEL 2 : A Unified Approach to Data Definition, Manipulation, and Control / [D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran et al.] // IBM J. Res. and Dev., 1976. — 20. — № 6. — P. 560—575 [<http://www.research.ibm.com/journal/rd/206/ibmrd2006E.pdf>].

20. *Astrahan M. M.* System R: A Relational Approach to Data Base Management / [M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin et al.] //

ACM Trans. Database Syst., 1976. — 1. — No 2. — P. 97—137 [<http://www.seas.upenn.edu/~zives/cis650/papers/System-R.PDF>].

21. *Gray J. N.* Granularity of Locks and Degrees of Consistency in a Shared Database / [J. N. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger] // Proc. IFIP Work. Conf. Model. Data Base Manag. Syst., Freudenstadt, Germany, Jan. 1976. New-York, 1976. — P. 695—723 [<http://research.microsoft.com/~Gray/papers/Granularity%20of%20Locks%20and%20Degrees%20of%20Consistency%20RJ%201654.pdf>].

22. *Eswaran K. P.* The Notions of Consistency and Predicate Locks in a Database System / K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger // Commun. ACM, 1976. — 19. — No 11. — P. 624—633 [<http://research.microsoft.com/~Gray/papers/On%20the%20Notions%20of%20Consistency%20and%20Predicate%20Locks%20in%20a%20Database%20System%20CACM.pdf>].

23. *Griffiths P. P.* An Authorization Mechanism for a Relational Database System / P. P. Griffiths, B. W. Wade // ACM Trans. Database Syst., 1976. — 1. — No 3. — P. 242—255 [<http://homes.cerias.purdue.edu/~bhargav/cs526/p242-griffiths.pdf>].

24. *Blasgen M. W.* Storage and Access in Relational Data Bases / M. W. Blasgen, K. P. Eswaran // IBM Syst. J., 1977. — Vol. 16. — No 4. — P. 363—377 [<http://www.research.ibm.com/journal/sj/164/ibmsj1604E.pdf>].

25. *Blasgen M. W.* The Convoy Phenomenon / [M. W. Blasgen, J. N. Gray, M. Mitoma, T. G. Price] // ACM Oper. Syst. Rev., 1979. — 13. — No 2. — P. 20—25 [<http://research.microsoft.com/~Gray/papers/Convoy%20Phenomenon%20RJ%202516.pdf>].

26. *Lorie R. A.* An Access Specification Language for a Relational Data Base System / R. A. Lorie, J. F. Nilsson // IBM J. Res. and Dev., 1979. — Vol. 23. — No 3. — P. 1—13 [<http://www.research.ibm.com/journal/rd/233/ibmrd2303F.pdf>].

27. *Gray J. N.* Notes on Database Operating Systems // Lect. Notes Comput. Sci., 1979. — Vol. 60. — P. 396—481 [<http://research.microsoft.com/~Gray/papers/DBOS.pdf>].

28. *Selinger P. G.* Access Path Selection in a Relational Database Management System / [P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie et al.] // Proc. ACM SIGMOD Int. Conf. Manag. Data, Boston, Mass., May 30 — June 1, 1979; N. Y., 1979. — P. 23—34 [<http://www.cs.berkeley.edu/~brewer/cs262/3-selinger79.pdf>].

29. *Gray J. N.* The Recovery Manager of the System R Database Manager / [J. N. Gray, P. McJones, M. W. Blasgen, B. G. Lindsay et al.] // ACM Comput. Surv., 1981. — Vol. 13. — No 2. — P. 223—242 [<http://research.microsoft.com/~Gray/papers/Recovery%20Manager%20of%20a%20Database%20System%20RJ%202623.pdf>].

30. *Chamberlin D. D.* A History and Evaluation of System R / [D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen et al.] // Commun.

ACM, 1981. — 24. — No 10. — P. 632—646 [<http://www.cs.berkeley.edu/~brewer/cs262/SystemR.pdf>].

31. *Chamberlin D. D.* Support for Repetitive Transactions and Ad Hoc Queries in System R / [D. D. Chamberlin, M. M. Astrahan, W. F. King et al.] // ACM Trans. Database Syst., 1981. — Vol. 6. No 1. — P. 70—94 [<http://research.microsoft.com/~Gray/papers/Support%20for%20Repetitive%20Transactions%20and%20Ad%20Hoc%20Query%20in%20System%20R%20RJ%202551.pdf>].

32. *Blasgen M. W.* System R : An Architectural Overview / [M. W. Blasgen, M. M. Astrahan, D. D. Chamberlin et al.] // IBM Syst. J., 1981. — Vol. 20. — No 1. — P. 41—62 [<http://www.research.ibm.com/journal/sj/382/blasgen.pdf>].

33. *Кузнецов С. Д.* Развитие идей и приложений реляционной СУБД System R. — М. : ВИНТИ, 1989, Тем. изд. «Итоги науки и техники. Вычислительные науки». — Т.1. — С. 3—75 [[http://www.citforum.ru/database/articles/art\\_27.shtml](http://www.citforum.ru/database/articles/art_27.shtml)].

34. *Carey M. J.* The EXODUS Extensible DBMS Project : An Overview / [M. J. Carey, D. J. DeWitt, G. Graefe et al.] // Readings in object-oriented database systems. Morgan Kaufmann, 1989. — P. 474—499 [<http://pages.cs.wisc.edu/~dewitt/includes/oodbms/exodus88.pdf>].

35. *Fagin R.* Extendible Hashing-A Fast Access Method for Dynamic Files / [R. Fagin, J. Nievergelt, N. Pippenger, H. R. Strong et al.] // ACM Transactions on Database Systems. — Vol. 4. — No. 3, September 1979. — P. 315—344 [<http://www.almaden.ibm.com/cs/people/fagin/tods79.pdf>].

36. *Litwin W.* Linear Hashing: A New Tool for File and Table Addressing // Proceedings of the Sixth International Conference on Very Large Data Bases, 1980, October. 1—3. — P. 212—223 [<http://www.cs.cmu.edu/afs/cs.cmu.edu/user/christos/www/courses/826-resources/PAPERS+BOOK/linear-hashing.PDF>].

37. *Кузнецов С.* Семейство алгоритмов ARIES / С. Кузнецов, П. Чардин // Открытые системы, 2004. — № 3. — С. 66—71 [<http://www.osp.ru/os/2004/03/184072/>]. Более подробный вариант статьи опубликован на странице [<http://www.citforum.ru/database/articles/aries/>].

38. *Кузнецов С. Д.* Базы данных: языки и модели. — М. : Бином, 2008.

39. *Кузнецов С. Д.* Основы баз данных. — М. : Бином, 2007.

# ОГЛАВЛЕНИЕ

|                   |   |
|-------------------|---|
| Предисловие ..... | 3 |
|-------------------|---|

## ЧАСТЬ I. БАЗЫ ДАННЫХ, СУБД И МОДЕЛИ ДАННЫХ

|                                                                                                                       |          |
|-----------------------------------------------------------------------------------------------------------------------|----------|
| <b>Глава 1. Назначение технологии баз данных. Функции и основные компоненты систем управления базами данных .....</b> | <b>6</b> |
|-----------------------------------------------------------------------------------------------------------------------|----------|

|                                                               |   |
|---------------------------------------------------------------|---|
| 1.1. Информационные системы и устройства внешней памяти ..... | 6 |
|---------------------------------------------------------------|---|

|                             |    |
|-----------------------------|----|
| 1.2. Файловые системы ..... | 10 |
|-----------------------------|----|

|                               |    |
|-------------------------------|----|
| 1.2.1. Структуры файлов ..... | 11 |
|-------------------------------|----|

|                                                                       |    |
|-----------------------------------------------------------------------|----|
| 1.2.2. Логическая структура файловых систем и именование файлов ..... | 13 |
|-----------------------------------------------------------------------|----|

|                                           |    |
|-------------------------------------------|----|
| 1.2.3. Авторизация доступа к файлам ..... | 16 |
|-------------------------------------------|----|

|                                                           |    |
|-----------------------------------------------------------|----|
| 1.2.4. Синхронизация многопользовательского доступа ..... | 17 |
|-----------------------------------------------------------|----|

|                                                  |    |
|--------------------------------------------------|----|
| 1.2.5. Области разумного применения файлов ..... | 17 |
|--------------------------------------------------|----|

|                                              |    |
|----------------------------------------------|----|
| 1.3. Потребности информационных систем ..... | 19 |
|----------------------------------------------|----|

|                               |    |
|-------------------------------|----|
| 1.3.1. Структуры данных ..... | 20 |
|-------------------------------|----|

|                                 |    |
|---------------------------------|----|
| 1.3.2. Целостность данных ..... | 23 |
|---------------------------------|----|

|                             |    |
|-----------------------------|----|
| 1.3.3. Языки запросов ..... | 25 |
|-----------------------------|----|

|                                                                     |    |
|---------------------------------------------------------------------|----|
| 1.3.4. Транзакции, журнализация и многопользовательский режим ..... | 26 |
|---------------------------------------------------------------------|----|

|                                               |    |
|-----------------------------------------------|----|
| 1.4. Основные функции и компоненты СУБД ..... | 28 |
|-----------------------------------------------|----|

|                                                       |    |
|-------------------------------------------------------|----|
| 1.4.1. СУБД как независимый системный компонент ..... | 28 |
|-------------------------------------------------------|----|

|                           |    |
|---------------------------|----|
| 1.4.2. Функции СУБД ..... | 30 |
|---------------------------|----|

|                                                   |    |
|---------------------------------------------------|----|
| 1.4.3. Типовая организация современной СУБД ..... | 34 |
|---------------------------------------------------|----|

|                                                                                  |           |
|----------------------------------------------------------------------------------|-----------|
| <b>Глава 2. Понятие модели данных. Обзор разновидностей моделей данных .....</b> | <b>38</b> |
|----------------------------------------------------------------------------------|-----------|

|                          |    |
|--------------------------|----|
| 2.1. Модель данных ..... | 38 |
|--------------------------|----|

|                                 |    |
|---------------------------------|----|
| 2.2. Ранние модели данных ..... | 39 |
|---------------------------------|----|

|                                                              |    |
|--------------------------------------------------------------|----|
| 2.2.1. Модель данных инвертированных таблиц .....            | 41 |
| 2.2.2. Иерархическая модель данных .....                     | 42 |
| 2.2.3. Сетевая модель данных .....                           | 44 |
| 2.3. Неформальное введение в реляционную модель данных. .... | 47 |
| 2.3.1. Реляционные структуры данных. ....                    | 48 |
| 2.3.2. Манипулирование реляционными данными .....            | 49 |
| 2.3.3. Целостность в реляционной модели данных .....         | 51 |
| 2.4. Современные модели данных. ....                         | 53 |
| 2.4.1. Объектно-ориентированная модель данных. ....          | 55 |
| 2.4.2. Модель данных SQL. ....                               | 60 |
| 2.4.3. Истинная реляционная модель .....                     | 64 |

## ЧАСТЬ II. РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ

|                                                                                                                                    |           |
|------------------------------------------------------------------------------------------------------------------------------------|-----------|
| <b>Глава 3. Реляционная модель данных. Понятия и определения. Основные свойства отношений. Целостность сущности и ссылок .....</b> | <b>68</b> |
|------------------------------------------------------------------------------------------------------------------------------------|-----------|

|                                                                                                    |    |
|----------------------------------------------------------------------------------------------------|----|
| 3.1. Базовые понятия реляционных баз данных .....                                                  | 69 |
| 3.1.1. Тип данных. ....                                                                            | 69 |
| 3.1.2. Домен. ....                                                                                 | 70 |
| 3.1.3. Заголовок отношения, кортеж, тело отношения, значение отношения, переменная отношения ..... | 71 |
| 3.1.4. Первичный ключ и интуитивная интерпретация реляционных понятий .....                        | 73 |
| 3.2. Фундаментальные свойства отношений .....                                                      | 74 |
| 3.2.1. Отсутствие кортежей-дубликатов, первичный и возможные ключи отношений. ....                 | 74 |
| 3.2.2. Отсутствие упорядоченности кортежей .....                                                   | 77 |
| 3.2.3. Отсутствие упорядоченности атрибутов. ....                                                  | 78 |
| 3.2.4. Атомарность значений атрибутов, первая нормальная форма отношения .....                     | 79 |
| 3.3. Реляционная модель данных. ....                                                               | 81 |
| 3.3.1. Общая характеристика. ....                                                                  | 81 |
| 3.3.2. Целостность сущности и ссылок .....                                                         | 82 |

|                                                        |           |
|--------------------------------------------------------|-----------|
| <b>Глава 4. Реляционные алгебра и исчисление .....</b> | <b>88</b> |
|--------------------------------------------------------|-----------|

|                                   |    |
|-----------------------------------|----|
| 4.1. Алгебра Кодда .....          | 89 |
| 4.1.1. Общая характеристика. .... | 90 |

|                                                                                  |     |
|----------------------------------------------------------------------------------|-----|
| 4.1.2. Замкнутость реляционной алгебры и операция переименования . . . . .       | 91  |
| 4.1.3. Особенности теоретико-множественных операций реляционной алгебры. . . . . | 92  |
| 4.1.4. Специальные реляционные операции . . . . .                                | 97  |
| 4.2. Реляционная алгебра А К. Дейта и Х. Дарвена . . . . .                       | 106 |
| 4.2.1. Базовые операции Алгебры А . . . . .                                      | 107 |
| 4.2.2. Полнота Алгебры А . . . . .                                               | 117 |
| 4.2.3. Избыточность Алгебры А. . . . .                                           | 125 |
| 4.2.4. Заключительные замечания. . . . .                                         | 127 |
| 4.3. Реляционное исчисление кортежей . . . . .                                   | 127 |
| 4.3.1. Кортежные переменные . . . . .                                            | 130 |
| 4.3.2. Правильно построенные формулы . . . . .                                   | 130 |
| 4.3.3. Целевые списки и выражения реляционного исчисления . . . . .              | 138 |
| 4.4. Реляционное исчисление доменов . . . . .                                    | 139 |
| 4.4.1. Условия членства . . . . .                                                | 140 |
| 4.4.2. Выражения исчисления доменов. . . . .                                     | 141 |

## ЧАСТЬ III. ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

|                                                                                                                                                                                       |            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|
| <b>Глава 5. Проектирование реляционных баз данных на основе учета функциональных зависимостей. Вторая и третья нормальные формы отношений, нормальная форма Бойса—Кодда . . . . .</b> | <b>143</b> |
| 5.1. Элементы теории функциональных зависимостей . . . . .                                                                                                                            | 145        |
| 5.1.1. Базовые определения и утверждения теории функциональных зависимостей . . . . .                                                                                                 | 146        |
| 5.1.2. Декомпозиция без потерь и функциональные зависимости . . . . .                                                                                                                 | 154        |
| 5.2. Минимальные функциональные зависимости и вторая нормальная форма . . . . .                                                                                                       | 159        |
| 5.2.1. Аномалии обновления, возникающие из-за наличия неминимальных функциональных зависимостей. . . . .                                                                              | 160        |
| 5.2.2. Возможная декомпозиция . . . . .                                                                                                                                               | 161        |
| 5.2.3. Вторая нормальная форма . . . . .                                                                                                                                              | 163        |
| 5.3. Нетранзитивные функциональные зависимости и третья нормальная форма . . . . .                                                                                                    | 164        |

|                                                                                                         |     |
|---------------------------------------------------------------------------------------------------------|-----|
| 5.3.1. Аномалии обновления, возникающие из-за наличия транзитивных функциональных зависимостей. . . . . | 164 |
| 5.3.2. Возможная декомпозиция . . . . .                                                                 | 164 |
| 5.3.3. Третья нормальная форма. . . . .                                                                 | 166 |
| 5.3.4. Независимые проекции отношений. Теорема Риссанена                                                | 167 |
| 5.4. Перекрывающиеся возможные ключи и нормальная форма Бойса—Кодда . . . . .                           | 168 |
| 5.4.1. Аномалии обновлений, связанные с наличием перекрывающихся возможных ключей. . . . .              | 169 |
| 5.4.2. Нормальная форма Бойса—Кодда . . . . .                                                           | 170 |
| 5.4.3. Всегда ли следует стремиться к BCNF? . . . . .                                                   | 171 |

## **Глава 6. Проектирование реляционных баз данных: дальнейшая нормализация . . . . . 176**

|                                                                                                     |     |
|-----------------------------------------------------------------------------------------------------|-----|
| 6.1. Многозначные зависимости и четвертая нормальная форма. . . . .                                 | 177 |
| 6.1.1. Аномалии обновлений при наличии многозначных зависимостей и возможная декомпозиция . . . . . | 177 |
| 6.1.2. Многозначные зависимости. Теорема Фейджина. Четвертая нормальная форма . . . . .             | 179 |
| 6.2. Зависимости проекции/соединения и пятая нормальная форма. . . . .                              | 183 |
| 6.2.1. N-декомпозируемые отношения. . . . .                                                         | 183 |
| 6.2.2. Зависимость проекции/соединения . . . . .                                                    | 185 |
| 6.2.3. Аномалии, вызываемые наличием зависимости проекции/соединения. . . . .                       | 186 |
| 6.2.4. Устранение аномалий обновления в 3-декомпозиции . .                                          | 187 |
| 6.2.5. Пятая нормальная форма . . . . .                                                             | 189 |

## **Глава 7. Проектирование реляционных баз данных с использованием диаграмм «Сущность-Связь» и диаграмм классов языка UML . . . . . 194**

|                                                                |     |
|----------------------------------------------------------------|-----|
| 7.1. Семантические модели данных . . . . .                     | 195 |
| 7.2. Семантическая модель Entity-Relationship (Сущность-Связь) | 198 |
| 7.2.1. Основные понятия ER-модели. . . . .                     | 199 |
| 7.2.2. Уникальные идентификаторы типов сущности . . . . .      | 203 |
| 7.2.3. Нормальные формы ER-диаграмм . . . . .                  | 206 |
| 7.2.4. Более сложные элементы ER-модели . . . . .              | 210 |
| 7.2.5. Получение реляционной схемы из ER-диаграммы. . . .      | 215 |

|                                                                               |     |
|-------------------------------------------------------------------------------|-----|
| 7.3. Диаграммы классов языка UML .....                                        | 219 |
| 7.3.1. Основные понятия диаграмм классов UML .....                            | 220 |
| 7.3.2. Ограничения целостности и язык OCL .....                               | 231 |
| 7.3.3. Получение схемы реляционной базы данных из диаграммы классов UML ..... | 240 |

## **ЧАСТЬ IV. АЛГОРИТМЫ И МЕТОДЫ ПОСТРОЕНИЯ РЕЛЯЦИОННЫХ СУБД**

### **Глава 8. Пример общей организации СУБД. Физическое представление реляционных баз данных во внешней памяти. Индексные структуры .....**

|                                                                                           |     |
|-------------------------------------------------------------------------------------------|-----|
| 8.1. Основные понятия, цели и общая организация System R ....                             | 244 |
| 8.1.1. Используемая терминология .....                                                    | 244 |
| 8.1.2. Цели System R и их связь с общей организацией системы .....                        | 245 |
| 8.1.3. Организация внешней памяти в базах данных System R .....                           | 250 |
| 8.1.4. Интерфейс RSS .....                                                                | 256 |
| 8.2. Общие принципы организации данных во внешней памяти в SQL-ориентированных СУБД. .... | 264 |
| 8.2.1. Хранение таблиц. ....                                                              | 266 |
| 8.2.2. Индексы. ....                                                                      | 268 |
| 8.2.3. Журнальная информация .....                                                        | 274 |
| 8.2.4. Служебная информация. ....                                                         | 274 |

### **Глава 9. Методы управления транзакциями. Схронизационные блокировки, временные метки и версии. ....**

|                                                                          |     |
|--------------------------------------------------------------------------|-----|
| 9.1. Общее понятие транзакции и основные характеристики транзакций ..... | 276 |
| 9.1.1. Атомарность транзакций .....                                      | 277 |
| 9.1.2. Транзакции и целостность баз данных .....                         | 278 |
| 9.1.3. Изолированность транзакций .....                                  | 280 |
| 9.1.4. Сериализация транзакций .....                                     | 284 |
| 9.2. Методы сериализации транзакций .....                                | 285 |
| 9.2.1. Синхронизационные блокировки .....                                | 286 |
| 9.2.2. Синхронизационные тупики, их распознавание и разрушение .....     | 296 |
| 9.2.3. Метод временных меток. ....                                       | 300 |



|                                                                                             |     |
|---------------------------------------------------------------------------------------------|-----|
| 9.2.4. Методы сериализации транзакций на основе поддержки версий объектов базы данных ..... | 301 |
|---------------------------------------------------------------------------------------------|-----|

## **Глава 10. Средства журнализации и восстановления баз данных .....**

|                                                                                         |     |
|-----------------------------------------------------------------------------------------|-----|
| 10.1. Буферизация блоков базы данных в основной памяти и ее связь с журнализацией. .... | 310 |
| 10.1.1. Управление буферным пулом базы данных .....                                     | 311 |
| 10.1.2. Физическая синхронизация .....                                                  | 313 |
| 10.1.3. Протокол упреждающей записи в журнал и его связь с буферизацией .....           | 315 |
| 10.2. Индивидуальный откат транзакции .....                                             | 316 |
| 10.3. Восстановление после мягкого сбоя .....                                           | 317 |
| 10.3.1. Схема восстановления от точки физической согласованности. ....                  | 318 |
| 10.3.2. Восстановление физической согласованности базы данных .....                     | 320 |
| 10.4. Восстановление базы данных после жесткого сбоя .....                              | 325 |

## **ЧАСТЬ V. МОДЕЛЬ ДАННЫХ SQL**

### **Глава 11. История стандарта языка SQL. Типы данных. Средства языка SQL для определения и изменения доменов, базовых таблиц и ограничений целостности .....**

|                                                       |     |
|-------------------------------------------------------|-----|
| 11.1. История стандарта SQL и структура языка. ....   | 330 |
| 11.1.1. Этапы процесса стандартизации языка SQL ..... | 330 |
| 11.1.2. Структура языка SQL .....                     | 334 |
| 11.2. Типы данных SQL .....                           | 336 |
| 11.2.1. Точные числовые типы. ....                    | 337 |
| 11.2.2. Приближенные числовые типы .....              | 338 |
| 11.2.3. Типы символьных строк. ....                   | 339 |
| 11.2.4. Типы битовых строк. ....                      | 340 |
| 11.2.5. Типы даты и времени. ....                     | 341 |
| 11.2.6. Типы временных интервалов. ....               | 343 |
| 11.2.7. Булевский тип. ....                           | 345 |
| 11.2.8. Типы коллекций .....                          | 346 |
| 11.2.9. Анонимные строчные типы .....                 | 348 |
| 11.2.10. Типы, определяемые пользователем .....       | 349 |

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| 11.2.11. Ссылочные типы .....                                                | 349 |
| 11.3. Средства определения доменов .....                                     | 350 |
| 11.3.1. Определение домена .....                                             | 350 |
| 11.3.2. Изменение определения домена .....                                   | 352 |
| 11.3.3. Отмена определения домена. ....                                      | 354 |
| 11.4. Средства определения, изменения и ликвидации базовых<br>таблиц .....   | 354 |
| 11.4.1. Определение базовой таблицы .....                                    | 355 |
| 11.4.2. Изменение определения базовой таблицы .....                          | 367 |
| 11.4.3. Отмена определения (уничтожение) базовой<br>таблицы .....            | 371 |
| 11.5. Средства определения и отмены общих ограничений целост-<br>ности ..... | 372 |
| 11.5.1. Определение общих ограничений целостности .....                      | 372 |
| 11.5.2. Отмена определения общего ограничения целостно-<br>сти .....         | 375 |
| 11.5.3. Немедленная и откладываемая проверка ограниче-<br>ний .....          | 375 |

## **Глава 12. Базовые возможности выборки данных в языке SQL .....**

|                                                              |     |
|--------------------------------------------------------------|-----|
| 12.1. Общая структура оператора выборки в языке SQL .....    | 379 |
| 12.1.1. Семантика оператора выборки .....                    | 380 |
| 12.1.2. Ссылки на таблицы раздела <b>FROM</b> .....          | 385 |
| 12.2. Представляемые таблицы, или представления (VIEW). .... | 390 |
| 12.3. Логические выражения раздела <b>WHERE</b> . ....       | 392 |
| 12.3.1. Предикат сравнения .....                             | 392 |
| 12.3.2. Предикат <b>between</b> .....                        | 396 |
| 12.3.3. Предикат <b>null</b> .....                           | 398 |
| 12.3.4. Предикат <b>in</b> .....                             | 399 |
| 12.3.5. Предикат <b>like</b> .....                           | 400 |
| 12.3.6. Предикат <b>similar</b> . ....                       | 401 |
| 12.3.7. Предикат <b>exists</b> .....                         | 404 |
| 12.3.8. Предикат <b>unique</b> . ....                        | 405 |
| 12.3.9. Предикат <b>overlaps</b> .....                       | 405 |
| 12.3.10. Предикат сравнения с квантором. ....                | 406 |
| 12.3.11. Предикат <b>match</b> .....                         | 408 |
| 12.3.12. Предикат <b>distinct</b> . ....                     | 409 |

|                                                                                                                      |     |
|----------------------------------------------------------------------------------------------------------------------|-----|
| 12.4. Логические выражения раздела <b>HAVING</b> .....                                                               | 410 |
| 12.4.1. Предикаты сравнения .....                                                                                    | 410 |
| 12.4.2. Предикат <b>in</b> .....                                                                                     | 412 |
| <b>Глава 13. Базовые возможности модификации баз данных в языке SQL</b> .....                                        | 414 |
| 13.1. Базовые средства манипулирования данными .....                                                                 | 415 |
| 13.1.1. Оператор <b>INSERT</b> для вставки строк в существующие таблицы .....                                        | 415 |
| 13.1.2. Оператор <b>UPDATE</b> для модификации существующих строк в существующих таблицах .....                      | 419 |
| 13.1.3. Оператор <b>DELETE</b> для удаления строк в существующих таблицах .....                                      | 421 |
| 13.2. Представления, над которыми возможны операции обновления .....                                                 | 422 |
| 13.2.1. Представления, допускающие применение операций обновления, в стандарте SQL/92 .....                          | 422 |
| 13.2.2. Представления, допускающие применение операций обновления, в стандарте SQL:1999 .....                        | 425 |
| 13.2.3. Раздел <b>WITH CHECK OPTION</b> определения представления .....                                              | 426 |
| 13.2.4. Исторический очерк .....                                                                                     | 433 |
| 13.3. Операции обновления баз данных и механизм триггеров ...                                                        | 434 |
| 13.3.1. Понятие триггера в SQL:1999 .....                                                                            | 435 |
| 13.3.2. Синтаксис определения триггеров и типы триггеров                                                             | 437 |
| 13.3.3. Выполнение триггеров .....                                                                                   | 441 |
| 13.3.4. Триггеры и ссылочные действия .....                                                                          | 446 |
| <b>Глава 14. Механизмы авторизации доступа и управления подключениями, сессиями и транзакциями в языке SQL</b> ..... | 448 |
| 14.1. Поддержка авторизации доступа к данным в языке SQL ..                                                          | 449 |
| 14.1.1. Пользователи и роли .....                                                                                    | 451 |
| 14.1.2. Использование идентификаторов пользователей и имен ролей .....                                               | 454 |
| 14.1.3. Создание и ликвидация ролей .....                                                                            | 455 |
| 14.1.4. Передача привилегий и ролей .....                                                                            | 456 |
| 14.1.5. Изменение текущих идентификаторов пользователей и имен ролей .....                                           | 460 |

|                                                             |     |
|-------------------------------------------------------------|-----|
| 14.1.6. Аннулирование привилегий и ролей . . . . .          | 461 |
| 14.2. Управление транзакциями в SQL . . . . .               | 464 |
| 14.2.1. Порождение транзакций в SQL . . . . .               | 464 |
| 14.2.2. Уровни изоляции SQL-транзакции . . . . .            | 467 |
| 14.2.3. Завершение транзакций . . . . .                     | 468 |
| 14.2.4. Транзакции и ограничения целостности . . . . .      | 469 |
| 14.2.5. Точки сохранения . . . . .                          | 470 |
| 14.3. Подключения и сессии . . . . .                        | 472 |
| 14.3.1. Установление соединений . . . . .                   | 474 |
| 14.3.2. Операторы SQL для управления соединениями . . . . . | 475 |
| Список литературы . . . . .                                 | 479 |

*Учебное издание*

**Кузнецов Сергей Дмитриевич**

**Базы данных**

**Учебник**

Технический редактор *О. Н. Крайнова*

Компьютерная верстка: *Д. В. Федотов*

Корректор *Н. Л. Котелина*

Изд. № 101113602. Подписано в печать 20.12.2011. Формат 60×90/16.

Гарнитура «Roman». Бумага офсетная № 1. Печать офсетная. Усл. печ. л. 31,0.

Тираж 1 200 экз. Заказ №

ООО «Издательский центр «Академия». [www.academia-moscow.ru](http://www.academia-moscow.ru)

125252, Москва, ул. Зорге, д. 15, корп. 1, пом. 26б.

Адрес для корреспонденции: 129085, Москва, пр-т Мира, 101В, стр. 1, а/я 48.

Тел./факс: (495) 648-0507, 616-00-29.

Отпечатано с электронных носителей издательства.

ОАО «Тверской полиграфический комбинат», 170024, г. Тверь, пр-т Ленина, 5.

Телефон: (4822) 44-52-03, 44-50-34. Телефон/факс: (4822) 44-42-15.

Home page [www.tverpk.ru](http://www.tverpk.ru) Электронная почта (E-mail) [sales@tverpk.ru](mailto:sales@tverpk.ru)

## **Редакционный совет серии**

### **Председатели совета:**

**академик РАН Ю. И. Журавлев,  
академик РАН В. А. Садовничий**

### **Члены совета:**

**О. М. Белоцерковский (академик РАН),  
В. П. Дымников (академик РАН),  
Ю. Г. Евтушенко (академик РАН),  
И. И. Еремин (академик РАН),  
В. А. Ильин (академик РАН),  
П. С. Краснощеков (академик РАН),  
Е. И. Моисеев (академик РАН),  
Л. Н. Королев (член-корреспондент РАН),  
Д. П. Костомаров (член-корреспондент РАН),  
Г. А. Михайлов (член-корреспондент РАН),  
Ю. Н. Павловский (член-корреспондент РАН),  
К. В. Рудаков (член-корреспондент РАН),  
Е. Е. Тыртышников (член-корреспондент РАН),  
И. Б. Федоров (член-корреспондент РАН),  
Б. Н. Четверушкин (член-корреспондент РАН)**

### **Ответственный редактор серии**

**доктор физико-математических наук  
Ю. И. Димитриенко**