ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

А.И. Муравьев

БАЗЫ ДАННЫХ

Учебное пособие

2006

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра промышленной электроники

А.И. Муравьев

БАЗЫ ДАННЫХ

Учебное пособие

2006

Рецензент: доцент кафедры ПМЭ ТПУ,
канд. техн. наук Цехановский С.А.

Муравьев А.И.
Базы данных: Учебное пособие. — Томск: Томский государственный университет систем управления и радиоэлектроники. — 136 с.

Представлено учебное пособие для подготовки студентов по дисциплине «Базы данных» в виде обобщенного конспекта лекций.

Для студентов, обучающихся по специальностям 210106 и 210104. Представляет интерес для студентов других специальностей.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Важнейшая задача компьютерных систем — хранение и обработка данных. Для ее решения были предприняты усилия, которые привели к появлению в конце 60-х — начале 70-х годов специализированного программного обеспечения — систем управления базами данных (DataGBase Management Systems — DBMS (СУБД)). СУБД позволяют структурировать, систематизировать и организовывать данные для их компьютерного хранения и обработки. Сегодня невозможно представить себе деятельность любого современного предприятия или организации без использования профессиональных СУБД. Несомненно, они составляют фундамент информационной деятельности во всех сферах — начиная с производства и заканчивая финансами и телекоммуникациями.

Глава 1. СОСТАВ ИНФОРМАЦИОННОЙ СИСТЕМЫ

1.1 Численные и информационные прикладные системы

Во всей истории вычислительной техники можно проследить две основных области ее использования. Первая область — применение вычислительной техники для выполнения численных расчетов, которые слишком долго или вообще невозможно производить вручную. Развитие этой области способствовало интенсификации методов численного решения сложных математических задач, развитию класса языков программирования, ориентированных на удобную запись численных алгоритмов, становлению обратной связи с разработчиками новых архитектур ЭВМ.

Вторая область, которая непосредственно относится к теме наших лекций, — это использование средств вычислительной техники в автоматических или автоматизированных информационных системах. В самом широком смысле информационная система представляет собой программно-аппаратный комплекс, функции которого состоят в надежном хранении информации в памяти компьютера, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса. Обычно такие системы имеют дело с большими объемами информации, и эта информация имеет достаточно сложную структуру. Классическими примерами информационных систем являются банковские системы, системы резервирования авиационных или железнодорожных билетов, мест в гостиницах и т.д.

Вторая область использования вычислительной техники возникла несколько позже первой. Это связано с тем, что на заре вычислительной техники возможности компьютеров по хранению информации были очень ограниченными. Говорить о надежном и долговременном хранении информации можно только при наличии запоминающих устройств, сохраняющих информацию после выключения электрического питания. Оперативная (основная) память компьютеров этим свойством обычно не обладает. В первых компьютерах использовались два вида устройств внешней памяти — магнитные ленты и барабаны. Емкость магнитных

лент была достаточно велика, но по своей физической природе они обеспечивали последовательный доступ к данным. Магнитные же барабаны (они больше всего похожи на современные магнитные диски с фиксированными головками) давали возможность произвольного доступа к данными, но были ограниченного размера.

Эти ограничения не являлись слишком существенными для чисто численных расчетов. Даже если программа должна обработать (или произвести) большой объем информации, при программировании можно продумать расположение этой информации во внешней памяти (например, на последовательной магнитной ленте), обеспечивающее эффективное выполнение этой программы.

Но для информационных систем, в которых потребность в текущих данных определяется конечным пользователем, наличие только магнитных лент и барабанов неудовлетворительно. Представьте себе покупателя билета, который, стоя у кассы, должен дождаться полной перемотки магнитной ленты. Одним из естественных требований к таким системам является удовлетворительная средняя скорость выполнения операций.

Как кажется, именно требования нечисленных приложений вызвали появление съемных магнитных дисков с подвижными головками, что явилось революцией в истории вычислительной техники. Эти устройства внешней памяти обладали существенно большей емкостью, чем магнитные барабаны, обеспечивали удовлетворительную скорость доступа к данным в режиме произвольной выборки, а возможность смены дискового пакета на устройстве позволяла иметь практически неограниченный архив данных.

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 памятью и устройствами внешней памяти с помощью программно-аппаратных средств низкого уровня (машинных команд или вызовов соответствующих программ операционной системы). Такой режим работы не позволяет или очень затрудняет 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.

Сегодня управление предприятием без компьютера просто немыслимо. Компьютеры прочно вошли в такие области управления, как бухгалтерский учет, управление складами, планирование и организация закупок, учет предоставляемых услуг и т.п. Тип используемой СУБД обычно определяется масштабом информационной системы. Малые информационные системы могут использовать локальные СУБД, в корпоративных же информационных системах используются мощные клиент-серверные СУБД, поддерживающих работу одновременно многих пользователей.

Одиночные информационные системы реализуются, как правило, на автономном персональном компьютере (сеть не используется). Такая система может содержать несколько простых приложений, связанных общим информационным фондом, и рассчитана на работу одного пользователя пли группы пользователей, разделяющих по времени одно рабочее место. Подобные приложения создаются с помощью так называемых настольных (локальных) систем управления базами данных Среди локальных СУБД наиболее известными являются Clarion, Clipper, FoxPro, Paradox, dBase и Microsoft Access.

Групповые информационные системы ориентированы на коллективное использование информации членами рабочей группы и чаще всего строятся на базе локальной вычислительной сети. При разработке таких приложении используются серверы баз данных (называемые также SQL-серверами) для рабочих групп. Существует довольно большое количество различных SQL-серверов, как коммерческих, так и свободно распространяемых. Среди них наиболее известны такие серверы баз данных, как Oracle, DB2, Microsoft SQL Server, InterBase. Sybase, Informix.

Корпоративные информационные системы являются развитием систем для рабочих групп, они ориентированы на крупные компании и могут поддерживать территориально разнесенные узлы или сети. В основном они имеют иерархическую структуру

из нескольких уровней. Для таких систем характерна архитектура клиент-сервер со специализацией серверов или же многоуровневая архитектура. При разработке таких систем могут использоваться те же серверы баз данных, что и при разработке групповых информационных систем. Однако в крупных информационных системах наибольшее распространение получили серверы Oracle, DB2 и Microsoft SQL Server.

Более подробно модели информационных систем рассмотрены в конце лекций.

1.2 Состав информационной системы

Состав информационной системы корпоративного предприятия приведен на рис. 1.

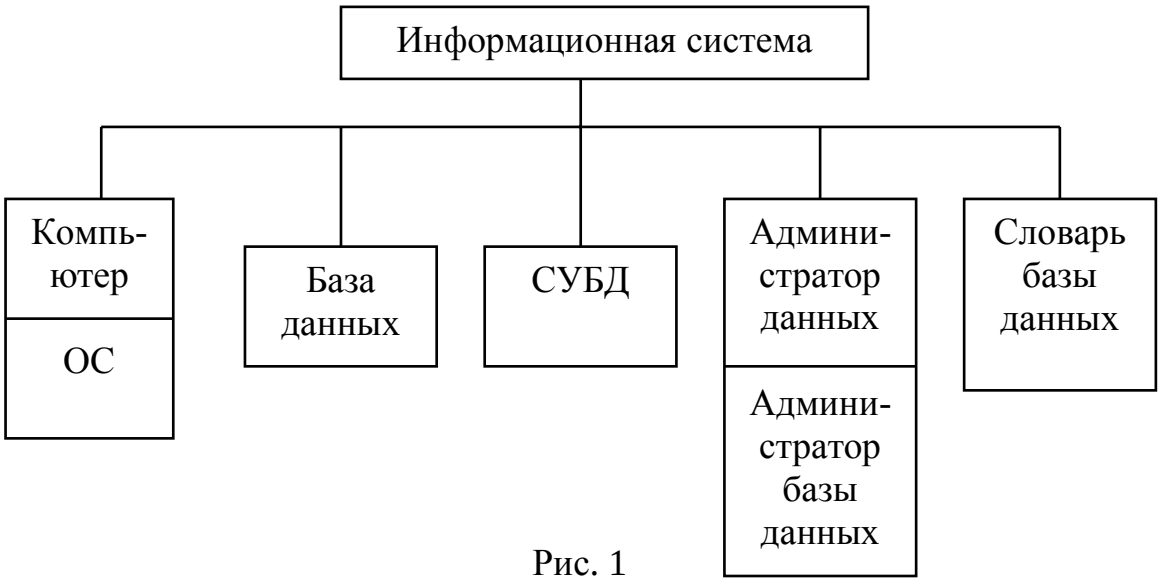


Рис. 1

Компьютер включает собственно сам компьютер и операционную систему (ОС), под которой работает сам компьютер. Понятно, что чем мощнее компьютер, тем быстрее будет осуществляться обработка данных и доступ к ним. Очень важное значение приобретает объем оперативной памяти установленной в компьютере и быстродействие накопителей на жестких магнитных дисках.

База данных (БД) состоит из некоторого набора связанных данных, которые используются прикладными системами. Более подробно БД рассмотрена ниже.

Наиболее важный компонент информационной системы — система управления базой данных (СУБД), которая является связующим звеном между пользователем и БД. Все запросы пользователей на доступ к БД обрабатываются СУБД: возможности добавления файлов (или таблиц) выборки или обновления данных также обеспечивает СУБД. Основная функция выполняемая СУБД — это предоставление пользователю БД возможности работы с ней, не вникая в детали на уровне аппаратного обеспечения.

Администратор данных — это специалист, который несет основную ответственность за данные предприятия. Он должен разбираться в данных и понимать нужды предприятия на уровне управления высшего руководства предприятием. Таким образом, в обязанности администратора данных входит: принимать решение, какие данные необходимо вносить в базу данных в первую очередь, а также обеспечивать поддержание порядка при обслуживании данных и использовании их после занесения в базу данных. Технический специалист (или группа специалистов), ответственный за реализацию решений администратора данных — это администратор базы данных или АБД. АБД отвечает за общее управление системой на техническом уровне.

АБД должен:

- определять, какие именно данные необходимо сохранять в БД, т.е. определять о каких объектах и их свойствах необходимо хранить информацию;
- решать, в каком виде нужно хранить необходимые данные;
- взаимодействовать с пользователями, обеспечивать наличие необходимых им данных;
- определять правила безопасности и целостности (непротиворечивости) данных;
- определять процедуры резервного копирования и восстановления БД в случае сбоя системы;

- обеспечивать необходимую производительность системы и реагировать на изменение требований к возможностям БД;
- обеспечивать правила безопасности и управление доступом к БД различных групп пользователей.

Словарь БД содержит «данные о данных» (называемых еще метаданными) и, представляет собой системную базу данных, в которой содержатся информация об объектах БД, их свойствах, описания столбцов и таблиц, формат представления данных, группы пользователей и их доступ к определенной части БД, о программах БД. По существу словарь базы представляет собой описание структуры БД.

Рассмотрим теперь подробнее, что представляет собой база данных. В узком смысле слова, база данных — это некоторый набор данных, необходимых для работы. Однако данные — это абстракция, никто никогда не видел просто данные, они не возникают и не существуют сами по себе. Данные — суть отражение объектов реального мира. Пусть, например, требуется хранить сведения о деталях, поступивших на склад. Как объект реального мира — деталь — будет отображена в базе данных? Для того, чтобы ответить на этот вопрос, необходимо знать, какие признаки или стороны детали будут актуальны, необходимы для работы. Среди них могут быть название детали, ее вес, размер, цвет, дата изготовления, материал, из которого она сделана и т.д. В традиционной терминологии объекты реального мира, сведения о которых хранятся в базе данных, называются сущностями, а их актуальные признаки — атрибутами.

Каждый признак конкретного объекта есть значение атрибута. Так, деталь ***двигатель*** имеет значение атрибута вес, равное 50, что отражает тот факт, что данный двигатель весит 50 килограммов. Было бы ошибкой считать, что в базе данных отражаются только физические объекты. Она способна вобрать в себя сведения об абстракциях, процессах, явлениях — то есть обо всем, с чем сталкивается человек в своей деятельности. Так, например, в базе данных можно хранить информацию о заказах на поставку деталей на склад (хотя заказ суть не физический объект,

а процесс). Атрибутами сущности ***заказ*** будут название поставляемой детали, количество деталей, название поставщика, срок поставки и т.д.

Определение: **сущность** — это объект, процесс или явление реального мира, о котором необходимо хранить информацию. Именованное свойство сущности — это **атрибут**. Хранится информация о сущностях, которая актуальна для конкретной предметной области и для конкретного применения. Например, атрибуты ***цвет глаз*** и ***рост*** вряд ли будут актуальны для хранения его в базе данных сотрудников предприятия, но в базе службы знакомств этот атрибут может быть использован.

Объекты реального мира имеют друг с другом множество сложных связей и зависимостей, которые необходимо учитывать в информационной деятельности. Например, детали на склад поставляются их производителями. Следовательно, в число атрибутов детали необходимо включить атрибут ***название фирмы-производителя***. Однако этого недостаточно, так как могут понадобиться дополнительные сведения о производителе конкретной детали — его адрес, номер телефона и т.д. Следовательно, база данных должна содержать не только сведения о деталях и заказах на поставку, но и сведения об их производителях. Более того, база данных должна отражать связи между деталями и производителями (каждая деталь выпускается конкретным производителем) и между заказами и деталями (каждый заказ оформляется на конкретную деталь).

Отметим, что в базе данных нужно хранить только актуальные, значимые связи.

Таким образом, в широком смысле слова база данных — это совокупность описаний объектов реального мира и связей, актуальных для конкретной прикладной области, между ними.

Глава 2. РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

В СУБД данные организованы таким образом, чтобы пользователи и прикладные программы могли считывать и обрабатывать эти данные. Организация данных и способы доступа к ним, обеспечиваемые конкретной СУБД, называются ее моделью данных. Вид модели определяется типом связей между данными. Модель данных определяет тип СУБД и круг приложений, для которых она подходит наилучшим образом. В 70—80-е годы появилось множество моделей данных, у каждой из них имелись свои достоинства и недостатки.

Кратко рассмотрим основные модели.

2.1 Системы управления файлами

До появления СУБД все данные, которые постоянно содержались в компьютерной системе, хранились в виде отдельных файлов. Система управления файлами, которая обычно является частью операционной системы компьютера, следила за именами файлов и местами их расположения. В системах управления файлами модели данных, как правило, не использовались; эти системы ничего не знали о внутреннем содержимом файлов.

Знание о содержимом файлов (какие данные в нем хранятся и какова их структура) было уделом прикладных программ, использующих этот файл. Если структура данных изменялась, необходимо было модифицировать каждую из программ, обращавшихся к файлу.

2.2 Иерархические СУБД

Структура многих сущностей является по своей природе иерархической структурой. Например, в структуре предприятия можно выделить главную часть, которой подчиняются другие части предприятия. Этим частям предприятия, в свою очередь, подчиняются другие части предприятия и т.д. Список составных частей какого-либо изделия также является по своей структуре иерархическим. Для хранения данных, имеющих такую структуру, была разработана иерархическая модель данных. На рис. 2

представлена иерархическая база данных, содержащая информацию о составных частях автомобиля. В этой модели каждая запись базы представляет конкретную деталь. Между записями существуют отношения предок-потомок, связывающие каждую часть с деталями, входящими в нее. Каждый элемент данных может иметь сколько угодно потомков и только одного предка, разумеется, кроме главного предка, который не может иметь своего предка. Для доступа к данным производились следующие операции:

- найти конкретную деталь по ее номеру;
- перейти вниз к первому потомку;
- перейти вверх к предку;
- перейти в сторону к другому потомку.

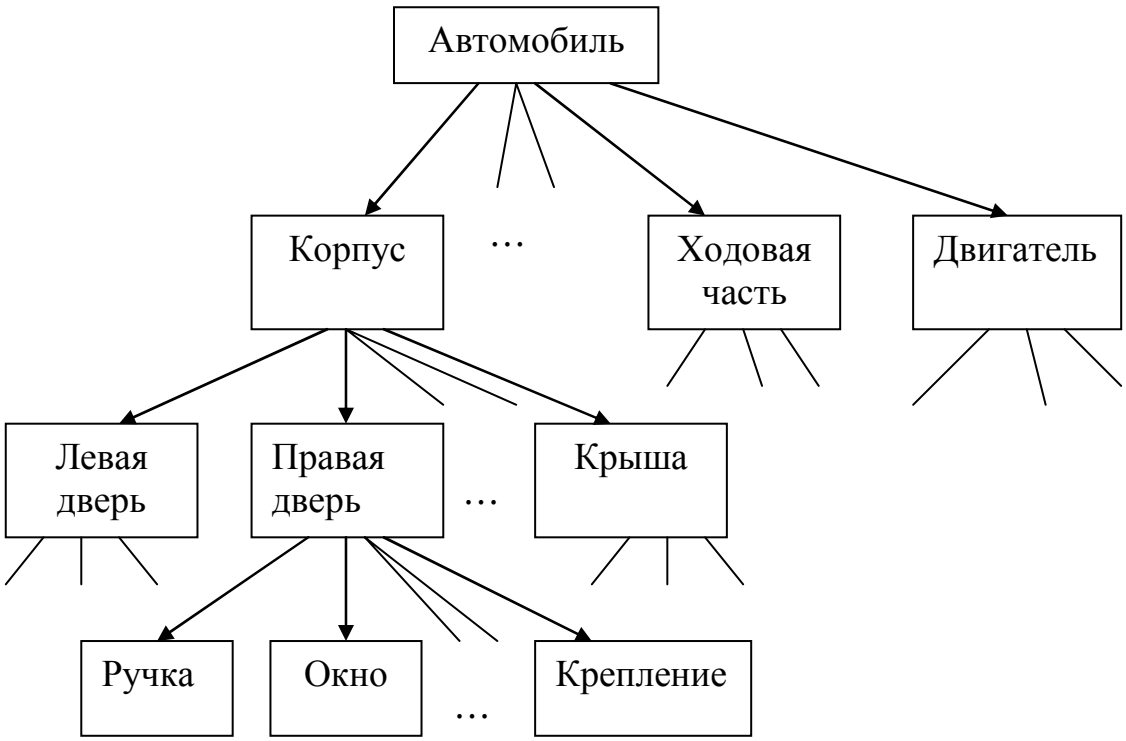


Рис. 2

Таким образом, для чтения данных из иерархической базы данных требовалось перемещение по записям, за один шаг переходя на одну запись вверх, вниз или в сторону. Иерархические СУБД характеризуются простотой представления данных (хотя не все сущности могут быть представлены в виде иерархической структуры) и исключительным быстродействием. К недостаткам

можно отнести отсутствие математической базы модели и жесткую привязку приложений к структуре базы данных. Изменение структуры базы данных приводит к тому, что нужно переписывать приложения, иногда очень существенно. В настоящее время они функционируют в основном на больших ЭВМ, где их доля среди других СУБД составляет примерно 25 %.

2.3 Сетевые базы данных

Если структура данных оказывалась сложнее, чем обычная иерархия, простота структуры иерархической базы становилась ее недостатком, например если какой-то элемент данных имеет больше одного предка. Для таких приложений была разработана сетевая модель данных, представляющая собой граф произвольной структуры. В такой модели каждое данное может быть связано отношением с любым данным, в том числе само с собой (рекурсивная связь). Был опубликован официальный стандарт сетевых баз данных, который получил название CODASYL. Появление стандарта увеличило популярность сетевой модели, и многие компании создали свои версии сетевой СУБД. На рис. 3 показан пример сетевой базы данных.

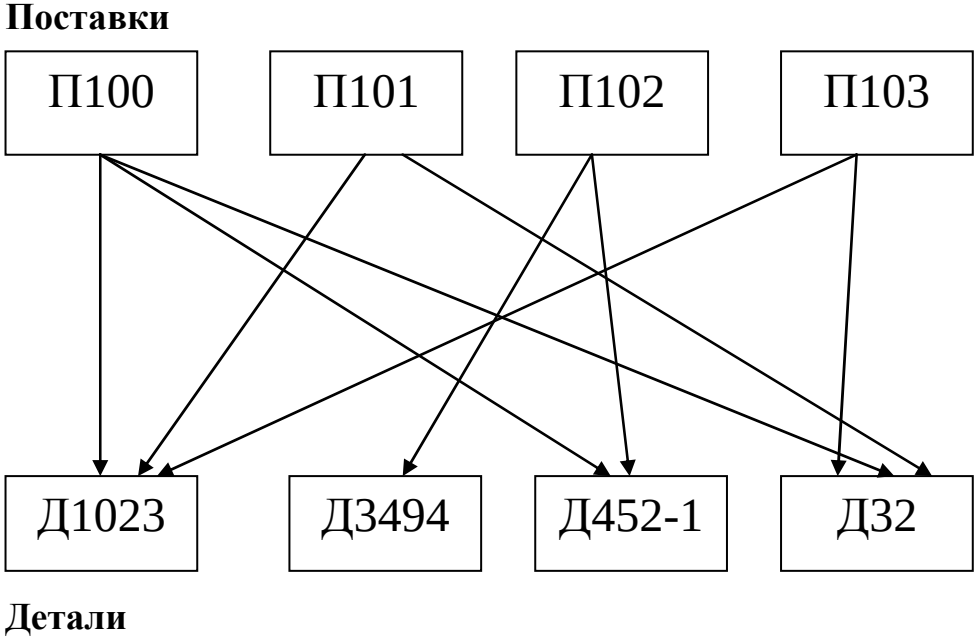


Рис. 3

Как в иерархической, так и в сетевой базе данных для манипулирования данными программистам приходилось писать большие программы, чтобы осуществить навигацию в базе. Изменение структуры базы приводило к модификации программ. Недостатки иерархической и сетевой моделей привели к созданию реляционной модели.

2.4 Реляционные базы данных

Реляционная модель была предложена математиком фирмы IBM Е.Ф. Коддом в 1970 году и была попыткой упростить структуру базы данных. Основным фундаментальным свойством реляционной модели является то, что данные хранятся в виде таблиц и только таблиц. Для пользователей информационной системы недостаточно, чтобы база данных просто отражала объекты реального мира. Важно, чтобы такое отражение было однозначным и непротиворечивым. В этом случае говорят, что база данных удовлетворяет условию целостности (integrity).

Для того чтобы гарантировать корректность и взаимную непротиворечивость данных, на базу данных накладываются ограничения, которые называют ограничениями целостности (data integrity constraints).

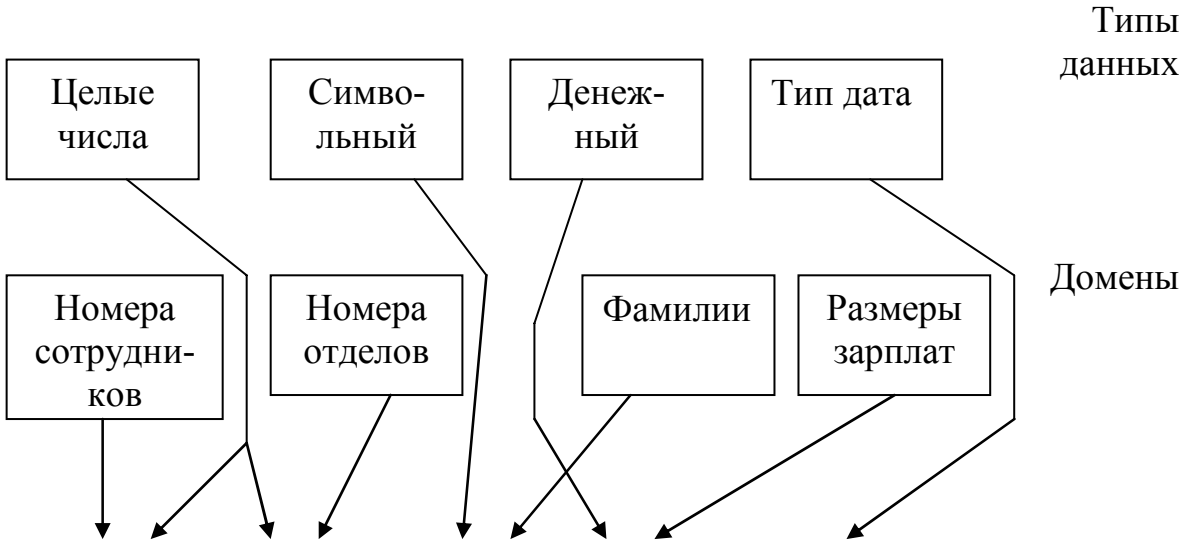
Реляционной считается такая база данных, в которой все данные представлены для пользователя в виде плоских нормализованных прямоугольных таблиц значений данных, и все операции над базой данных сводятся к манипуляциям с таблицами. Таблица состоит из строк и столбцов и имеет имя, уникальное внутри базы данных. Таблица отражает тип объекта реального мира (сущность), а каждая ее строка — конкретный объект. Так, таблица ***Деталь*** содержит сведения обо всех деталях, хранящихся на складе, а ее строки суть набор значений атрибутов каждой конкретной детали. Каждый столбец таблицы — это совокупность значений конкретного атрибута объекта. Так, столбец ***материал*** представляет собой множество значений ***сталь, медь, цинк, никель*** и т.д. В столбце ***количество*** содержатся целые положительные числа. Значения в столбце ***вес*** суть вещественные числа, равные весу детали в килограммах (или в граммах).

2.5 Базовые понятия реляционных баз данных

Основными понятиями реляционных баз данных являются:

- отношение
- атрибут;
- кортеж;
- тип данных;
- домен;
- возможный (потенциальный) ключ;
- первичный ключ;
- внешний ключ.

Для начала покажем смысл этих понятий на примере отношения ***Сотрудник***, содержащего информацию о сотрудниках некоторой организации (рис. 4).



Номер сотрудника	**Номер отдела**	**Фамилия**	**Зарплата**	**Дата рождения**
2954	100	Иванов	2000	01.01.1977
4233	120	Петров	3500	02.02.1980
4233	122	Сидоров	1800	12.03.1955
3655	100	Сергеев	3500	8.03.1970
2232	111	Петров	2200	31.12.1950
…….	……	…….	……	…….

Рис. 4

2.5.1 Тип данных

Понятие тип данных в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких, как деньги), а также специальных темпоральных данных (дата, время, временной интервал). Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres). В нашем примере мы имеем дело с данными четырех типов: строки символов, целые числа, деньги и тип дата.

Понятие домена более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных. Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа.

Домен ***Фамилии*** в вышерассмотренном примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать фамилию (в частности, такие строки не могут начинаться с мягкого или твердого знака). Домен номера отделов может иметь значения, допустимые для данного предприятия. Например, на предприятии имеются отделы с номерами 100-112, 116-125, 130 (отделы с номерами 113-115,126-129 не существуют). Тогда значения домена включают значения от 100 до 130 с исключением значений номеров несуществующих отделов. Значения домена могут меняться с течением времени (например, при добавлении или удалении отдела). Считается, что значение атрибута имеет правильное значение, если его значение берется из соответствующего домена (это необходимое, хотя и недостаточное условие).

Пусть в отношении ***Деталь*** имеются атрибуты ***артикул, название, материал*** и ***регион_поставки*** (предполагается, что поставки идут только из России). Тогда допустимыми значения-

ми домена ***регион_поставки*** будут значения, включающие названия областей и республик, из которых могут идти поставки детали, и значения атрибута регион_поставки берут свои значения из этого домена.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере (рис.4) значения доменов номера сотрудников и номера отделов относятся к типу целых чисел, но не являются сравнимыми, поскольку они относятся к разным доменам. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle, начиная с версии V.7, оно уже поддерживается.

2.5.2 Схема отношения, схема базы данных

Схема отношения — это именованное множество пар имя атрибута — имя домена (или типа, если понятие домена не поддерживается). Степень или арность схемы отношения — мощность этого множества. Степень отношения ***Сотрудник*** равна пяти, то есть оно является 5-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно 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). Схема БД (в структурном смысле) — это набор именованных схем отношений.

2.5.3 Кортеж, отношение

Кортеж, соответствующий данной схеме отношения, — это множество пар имя атрибута, значение, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. Значение является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень кортежа, или арность кортежа, т.е. число элементов в нем, совпадает с арностью соответствующей схемы отношения. Попросту говоря, кортеж — это набор именованных значений заданного типа. Отношение — это мно-

жество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят отношение-схема и отношение-экземпляр, иногда схему отношения называют заголовком отношения, а отношение как набор кортежей — телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем — одно или несколько отношений с данной схемой, однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. Отношение характеризуется кардинальным числом, значение которого равно числу кортежей. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть эволюцией схемы базы данных.

Обычным житейским представлением отношения является таблица, заголовком которой является схема отношения, а строками — кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят столбец таблицы, имея в виду атрибут отношения. Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД. Реляционная база данных — это набор связанных отношений, имена которых совпадают с именами схем отношений в схеме БД. Как видно, основные структурные понятия реляционной модели данных (если не считать понятия домена) имеют очень простую интуитивную интерпретацию, хотя в теории реляционных БД все они определяются абсолютно формально и точно.

2.5.4 Пустые значения атрибутов

В некоторых случаях какой-либо атрибут отношения может быть неприменим. Например, если в отношении в отношении **сотрудник** хранится информация не только о сотрудниках, работающих в отделах, но и о руководящих сотрудниках (которые не

работают ни в одном отделе), тогда атрибут номер отдела для руководящих сотрудников не применим. Или же, иногда при вводе значений в строку реляционной таблицы некоторые данные могут быть неизвестны и выясняться позже (для нашего примера зарплата какого-либо сотрудника еще не определена и будет определена позднее). В этих случаях в соответствующие атрибуты этих сотрудников ничего не заносится и строка записывается в базу данных с пустыми значениями этих атрибутов. Следует понимать, что пустое значение — это не нулевое значение и не пустое символьное значение, а неизвестное (на данный момент) или не применимое значение. Если для сотрудника, не работающего ни в одном отделе, мы занесем в поле номера отдела нулевое значение, это будет неверно (будет означать, что сотрудник работает в несуществующем отделе с номером 0). Для обозначения пустых значений атрибутов используется слово **NULL**. При сравнении пустых значений не действуют стандартные правила сравнения: одно пустое значение никогда не считается равным другому пустому значению. Правила сравнения пустых значений рассматриваются в языке SQL.

2.6 Фундаментальные свойства отношений

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

2.6.1 Отсутствие кортежей-дубликатов

То свойство, что отношения не содержат кортежей-дубликатов, следует из определения отношения как множества кортежей. В классической теории множеств, по определению, каждое множество состоит из различных элементов. Из этого свойства вытекает наличие у каждого отношения так называемого **возможного ключа** — атрибута или набора атрибутов, значения которых однозначно определяют (идентифицируют) кортеж отношения. Для каждого отношения, в предельном случае, полный набор его атрибутов обладает этим свойством. Однако при формальном определении возможного ключа требуется обеспечение его минимальности, т.е. в набор атрибутов возможного ключа не

должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства — однозначно определять кортеж. Кроме того, атрибуты, входящие в возможный ключ, не должны допускать пустых значений. Среди всех возможных ключей один выбирается в качестве **первичного ключа,** который является главным, все остальные называются **вторичными**. Понятие первичного ключа является исключительно важным в связи с понятием целостности баз данных. Забегая вперед, заметим, что во многих практических реализациях РСУБД 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 иногда приводит к серьезным проблемам. В приведенном выше примере в качестве возможного ключа может выступать атрибут **номер сотрудника**, при условии, что номер является уникальным на предприятии (номер сотрудника в данном случае не должен допускать пустых значений, т.е. каждый сотрудник должен иметь номер). Если же номер сотрудника является уникальным значением в пределах отдела, тогда возможным ключом будет составной атрибут {**номер сотрудника, номер отдела}**. Если бы в приведенном выше примере присутствовал атрибут ***номер паспорта,*** тогда этот атрибут мог быть выбран в качестве возможного ключа (поскольку каждый сотрудник должен иметь паспорт и значения номеров паспортов должны быть уникальны). Если бы присутствовал атрибут ***номер страхового свидетельства*** (государственного пенсионного страхования), этот атрибут не мог быть выбран в качестве возможного ключа, поскольку сотрудник может не иметь такого свидетельства. Атрибуты ***номер отдела, фамилия, зарплата и дата рождения*** не могут быть выбраны в качестве возможного ключа, поскольку они допускают дублирующие значения (значения этих атрибутов не уникальны).

2.6.2 Отсутствие упорядоченности кортежей

Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения-экземпляра как множества кортежей. Отсутствие требования к поддержанию

порядка на множестве кортежей отношения дает дополнительную гибкость СУБД при хранении баз данных во внешней памяти и при выполнении запросов к базе данных, хотя упорядочение кортежей влияет на скорость доступа к данным. Это не противоречит тому, что при формулировании запроса к БД, например на языке SQL, можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых столбцов, и такой результат, вообще говоря, не отношение, а некоторый упорядоченный список кортежей.

2.6.3 Отсутствие упорядоченности атрибутов

Атрибуты отношений могут быть не упорядочены, поскольку по определению схема отношения есть множество пар имя атрибута + имя домена. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Это свойство теоретически позволяет, например, модифицировать схемы существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих атрибутов. Однако в большинстве существующих систем такая возможность не допускается, и хотя упорядоченность набора атрибутов отношения явно не требуется, часто в качестве неявного порядка атрибутов используется их порядок в линейной форме определения схемы отношения.

2.6.4 Атомарность значений атрибутов

Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого (скалярного) типа данных, т.е. среди значений домена не могут содержаться множества значений (отношения). Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме. Потенциальный пример ненормализованного отношения показан на рис. 5

Номер_отдела	Номер сотруд	Фамилия сотруд	Зарплата сотруд
310	2934	Иванов	112,00
	2543	Сидоров	345,00
	4354	Петров	345,00
314	3733	Федоров	423,00
315	5481	Иванова	250,00

Рис. 5

а нормализованное отношение показано на рис. 6.

Номер отдела	Номер сотруд	Фамилия сотруд	Зарплата сотруд
310	2934	Иванов	112,00
310	2543	Сидоров	345,00
310	4354	Петров	345,00
314	3733	Федоров	423,00
315	5481	Иванова	250,00

Рис.6

Нормализованные 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 сотрудника Кузнецова (номер 3000, зарплата 115,000) в отдел номер 320 и Зачислить сотрудника Кузнецова (номер 3000, зарплата 115,000) в отдел номер 310. Если информация о сотрудниках представлена в виде отношения (рис. 5), оба оператора будут выполняться одинаково (вставить кортеж в отношение **Сотрудник**). Если же работать с ненормализованным отношением, то первый оператор выразится в занесение кортежа, а второй — в добавление информации о Кузнецове в множественное значение атрибута Отдел кортежа с первичным ключом 310.

Другой пример ненормализованного отношения: пусть сотрудники могут одновременно числиться в нескольких отделах

Номер сотрудника	Номер отдела	Фамилия	Зарплата
234	100 110	Иванов	2000
354	120	Петров	3000
432	100 101 110	Сидоров	3000

Здесь атрибут номера отдела имеет множественное значение (нарушается фундаментальное свойство отношения — атомарность значений атрибутов). Для нормализации данного отношения нужно для каждого сотрудника иметь столько строк, в скольких отделах он работает.

Номер сотрудника	Номер отдела	Фамилия	Зарплата
234	100	Иванов	2000
234	110	Иванов	2000
354	120	Петров	3000
432	100	Сидоров	3000
432	101	Сидоров	3000
432	110	Сидоров	3000

Примечание: для данной таблицы номер сотрудника не может быть возможным ключом, даже если сотрудник имеет уникальный номер в пределах предприятия, поскольку значения этого номера в отношении не уникальны. Возможным ключом здесь может быть составной атрибут:

{номер сотрудника, номер отдела}

2.7 Связанные отношения

В реляционной модели данные представляются в виде совокупности взаимосвязанных отношений. Это вытекает из-за того, что между сущностями могут быть установлены связи — бинарные ассоциации, показывающие, каким образом сущности соотносятся между собой. Если связь устанавливается между двумя сущностями, то она показывает взаимосвязь между экземплярами

обоих сущностей. Например, у нас есть связь между сущностью ***сотрудник*** и сущностью ***отдел***, поскольку сотрудники работают в отделах и отделы имеют в наличии сотрудников.

Существует три типа связей:

- «один-к-одному»
- «один-ко-многим»
- «многие-ко-многим»

Приведенный выше пример (при условии, что сотрудник работает только в одном отделе) определяет связь типа «один-ко-многим», поскольку одному экземпляру сущности **отдел** соответствует несколько экземпляров сущности ***сотрудник*** (в отделе могут работать несколько сотрудников) и одному экземпляру сущности **сотрудник** соответствует один экземпляр сущности **отдел** (сотрудник работает только в одном отделе). Для примера связи «один-к-одному» возьмем сущность ***сотрудник*** и сущность ***служебный автомобиль*** (за сотрудником может быть закреплено не более одного автомобиля, а автомобиль может быть выделен не более чем одному сотруднику). Одному экземпляру сущности ***автомобиль*** соответствует не более одного экземпляра сущности ***сотрудник*** и наоборот. Связь типа «многие-ко-многим» присутствует между сущностями ***студент*** и ***преподаватель***. Студент обучается у нескольких преподавателей, и преподаватель обучает несколько студентов, т.е. одному экземпляру сущности ***студент*** может соответствовать несколько экземпляров сущности ***преподаватель*** и, наоборот, одному экземпляру сущности ***преподаватель*** может соответствовать несколько экземпляров сущности ***студент***. Связь типа «многие-ко-многим» присутствует и между сущностью ***студент*** и сущностью ***изучаемый предмет***. Определите сами, почему?

Связь необязательно может существовать между разными сущностями, она может быть между одной и той же сущностью (рекурсивная связь). В сущности ***сотрудник*** может существовать связь между разными экземплярами этой сущности, например сотрудники могут быть связаны тем, что живут по одному и тому же адресу (или же быть в браке).

Между двумя сущностями может быть несколько различных связей. Между сущностями ***преподаватель*** и ***студент*** могут быть связи «чтение лекций», «дипломное проектирование», «практические занятия».

Связь может быть обязательной или необязательной, причем может быть обязательной с обеих сторон сущностей, необязательной с обеих сторон и обязательной с одной стороны и не обязательной с другой. Если не каждому экземпляру первой сущности должны соответствовать экземпляры второй сущности, тогда связь является необязательной со стороны первой сущности. Связь «практические занятия» между сущностями ***преподаватель*** и ***студент*** является необязательной со стороны сущности ***преподаватель*** (не каждый преподаватель обязан проводить практические занятия) и обязательной со стороны сущности ***студент*** (каждый студент обязан выполнять практические занятия). Такая же связь и между сущностями ***сотрудник*** и ***дети сотрудников*** (не каждый сотрудник имеет детей (связь со стороны сотрудника необязательна), а у каждого ребенка, помещенного в базу, должен быть родитель, поскольку в базу заносятся только дети сотрудников — связь со стороны отношения ***дети сотрудников*** обязательна).

2.8 Внешние ключи отношения

В базах данных одни и те же атрибуты часто используются в разных отношениях для установления логических связей между отношениями. Например, в отношениях ***сотрудник*** и ***отдел*** может присутствовать атрибут ***номер отдела*** (для отношения ***отдел*** этот атрибут является первичным ключом, а для отношения ***сотрудник*** — внешним ключом).

Внешний ключ — это атрибут (простой или составной) отношения, вводимый в это отношение для связи с другим отношением, который является возможным ключом для этого другого отношения. Связь между отношениями осуществляется путем равенства внешнего ключа одного отношения и возможного (часто первичного) ключа другого отношения. Внешний ключ при связи «один-ко-многим», как правило, имеет дублирующие значения (в отношении ***сотрудник***, например для отдела с номером

100, будет столько этих значений, сколько сотрудников работает в отделе с номером 100). При типе связей «один-ко-многим» отношение, которое связано посредством первичного (возможного) ключа, является **родительским**, а отношение, связанное внешним ключом является **дочерним**. На нашем примере отношение ***отдел*** — это родительское отношение, а отношение ***сотрудник*** — дочернее. На рис. 7 показано, каким образом отношения связаны посредством первичного и внешнего ключей.

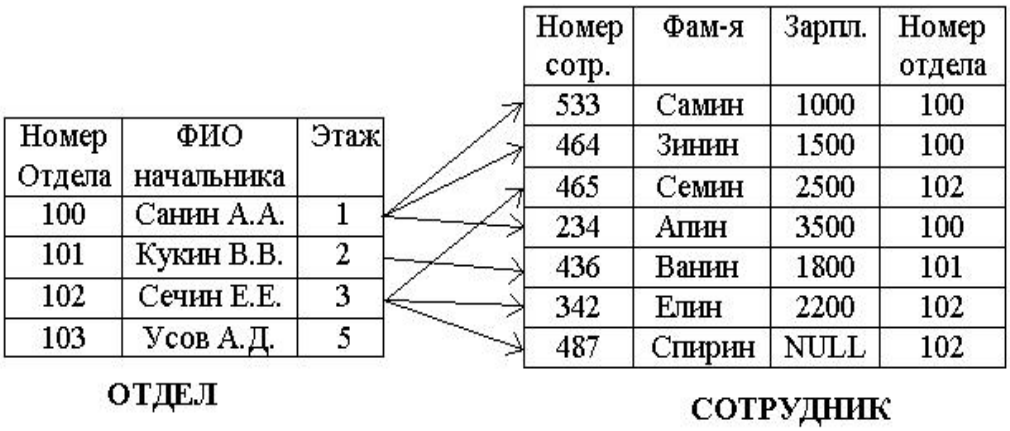


Рис. 7

2.9 Целостность

Для того чтобы хранящаяся в базе данных информация была однозначной и непротиворечивой, в реляционной модели устанавливаются некоторые ограничительные условия (правила целостности, определяющие возможные значения данных и поддерживающие правильные условия связи таблиц). Эти ограничения можно задавать при создании таблиц, таким образом можно свести к минимуму ошибки при вводе, удалении и модификации данных. Совокупность данных обладает целостностью, если данные логически согласованы. Когда данные дублируются, это может нарушать их целостность. Например, если фамилия сотрудника дублирована в нескольких таблицах, то при вводе нужно будет вводить одинаковые значения, а при изменении нужно обновлять данные в нескольких таблицах. Существует опасность, что обновлены будут не все таблицы и между ними появятся несоответствия. Проблемы целостности являются очень серьезны-

ми. Если данные противоречат друг другу, это приведет к несогласованным результатам и неопределенности. Выделяют несколько типов ограничений целостности, мы рассмотрим важнейшие из них:

- Целостность таблицы.
- Целостность данных.
- Ссылочная целостность (или целостность ссылок).

Целостность таблицы заключается в том, что никакой атрибут возможного (и, естественно, первичного) ключа не может иметь пустое значение. Для извлечения данных строки из таблицы или для манипулирования данными нужно знать значение первичного ключа для этой строки, и если оно будет пустым, то получится неоднозначность. При определении таблицы можно указать первичный ключ, и система не допустит ввод пустого значения любого атрибута, входящего в первичный ключ.

Целостность данных заключается в том, чтобы все данные имели допустимое значение. Это также можно определить на уровне создания таблиц, хотя такое ограничение может быть более сложным, чем предыдущее, и часто может быть реализовано не декларативно, а только написанием специальных процедур проверки правильности ввода значений (триггеров баз данных).

Ссылочная целостность реализуется для правильной связи таблиц и заключается в том, чтобы каждому значению внешнего ключа дочерней таблицы соответствовало значение первичного ключа дочерней таблицы (кроме значений NULL внешнего ключа, которые могут присутствовать при необязательном типе связи со стороны дочерней таблицы). Например, если в таблице на рис. 7 будет присутствовать сотрудник со значением внешнего ключа 110, получится, что в базе хранятся данные сотрудника работающего в несуществующем отделе. В разных СУБД поддержка ссылочной целостности реализована по-разному, рассмотрим наиболее типичные случаи. В случае вставки строки в дочернюю таблицу значения внешнего ключа, которое не соответствует первичному ключу, в родительской таблице выдается сообщение об ошибке и вставка строки в этом случае игнорируется. В случае удаления строки из родительской таблицы порожденные строки в

дочерней таблице либо также каскадно удаляются, либо значения внешних ключей приобретают значение NULL, либо удаление игнорируется с выдачей ошибки. При изменении внешнего ключа на значение, которого нет в родительской таблице, выдается сообщение об ошибке и изменение аннулируется. При изменении первичного ключа в случае присутствия порожденных строк изменение либо игнорируется, либо значения внешних ключей порожденных строк приобретают значения NULL. Какой тип реакции будет осуществляться, задается при создании таблицы. Поддерживать ссылочную целостность можно также с помощью специальных процедур.

Глава 3. ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

При проектировании базы данных решаются две основные проблемы:

1. Отображение объектов предметной области в абстрактные объекты модели данных таким образом, чтобы это отображение не противоречило семантике предметной области и было по возможности лучшим (эффективным, удобным и т.д.). Часто эту проблему называют проблемой логического проектирования баз данных (или проблемой инфологического проектирования).

2. Обеспечение эффективного выполнения запросов к базе данных, т.е. рациональное расположение данных во внешней памяти, создание полезных дополнительных структур (например, индексов) с учетом особенностей конкретной СУБД. Эту проблему называют проблемой физического проектирования баз данных (или проблемой датологического проектирования).

В общем, проблема проектирования формулируется следующим образом: как в некоторой базе данных для заданного набора данных выбрать подходящую логическую структуру? Иначе говоря, нужно решить вопрос, какие отношения и с какими атрибутами следует задать. Также необходимо выяснить, какие домены следует использовать.

3.1 Модель «Сущность-Связь» (ER-модель)

Модель «сущность-связь» (ER-модель) была предложена Питером Ченом для представления семантики (смысла) предметной области и формируется после словесного описания предметной области на ранних стадиях разработки. Эта модель используется во многих CASE-инструментах. Разработаны методы автоматического преобразования проекта базы данных из ER-модели в реляционную.

Основными понятиями ER-модели являются сущности, атрибуты, первичные ключи и связи.

Сущность — это объект, явление или процесс реального мира, о котором необходимо хранить информацию. Сущность имеет имя, уникальное в пределах моделируемой системы. Экземпляру сущности соответствует определенный объект. Например, для сущности ***сотрудник*** экземпляру соответствует определенный сотрудник. Каждый экземпляр сущности имеет свой набор атрибутов — характеристик, определяющих свойства экземпляра. Например, для сущности ***книга*** имеется набор атрибутов: название, автор, количество страниц, издательство, ISBN и т.д. Имена атрибутов должны быть уникальны в пределах соответствующей им сущности. Понятие первичного ключа мы рассмотрели ранее. Сущность изображается в виде прямоугольника, в верхней части которого записано название сущности, ниже перечисляются атрибуты с подчеркиванием ключевых атрибутов. Пример представления сущности ***книга*** в ER- модели представлен на рис. 8.

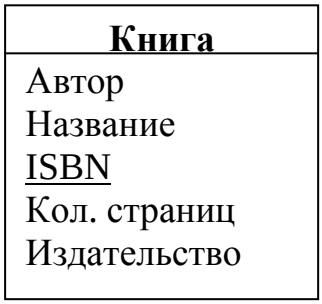
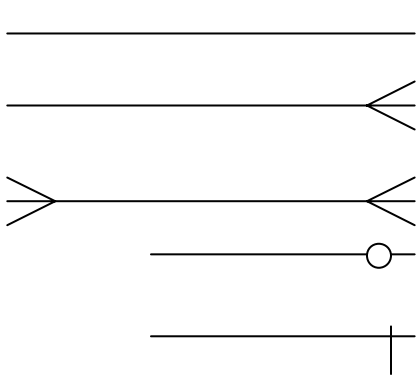


Рис. 8

Связи (их типы мы рассмотрели ранее) показывают, как соотносятся между собой сущности. Ниже показано графическое изображение различных типов связей.



Напомним, что при обязательном типе связей каждый экземпляр сущности должен быть связан с экземпляром (или экземплярами) другой сущности, при необязательном он может быть не связан.

Для сущностей ***отдел*** и ***сотрудник***, при условии, что сотрудник может работать только в одном отделе, в отношении ***сотрудник*** хранится информация только о сотрудниках, работающих в отделах, и в отношение ***отдел*** может быть внесена информация об отделе, не имеющем сотрудников (после создания отдела не принято в него ни одного сотрудника), приведена диаграмма на рис. 9

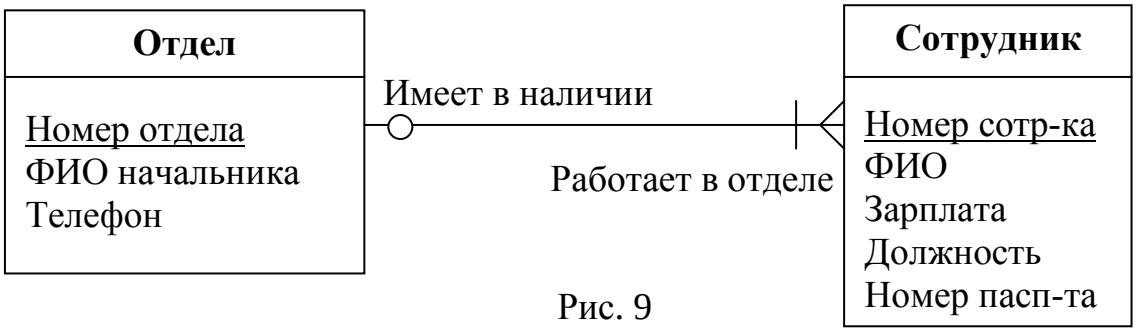


Рис. 9

Приведем еще один пример диаграммы с большим количеством сущностей. Пусть в базе кроме информации об отделах и сотрудниках должна храниться информация о детях сотрудников и о школах, в которых обучаются дети. Сотрудник как может иметь детей, так и не может. В случае, если ребенок имеет обоих родителей среди сотрудников, он связан только с одним из родителей. В отношение ***дети*** вносится информация о детях, находящихся на иждивении родителей, и дети могут или не могут учиться в школе. У одного родителя не может быть два ребенка с одинаковыми именами. В отношение ***школа*** вносится информация о всех школах города (общеобразовательные, художественные, музыкальные, спортивные и т.п., т.е. один ребенок может учиться в нескольких школах одновременно). Диаграмма представлена на рис. 10.

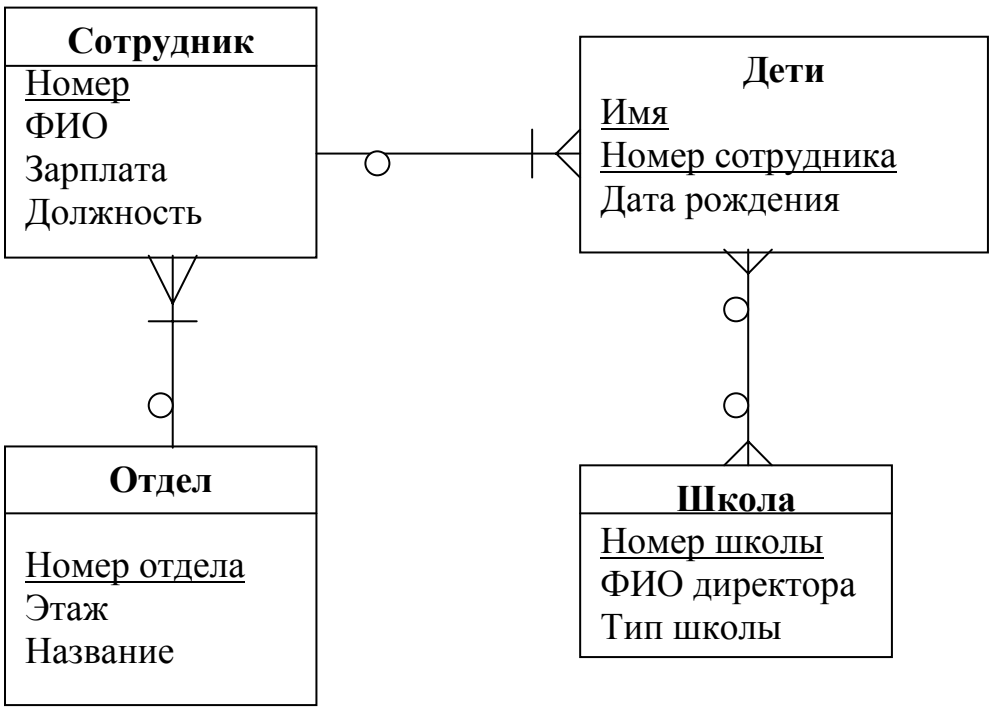


Рис. 10

Рассмотрим правила перехода от ER-модели к реляционной модели:

1. Каждой сущности ставится в соответствие отношение реляционной модели, при этом имена сущностей и имена отношений могут быть различными, поскольку на имена отношений накладываются синтаксические ограничения конкретной СУБД по именованию имен. Например, для отношения, соответствующего сущности ***отдел,*** может быть использовано имя OTDEL.

2. Каждый атрибут сущности становится атрибутом соответствующего отношения, именование сущностей происходит по тем же правилам, что и именование отношений. Для каждого атрибута задаются соответствующий тип атрибута и обязательность или необязательность атрибута (допустимость или недопустимость значений NULL).

3. Первичный ключ становится первичным ключом соответствующего отношения и получает свойство NOT NULL (недопустимость пустых значений).

4. В каждое дочернее отношение добавляется первичный ключ родительского отношения и становится внешним ключом отношения. Внешний ключ приобретает свойство NOT NULL

при обязательном типе связей со стороны дочернего отношения и свойство NULL при необязательном типе связи.

5. Поскольку в реляционной модели не поддерживаются типы связи «многие-ко-многим», то при таком типе создается специальное дополнительное отношение связи, в которое вносятся первичные ключи связываемых отношений и эта совокупность первичных ключей становится первичным ключом этого дополнительного отношения. У дополнительного отношения получается два внешних ключа для связи с соответствующими ему отношениями.

Приведем пример перехода в реляционную модель (рис. 11) из приведенной выше ER-модели. При этом отметим, что копирования первичного ключа из отношения ***сотрудник*** в отношение ***дети*** не будет, поскольку этот атрибут уже присутствует в этом отношении для определения первичного ключа (без атрибута ***номер сотрудника*** невозможно было определить первичный ключ в отношении ***дети***, поскольку составной атрибут ***имя+дата рождения*** не является уникальным):

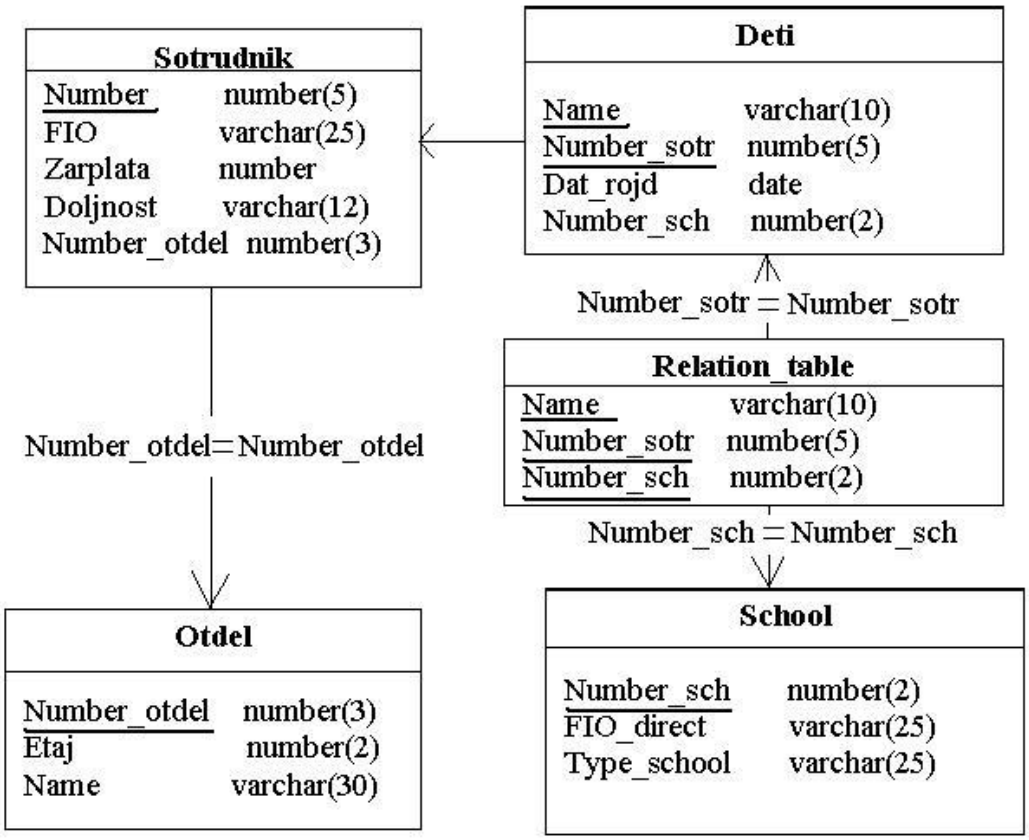


Рис. 11

3.2 Нормализация отношений

Рассмотрим классический подход, при котором весь процесс проектирования производится в терминах реляционной модели данных методом последовательных приближений к удовлетворительному набору схем отношений.

Исходной точкой является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования составляется некоторый набор схем отношений, обладающих лучшими свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причем каждая следующая нормальная форма обладает свойствами лучшими, чем предыдущая. При модификации данных в отношениях возможны аномалии, обусловленные избыточностью данных, а избыточность, как отмечалось ранее, может привести к потере целостности базы данных. Нормализация отношений устраняет избыточность данных и аномалии модификации.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. Примером является ограничение первой нормальной формы: значения всех атрибутов отношения должны быть атомарными (отметим, что понятие первой нормальной формы и ее свойства отличаются некоторой тавтологичностью, поскольку являются следствиями основных определений реляционных категорий). Поскольку требование первой нормальной формы 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 (NFBK) (правильнее было бы считать эту нормальную форму третьей, однако по истори-

ческим причинам третья ступень оказалась занятой к моменту выявленения NFBK, из-за чего она и получила нестандартное название);

- четвертая нормальная форма (4NF);
- пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).
- Доменно-ключевая нормальная форма.

Основные свойства нормальных форм:

- каждая следующая нормальная форма в некотором смысле улучшает свойства предыдущей;
- при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе классического процесса проектирования лежит метод нормализации, который опирается на декомпозицию (на основе проекции) отношения, находящегося в предыдущей нормальной форме, в два или более отношений, удовлетворяющих требованиям следующей нормальной формы.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии функциональной зависимости. Для дальнейшего изложения нам потребуются несколько определений.

Рассмотрим понятие функциональной зависимости.

Определение 1: **Функциональная зависимость** (functional dependence — FD).

В отношении R атрибут Y функционально зависит от атрибута X (X и Y могут быть составными атрибутами, т.е. реально состоять из нескольких атомарных атрибутов) в том и только в том случае, если каждому значению X соответствует в точности одно значение Y, что в символическом виде записывается как

$$\mathbf{\{R.X \rightarrow R.Y\}}$$

(и читается либо как «X функционально определяет Y», либо как «X стрелка Y»). Иначе говоря, если два кортежа совпадают по значению X, они должны совпадать и по значению Y (каждое

значение атрибута X связано только с одним значением атрибута Y). Если зависимости рассматриваются для одного конкретного отношения, тогда имя отношения можно опустить.

Левую и правую части символической записи иногда называют **детерминантом** и **зависимой частью** соответственно. Если детерминант или зависимая часть содержит только один атрибут, они называются одноэлементными и фигурные скобки в символической записи можно опустить. Для примера определим отношение (рис.12), в котором атрибут S означает номер поставщика, P — номер детали, CITY — город, в котором находится поставщик, а атрибут QTY — количество поставляемых деталей. Кроме того, одна и та же деталь может поставляться разными поставщиками. Отношение удовлетворяет приведенной ниже функциональной зависимости, поскольку все кортежи отношения с одинаковым значением атрибута S имеют одинаковое значение атрибута CITY.

S	CITY	P	QTY
S1	Москва	P1	100
S1	Москва	P2	200
S2	Красноярск	P1	200
S2	Красноярск	P2	200
S3	Красноярск	P2	300
S4	Москва	P2	400
S4	Москва	P4	300
S4	Москва	P5	400

Рис.12

S → CITY

На самом деле это отношение удовлетворяет нескольким функциональным зависимостям:

{S,P} → QTY
{S,P} → CITY
{S,P} → {CITY,QTY}

{S,P} → S
{S,P} → {S,P,CITY,QTY}

Необходимо рассматривать только те функциональные зависимости, которые распространяются на множество всех возможных значений атрибутов отношения и поэтому всегда остаются справедливыми. Кроме того, необходимо исключить из рассмотрения все тривиальные функциональные зависимости. Функциональная зависимость называется тривиальной, если она остается справедливой при любых условиях. К тому же зависимость является тривиальной, если и только если в правой части выражения, определяющего зависимость, приведено подмножество (но необязательно собственное подмножество) множества, которое указано в левой части (детерминанте) выражения. Примеры тривиальной зависимости:

{S,P} → S
{S,P} → {S,P,CITY,QTY}

Замечание. Термин **функциональная зависимость** не случаен, поскольку в точности соответствует математическому понятию функции.

Определение 2: **Полная функциональная зависимость.**

Функциональная зависимость R.X → R.Y называется полной, если атрибут Y не зависит функционально от любого точного подмножества X (точным подмножеством множества X называется любое его подмножество, не совпадающее с X). Иначе говоря, неполная функциональная зависимость существует, если атрибут Y зависит от части составного атрибута X. Неполная функциональная зависимость возможна только в том случае, если атрибут X является составным (состоящим из нескольких атрибутов). Для составного атрибута X, состоящего из трех атрибутов x_1,x_2,x_3, может быть 6 подмножеств:

x_1 ; x_2 ; x_3 ; x_1,x_2 ; x_1,x_3 ; x_2,x_3

Определение 3: **Транзитивная функциональная зависимость.**

Функциональная зависимость R.X → R.Y называется транзитивной, если существует такой атрибут Z, что имеются функциональные зависимости R.X → R.Z и R.Z → R.Y.

Определение 4**: Возможный ключ (alternative key).**

Возможным ключом отношения называется его атомарный или составной атрибут, значения которого полностью функционально определяют значения всех остальных атрибутов отношения.

Определение 5: **Неключевой атрибут.**

Неключевым атрибутом называется любой атрибут отношения, не входящий в состав возможного ключа.

Определение 6: **Взаимно независимые атрибуты.**

Два или более атрибута называются взаимно независимыми, если ни один из этих атрибутов не является функционально зависимым от других неключевых атрибутов.

3.2.1 Первая нормальная форма

Отношение находится в первой нормальной форме, если значения всех его атрибутов простые (атомарные), т.е. значение атрибута не должно быть множеством или повторяющейся группой. Отношение всегда должно находиться минимум в первой форме, поскольку это вытекает из фундаментального свойства отношений.

3.2.2 Вторая нормальная форма

Пусть имеется отношение ***ПОСТАВКИ***, содержащее информацию о поставщиках (идентифицируемых первичным ключом П), поставляемых ими товарах и их ценах:

ПОСТАВКИ (П, ТОВАР, ЦЕНА)

Предположим, что поставщик может поставлять различные товары, а один и тот же товар могут поставлять разные поставщики. Таким образом, возможный ключ отношения (выделенный подчеркиванием) будет составным из атрибутов П и ТОВАР. Известно, что цена любого товара зафиксирована (т.е. поставщики поставляют товар по одной и той же цене). Зависимости отношения будут:

{П,ТОВАР} → ЦЕНА (по определению возможного ключа)
ТОВАР → ЦЕНА

Можно отметить неполную функциональную зависимость атрибута ЦЕНА от возможного ключа. Это приводит к следующим аномалиям:

• Аномалия включения. Если у поставщика появляется новый товар, информация о товаре и его цене не может храниться в базе данных до тех пор, пока поставщик не станет поставлять его.

• Аномалия удаления. Если поставки некоторого товара прекращаются, из базы данных придется удалить сведения о товаре и его цене, даже если он имеется в наличии у поставщика.

• Аномалия обновления. При изменении цены товара необходим полный просмотр отношения, с целью найти все поставки этого товара, чтобы изменение цены было отражено для всех поставщиков. Таким образом, изменение значения атрибута одного объекта влечет необходимость изменений в нескольких кортежах отношения: в противном случае база данных окажется несогласованной.

Причиной этих аномалий является неполная функциональная зависимость атрибута ***ЦЕНА*** от ключа, что обусловлено объединением в отношении ***ПОСТАВКИ*** двух семантических фактов в одной структуре. Разложение отношения ***ПОСТАВКИ*** на два отношения устраняет неполную функциональную зависимость.

Отношение находится во второй нормальной форме, если оно находится в первой нормальной форме и каждый неключевой

атрибут функционально полно зависит от любого возможного ключа.

Следующее разложение приводит к отношению во второй нормальной форме (2НФ):

ПОСТАВКИ_1(П,ТОВАР)
ЦЕНА_ТОВАРА(ТОВАР,ЦЕНА)

Здесь два отношения связаны типом «один-ко-многим», родительским отношением будет отношение ***ЦЕНА_ТОВАРА***, дочерним — отношение ***ПОСТАВКИ_1*** (с внешним ключом ***ТОВАР***, который является частью первичного ключа). Тип связи — «один-ко-многим».

Цену товара конкретной поставки можно определить путем соединения двух отношений по атрибуту ***ТОВАР***, как будет рассмотрено ниже в реляционных операциях. Изменение цены товара вызовет модификацию лишь одного кортежа второго отношения.

3.2.3 Третья нормальная форма

Пусть имеется отношение **ХРАНЕНИЕ** (ФИРМА,СКЛАД, ОБЬЕМ), которое содержит информацию о фирмах, получающих товары со складов, и объемах этих складов. Фирма получает товары только с одного склада, с одного склада получают товары разные фирмы. Отношение находится во второй нормальной форме, т.к. единственно возможный ключ является простым. В отношении имеются функциональные зависимости:

ФИРМА $\rightarrow$ СКЛАД (фирма получает товары только с одного склада)
СКЛАД $\rightarrow$ ОБЪЕМ

В отношении присутствует транзитивная зависимость

ФИРМА $\rightarrow$ ОБЪЕМ,

т.к так как ФИРМА $\rightarrow$ СКЛАД и СКЛАД $\rightarrow$ ОБЪЕМ (поскольку есть функциональная зависимость между неключевыми атрибутами СКЛАД и ОБЪЕМ).

Аномалии. Если на данный момент отсутствует фирма, получающая товар со склада, то в базу данных нельзя ввести информацию об объеме склада (аномалия включения). Если последняя фирма перестает получать товар со склада, данные о складе и его объеме нельзя сохранить в базе данных (аномалия удаления). Если объем склада изменяется, необходим просмотр всего отношения и изменение кортежей для фирм, связанных со складом (аномалия обновления). Присутствующая здесь транзитивная зависимость (аналогично неполной функциональной зависимости в предыдущем примере) вызвана наличием в отношении двух семантических различных фактов.

Отношение находится в третьей нормальной форме (3НФ), если оно находится в 2НФ и в нем отсутствуют транзитивные зависимости непервичных атрибутов от любого возможного ключа (или, иначе говоря, в отношении отсутствуют функциональные зависимости между неключевыми атрибутами). Следующая декомпозиция приводит к отношениям в 3НФ:

ХРАНЕНИЕ_1(ФИРМА,СКЛАД)
С_ОБЪЕМ(СКЛАД,ОБЪЕМ)

Определите, какое из этих отношений является родительским, а какое — дочерним?

3.2.4 Нормальная форма Бойса-Кодда

Рассмотрим отношение с перечисленными ниже условиями.

1. Отношение имеет два (или более) возможных ключа.
2. Два возможных ключа являются составными.
3. Они перекрываются (т.е. имеют, по крайней мере, один общий атрибут).

Таким условиям удовлетворяет отношение, в котором хранятся данные изучения предметов студентами и преподавателей, которые эти предметы преподают:

ОБУЧЕНИЕ(НОМЕР_ЗАЧ_КНИЖ, ПРЕДМЕТ, НОМЕР_ПРЕП.**)**

Номера зачетных книжек и номера преподавателей являются уникальными для студента и преподавателя. Наложим ограничение, что каждый преподаватель может быть связан только с одним предметом и любой предмет может преподаваться разными преподавателями. Для каждого студента в отношении будет столько кортежей, сколько предметов он изучает, поэтому номер зачетной книжки будет дублирован и не может быть возможным ключом. Номера преподавателей также не могут быть уникальны в отношении (один и тот же предмет изучается разными студентами). Таким образом, ни один скалярный атрибут не может выступать в качестве возможного ключа. В качестве возможных ключей здесь два составных атрибута — (НОМЕР_ЗАЧ_КНИЖ, ПРЕДМЕТ) и (НОМЕР_ЗАЧ_КНИЖ, НОМЕР_ПРЕП). Отношение находится в третьей нормальной форме хотя бы потому, что в нем отсутствуют неключевые атрибуты. Аномалии тем не менее присутствуют. Например, если преподаватель переходит на другой предмет, придется просматривать все отношение для изменения. Если преподаватель в данный момент не преподает ни один предмет, нет информации, с каким предметом он связан. В отношении имеются функциональная зависимость:

$$\text{НОМЕР\_ПРЕП} \rightarrow \text{ПРЕДМЕТ}$$

что и определяет аномалии в отношении. Устранить аномалии можно декомпозицией отношения на два отношения

ОБУЧЕНИЕ1(НОМЕР_ЗАЧ_КНИЖ**,** НОМЕР_ПРЕП**)**

ПРЕП_ПРЕДМЕТ(НОМЕР_ПРЕП, ПРЕДМЕТ**)**

Отношение находится в нормальной форме Бойса-Кодда (NFBK), если оно находится в третьей нормальной форме и детерминанты являются возможными ключами.

3.2.5 Четвертая нормальная форма

НФБК позволяет устранить любые аномалии, вызванные функциональными зависимостями. Однако в результате теоретических исследований был выявлен еще один тип зависимости — многозначная зависимость, которая при проектировании отношений также может вызвать проблемы, связанные с избыточностью данных. Возможность существования в отношении многозначных зависимостей возникает вследствие приведения исходных таблиц к форме 1НФ, для которой не допускается наличие некоторого набора значений на пересечении одной строки и одного столбца. Например, при наличии в отношении двух многозначных атрибутов для достижения непротиворечивого состояния строк необходимо повторить в них каждое значение одного из атрибутов в сочетании с каждым значением другого атрибута.

Пусть дано ненормализованное отношение **ПРЕПОДАВАТЕЛЬ**, в котором содержится информация о номерах преподавателей, предметах, которые они преподают (разные преподаватели могут вести один и тот же предмет) и именах детей преподавателей. Для преподавателя с номером 100, ведущего правоведение и политологию и имеющего детей с именами Петр, Мария и Егор часть ненормализованного отношения будет выглядеть следующим образом (атрибуты ПРЕДМЕТ и ИМЯ_РЕБЕНКА являются многозначными атрибутами):

Номер_преп	Предмет	Имя_ребенка
…..	…..	……
100	Политология Правоведение	Петр Мария Егор
….	…..	…..

Поскольку предмет и имена детей никак не связаны друг с другом, нормализованное отношение имеет вид:

Номер_преп	Предмет	Имя_ребенка
…	…	…
100	Политология	Петр
100	Политология	Мария
100	Политология	Егор
100	Правоведение	Петр
100	Правоведение	Мария
100	Правоведение	Егор
…	…	…

Любой составной атрибут, состоящий из двух одиночных атрибутов имеет дублирующие значения и не может быть возможным ключом. Единственный здесь возможный ключ состоит из всех трех атрибутов. Очевидно, что отношение характеризуется значительной избыточностью и приводит к аномалии обновления. Для ввода нового предмета для данного преподавателя придется создавать три новых кортежа. Тем не менее отношение находится в нормальной форма Бойса-Кодда, поскольку имеет всего один возможный ключ (является «полностью ключевым»). Имеется какая либо связь между атрибутами? Номеру преподавателя соответствует некоторое множество предметов, равно как и номеру преподавателя также соответствует множество значений имен детей. Такие зависимости называются многозначными и приводят к аномалиям модификации.

Определение: **Многозначная зависимость.**

В отношении R(X, Y, Z) существует многозначная зависимость (обзначается как $X \rightarrow\rightarrow Y$) в том и только в том случае, если множество значений Y, соответствующее паре значений X и Z, зависит только от X и не зависит от Z. В рассмотренном выше отношении есть две многозначный зависимости:

$$\text{Номер\_преп} \rightarrow\rightarrow \text{Предмет}$$

$$\text{Номер\_преп} \rightarrow\rightarrow \text{Имя\_ребенка}$$

Для устранения аномалий отношение можно спроецировать без потерь в два отношения: **ПРЕП_ПРЕДМЕТ**(Номер_преп, Предмет) и ПРЕП_ДЕТИ(Номер_преп, Имя_ребенка).

Отношение находится в в четвертой нормальной форме, если оно находится в нормальной форме Бойса-Кодда и, в случае существования многозначной зависимости $X \rightarrow\rightarrow Y$ все остальные атрибуты функционально зависят от A (иначе говоря, в отношении не должно быть двух многозначных зависимостей).

Заметим, что процедура нормализации обратима, т.е. всегда можно использовать ее результат (например, множество отношений, находящихся в 3НФ) для обратного преобразования (в исходное отношение, находящееся в 2НФ). Возможность обратного преобразования является очень важной характеристикой, поскольку означает, что в процессе нормализации информация не утрачивается.

Иногда осуществляют декомпозицию, т.е. понижают нормы отношений для увеличения быстродействия выполнения запросов, поскольку соединение двух отношений в одно требует определенное время для выполнения.

Глава 4. БАЗИСНЫЕ СРЕДСТВА МАНИПУЛИРОВАНИЯ РЕЛЯЦИОННЫМИ ДАННЫМИ

Манипуляция данными является частью реляционной модели данных. В манипуляционной составляющей 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 рассматриваются два вида реляционного исчисления — исчисление доменов и исчисление предикатов. 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 результатами вычислений также являются отношения. Как следствие, любое выражение или формула могут интерпретироваться как отношение, что позволяет использовать их в других выражениях или формулах. Как мы увидим, 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. По этой причине именно эти механизмы включены в реляционную модель данных. Конкретный язык манипулирования реляционными БД называется реляционно-полным, если любой запрос, выражаемый с помощью одного выражения реляционной алгебры или одной формулы реляционного исчисления, может быть выражен с помощью одного оператора этого языка.

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 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, выра-

жение реляционной алгебры также имеет процедурную интерпретацию. Другими словами, запрос, представленный на языке реляционной алгебры, может быть определен на основе вычисления элементарных алгебраических операций с учетом их старшинства и возможного наличия скобок. Для формулы реляционного исчисления подобная интерпретация, вообще говоря, отсутствует. Формула только устанавливает условия, которым должны удовлетворять кортежи результирующего отношения. Поэтому языки реляционного исчисления являются более непроцедурными или декларативными. Обычно (как, например, в случае языка SQL) язык основывается на некоторой смеси алгебраических и логических конструкций.

4.1 Реляционная алгебра

Основная идея реляционной алгебры состоит в том, что если отношения являются множествами, то средства манипулирования отношениями могут базироваться на традиционных теоретико-множественных операциях, дополненных специальными операциями, специфичными для баз данных. Существует много подходов к определению реляционной алгебры, которые различаются набором операций и способами их интерпретации, но, в принципе, они более или менее равносильны. Опишем немного расширенный начальный вариант алгебры, который был предложен Коддом. В этом варианте 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 операции (модифицированные с учетом того, что их операндами являются отношения, а не произвольные множества):

- объединения отношений;
- пересечения отношений;
- взятия разности отношений;
- Декартова произведения отношений.

Специальные реляционные операции включают:

- ограничение отношения;

- проекцию отношения;
- соединение отношений;
- деление отношений.

Кроме того, в состав алгебры включаются операция присваивания, позволяющая сохранить в базе данных результаты вычисления алгебраических выражений, и операция переименования атрибутов, дающая возможность корректно сформировать заголовок (схему) результирующего отношения. Главное назначение реляционной алгебры — определение области выборки и обновления данных.

4.2 Общая интерпретация реляционных операций

Операция объединения

При выполнении операции объединения двух совместимых по типу отношений A и B (A UNION B) производится отношение с тем же заголовком, как и в отношениях A и B, и с телом, включающим все кортежи, входящие хотя бы в одно из отношений-операндов. Повторяющиеся кортежи в результирующем отношении удаляются по определению. Эта операция коммутативна, т.е. A UNION B= B UNION A

Операция пересечения

Операция пересечения двух совместимых по типу отношений A и B (A INTERSECT B) производит отношение с тем же заголовком, как и в отношениях A и B, и с телом, включающим кортежи, входящие в оба отношения-операнды. Эта операция также коммутативна, т.е.

A INTERSECT B=B INTERSECT A

Операция разности

Отношение, являющееся разностью двух совместимых по типу отношений A и B (A MINUS B) и с тем же заголовком, включает все кортежи, входящие в отношение — первый операнд, такие, что ни один из них не входит в отношение, являющееся вторым операндом. Операция разности не коммутативна.

Операция произведения

При выполнении прямого (Декартова) произведения двух отношений A и B (A TIMES B), где A и B не имеют общих имен атрибутов, производится отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго операндов во всех возможных сочетаниях и кардинальным числом, равном произведению кардинальных чисел отношений A и B.

A TIMES B= B TIMES A

На рис. 13—19 показан пример рассмотренных операций.

Отношение A

S	SNAME	CITY	STATUS
S1	Омега	Москва	50
S2	Истэн	Красноярск	25
S3	Рест	Красноярск	25
S4	Вектор	Омск	30

Рис. 13

Отношение B

S	SNAME	CITY	STATUS
S5	Крук и К°	Томск	35
S3	Рест	Красноярск	25
S7	ИВК	Омск	30

Рис.14

(A UNION B)

S	SNAME	CITY	STATUS
S1	Омега	Москва	50
S2	Истэн	Красноярск	25
S3	Рест	Красноярск	25
S4	Вектор	Омск	30
S5	Крук и К	Томск	35
S7	ИВК	Омск	30

Рис.15

(A INTERSECT B)

S	SNAME	CITY	STATUS
S3	Rest	Красноярск	25

Рис.16

(A MINUS B)

S	SNAME	CITY	STATUS
S1	Омега	Москва	50
S2	Истэн	Красноярск	25
S4	Вектор	Омск	30

Рис. 17

(B MINUS A)

S	SNAME	CITY	STATUS
S5	Крук и К	Томск	35
S7	ИВК	Омск	30

Рис.18

A

S
S1
S2

B

P
P1
P2
P3

A TIMES B

S	P
S1	P1
S1	P2
S1	P3
S2	P1
S2	P2
S2	P3

Рис. 19

Операция ограничения

Результатом ограничения отношения A по некоторому условию является отношение, имеющее тот же заголовок, что и отношение A, и включающее кортежи отношения A, удовлетворяющее этому условию, причем оператор условия должен иметь смысл. Операция ограничения унарна, т.е. применима только к

одному отношению. Для обозначения операции ограничения будем использовать конструкцию A WHERE comp, где A — ограничиваемое отношение, а comp — простое условие сравнения.

На рис. 15—17 приведено несколько примеров.

Пусть comp1 и comp2 — два простых условия ограничения. Тогда по определению:

A WHERE comp1 AND comp2 обозначает то же самое, что и
(A WHERE comp1) INTERSECT (A WHERE comp2)

A WHERE comp1 OR comp2 обозначает то же самое, что и
(A WHERE comp1) UNION (A WHERE comp2)

A WHERE NOT comp1 обозначает то же самое, что и
A MINUS (A WHERE comp1)

С использованием этих определений можно использовать операции ограничения, в которых условием ограничения является произвольное булевское выражение, составленное из простых условий с использованием логических связок AND, OR и AND и скобок. На интуитивном уровне операцию ограничения лучше всего представлять как взятие некоторой горизонтальной вырезки из отношения-операнда.

Операция проекции

При выполнении проекции отношения на заданный набор его атрибутов производится отношение, кортежи которого производятся путем взятия соответствующих значений из заданных столбцов кортежей отношения-операнда и последующим исключением дублирующих кортежей из того, что осталось.

Операция соединения

При естественном соединении двух отношений A и B (A JOIN B) по некоторому условию образуется результирующее отношение, кортежи которого являются конкатенацией (сцеплением) кортежей первого и второго отношений и удовлетворяют этому условию.

Операция деления

У операции реляционного деления два операнда — бинарное и унарное отношения. Результирующее отношение состоит из одноатрибутных кортежей, включающих значения первого атрибута кортежей первого операнда, таких, что множество значений второго атрибута (при фиксированном значении первого атрибута) совпадает со множеством значений второго операнда.

На рис. приведены примеры вышерассмотренных операций.

A WHERE CITY =’Красноярск’

S	SNAME	CITY	STATUS
S2	Истэн	Красноярск	25
S3	Рест	Красноярск	25

Рис. 20

A WHERE STATUS>=30

S	SNAME	CITY	STATUS
S1	Омега	Москва	50
S4	Вектор	Омск	30

Рис. 21

A WHERE STATUS<30 OR CITY=’Москва’

S	SNAME	CITY	STATUS
S1	Омега	Москва	50
S2	Истэн	Красноярск	25
S3	Рест	Красноярск	25

Рис. 22

Операция переименования

Операция переименования производит отношение, тело которого совпадает с телом операнда, но имена атрибутов изменены.

Операция присваивания

Операция присваивания позволяет сохранить результат вычисления реляционного выражения в существующем отношении БД. Поскольку результатом любой реляционной операции (кроме операции присваивания) является некоторое отношение, можно образовывать реляционные выражения, в которых вместо отношения-операнда некоторой реляционной операции находится вложенное реляционное выражение.

4.3 Замкнутость реляционной алгебры и операция переименования

Как мы говорили в предыдущей лекции, каждое отношение характеризуется схемой (или заголовком) и набором кортежей (или телом). Поэтому, если операции алгебры 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 будет действительно возможно строить вложенные выражения. Заголовок отношения представляет собой множество пар: имя-атрибута, имя-домена. Если посмотреть на общий обзор реляционных операций, приведенный в предыдущем разделе, то видно, что домены атрибутов результирующего отношения однозначно определяются доменами отношений-результатов. Однако с именами атрибутов результата не всегда бывает все так просто. Например, представим себе, что у отношений-операндов операции прямого произведения имеются одноименные атрибуты с одинаковыми доменами. Каким должен быть заголовок результирующего отношения? Поскольку заголовок — это множество, в нем не должны содержаться одинаковые элементы. Но и потерять атрибут в результате операции недопустимо. А это значит, что в этом случае вообще невозможно корректно выполнить операцию прямого произведения. Аналогичные проблемы могут возникать и в случаях других двуместных операций. Для их разрешения в состав операций реляционной алгебры вводится операция переименования. Ее следует применять в любом 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 безо всяких проблем. В дальнейшем изложении мы будем предполагать применение операции переименования во всех конфликтных случаях.

4.4 Особенности теоретико-множественных операций реляционной алгебры

Хотя в основе теоретико-множественной части реляционной алгебры лежит классическая теория множеств, соответствующие операции реляционной алгебры обладают некоторыми особенностями. Начнем с операции объединения (все, что будет говориться по поводу объединения, переносится на операции пересечения и взятия разности). 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 допустить в реляционной алгебре возможность теоретико-множественного объединения произвольных двух отношений (с разными схемами), то, конечно, результатом операции будет множество, состоящее из разнотипных кортежей, т.е. оно не является отношением. Если исходить из требования замкнутости реляционной алгебры относительно понятия отношения, то такая операция объединения является бессмысленной. Все эти соображения приводят к появлению понятия совместимости отношений по объединению: два отношения совместимы по объединению в том и только в том случае, когда обладают одинаковыми заголовками. Более точно, 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. Если два отношения совместимы по объединению, то при обычном выполнении над ними операций объединения, пересечения и взятия разности 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 со-

единения эти отношения можно сделать полностью совместимыми по объединению путем применения операции переименования. Заметим, что включение в состав операций реляционной алгебры трех операций объединения, пересечения и взятия разности является очевидно избыточным, поскольку известно, что любая из этих операций выражается через две других. Тем не менее Кодд в свое время решил включить все три операции, исходя из интуитивных потребностей потенциального пользователя системы реляционных БД, далекого от математики. Другие проблемы связаны с операцией взятия Декартова (или прямого) произведения двух отношений. В теории множеств прямое произведение может быть получено для любых двух множеств, и элементами результирующего множества являются пары, составленные из элементов первого и второго множеств. Поскольку отношения являются множествами, то и для любых двух отношений возможно получение прямого произведения. Но результат не будет отношением. Элементами результата будут являться не кортежи, а пары кортежей. Поэтому в реляционной алгебре используется специализированная форма операции взятия прямого произведения — расширенное прямое произведение отношений.

При взятии расширенного прямого произведения двух отношений элементом результирующего отношения является кортеж, являющийся конкатенацией (или слиянием) одного кортежа первого отношения и одного кортежа второго отношения. Но теперь возникает второй вопрос — как получить корректно сформированный заголовок отношения-результата? Очевидно, что проблемой может быть именование атрибутов результирующего отношения, если отношения-операнды обладают одноименными атрибутами. Эти соображения приводят к появлению понятия совместимости по взятию расширенного прямого произведения.

Два отношения совместимы по взятию прямого произведения в том и только в том случае, если множества имен атрибутов этих отношений не пересекаются. Любые два отношения могут быть сделаны совместимыми по взятию прямого произведения путем применения операции переименования к одному из этих отношений. Следует заметить, что операция взятия прямого произведения не является слишком осмысленной на практике. Во-первых, мощность ее результата очень велика даже при допусти-

мых мощностях операндов, а во-вторых, результат операции не более информативен, чем взятые в совокупности операнды. Как мы увидим ниже, основной смысл включения операции расширенного прямого произведения в состав реляционной алгебры состоит в том, что на ее основе определяется действительно полезная операция соединения. По поводу теоретико-множественных операций реляционной алгебры следует еще заметить, что все четыре операции являются ассоциативными. То есть если обозначить через OP любую из четырех операций, то

(A OP B) OP C = A OP (B OP C),

и, следовательно, без введения двусмысленности можно писать

A OP B OP C

(A, B и C — отношения, обладающие свойствами, требуемыми для корректного выполнения соответствующей операции). Все операции, кроме взятия разности, являются коммутативными, т.е. A OP B = B OP A.

Глава 5. ЯЗЫК SQL

Язык SQL является единственным стандартным реляционным языком и в настоящее время поддерживается практически всеми СУБД. SQL — это сокращенное название структурированного языка запросов (Structured Query Language). Язык SQL является языком программирования, который применяется для организации взаимодействия с базой данных. SQL реализует все функциональные возможности, которые СУБД предоставляет пользователю, а именно:

- Организацию данных. SQL дает пользователю возможность изменять структуру представления данных.
- Чтение данных. SQL дает пользователю или приложению возможность выбирать из базы данных содержащиеся в ней данные и пользоваться ими.
- Обработки данных. SQL дает возможность пользователю или приложению возможность изменять базу данных, т.е. добавлять в нее новые данные, а также удалять или обновлять уже имеющиеся в ней данные.
- Управление доступом. С помощью SQL можно ограничить возможности пользователя по чтению или изменению данных и защитить их от несанкционированного доступа.
- Совместное использование данных. SQL координирует совместное использование данных пользователями, работающими параллельно, так, чтобы они не мешали друг другу.
- Целостность данных. SQL позволяет обеспечить целостность базы данных.

SQL является декларативным языком, в нем нет операторов цикла или логических операторов, поэтому он обычно встраивается в какой-либо процедурный базовый язык (PL/1, ADA, COBOL, и др.).

По стандарту ANSI команды SQL объединены по группам. Эти группы команд SQL называются подразделами. Приведем список основных подразделов.

Язык определения данных (Data Definition Language, DDL). В эту группу входят команды, предназначенные для создания, модификации и удаления объектов баз данных, таких, как таблицы, представления, индексы и т.д. Команда CREATE создает объект базы, команда DROP — удаляет, а с помощью команды ALTER можно изменить объект. Приведем несколько наиболее используемых команд DDL для таких объектов, как таблицы, пользователи и триггеры:

- CREATE TABLE, DROP TABLE, ALTER TABLE;
- CREATE USER, DROP USER, ALTER USER;
- CREATE TRIGGER, DROP TRIGGER.

Язык манипулирования данными (Data Manipulation language, DML). Эта группа содержит команды, используемые для манипулирования данными в таблицах и представлениях. Проще говоря, с помощью команд DML выполняется выборка данных, вставка новых строк, изменение и удаление существующих.

К командам DML относятся следующие команды:

- SELECT — выбрать строки из таблиц;
- INSERT — добавить строки в таблицу;
- UPDATE — изменить строки в таблице;
- DELETE — удалить строки в таблице.

Команды управления транзакциями (Transaction Conrol Statement, TCS). Рассматриваемые команды используются совместно с командами DML и позволяют контролировать изменения данных. Приведем список команд управления транзакциями:

- SET TRANSACTION — начать транзакцию;
- SAVEPOINT — создать внутри транзакции контрольную точку (или, как ее еще называют, точку сохранения), далее в транзакции можно будет привести данные в состояние, в котором они были на момент определения контрольной точки;
- COMMIT — фиксация транзакции или контрольной точки, после чего данные не смогут быть восстановлены в состояние, в котором они были до начала транзакции, и все изменения, сделанные в теле транзакции, становятся доступными другим пользователям;

- ROLLBACK — отмена всех сделанных в теле транзакции изменений, данные восстанавливаются в состояние, в котором они были до начала транзакции.

К командам управления данными можно отнести команды предоставления доступа к данным различным группам пользователей. Если оператор использует таблицу не из схемы пользователя, тогда имя таблицы предваряется именем схемы с точкой, например Mai.Sotr

5.1 Создание таблиц

Для того чтобы изучить команду создания таблиц, рассмотрим сначала типы данных, используемых в таблицах. Каждая система использует свой набор типов данных, где-то они пересекаются. Поскольку в дальнейшем некоторые лабораторные и практические занятия будут выполняться в СУБД ORACLE, рассмотрим основные типы данных, используемых этой системой баз данных (параметры заключенные в квадратные скобки необязательны и могут быть опущены).

- Varchar2(N) — символьный тип переменной длины, может содержать последовательность N символов (N может быть задано от 1 до 2000);
- Char[(N)] — символьный тип фиксированной длины (N может быть от 1 до 255), если параметр не задан, по умолчанию принимается один символ. Отличается от типа Varchar2 тем, что дополняется до длины N пробелами;
- Number[(N1[,N2])] — числовой тип, значение N1 задает разрядность числа (до 38 разрядов), N2 (необязательный параметр) — точность, задает количество разрядов после точки (значение N2 от –84 до 127). Может иметь отрицательное значение, в этом случае округление идет в обратную сторону. Если параметр N2 опущен, считается, что он равен 0 (в этом случае задан целый тип). Примеры задания этого типа для числа 123.4567:
  - Number — число равно 123.4567;

- Number(8,3) — число равно 123.457 (осуществляется округление до заданной точности);
- Number(8) — число равно 123;
- Number(8,-2) — число равно 100 (округление до сотен);

- Date — тип Дата, может хранить информацию о дате и времени;
- Long — Символьные данные переменной длины вплоть до 2 гигабайт;
- Raw(N) — Двоичные данные переменной длины до 255 байт;
- Long Raw — Двоичные данные переменной длины до 2 гигабайт;

В общем виде команда создания таблиц имеет вид (в приведенной команде и последующих командах символ «|» означает один их двух вариантов выбора, параметры заключенные в квадратные скобки необязательны (могут присутствовать и отсутствовать в команде), при написании выражений в команде угловые скобки не пишутся):

```
CREATE TABLE имя_таблицы
      (определение столбца 1[, определение столбца 2
       [,определение столбца 3, …]]
       [,ограничения таблицы]);
```

определение столбца:

имя_столбца тип столбца [null|not null] [значение по умолчанию] [ограничение_1 столбца[ограничение_2 столбца…]]

null или not null — допустимость или недопустимость пустых значений в поле (по умолчанию (при отсутствии этого параметра) пустые значения допускаются).

Значение по умолчанию:

DEFAULT <выражение> | :=<выражение>
(тип выражения должен совпадать с типом поля).

Ограничения:

[Constraint имя_ограничения] тип ограничения

Типы ограничений:

- Primary Key — задание первичного ключа (первичный ключ в таблице может быть только один);
- References <ссылка на родительскую таблицу> — задание внешнего ключа дочерней таблицы для поддержки ссылочной целостности и связи таблиц;
- Unique — задание уникального поля, обладает всеми свойствами первичного ключа, если для поля установлено свойство Not Null. Вторичных ключей в таблице, в отличие от первичного ключа, может быть несколько;
- Check(<логическое выражение>) — задание условий проверок при вводе значений или при их модификации. Значения, входящие в логическое выражения не могут динамически меняться и фиксируются при создании таблицы. Например, если в выражении стоит функция, возвращающая системную дату, функция возвратит текущее значение даты и оно будет зафиксировано в выражении проверки.

Если имя ограничения при создании таблицы не будет задано, система сама сгенерирует имя ограничения. При нарушении ограничения при вводе строки или модификации данных будет выведено сообщение о нарушении ограничения и строка или не будет введена или игнорируется модификация.

Ограничения декларативно поддерживают относительно простую целостность данных, более сложные правила поддержки целостности данных реализуются в триггерах, которые рассматриваются в главе PL/SQL.

В ограничениях таблицы обычно ставят ограничения (через запятую, а не через пробел, как в ограничениях полей), если они должны относиться к нескольким полям (например, если первичный или внешний ключи составные или ограничение check должно быть применимо к нескольким полям). Можно в ограничении таблицы использовать ограничения и для одного поля, только это менее наглядно. К полям типа Long и Long Raw ограничения не-

применимы. При задании внешнего ключа в ограничении таблицы ограничение имеет формат:

FOREIGN KEY <выражение внешнего ключа> REFERENCES <ссылка на родительскую таблицу>

Таким образом, команда создания таблицы имеет вид:

```
CREATE TABLE имя_таблицы
  (
имя_поля1  тип  [(ширина поля[,точность])]
      [NULL | NOT NULL]
      [CHECK <логическое _выражение>]
      [DEFAULT <выражение>]
      [PRIMARY KEY | UNIQUE]
      [REFERENCES <ссылка_на_родительскую_таблицу>]
      [, определение_поля2… ]
      [, PRIMARY KEY <выражение>]
      |, UNIQUE <выражение>]
      [, FOREIGN KEY <выражение>
            REFERENCES <имя родительской таблицы>]
      [, CHECK(<логическое выражение>)]
)
```

Для примера создадим таблицы для хранения данных отделов и сотрудников. Зададим ограничения на данные вводимых таблиц. В таблице отделов будем хранить номера отделов (номер отдела уникален и может быть выбран в качестве первичного ключа), номера отделов могут быть в диапазоне от 100 до 120 с исключением номеров 110,115. Также в этой таблице будем хранить название отдела, номер телефона отдела и этаж, на котором находится отдел (номера этажей от 1 до 9). Определим ограничения на поля таблицы сотрудников:

- номер сотрудника (первичный ключ, номер сотрудника начинается с цифры 1000);
- Фамилия сотрудника (не может быть пустым);
- Зарплата (не может быть меньше 0 и больше 30000, по умолчанию заносится нулевое значение);
- Адрес (по умолчанию Кирова 12-22);

- Номер отдела, в котором работает сотрудник (внешний ключ). Считаем, что в таблице данные только сотрудников, работающих в отделах, поэтому внешний ключ не может быть пустым;
- Номер паспорта (уникальный).
- Дата рождения.

Дополнительное ограничение (оно искусственно, но позволяет показать пример ограничения таблицы): в одном отделе не могут работать однофамильцы

```
CREATE TABLE OTDEL
(Nom_otd number(6) constraint pk1 primary key
  Constraint nm_otd check(nom_otd>100 and nom_otd<120
and nom_otd<>110 and nom_otd<>115),
name  Varchar2(25) not null,
nom_tel number(6),
etaj number(1) not null Constraint nom_etaj check(etaj>0 and etaj<10)
)

CREATE TABLE SOTR
(Nom_sotr number(8) Constraint pk_sotr primary key
Constraint n_s  check(nom_sotr>1000),
Famil varchar(15) not null,
Zarpl number(5) DEFAULT 0
Constraint  val_zarpl check(zarpl>=0 and zarpl<=30000),
Adres varchar2(25) DEFAULT ‘Кирова 12-22’,
Nom_otd number(6) not null Constraint fk_sotr references otdel,
Nom_pasp number (10) Constraint uniq_pasp unique,
Dat_rojd date,
Constraint uniq_fam_otd Unique Famil,nom_otd
)
```

После создания таблицы в нее можно добавлять записи (строки), модифицировать их или удалять.

5.2 Выборка данных

Рассмотрим теперь выборку из таблиц. Операция выборки в SQL — это, по существу, табличное выражение, которое может

быть сколько угодно сложным. В общем виде сокращенная структура оператора SELECT:

```
SELECT [ ALL|DISTINCT ] <что выводится>  FROM <откуда>
   [WHERE <условие отбора строк>]
     [GROUP BY <список группировки>]
       [HAVING <условие отбора групп>]
         [ORDER BY <список сортировки>  [ASC|DESC] ]
```

Если стоит необязательный параметр ALL (который действует по умолчанию), тогда выводятся повторяющиеся строки (если они есть), параметр DISTINCT подавляет вывод повторяющихся строк. В предложении <что выводится> указывается список столбцов, которые должны быть возвращены оператором SELECT. Возвращаемые столбцы могут содержать значения, считываемые из столбцов таблиц базы данных, или значения, вычисляемые во время выполнения запроса, в том числе встроенные и агрегированные функции SQL. Можно указать символ « * «, который указывает, что выводится все поля. После имени поля или выражения через пробел можно указать псевдоним, которые может использоваться в опциях оператора. В предложении FROM указывается список таблиц или представлений, считываемые запросом. После имени таблицы (представления) можно также указать псевдоним, использование которого весьма удобно при длинном имени таблицы (особенно при удаленной базе или осуществляется выборка из таблицы другой схемы).

Предложение WHERE показывает, что в результат запроса включаются только некоторые строки, если условие истинно (условие должно возвращать логическое значение). Если условие ложно или равно NULL, соответствующие строки не выводятся. Если при выборке из нескольких таблиц не указать условия связи строк, результирующая таблица имеет кардинальное число, равное произведению кардинальных чисел таблиц-источников (как правило, если осуществляется выборка из нескольких таблиц, должна присутствовать опция WHERE). Остальные опции рассмотрим позже. Для примера рассмотрим две таблицы (таблица Otdel, содержит сведения об отделах (номер, название, номер телефона и этаж, на котором расположен отдел) и таблица Sotr

(номер сотрудника, фамилия, зарплата, адрес, номер отдела, в котором работает сотрудник, номер паспорта и дата рождения), содержит сведения о сотрудниках):

Otdel(nom_otd, name, nom_tel, etaj)

с первичным ключом nom_otd

Sotr(nom_sotr, famil, zarpl, adres, nom_otd, nom_pasp, dat_rojd)

с первичным ключом nom_sotr и с внешним ключом nom_otd

Рассмотрим простые однотабличные запросы:

- Вывести всю таблицу сотрудников.

```
SELECT * FROM Sotr
```

- Вывести фамилии, зарплату и адрес сотрудников.

```
SELECT famil,zarpl,adres FROM Sotr
```

- Вывести данные сотрудников, зарплата которых больше 5000.

```
SELECT * FROM Sotr WHERE zarpl>5000
```

- Вывести номер сотрудника и его зарплату в тысячах рублей.

```
SELECT nom_sotr, zarpl/1000 FROM Sotr
```

- Вывести данные сотрудников с фамилией Иванов, зарплата которых больше 5000.

```
SELECT * FROM Sotr
    WHERE zarpl>5000 and famil='Иванов'
```

• Вывести данные сотрудников с пустым значением адреса.

```
SELECT * FROM Sotr WHERE adres is NULL
```

• Вывести данные сотрудников с не пустым значением адреса.

```
SELECT * FROM Sotr WHERE adres is not NULL
```

Если в опции WHERE использовать выражение adres=NULL, это выражение вернет значение NULL и не будет выведено ни одной строки (любое сравнение с пустым значением возвратит пустое значение NULL), поэтому используется специальная конструкция is для определения пустого или не пустого значения.

Рассмотрим логические выражения, которые можно использовать в выражении WHERE (значение A и выражения должны быть одного типа):

♦ **A between N1 and N2** — диапазон значений (для числового типа и типа дата) возвращает истину, если A находится в диапазоне от N1 до N2, например 2500 BETWEEN 2000 and 5000 возвратит значение TRUE, а 2500 BETWEEN 3000 and 4000 возвратит значение FALSE;

♦ **A in (список выражений)** — вхождение во множество, возвратит TRUE, если A находится в списке и FALSE в противном случае. 5 in (22,6,5,12,11,17)=TRUE, 6 in (66,44,11,3,4,10)=FALSE.

♦ **A like <маска>** — соответствие маске (значение A и маска должны быть символьного типа), возвращает TRUE, если A соответствует маске, в маске можно использовать символы «%» для определения любого набора символов и символ «_» для определения любого одиночного символа. Выражение `A like 'Ив%'` возвратит TRUE для всех A, значения которых начинаются на `'Ив'`, например, если A имеет значение ‘Иванов’, или ’Иванихин’, или ’Иванченко’ и т.д.

• Вывести данные сотрудников из отделов 103, 105, 110, 111, 112, 120. Использование логического выражения с операци-

ей логического сложения OR было бы достаточно длинным. Гораздо короче такое выражение:

```
SELECT * FROM Sotr
      WHERE nom_otd in (103,105,110,111,112,120)
```

- Вывести неповторяющиеся фамилии сотрудников.

```
SELECT DISTINCT Famil FROM Sotr
```

Здесь мы включили опцию DISTINCT для подавления вывода повторяющихся строк, поскольку среди сотрудников имеются однофамильцы.

Группировка (GROUP BY) обычно используется с агрегированными функциями:

- SUM(поле) — возвращает суммарное значение поля;
- MAX(поле) — возвращает максимальное значение поля;
- MIN(поле) — возвращает минимальное значение поля;
- AVG(поле) — возвращает среднее значение поля;
- COUNT(поле) — возвращает количество строк с непустыми значениями поля;
- COUNT(*) — возвращает общее количество строк.

Замечание: агрегированные функции нельзя включать в опцию WHERE.

Если агрегированные функции используются без опции GROUP BY, команда SELECT возвращает только одну строку (в данном случае выводить совместно с функцией значения полей нельзя).

- Вывести суммарное, среднее значения зарплаты сотрудников и максимальный номер сотрудника:

```
SELECT SUM(Zarpl),AVG(Zarpl),MAX(Nom_sotr) FROM Sotr
```

• Вывести количество строк с пустыми и непустыми значениями адресов и их относительное значение:

```
SELECT COUNT(Adres),COUNT(*), COUNT(Adres)/COUNT(*)
   FROM Sotr
```

• Сколько сотрудников с фамилией, заканчивающейся на 'ин', имеют пустое значение адреса?

```
SELECT COUNT(*) FROM Sotr WHERE Famil like '%ин' and
Adres is NULL
```

• Вывести количество сотрудников в отделе 105.

```
SELECT COUNT(*) FROM Sotr WHERE Nom_otd=105
```

Группировка позволяет вычислять значения агрегированных функций для значений, определяемых группировкой. Сначала таблица группируется по значению группировки и затем для каждой группировки вычисляются значения. Выражение группировки должно включаться в вывод, иначе выводимое табличное значение теряет смысл.

• Вывести суммарное и среднее значения зарплаты отделов.

```
SELECT SUM(Zarpl),AVG(Zarpl),Nom_otd
                    FROM Sotr GROUP BY Nom_otd
```

• Вывести количество сотрудников в отделах.

```
SELECT COUNT(*),Nom_otd FROM Sotr GROUP BY Nom_otd
```

Опция HAVING позволяет отбирать вывод по условиям:

• Вывести количество сотрудников в отделах и суммарные зарплаты отделов, если количество сотрудников в отделах больше 30.

```
SELECT COUNT(*), SUM(Zarpl), Nom_otd FROM Sotr
        GROUP BY Nom_otd HAVING COUNT(*)>30
```

Опция ORDER BY сортирует выводимую таблицу по какому-либо параметру.

• Вывести фамилии, зарплату и номера сотрудников с сортировкой по фамилии.

```
SELECT Famil, Zarpl, Nom_Sotr FROM Sotr ORDER BY Famil
```

В выборке

```
SELECT Famil, Zarpl, Nom_otd FROM Sotr
       ORDER BY Famil, Zarpl
```

сортировка вначале осуществляется по фамилии, затем по зарплате (вывод сотрудников с одинаковой фамилией осуществляется с сортировкой по зарплате). Параметр ASC (действует по умолчанию) задает сортировку по возрастанию, параметр DESC — по убыванию.

Более сложные запросы могут включать запросы из двух и более таблиц, при выборке из связанных таблиц нужно задать условие связи таблиц, в котором первичный ключ родительской таблицы сравнивается с внешним ключом дочерней таблицы. Когда осуществляется выборка из двух таблиц, происходит декартово произведение этих таблиц (сочетание всевозможных сцеплений строк во всех комбинациях). Нельзя использовать простые имена полей, если в многотабличной выборке существуют одинаковые имена полей, поскольку появляется неоднозначность, к какой таблице относить имя поля. Для уточнения имени поля в этом случае необходимо использовать полное имя поля, т.е. использовать точечную нотацию (предварять имя поля именем таблицы с точкой), например Sotr.nom_otd или Otdel.nom_otd. Полное имя можно использовать, даже если в этом нет необходимости. Команда

```
SELECT * FROM Otdel, Sotr
```

возвратит набор данных, состоящий из 11-ти столбцов и количеством строк равный произведению строк таблиц. Когда строка отдела сцепляется со строкой сотрудника, который в этом отделе не работает, в большинстве случаев такое сцепление лишено смысла. Поэтому набор данных нужно ограничить (равенством первичных и внешних ключей):

```
SELECT * FROM Otdel, Sotr WHERE Ot-
    del.nom_otd=Sotr.nom_otd
```

Если таблицы находятся в схеме P_Ivanov имена таблиц нужно предварять именем схемы:

```
SELECT * FROM P_Ivanov.Otdel, P_Ivanov.Sotr WHERE Ot-
            del.nom_otd=Sotr.nom_otd
```

и то же самое с использованием псевдонимов:

```
SELECT * FROM P_Ivanov.Otdel Q, P_Ivanov.Sotr S
          WHERE Q.nom_otd=S.nom_otd
```

- Вывести имена, номера, фамилии и зарплату сотрудников и имена отделов, в которых работают сотрудники, получающие зарплату больше 5000.

```
SELECT Nom_sotr,Famil,Zarpl,Name
          FROM Otdel, Sotr
WHERE Otdel.nom_otd=Sotr.nom_otd and Zarpl>5000
```

Подзапросы: первичное число строк при многотабличных выборках может получиться очень большим (предложение WHERE работает после декартова умножения). Например, при выборке из четырех таблиц, содержащих 20, 1000, 2000 и 50 строк (из таблиц отделов, сотрудников, детей и школ) получается число строк, равное 2 млрд. Мало того, что время выборки будет достаточно большим, результаты выборки занимают в этом случае значительную память компьютера. Для оптимизации выборки в правой части предложения WHERE можно использовать вы-

борку из какой-либо таблицы (оператор выборки в операции сравнения всегда должен стоять в правой части). Если подзапрос возвращает несколько значений, нельзя использовать простое сравнение (больше, меньше, равно и т.п.).

• Выбрать сотрудников, зарплата которых больше средней зарплаты сотрудников.

```
SELECT * FROM Sotr
     WHERE Zarpl>(SELECT AVG(Zarpl) FROM Sotr)
```

• Выбрать сотрудников, работающих на 5 этаже.

```
SELECT * FROM Sotr
      WHERE nom_otd in (SELECT nom_otd FROM OTDEL
                                       WHERE etaj=5)
```

5.3 Изменение данных

Оператор INSERT позволяет вставлять в таблицу новую строку или строки. Однострочный оператор имеет структуру:

```
   INSERT INTO имя_таблицы [(список полей)]
            VALUES(список выражений)
```

Если не указан список полей, тогда в списке выражений должны быть значения всех полей (могут быть значения NULL, если поле это допускает), причем типы значений выражения и столбцов должны совпадать. Если в списке полей не указано имя поля, тогда при вставке значение поля получает значение NULL (или значение по умолчанию, если оно указано при создании таблицы).

Вставим строку в таблицу Sotr:

```
   INSERT INTO Sotr (nom_sotr, famil, zarpl, adres,
nom_otd, nom_pasp, dat_rojd)
  VALUES(1354,'Сергеев',2550,'Киевская 115-27',
111,2343345653,{^12.02.1965})
```

Если бы мы пытались ввести однофамильца в отдел, была бы выдана ошибка, поскольку в примере создания таблицы сотрудников мы наложили ограничение (искусственное), что в одном отделе не могут быть однофамильцы. Если бы из списка исключили бы дату рождения, в это поле было бы введено значение NULL, а если бы исключили адрес, было бы внесено значение по умолчанию ‘Кирова 12-22’.

В многострочном операторе источником новых строк служит оператор SELECT. Например, для включения строк в таблицу Sotr строк из таблицы Sotr1 с такими же полями, как в таблице Sotr:

- Скопировать в таблицу Sotr строки из таблицы Sotr1, где сотрудники получают зарплату>1500

```
INSERT INTO Sotr
   SELECT * FROM Sotr1 WHERE Zarpl>1500
```

Оператор удаления строк имеет структуру:

```
DELETE FROM имя_таблицы
  [WHERE условие_выбора_строк>]
```

Здесь предложение WHERE может отсутствовать, тогда удаляются все строки. В предложении WHERE может быть использован подзапрос. Удаление строк в таблице Sotr, где сотрудники получают зарплату больше 2500, формулируется следующим образом:

```
DELETE FROM Sotr WHERE Zarpl>2500
```

- Удалить сотрудников, работающих на 5-м этаже.

```
DELETE FROM Sotr WHERE nom_otd in
   (SELECT nom_otd FROM Otdel WHERE etaj=5)
```

Оператор изменения данных UPDATE позволяет изменять данные в столбцах и имеет структуру:

```
    UPDATE имя_таблицы SET имя_столбца1=выражение1)
[, имя_столбца2=выражение2…]
     [WHERE условие_выбора_строк> ]
```

типы столбцов и выражений должны совпадать (или допускать неявное преобразование). Если опущено предложение WHERE, изменяются все строки. В предложении WHERE может быть использован подзапрос.

Рассмотрим несколько примеров изменения данных:

- Увеличить на 15 % зарплату сотрудникам, зарплата которых меньше 1200

```
UPDATE Sotr SET Zarpl=Zarpl*1.15
  WHERE Zarpl<1200
```

- Перевести сотрудников из отдела 110 в отдел 111

```
UPDATE Sotr SET nom_otd=111 WHERE nom_otd=110
```

С операторами изменения данных тесно связано понятие **транзакции** в клиент-серверных СУБД, которая является логической единицей работы с базой данных. Когда пользователь изменяет данные (вставляет, модифицирует или удаляет строки), другие пользователи заблокированы от изменений, пока пользователь, изменивший данные, не завершит транзакцию. Команда **COMMIT** подтверждает транзакцию, данные фиксируются в базе данных и разблокируются для изменений другим пользователям. Точки сохранения используются в сочетании с командой ROLLBACK, чтобы отменять порции текущей транзакции и имеет вид **SAVEPOINT имя_точки_сохранения**. Точки сохранения полезны в интерактивных программах, потому что вы можете создавать промежуточные шаги такой программы и давать этим шагам имена. Это обеспечивает большую степень контроля над длинными, более сложными программами. Например, вы можете использовать точки сохранения вдоль длинного сложного ряда

обновлений, так что, если вы сделаете ошибку, вам не понадобится перезапускать каждое предложение. Имена точек сохранения должны быть различными внутри данной транзакции. Если вы создаете в одной транзакции вторую точку сохранения с таким же идентификатором, как и у более ранней точки сохранения, то старая точка сохранения стирается. После того как точка сохранения создана, вы можете продолжать работу, подтвердить вашу транзакцию, выполнить откат всей транзакции либо откат к точке сохранения. Команда **ROLLBACK [точка_сохранения]** отменяет все изменения, выполненные пользователем после начала транзакции (или до точки сохранения, если она указана в транзакции). Рассмотрим пример: в системе банка есть две таблицы ведения счетов, один накопительный в таблице Nakop(nom_klient,schet) и второй текущий в таблице Tek(nom_klient,schet), где поля в таблицах nom_klient — номер клиента, а schet — сумма счета. Нужно перевести 5000 р. для клиента с номером 1234 с накопительного счета на текущий счет. В этом случае понадобятся две команды модификации данных.

```
UPDATE Nakop SET Schet=Schet-5000 WHERE nom_klient=1234;
UPDATE Tek     SET Schet=Schet+5000 WHERE nom_klient=1234;
```

Эти команды логически связаны друг с другом, поскольку, если одна из них выполнится, а другая нет, нарушится целостность данных. Таким образом, должны выполниться либо обе команды, либо не выполниться ни одной. Команды которые должны выполняться вместе, может быть и больше. Например, если имеется таблица ведения счетов Ved_schet(nom_klient, vid, table1, summa, date_oper), где vid — поле вида операции, table1 — имя таблицы, где осуществляется операция, summa — сумма и date_oper — дата операции, тогда к этим двум командам должны добавиться еще две (функция DATE() возвращает системную дату):

```
INSERT INTO Ved_schet values(1234,’снятие’,’Nakop’,5000,date())
INSERT INTO Ved_schet values(1234,’добавление’,’Tek’,5000,date())
```

Таким образом, должны выполниться уже 4 команды либо не выполниться ни одной, т.к. невыполнение одной из команд приведет к потере целостности данных.

Транзакцией называется совокупность логически связанных последовательных операций изменения данных, до выполнения и после выполнения которых база данных должна находиться в целостном состоянии и которые должны выполниться как единое целое. Таким образом транзакция является логической единицей работы с базой данных.

Свойства транзакции:

- Атомарность — должны выполниться либо все команды в транзакции, либо не выполниться ни одной.
- Согласованность — после транзакции база должна находиться в целостном состоянии.
- Изолированность — изменения данных, выполняемые в одной транзакции, не должны зависеть от изменений, выполняемых другой транзакцией.
- Устойчивость — после того как транзакция завершена, она сохраняется в базе данных.

Транзакция должна содержать минимальный набор команд. Если операторы не влияют на целостность данных, они должны быть вынесены за пределы транзакции.

5.4 Представления

В базе данных могут быть особые объекты — представления, которые еще называются «виртуальными» таблицами и формируют данные табличного вида на основе оператора SELECT. Сами представления данных не содержат, а делают выборку из одной или нескольких таблиц «на лету», в момент обращения к представлениям. Подобно реальным таблицам представления содержат строки и именованные столбцы и могут использоваться в операторах выборки и изменения данных как обычные таблицы. Обычно представления используются для следующих целей:

- Для группировки столбцов из разных таблиц в виде одного объекта. Оператор SELECT представления может быть очень сложен, но при обращении достаточно указать имя представления.
- Для ограничения доступа пользователей к определенным строкам таблицы (вертикальная фильтрация). Например, таблица сотрудников содержит сведения обо всех сотрудниках, но со-

труднику разрешается просматривать данные, относящиеся только к нему одному.

- Для ограничения доступа пользователей к определенным столбцам таблицы (горизонтальная фильтрация). Например, начальникам отделов можно просматривать все данные сотрудников других отделов, кроме их зарплаты или любой другой конфиденциальной информации. Когда пользователю дается привилегия на выборку таблицы, он может просмотреть всю таблицу. Можно создать представление с сокрытием той или иной информации и дать пользователю привилегию на выборку из этого представления.
- Для просмотра информации, получающейся в результате преобразования данных столбцов. Например, с помощью представлений можно просмотреть сумму значений столбца, минимальное или максимальное значение столбца без отображения всех данных этого столбца. Также представление может содержать агрегированную информацию, получаемую в результате сложных вычислений.

Можно изменять данные в представлениях, и это изменение вызовет изменения в таблицах, но только в том случае, если данные в представлении можно сопоставить данным таблицы. Например, нельзя модифицировать представление, если в представлении есть агрегированные функции или выражения, или же запрос осуществляется из нескольких таблиц.

Дадим вид команды:

```
CREATE VIEW [(список имен столбцов представления)]
AS оператор_SELECT
```

Если список столбцов представления опущен, тогда имена столбцов представления приобретают имена столбцов таблиц оператора SELECT. Список столбцов обязательно нужно задавать, если в операторе SELECT присутствуют выражения или агрегированные функции или же имеются столбцы с одинаковыми именами при отсутствии псевдонимов столбцов.

Создадим представление с именем Sotr_Otd_Zarpl, которое будет извлекать информацию о названиях отделов, номерах сотрудниках и их зарплатах и фамилиях.

```
CREATE VIEW Sotr_Otd_Zarpl AS
    SELECT Name, Nom_sotr, Zarpl, Famil FROM Sotr, Otdel
                WHERE Otdel.nom_otd=Sotr.nom_otd
```

Представление которое сотруднику выдает только его данные, будет иметь вид (при условии, что имя сотрудника при подключении к базе будет состоять из символов ‘SOTR’ и его номера. Встроенная функция USER возвращает имя пользователя):

```
CREATE VIEW Sotr_Data_Privat AS
  SELECT * FROM Sotr WHERE User=’SOTR’||nom_sotr
```

Следующее представление не позволит начальникам отделов просмотреть зарплаты сотрудников

```
CREATE VIEW Sotr_Data AS
SELECT nom_sotr, famil, adres, nom_otd, nom_pasp,
dat_rojd FROM Sotr
```

И наконец, представление, которое будет выдавать информацию о номерах отделов и их суммарных и средних зарплатах, будет иметь вид:

```
CREATE VIEW Otd_Zarpl(nom_otd,sum_zarpl,sredn_zarpl)
AS SELECT nom_otd,sum(zarpl),avg(zarpl) FROM Sotr
GROUP BY nom_otd
```

Теперь, выполнив команду

```
SELECT * FROM Otd_Zarpl
```

получим в табличном виде суммарную и среднюю зарплату отделов.

Глава 6. СЕРВЕР БАЗ ДАННЫХ

Современная СУБД должна удовлетворять ряду требований, важнейшее из которых — высокопроизводительный интеллектуальный сервер базы данных. Далее мы рассмотрим основные тенденции его развития и обсудим конкретные механизмы, в которых они находят свое воплощение.

Процесс технического совершенствования сервера базы данных пока остается невидимым для большинства пользователей современных СУБД. Поэтому при выборе той или иной системы они, как правило, не учитывают ни технический уровень решений, заложенных в механизм его функционирования, ни влияние этих решений на общую производительность СУБД. Между тем ее качество определяется отнюдь не богатством интерфейсов с пользователем, не разнообразием средств поддержки разработок, а в первую очередь зависит от особенностей архитектуры сервера базы данных. Далее будут рассмотрены модели технологии клиент-сервер, определено место сервера БД в этих моделях.

Клиент-сервер — это модель взаимодействия компьютеров в сети. Как правило, компьютеры не являются равноправными. Каждый из них имеет свое, отличное от других, назначение, играет свою роль. Некоторые компьютеры в сети владеют и распоряжаются информационно-вычислительными ресурсами, такими, как процессоры, файловая система, почтовая служба, служба печати, база данных. Другие же компьютеры имеют возможность обращаться к этим службам, пользуясь услугами первых. Компьютер, управляющий тем или иным ресурсом, принято называть сервером этого ресурса, а компьютер, желающий им воспользоваться — клиентом. Конкретный сервер определяется видом ресурса, которым он владеет. Так, если ресурсом являются базы данных, то речь идет о сервере баз данных, назначение которого — обслуживать запросы клиентов, связанные с обработкой данных; если ресурс — это файловая система, то говорят о файловом сервере, или файл-сервере, и т.д.

В сети один и тот же компьютер может выполнять роль как клиента, так и сервера. Например, в информационной системе, включающей персональные компьютеры, большую ЭВМ и мини-компьютер под управлением UNIX, последний может выступать как в качестве сервера базы данных, обслуживая запросы от клиентов — персональных компьютеров, так и в качестве клиента, направляя запросы большой ЭВМ.

Этот же принцип распространяется и на взаимодействие программ. Если одна из них выполняет некоторые функции, предоставляя другим соответствующий набор услуг, то такая программа выступает в качестве сервера. Программы, которые пользуются этими услугами, принято называть клиентами. Так, ядро реляционной SQL-ориентированной СУБД часто называют сервером базы данных, или SQL-сервером, а программу, обращающуюся к нему за услугами по обработке данных, — SQL-клиентом.

Первоначально СУБД имели централизованную архитектуру (рис. 23). В ней сама СУБД и прикладные программы, которые работали с базами данных, функционировали на центральном компьютере (большая ЭВМ или мини-компьютер). Там же располагались базы данных. К центральному компьютеру были подключены терминалы, выступавшие в качестве рабочих мест пользователей. Все процессы, связанные с обработкой данных, как то: поддержка ввода, осуществляемого пользователем, формирование, оптимизация и выполнение запросов, обмен с устройствами внешней памяти и т.д., выполнялись на центральном компьютере, что предъявляло жесткие требования к его производительности. Особенности СУБД первого поколения напрямую связаны с архитектурой систем больших ЭВМ и мини-компьютеров и адекватно отражают все их преимущества и недостатки. Однако нас больше интересует современное состояние многопользовательских СУБД, для которых архитектура клиент-сервер стала фактическим стандартом.

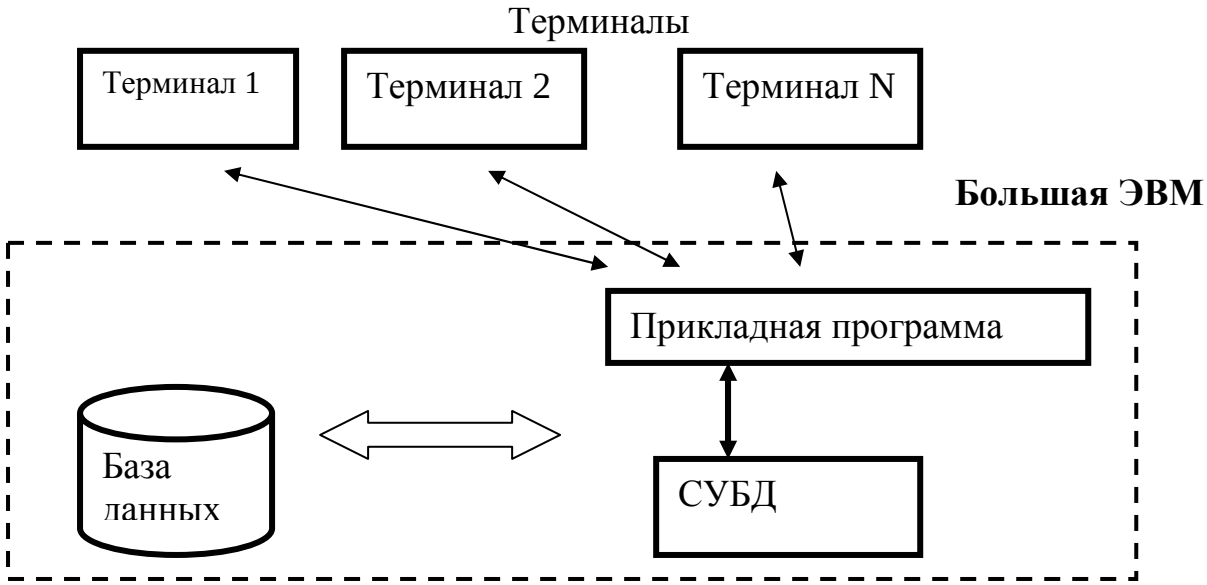


Рис. 23

Если предполагается, что проектируемая информационная система (ИС) будет иметь технологию клиент-сервер, то это означает, что прикладные программы, реализованные в ее рамках, будут иметь распределенный характер. Иными словами, часть функций прикладной программы (или, проще, приложения) будет реализована в программе-клиенте, другая — в программе-сервере, причем для их взаимодействия будет определен некоторый протокол.

Основной принцип технологии клиент-сервер заключается в разделении функций стандартного интерактивного приложения на четыре группы, имеющие различную природу.

Первая группа объединяет функции ввода и отображения данных.

Вторая группа объединяет чисто прикладные функции, характерные для данной предметной области (например, для банковской системы — открытие счета, перевод денег с одного счета на другой и т.д.).

К третьей группе относятся фундаментальные функции хранения и управления информационными ресурсами (базами данных, файловыми системами и т.д.).

К четвертой группе относятся служебные функции (играющие роль связок между функциями первых трех групп).

В соответствии с этим в любом приложении выделяются следующие логические компоненты:

- компонент представления, реализующий функции первой группы;
- прикладной компонент, поддерживающий функции второй группы;
- компонент доступа к информационным ресурсам, поддерживающий функции третьей группы, а также вводятся и уточняются соглашения о способах их взаимодействия (протокол взаимодействия).

Различия в реализациях технологии клиент-сервер определяются четырьмя факторами.

- Во-первых, тем, в какие виды программного обеспечения интегрированы каждый из этих компонентов.
- Во-вторых, тем, какие механизмы программного обеспечения используются для реализации функций всех трех групп.
- В-третьих, тем, как логические компоненты распределяются между компьютерами в сети.
- В-четвертых, тем, какие механизмы используются для связи компонентов между собой.

Выделяются четыре подхода, реализованные в моделях:

- модель файлового сервера (File Server — FS);
- модель доступа к удаленным данным (Remote Data Access — RDA);
- модель сервера базы данных (DataBase Server — DBS);
- модель сервера приложений (Application Server — AS).

FS-модель на рис. 24 является базовой для локальных сетей персональных компьютеров. Один из компьютеров в сети считается файловым сервером и предоставляет услуги по обработке файлов другим компьютерам. Файловый сервер работает под управлением сетевой операционной системы (например, Novell NetWare) и играет роль компонента доступа к информационным ресурсам (то есть к файлам). На других компьютерах в сети

функционирует приложение, в кодах которого совмещены компонент представления и прикладной компонент (рис. 19). Протокол обмена представляет собой набор низкоуровневых вызовов, обеспечивающих приложению доступ к файловой системе на файл-сервере.

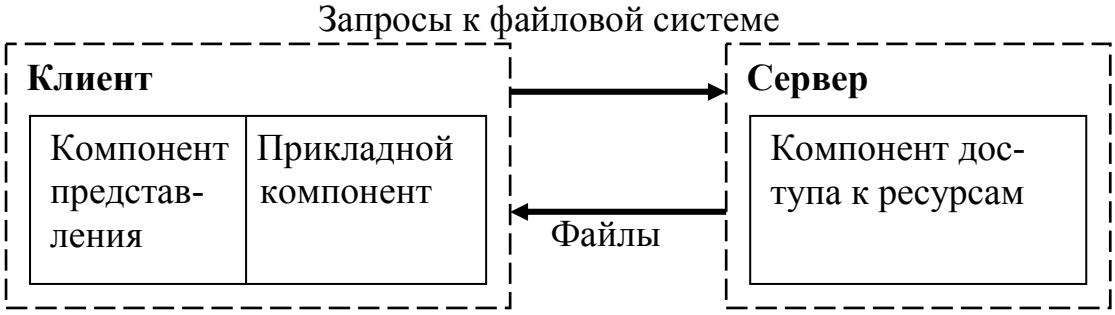


Рис. 24

FS-модель послужила фундаментом для расширения возможностей персональных СУБД в направлении поддержки многопользовательского режима. В таких системах на нескольких персональных компьютерах выполняется как прикладная программа, так и копия СУБД, а базы данных содержатся в разделяемых файлах, которые находятся на файловом сервере. Когда прикладная программа обращается к базе данных, СУБД направляет запрос на файловый сервер. В этом запросе указаны файлы, где находятся запрашиваемые данные. В ответ на запрос файловый сервер направляет по сети требуемый блок данных. СУБД, получив его, выполняет над данными действия, которые были декларированы в прикладной программе.

К технологическим недостаткам модели относят высокий сетевой трафик (передача множества файлов, необходимых приложению), узкий спектр операций манипуляции с данными (данные — это файлы), отсутствие адекватных средств безопасности доступа к данным (защита только на уровне файловой системы) и т.д. Собственно, перечисленное не есть недостатки, но — следствие внутренне присущих FS-модели ограничений, определяемых ее характером.

Более технологичная RDA-модель существенно отличается от FS-модели характером компонента доступа к информационным ресурсам. Это, как правило, SQL-сервер. В RDA-модели ко-

ды компонента представления и прикладного компонента совмещены и выполняются на компьютере-клиенте. Последний поддерживает как функции ввода и отображения данных, так и чисто прикладные функции. Доступ к информационным ресурсам обеспечивается либо операторами специального языка (языка SQL, например, если речь идет о базах данных), либо вызовами функций специальной библиотеки (если имеется соответствующий интерфейс прикладного программирования — API).

Клиент направляет запросы к информационным ресурсам (например, к базам данных) по сети удаленному компьютеру. На нем функционирует ядро СУБД, которое обрабатывает запросы, выполняя предписанные в них действия, и возвращает клиенту результат, оформленный как блок данных (рис. 25). При этом инициатором манипуляций с данными выступают программы, выполняющиеся на компьютерах-клиентах, в то время как ядру СУБД отводится пассивная роль — обслуживание запросов и обработка данных. Такое распределение обязанностей между клиентами и сервером базы данных не догма — сервер БД может играть более активную роль, чем та, которая предписана ему традиционной парадигмой.

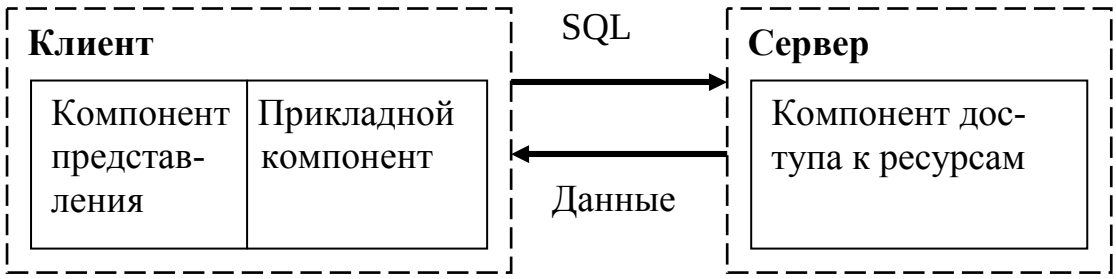


Рис. 25

RDA-модель избавляет от недостатков, присущих как системам с централизованной архитектурой, так и системам с файловым сервером.

Прежде всего, перенос компонента представления и прикладного компонента на компьютеры-клиенты существенно разгружает сервер БД, сводя к минимуму общее число процессов операционной системы. Сервер БД освобождается от несвойст-

венных ему функций; процессор или процессоры сервера целиком загружаются операциями обработки данных, запросов. Это становится возможным благодаря отказу от терминалов и оснащению рабочих мест компьютерами, которые обладают собственными локальными вычислительными ресурсами. С другой стороны, уменьшается загрузка сети, так как по ней передаются от клиента к серверу не запросы на ввод-вывод (как в системах с файловым сервером), а запросы на языке SQL, их объем существенно меньше.

Основное достоинство RDA-модели — унификация интерфейса клиент-сервер в виде языка SQL. Действительно, взаимодействие прикладного компонента с ядром СУБД невозможно без стандартизованного средства общения. Запросы, направляемые программой ядру, должны быть понятны обоим. Для этого их следует сформулировать на специальном языке. Но в СУБД уже существует язык SQL, о котором шла речь. Поэтому целесообразно использовать его не только в качестве средства доступа к данным, но и стандарта общения клиента и сервера.

К сожалению, RDA-модель не лишена ряда недостатков.

Во-первых, взаимодействие клиента и сервера посредством SQL-запросов при большом числе клиентов все же загружает сеть.

Во-вторых, удовлетворительное администрирование приложений в RDA-модели практически невозможно из-за совмещения в одной программе различных по своей природе функций (функции представления и прикладные).

Наряду с RDA-моделью все большую популярность приобретает перспективная DBS-модель (рис. 26). Последняя реализована в некоторых реляционных СУБД (Informix, Ingres, Sybase, Oracle). Ее основу составляет механизм хранимых процедур — средство программирования SQL-сервера. Процедуры хранятся в компилированном виде в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, где функционирует SQL-сервер. Язык, на котором разрабатываются хранимые процедуры, представляет собой процедурное

расширение языка запросов SQL и уникален для каждой конкретной СУБД.

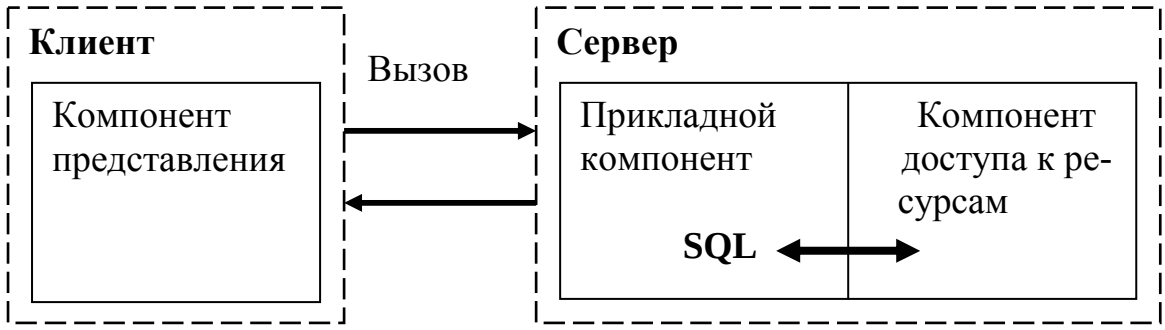


Рис. 26

В DBS-модели компонент представления выполняется на компьютере-клиенте, в то время как прикладной компонент оформлен как набор хранимых процедур и функционирует на компьютере-сервере БД. Там же выполняется компонент доступа к данным, то есть ядро СУБД. Достоинства DBS-модели очевидны: это и возможность централизованного администрирования прикладных функций, и снижение трафика (вместо SQL-запросов по сети направляются вызовы хранимых процедур с указанием их параметров), и возможность разделения процедуры между несколькими приложениями, и экономия ресурсов компьютера за счет использования единожды созданного плана выполнения процедуры. Кроме того, использование хранимых процедур повышает гибкость разделения доступа к базе данных. К недостаткам модели относится то, что кроме доступа к данным на сервере выполняется и компонент приложения, значительно загружая его. Можно также отнести ограниченность средств, используемых для написания хранимых процедур, которые представляют собой разнообразные процедурные расширения SQL, не выдерживающие сравнения по изобразительным средствам и функциональным возможностям с языками третьего поколения, такими, как C или Pascal. Сфера их использования ограничена конкретной СУБД, в большинстве СУБД отсутствуют возможности отладки и тестирования разработанных хранимых процедур.

На практике часто используются смешанные модели, когда поддержка целостности базы данных и некоторые простейшие прикладные функции поддерживаются хранимыми процедурами (DBS-модель), а более сложные функции реализуются непосредственно в прикладной программе, которая выполняется на компьютере-клиенте (RDA-модель).

В AS-модели (рис. 27) процесс, выполняющийся на компьютере-клиенте, отвечает, как обычно, за интерфейс с пользователем (то есть осуществляет функции первой группы). Обращаясь за выполнением услуг к прикладному компоненту, этот процесс играет роль клиента приложения (Application Client — AC). Прикладной компонент реализован как группа процессов, выполняющих прикладные функции, и называется сервером приложения (Application Server — AS). Все операции над информационными ресурсами выполняются соответствующим компонентом, по отношению к которому AS играет роль клиента. Из прикладных компонентов доступны ресурсы различных типов — базы данных, очереди, почтовые службы и др.

Рис. 27

RDA- и DBS-модели опираются на двухзвенную схему разделения функций.

В RDA-модели прикладные функции приданы программе-клиенту, в DBS-модели ответственность за их выполнение берет на себя ядро СУБД. В первом случае прикладной компонент сливается с компонентом представления, во-втором — интегрируется в компонент доступа к информационным ресурсам. В AS-модели реализована трехзвенная схема разделения функций, где

прикладной компонент выделен как важнейший изолированный элемент приложения, для его определения используются универсальные механизмы многозадачной операционной системы, и стандартизованы интерфейсы с двумя другими компонентами.

Глава 7. СЕРВЕР ORACLE

СУБД ORACLE — это многопользовательская реляционная система управления БД, эксплуатируемая на различных компьютерах и на различных операционных системах. СУБД ORACLE хранит большие объемы информации готовыми для немедленного использования или редактирования. Организация хранения данных в системе основана на двумерных таблицах и позволяет пользователю устанавливать связи как между элементами одной таблицы, так и между элементами, находящимися в разных таблицах. Доступ к СУБД осуществляется при помощи языка запросов высокого уровня SQL, процедурном расширении SQL, либо путем включения операторов языка SQL в программу, написанную на каком-нибудь языке программирования, например на C или Java.

7.1 Файлы ORACLE

Существуют три группы файлов, составляющий базу данных.

Файлы базы данных. Файлы базы данных содержат собственно данные, другие файлы важны для функционирования архитектуры. В зависимости от размеров, таблицы и другие объекты могут, очевидно, располагаться в одном файле базы данных, однако это не лучшее решение, поскольку оно не способствует гибкости структуры базы данных для управления доступом к различным пользовательским разделам, размещения базы данных на различных дисководах или резервного копирования и восстановления базы данных.

Управляющие файлы. В управляющие файлы заносится имя базы, дата и время ее создания, сведения о расположении файлов базы данных и журналов обновлений и информация синхронизации, что обеспечить согласованность работы всех групп файлов базы данных.

Журналы обновлений (транзакций). Каждая база должна иметь не менее двух оперативных журналов обновлений, которые работают поочередно и всегда используются во время работы экземпляра ORACLE. Когда какая либо транзакция фиксируется,

изменение данных записывается в журнал обновлений, так что в случае сбоя, не связанного с потерей данных (такого, как аварийный отказ компьютера) ORACLE в процессе запуска может воспользоваться информацией из журнала обновлений для автоматического восстановления без вмешательства администратора базы данных. Когда журнал заполняется (его размер фиксируется при создании базы и впоследствии не может быть увеличен), ORACLE переключается на другой журнал, а заполненный архивируется, образуя группу автономных (архивных) журналов обновлений. Если имеется полный набор автономных журналов обновлений с момента выполнения последней резервной копии БД, значит имеется возможность полного восстановления БД в случае ее краха.

7.2 Фоновые процессы

Для работы многих пользователей многопроцессная система ORACLE использует некоторые дополнительные процессы, называемые фоновыми. Цель этих процессов — увеличение производительности за счет закрепления за ними некоторых функций, которые будут выполняться в интересах всех пользователей системы ORACLE. Фоновые процессы асинхронно выполняют чтение/запись данных в базу, мониторинг процессов ORACLE для того, чтобы достигнуть большего параллелизма и, следовательно, большей производительности и достоверности работы системы ORACLE.

Обязательные системные процессы: к четырем системным процессам, которые должны быть всегда активными, что СУБД могла работать, относятся DBWR, LGWR, SMON и PMON.

DBWR — процесс записи в БД

Фоновый процесс записи в БД переписывает модифицированные блоки из системной глобальной области в файлы базы данных. Он записывает только модифицированные блоки, например, если блок содержит новую, измененную или удаленную запись. Сначала записываются редко используемые (относительно) блоки (они не обязательно записываются в базу данных при фиксации транзакции).

LGWR — процесс записи в журнал обновлений

Процесс записи в журнал переписывает элементы буфера обновлений в памяти для одной или нескольких транзакций. При фиксации транзакции этот процесс должен элементы буфера в журнал на диск, и только потом пользовательский процесс сообщение о успешном фиксации транзакции.

SMON — системный монитор

Процесс SMON осуществляет мониторинг экземпляра ORCAL. Если две транзакции оказались взаимно заблокированы, SMON распознает эту ситуацию и одна из транзакций выбирается в качестве «жертвы» для отката этой транзакции. Кроме того, SMON несет ответственность за очистку временных сегментов, необходимость в которых отпала и наконец — за восстановление «погибших» транзакций, пропущенных во время аварии и восстановления экземпляра, вызванных ошибками чтения файла. Когда СУБД простаивает, SMON дефрагментирует свободное пространство в файлах базы данных, подготавливая распределение внешней памяти под новые объекты.

PMON — пользовательский процесс

Назначение PMON — выполнять восстановление аварийно завершившихся пользовательских процессов. PMON отвечает также за очистку кэш-памяти и освобождение ресурсов, используемых процессом. Например — меняет статус активной таблицы транзакций, освобождает фиксации объектов системы и убирает идентификаторы процессов из списка активных процессов ORACLE. Также как и SMON, PMON периодически «пробуждается» сам, а также может быть вызван другим процессом, если он определит, что PMON должен выполнить свои функции.

Необязательные системные процессы:

ARCH — архиватор

Процесс ARCH производит копирование находящегося в online режиме оперативного журнала обновлений при его заполнении. Процесс ARCH активен, если используется режим

ARCHIVELOG и архивация активирована (автоматически или вручную).

RECO — процесс восстановления

Фоновый процесс восстановления запускается, если происходит сбой в распределенной транзакции (транзакция, в которой изменяются две или несколько баз данных) и одна или несколько баз данных требуют выполнить или фиксацию или откат транзакции. RECO, если он запущен, пытается автоматически фиксировать или откатить транзакцию на локальной базе данных синхронно с процессами RECO на других базах данных ORACLE.

SQL*Net listener — процесс прослушивания сети

Этот процесс направляет запросы, приходящие от компьютеров-клиентов, к соответствующему экземпляру ORACLE. Он взаимодействует с сетевым программным обеспечением, маршрутизируя запросы между сервером баз данных и компьютером-клиентом.

7.3 Оперативная память ORACLE

Глобальная область системы (SGA) представляет собой совместно используемую область в памяти компьютера и активно используется данными во время работы ORACLE. Размер SGA зависит от значения параметра, установленного в файле INIT.ORA. Глобальная область системы содержит данные и управляющую информацию для одного экземпляра ORACLE. SGA распределяется при запуске экземпляра ORACLE и удаляется при остановке. Текущим пользователям базы необходимо видеть текущие копии информации из SGA. Исходя из того, что SGA содержит столь часто используемую информацию, лучше (быстрее) держать SGA в основной памяти, нежели чем постоянно считывать с диска. Для оптимальной работы в большинстве систем SGA должна быть в постоянной готовности в реальной памяти. Если это не так, производительность системы может катастрофически снижаться, так как части SGA участвуют в страничном обмене между внешней и основной памятью. Структура SGA показана на рис. 28.

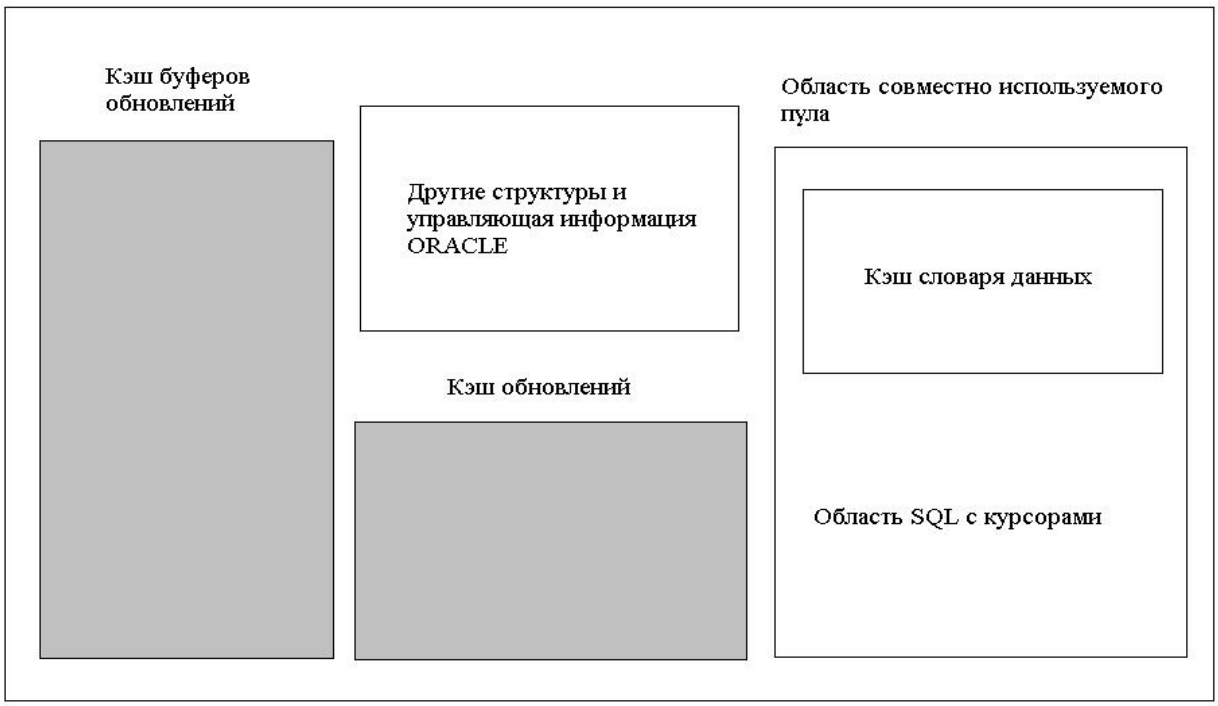


Рис. 28

В кэше буферов обновлений хранятся блоки ORACLE, которые были прочитаны из файлов базы данных. Если один процесс прочитал блоки таблицы в память, все процессы могут обратиться к этим блокам. Если процессу требуется обращение к некоторым данным, ORACLE проверяет, не считан ли уже этот блок в кэш, избегая таким образом чтения с диска. Блоки в буферном кэше упорядочиваются таким образом, что наиболее часто используемые блоки располагаются в одном конце списка буферов, и наименее используемые — в другом, причем этот список постоянно обновляется. Чем больше блоков данных помещается в памяти, тем быстрее работает экземпляр.

В оперативные журналы обновлений записываются все изменения пользовательских и системных объектов базы данных. До того, как изменения записываются в журнал, ORACLE хранит их в кэше обновлений. Элементы из кэша переписываются в журнал обновлений, когда кэш заполняется или когда фиксируется транзакция.

Оператор SQL, переданный для выполнения серверу баз данных, должен быть интерпретирован перед его выполнением. Область SQL в SGA содержит данные для связывания, временные буферы, дерево разбора, и план выполнения для каждого

оператора SQL. КЭШ словаря данных содержит фрагменты системных таблиц ORACLE. Он загружается начальным набором элементов при запуске экземпляра и затем, если требуется дополнительная информация, заполняется данными из словаря базы данных.

7.4 Внешняя память ORACLE

Из соображений управляемости, безопасности и производительности база данных логически разделяется на одно или несколько табличных пространств, состоящих из одного или нескольких файлов базы данных. Файл базы данных всегда связан только с одним табличным пространством. Табличные пространства могут быть в любое время переведены в состояние offline (недоступны для RDBMS) и наоборот. Таким образом, все файлы, принадлежащие табличному пространству, одновременно переводятся в состояния online или offline. Если файл принадлежит табличному пространству, он не может быть отдельно переведен в состояние offline. Любое табличное пространство может быть переведено в состояие offline, за двумя исключениями:

- Табличное пространство SYSTEM (а значит и входящие в него файлы) должно быть всегда в состоянии online;
- Любое табличное пространство, содержащее активный сегмент rollback (используемый активным или приостановленным экземпляром ORACLE), должно оставаться в состоянии online.

Администратор использует табличные пространства для:

- управления распределением памяти для объектов базы, таких как таблицы, индексы, и временные сегменты;
- установления квот памяти для пользователей базы;
- управления доступностью данных путем перевода отдельных табличных пространств в состояния online или offline;
- копирования и восстановления данных;
- распределения данных по устройствам для повышения производительности.

Такие объекты базы данных, как синонимы или представления не занимают места в базе данных, кроме места в таблице словаря базы данных, где хранятся их определения, наряду с определениями всех других типов объектов.

Все данные в табличных пространствах запоминаются в элементах пространства базы, называемых сегментами. Сегмент — это набор блоков базы, распределенных для хранения данных базы. Эти данные могут быть таблицей, индексом или временными данными, необходимыми для работы ORACLE. Сегменты являются следующим логическим уровнем памяти после табличных пространств. Сегмент не может перекрывать табличное пространство, но может перекрывать файлы внутри табличного пространства.

Базе требуется до пяти типов сегментов:

- сегменты данных;
- индексные сегменты;
- сегменты отката (rollback);
- временные сегменты;
- сегмент первоначальной загрузки.

Сегменты в зависимости от типа создаются различными способами. Большинство пользователей имеют дело только с сегментами данных и индексными сегментами. Вообще говоря, только DBA работает с сегментами отката, временными и сегментом первоначальной загрузки. DBA работает с сегментами отката напрямую, используя SQL — операторы, ORACLE же автоматически создает временные сегменты и сегмент первоначальной загрузки. Все сегменты имеют сходную структуру и состоят из экстентов. Внешняя память для какого-либо объекта в базе данных выделяется порциями из определенного числа блоков. В файлах базы данных эти блоки должны быть смежными. Группа смежных блоков называется экстентом. Например, когда создается таблица с использованием установок по умолчанию, для самого первого экстента таблицы распределяется пять блоков ORACLE (начальный экстент). По мере добавления и модификации строк в таблице эти пять блоков заполняются данными. Ко-

гда последний блок таблицы будет заполнен и должны вставляться новые строки, база данных автоматически добавляет для таблицы новый экстент (пять блоков). Это выделение новых экстентов будет продолжаться до тех пор, пока не будет исчерпано свободное место в табличном пространстве.

Вначале большинство сегментов имеют некоторый указанный размер (количество экстентов), впоследствии они динамически растут (добавляются экстенты) по мере необходимости. Например в момент создания таблица содержит только начальный экстент указанного размера. Несмотря на то, что в таблицу не занесено ни одной строки, это количество байтов (в блоках ORACLE) зарезервировано под таблицу. При наполнении таблицы данными он растет, пока не заполнит начальный экстент. Если для данных требуется дополнительная память, распределяется дополнительный экстент так, что теперь сегмент данных будет состоять из двух. Каждый сегмент содержит блок управления сегментом (segment control block), который описывает характеристики этого сегмента, а также содержит справочник экстентов данного сегмента. Подобная информация содержится в словаре данных в обзоре DBA_EXTENTS, который показывает все экстенты базы вне зависимости от принадлежности табличному пространству. (Словарь данных показывает свободную память и используется для поиска новых экстентов). Когда ORACLE необходимо распределить следующий экстент для сегмента, она просматривает «свободную память» в табличном пространстве и распределяет первый сплошной участок блоков требуемого размера. Заголовок сегмента и словарь данных затем обновляются, чтобы отметить появление нового экстента и уменьшение

Экстенты данных и индекса не освобождаются, пока не освободится соответствующий сегмент. Пока таблица существует, остаются распределенными все принадлежащие ей блоки памяти. Новая строка может быть добавлена в существующий блок памяти, если там есть достаточно места. Даже если удалены все строки таблицы, блоки данных не освобождаются для дальнейшего использования в табличном пространстве.

Если же удалена вся таблица (drop table), сегменты индекса и данных освобождаются для данного табличного пространства, а экстенты становятся доступны другим объектам этого пространства.

Аналогично для индексного сегмента все блоки распределенных экстентов остаются распределенными пока существует индекс. Когда индекс отменяется, все блоки освобождаются для использования в данном табличном пространстве.

Сегменты отката (rollback)

Каждая база содержит один или несколько сегментов отката. Сегмент отката — это часть базы, куда записываются действия, которые необходимо выполнить при определенных обстоятельствах. Сегменты отката используются для обеспечения:

- отмены транзакции;
- согласованного чтения;
- восстановления.

Сегмент отката содержит соответствующие данные для транзакций одного экземпляра ORACLE, которые (транзакции) могут быть связаны с любым табличным пространством базы.

Временные сегменты

При обработке запросов системе ORACLE обычно бывает необходима временная память на промежуточных стадиях обработки операторов. Эти области называются временными сегментами. Типично временный сегмент требуется как рабочая область для сортировки. Сегмент не создается, если сортировка может быть выполнена в оперативной памяти или ORACLE найдет другой путь сортировки, используя индексы.

Следующие SQL — операторы могут нуждаться в использовании временных сегментов:

- CREATE INDEX
- SELECT ... ORDER BY
- SELECT DISTINCT ...
- SELECT ... GROUP BY
- SELECT ... UNION
- SELECT ... INTERSECT
- SELECT ... MINUS

- неиндексированные объединения
- некоторые коррелированные подзапросы

Например, если запрос содержит фразы DISTINCT, GROUP BY и ORDER BY, потребуется не менее трех временных сегментов.

Глава 8. PL/SQL

PL/SQL — это принадлежащее Oracle процедурное расширение SQL и изначально предназначался для создания хранимых подпрограмм и триггеров базы данных. PL/SQL позволяет вставлять, удалять, обновлять и извлекать данные ORACLE и управлять потоком предложений для обработки этих данных. Можно объявлять константы и переменные, определять внутренние подпрограммы (процедуры и функции) и перехватывать ошибки времени выполнения. Таким образом, PL/SQL комбинирует мощь манипулирования данными SQL с мощью обработки данных процедурных языков. PL/SQL — это язык, структурированный блоками. Это значит, что основные единицы (процедуры, функции и анонимные блоки), составляющие программу PL/SQL, являются логическими блоками, которые могут содержать любое число вложенных в них подблоков. Обычно каждый логический блок соответствует некоторой проблеме или подпроблеме, которую он решает.

За счет расширения языка SQL, PL/SQL предлагает уникальную комбинацию мощи и простоты использования. и может гибко и безопасно манипулировать данными ORACLE, потому что PL/SQL поддерживает все команды манипулирования данными, команды управления транзакциями, функции, и операторы SQL. Однако PL/SQL НЕ поддерживает команд определения данных, таких как CREATE, команд управления сессией, таких как SET ROLE, и команду управления системой ALTER SYSTEM.

Для манипулирования данными ORACLE можно использовать команды INSERT, UPDATE, DELETE, SELECT.

Блок (или подблок) позволяет группировать логически связанные объявления и предложения. Благодаря этому можно размещать объявления близко к тем местам, где они используются. Объявления локальны в блоке, и перестают существовать, когда блок завершается.

Структура блока Pl/SQL:

```
[ DECLARE
  объявления локальных объектов  (декларативная часть)]
BEGIN
  Исполнительная часть  (выполняемые предложения)
```

```
 [ EXCEPTION
   обработчики исключений   ]
END;
```

8.1 Типы данных

Порядок частей блока логичен. Блок начинается с декларативной части, в которой объявляются объекты. С объявленными объектами осуществляются манипуляции в исполнительной части. Исключения, возбуждаемые во время исполнения, могут быть обработаны в части обработки исключений. Каждый блок может содержать другие блоки.

Типы данных используемые в Pl/SQL:

- Number — совпадает с типом Number таблиц;
- Varchar2(N) — совпадает с типом Varchar2 таблиц с отличием N (максимальная длина этого типа в PL/SQL составляет 32767);
- Char[(N)] — совпадает с типом Char таблиц (как и Varchar2 максимальная длина этого типа 32767);
- Date — совпадает с типом Date таблиц;
- Boolean — логический тип, может иметь значения TRUE, FALSE или NULL;
- Binary_Integer — целый тип, диапазон значений от — $2 \bullet 10^{31}-1$ до $2 \bullet 10^{31}-1$;

Можно также использовать типы Long, Raw и Long Raw (максимальная длина этих типов до 32767). Операции с вышеприведенными типами представлены в таблице на рис. 29

Оператор	Операция
** , NOT	возведение в степень, логическое отрицание
* , /	умножение, деление
+ , - , ||	сложение, вычитание, конкатенация (сцепление строк)
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	сравнение
AND	логическое умножение
OR	логичское сложение

Рис. 29

PL/SQL трактует любую строку нулевой длины как NULL. Поэтому для проверки пустых строк нужно использовать оператор IS NULL, как показано ниже:

```
IF my_string IS NULL THEN ...
```

Оператор конкатенации игнорирует пустые операнды. Например, выражение

```
'Иванов' || NULL || NULL || 'Петр'
```

даст значение 'ИвановПетр'.

Можно объявлять переменные и константы в декларативной части любого блока PL/SQL или подпрограммы. Объявление распределяет место для значения, специфицирует его тип данных и задает имя, по которому можно обращаться к этому значению. Объявление может также присвоить начальное значение и специфицировать ограничение NOT NULL. Примеры:

```
Date_Rojd DATE;
nom_count NUMBER(6) := 0;
Name VARCHAR2(15) NOT NULL DEFAULT 'Владимир';
```

Первое объявление именует переменную типа DATE. Второе объявление именует переменную типа NUMBER и использует оператор присваивания (:=), чтобы присвоить этой переменной нулевое начальное значение. Третье объявление именует переменную типа VARCHAR2, специфицирует для нее ограничение NOT NULL и присваивает ей начальное значение 'Владимир' (вместо оператора присваивания можно использовать зарезервированное слово DEFAULT).

Нельзя присваивать значения NULL переменным или константам, объявленным как NOT NULL. Если вы попытаетесь это сделать, будет возбуждено предопределенное исключение VALUE_ERROR. За ограничением NOT NULL должна следовать

фраза инициализации; в противном случае получится ошибка компиляции.

В объявлениях констант зарезервированное слово CONSTANT должно предшествовать спецификатору типа, как показывает следующий пример:

```
credit_limit  CONSTANT REAL := 5000.00;
```

Как показывают следующие примеры, инициализирующее выражение может быть сколь угодно сложным и может ссылаться на ранее инициализированные переменные и константы:

```
pi CONSTANT NUMBER := 3.14159;
radius  NUMBER DEFAULT 1;
area NUMBER := pi * radius**2;
```

Использование %TYPE

Атрибут %TYPE представляет тип данных переменной, константы или столбца базы данных. В следующем примере, %TYPE представляет тип данных переменной:

```
credit NUMBER(7,2);
debit credit%TYPE;
```

Переменные и константы, объявленные с атрибутом %TYPE, трактуются так, как если бы они были объявлены с явным типом данных. Например, в примере выше PL/SQL рассматривает переменную debit как переменную типа NUMBER(7,2).

Следующий пример показывает, что объявление через %TYPE может включать фразу инициализации:

```
balance NUMBER(7,2);
minimum_balance balance%TYPE := 10.00;
```

Атрибут %TYPE особенно полезен при объявлении переменных, которые ссылаются на столбцы базы данных. Можно ссылаться на таблицу и столбец, как показывает следующий пример:

```
my_name Sotr.famil%TYPE;
```

Использование атрибута %TYPE при объявлении my_name имеет два преимущества. Во-первых, не обязательно знать точный тип столбца famil. Во-вторых, если определение столбца famil изменится, то тип данных переменной my_name изменится соответственно во время выполнения. При задании переменной с атрибутом %TYPE можно также использовать инициализацию.

Использование %ROWTYPE

Атрибут %ROWTYPE возвращает тип записи, представляющей строку в таблице (или обзоре) или в курсоре. Такая запись может содержать целую строку данных, выбранных из таблицы или извлеченных курсором. В следующем примере объявляются две записи. Первая из них хранит строку, выбранную из таблицы Sotr. Вторая запись хранит строку, извлеченную курсором c1 (курсоры рассматриваются в соответствующем параграфе).

```
DECLARE
Sotr_rec  Sotr%ROWTYPE;
CURSOR c1 IS SELECT nom_otd, name FROM Otd;
Otd_rec c1%ROWTYPE;
...
```

Столбцы в строке таблицы (или курсора) и соответствующие поля в записи имеют одинаковые имена и типы данных.

8.2 Управляющие структуры

8.2.1 Условное управление: предложения IF

Часто бывает необходимо предпринять альтернативные действия в зависимости от обстоятельств. Предложение IF позволяет выполнить последовательность предложений условно. Это значит, что, будет выполнена эта последовательность или нет, зависит от значения условия.

```
IF условие1 THEN
        ряд_предложений1;
 [  ELSIF условие2 THEN
        ряд_предложений2;]
 [  ELSIF условие3 THEN
        ряд_предложений3;]
……….
 [  ELSE
        ряд_предложений;]
END  IF;
```

Если первое условие дает FALSE или NULL, фраза ELSIF проверяет следующее условие. В предложении IF может быть сколько угодно фраз ELSIF, последняя фраза ELSE необязательна. Условия вычисляются по одному сверху вниз. Если любое условие даст TRUE, выполняется соответствующая последовательность предложений, и управление передается на следующее за IF предложение (без вычисления оставшихся условий). Если все условия дадут FALSE или NULL, выполняется последовательность предложений в фразе ELSE, если она есть. Рассмотрим следующий пример:

```
    IF  zarplata> 50000 THEN
       Premia := 1500;
    ELSIF zarplata > 35000 THEN
       Premia := 500;
    ELSE
       Premia := 100;
    END IF;
INSERT INTO prem_sotr VALUES (nomer, premia, ...);
```

Если значение Zarplata превышает 50000, истинны как первое, так и второе условия. Тем не менее, переменной Premia присваивается правильное значение 1500, потому что второе условие проверяться не будет, а управление сразу будет передано на предложение INSERT.

8.2.2 Итеративное управление: Предложения LOOP и EXIT

Предложения LOOP позволяют выполнить последовательность предложений несколько раз. Есть три формы предложения LOOP:

LOOP, WHILE-LOOP и FOR-LOOP.

LOOP

Простейшую форму предложения LOOP представляет основной (или бесконечный) цикл, который окружает последовательность предложений между ключевыми словами LOOP и END LOOP:

```
LOOP
ряд_предложений
END LOOP;
```

При каждой итерации цикла последовательность предложений выполняется, а затем управление передается на начало цикла. Если дальнейшее повторение нежелательно или невозможно, вы можете использовать предложение EXIT, чтобы закончить цикл. Вы можете поместить сколько угодно предложений EXIT внутри цикла, но только не вне цикла. Есть две формы предложения EXIT: EXIT и EXIT WHEN.

EXIT

Предложение EXIT форсирует безусловное завершение цикла. Когда встречается предложение EXIT, цикл немедленно заканчивается, и управление передается на следующее (за END LOOP) предложение. Предложение EXIT можно применять только внутри цикла. Чтобы выйти из блока PL/SQL до достижения его нормального конца, можно использовать предложение RETURN.

Пример:

Пример:

```
LOOP
...
IF ... THEN
...
EXIT;          -- немедленно выходит из цикла
END IF;
END LOOP;   --управление передается сюда
```

EXIT-WHEN

Предложение EXIT-WHEN позволяет завершить цикл условно. Когда встречается это предложение, вычисляется условие в фразе WHERE. Если это условие дает TRUE, цикл завершается, и управление передается на предложение, следующее за циклом. Пример:

```
LOOP
   A1:=a1+1;
EXIT WHEN A1>100; -- выйти из цикла при условии
...
END LOOP;
CLOSE c1;
```

Пока условие не станет истинным, цикл не может завершиться. Поэтому, предложения внутри цикла должны изменять значение условия.

Метки циклов

Как и блоки PL/SQL, циклы могут иметь метки. Метка, (необъявляемый идентификатор в двойных угловых скобках) должна появиться в начале предложения LOOP:

```
<<имя_метки>>
LOOP
ряд_предложений
END LOOP;
```

Имя метки цикла может также (необязательно) появиться в конце цикла в предложении END LOOP. В предложении EXIT можно указать метку цикла, из которого необходимо выйти:

```
LOOP
…
<<out1>>
LOOP
…
LOOP
…
EXIT out1 WHEN ... -- выйти из цикла out1
END LOOP;
…
END LOOP out1;
…
 END LOOP;
```

WHILE-LOOP

Предложение WHILE-LOOP ассоциирует условие с последовательностью предложений, окруженной ключевыми словами LOOP и END LOOP:

```
WHEN условие LOOP
   ряд_предложений;
END LOOP;
```

Перед каждой итерацией цикла условие проверяется. Если оно дает TRUE, то последовательность предложений выполняется, и управление возвращается на начало цикла. Если условие дает FALSE или NULL, то цикл обходится, и управление передается на следующее предложение. Пример:

```
WHILE total <= 25000 LOOP
...
total := total + salary;
END LOOP;
```

Число повторений цикла зависит от условия и неизвестно до тех пор, пока цикл не завершится. Поскольку условие проверяется в начале цикла, последовательность предложений может не выполниться ни разу.

FOR-LOOP

В то время как число итераций цикла WHILE неизвестно до тех пор, пока цикл не завершится, для цикла FOR число итераций известно до того, как войти в цикл. Циклы FOR осуществляют свои итерации по заданному интервалу целых чисел. (Курсорные циклы FOR, которые повторяются по активному множеству курсора, обсуждаются ниже) Этот интервал является частью СХЕМЫ ИТЕРАЦИЙ, которая окружается ключевыми словами FOR и LOOP. Синтаксис имеет следующий вид:

```
FOR счетчик IN [REVERSE] нижняя_граница..верхняя_граница
   LOOP
      ряд_предложений;
   END LOOP;
```

Интервал вычисляется один раз, при первом входе в цикл, и больше не перевычисляется. Как показывает следующий пример, последовательность предложений выполняется один раз для каждого целого в заданном интервале. После каждой итерации выполняется приращение индекса цикла.

```
FOR i IN 1..3 LOOP -- присваивает переменной i значения 1, 2, 3
ряд_предложений; -- будет выполнен три раза
END LOOP;
```

Как показывает следующий пример, если нижняя граница интервала совпадает с верхней, цикл выполняется один раз:

```
FOR i IN 3..3 LOOP -- присваивает переменной i значение 3
   ряд_предложений; -- будет выполнен один раз
END LOOP;
```

По умолчанию индекс наращивается на 1 от нижней до верхней границы. Однако, если вы используете ключевое слово REVERSE, индекс будет изменяться в обратном направлении, от верхней границы к нижней, как показывает следующий пример. После каждой итерации индекс уменьшается на 1.

```
FOR i IN REVERSE 1..3 LOOP -- присваивает переменной i 3, 2, 1
ряд_предложений; -- будет выполнен три раза
END LOOP;
```

Внутри цикла FOR к индексу цикла можно обращаться как к константе. Поэтому индекс может встречаться в выражениях, но ему нельзя присваивать значений.

Предложение SELECT INTO

Предложение извлекает данные из одной или нескольких таблиц (или представлений) и помещает их в одну или несколько переменных. Предложение имеет структуру:

```
SELECT  список_выбора INTO список_переменных
  FROM … остаток_select_предложения.
```

Переменные в списке переменных должны быть объявлены в блоке объявления объектов блока и совпадать по типу со списком выбора или допускать неявные преобразования типов. Может быть и переменная типа запись. Поскольку данные извлекаются из таблиц, наиболее часто объявляется переменная, использующая тип на основе атрибута %ROWTYPE. Остаток select предложения может означать что угодно, что может следовать за фразой FROM оператора SELECT.

Предложение SELECT INTO должно возвращать ровно одну строку из таблицы. Если это предложение возвращает несколько строк, происходит возбуждение предопределенного исключения TOO_MANY_ROWS, которое может быть обработано в блоке обработки исключений. Если не возвращается ни одной стоки, возбуждается исключение NO_DATA_FOUND, которое также может быть обработано.

Рассмотрим пример извлечения данных всех столбцов одной строки таблицы SOTR для сотрудника с номером 1000 и занесения их в переменную типа запись (номер сотрудника является первичным ключом, поэтому оператор select возвратит одну строку (при условии, что сотрудник с таким номером существует):

```
DECLARE
  Data_sotr  Sotr%ROWTYPE;
……….
BEGIN
   SELECT * INTO Data_sotr FROM Sotr WHERE nom_sotr=1000;
……….
```

8.3 Процедуры и функции

Подпрограмма — это поименованный блок PL/SQL, который принимает параметры и может быть вызван. PL/SQL имеет два типа подпрограмм, называемых ПРОЦЕДУРАМИ и ФУНКЦИЯМИ. Обычно процедуру вызывают для того, чтобы выполнить некоторое действие, а функцию — для того, чтобы вычислить некоторое значение.

Как и непоименованные (АНОНИМНЫЕ) блоки PL/SQL, подпрограммы имеют декларативную часть, исполняемую часть и необязательную часть обработки исключений. Декларативная часть содержит объявления типов, курсоров, констант, переменных, исключений и вложенных подпрограмм. Все эти объекты локальны, и перестают существовать после выхода из подпрограммы. Исполняемая часть содержит предложения, которые присваивают значения, управляют выполнением и манипулируют данными ORACLE. Часть обработки исключений содержит обработчики, которые имеют дело с исключениями, возбуждаемыми при исполнении. Объявление процедуры имеет следующий вид:

```
PROCEDURE имя_процедуры [(список формальных параметров)]
  IS
[ объявления локальных объектов]
  BEGIN
     Исполняемая часть
  [ EXCEPTION  блок обработки исключений]
END [имя_процедуры];
```

Объявление формального параметра:

Имя_параметра [режим передачи параметра] тип_параметра

тип параметра:

- Имя_типа_записи;
- таблица.столбец%TYPE;
- таблица%ROWTYPE;
- скалярный тип данных;
- имя_переменной%TYPE.

Для типов NUMBER, CHAR и VARCHAR2 нельзя указывать размерность, это приведет к ошибке компиляции.

Режимы передачи параметров:

IN

Режим передачи IN позволяет передавать значения в подпрограмму. Внутри подпрограммы такой формальный параметр ведет себя как константа — его значение можно использовать, но нельзя изменять любым способом. Этот режим действует по умолчанию, т.е. если режим не задан, то считается, что параметр задан в режиме IN.

Фактический параметр, соответствующий формальному параметру с режимом IN, может быть константой, литералом, инициализированной переменной или выражением. В отличие от параметров OUT и IN OUT, параметры IN могут инициализироваться умалчиваемыми значениями, как это делается, будет показано ниже на примере.

OUT

Параметр OUT по своему назначению прямо противоположен параметру IN и позволяет возвращать значение вызывающей программе. Внутри подпрограммы такой параметр выступает как неинициализированная переменная. Поэтому его значение нельзя

присваивать другим переменным или переприсвоить самому себе. Фактический параметр, соответствующий формальному параметру с режимом OUT, должен быть переменной; он не может быть константой или выражением. Формальный параметр OUT может (но не обязан) иметь значение в момент вызова подпрограммы. Однако это значение теряется, когда вызывается подпрограмма. Внутри подпрограммы формальный параметр OUT нельзя использовать в выражении; единственная операция, допустимая на таком параметре — это присваивание ему значения.

Перед выходом из подпрограммы не нужно забывать явно присвоить значения параметрам OUT. В противном случае значения соответствующих фактических параметров будут не определены. При успешном выходе из подпрограммы PL/SQL присваивает значения фактическим параметрам. Однако, если происходит выход необработанным исключением, PL/SQL не присваивает значений фактическим параметрам.

IN OUT

Параметр IN OUT позволяет вам передавать в подпрограмму начальные значения и возвращать обновленные значения вызывающей программе. Внутри подпрограммы такой параметр выступает как инициализированная переменная. Поэтому ему можно присвоить значение, а его значение можно присваивать другим переменным. Иными словами, параметр IN OUT можно рассматривать как обычную переменную. Вы можете изменять его значение или обращаться к этому значению любыми способами. В таблице показаны сравнение режимов передачи параметров:

IN	OUT	IN OUT
используется по умолчанию	должен быть указан явно	должен быть указан явно
только для чтения	только для записи	для чтения и записи
предназначен для передачи значения в подпрограмму формальный параметр	предназначен для передачи значения из подпрограммы в вызывающий модуль	предназначен для передачи начального значения в подпрограмму и возврата измененного

IN	OUT	IN OUT
ведет себя как константа	формальный параметр ведет себя как непроинициализированная переменная	значения вызывающему модулю
значение формального параметра нельзя изменять	формальный параметр нельзя использовать в выражениях; ему должно быть присвоено значение	формальный параметр ведет себя как проинициализированная переменная значение формального параметра может быть изменено в подпрограмме
фактический параметр может быть литералом, константой, проинициализированной переменной или, в общем случае, произвольным выражением	фактический параметр должен быть переменной, полем записи или элементом PL/SQL-таблицы	фактический параметр должен быть переменной, полем записи или элементом PL/SQL-таблицы

Умалчиваемые значения параметров

Как показывает следующий пример, можно инициализировать параметры с режимом IN умалчиваемыми значениями. Это позволяет передавать подпрограмме различное число параметров, принимая или перекрывая умалчиваемые значения .

```
PROCEDURE create_otd
   (new_nom NUMBER, name_otd  VARCHAR2, tel_otd number,
                          etaj_otd number default 1)
IS
BEGIN
   INSERT INTO Otd
         VALUES (new_nom,name_otd,tel_otd,etaj_otd);
END create_otd;
```

Здесь при вызове процедуры можно опустить четвертый параметр и в самой процедуре он будет иметь значение 1.

Описание функции похоже на описание процедуры, с тем отличием, что указывается тип возвращаемого значения и в теле функции должен быть оператор присваивания функции значения:

```
FUNCTION имя_функции [(список формальных параметров)]
  RETURN тип_возвращаемого_значения
  IS
[ объявления локальных объектов]
  BEGIN
     Исполняемая часть
  [ EXCEPTION  блок обработки исключений]
END [имя_функции];
```

Как и при задании типа параметра, в типе возвращаемого значения нельзя указывать размерность типов NUMBER, CHAR и VARCHAR2. В теле функции должен быть хотя бы один оператор присваивания функции значения:

```
RETURN выражение;
```

Примечание: этот параметр без выражения можно использовать и в процедуре или в анонимном блоке для немедленного прекращения выполнения процедуры или анонимного блока.

PL/SQL предоставляет много мощных функций, помогающих манипулировать данными. Можно использовать функции всюду, где допускаются выражения того же типа. Более того, допускаются вложенные вызовы функций друг в друга.

Встроенные функции распадаются на следующие категории:

- функции сообщений об ошибках;
- числовые функции;
- символьные функции;
- функции преобразований;
- календарные функции;
- смешанные функции.

В предложениях SQL можно использовать все встроенные функции, за исключением функций сообщений об ошибках SQLCODE и SQLERRM. В процедурных предложениях можно использовать все встроенные функции, за исключением смешанной функции DECODE.

Групповые функции SQL AVG, MIN, MAX, COUNT, SUM, не встроены в PL/SQL. Тем не менее, вы можете использовать их в предложениях SQL (но не в процедурных предложениях PL/SQL). Для каждой встроенной функции приводятся ее аргументы, типы данных этих аргументов, и тип данных возвращаемого значения. Следующий пример показывает, что функция LENGTH принимает аргумент типа VARCHAR2 и возвращает значение типа NUMBER:

```
function LENGTH (str VARCHAR2) return NUMBER
```

Ниже приводятся примеры функций, которые могут понадобиться при выполнении практических и лабораторных работ.

ASCII

```
function ASCII (char VARCHAR2) return NUMBER
```

Возвращает код сопоставляющей последовательности, который представляет символ char в наборе символов базы данных. Функция ASCII является обратной к функции CHR.

CHR

```
function CHR (num NUMBER) return VARCHAR2
```

Возвращает символ, который имеет код n в сопоставляющей последовательности набора символов базы данных. Функция CHR является обратной к функции ASCII.

CONCAT

```
function CONCAT (str1 VARCHAR2, str2 VARCHAR2) return VARCHAR2
```

Присоединяет строку str2 к строке str1 и возвращает результат. Если один из аргументов пуст, CONCAT возвращает другой

аргумент. Если оба аргумента пусты, CONCAT возвращает NULL.

LENGTH
function LENGTH (str CHAR) return NUMBER
function LENGTH (str VARCHAR2) return NUMBER

Возвращает число СИМВОЛОВ в строке str. Если строка str имеет тип CHAR, то в длину входят хвостовые пробелы. Если строка str пуста, LENGTH возвращает NULL.

LPAD
function LPAD (str VARCHAR2, len NUMBER [, pad VARCHAR2]) return VARCHAR2

Возвращает строку str, дополненную слева до длины len цепочкой символов pad, повторяющейся столько раз, сколько необходимо. Если строка pad не указана, подразумевается пробел. Если строка str длиннее len символов, то LPAD возвращает первые len символов строки str.

RPAD
function RPAD (str VARCHAR2, len NUMBER [, pad VARCHAR2]) return VARCHAR2

Возвращает строку str, дополненную справа до длины len цепочкой символов pad, повторяющейся столько раз, сколько необходимо. Если строка pad не указана, подразумевается пробел. Если строка str длиннее, чем len символов, то RPAD возвращает первые len символов строки str.

LTRIM
function LTRIM (str VARCHAR2, [, set VARCHAR2]) return VARCHAR2

Возвращает строку str, из которой удалены начальные символы вплоть до первого символа, не принадлежащего множеству set. Если множество set не задано, подразумевается пробел.

RTRIM
function RTRIM (str VARCHAR2, [, set VARCHAR2]) return VARCHAR2

Возвращает символьную строку str, из которой удалены конечные символы после последнего символа, не принадлежащего множеству set. Если множество set не задано, подразумевается пробел.

REPLACE
function REPLACE (str1 VARCHAR2, str2 VARCHAR2
[,str3 VARCHAR2]) return VARCHAR2

Возвращает строку str1, в которой каждое вхождение подстроки str2 заменено строкой str3. Если строка str3 не задана, то все вхождения подстроки str2 удаляются из строки str1. Если не специфицированы ни поисковая подстрока, ни строка замены, то REPLACE возвращает NULL.

SUBSTR
function SUBSTR (str VARCHAR2, pos NUMBER [, len NUMBER]])
return VARCHAR2

Возвращает подстроку строки str, начинающуюся с СИМВОЛЬНОЙ позиции pos и содержащую len символов (или, если число len опущено, все символы до конца строки str). Значение pos не может быть нулевым. Если значение pos отрицательно, SUBSTR подсчитывает символы от конца строки str. Число len должно быть положительным.

LOWER
function LOWER (str CAR) return CHAR
function LOWER (str VARCHAR2) return VARCHAR2

Возвращает строку str, в которой все буквы преобразованы в строчные.

UPPER
function UPPER (str CHAR) return CHAR
function UPPER (str VARCHAR2) return VARCHAR2

Возвращает строку str, в которой все буквы преобразованы в прописные.

TO_DATE
function TO_DATE (str VARCHAR2 [, fmt VARCHAR2])
return DATE

Преобразует строку str или число num в значение даты в формате, заданном fmt. Допустимые модели формата приведены в следующей таблице:

Модель формата	Описание
CC,SCC	век (S префиксует даты до н.э. минусом)
YYYY,SYYYY	год (S префиксует даты до н.э. минусом)
IYYY	год в стандарте ISO
YYY,YY,Y	последние три, две или одна цифра года
IYY,IY,I	то же для года ISO
Y,YYY	год с запятой
YEAR,SYEAR	год прописью (S префиксует даты до н.э. минусом)
RR	последние две цифры года в новом веке
BC,AD	индикатор BC или AD
B.C.,A.D.	индикатор B.C. или A.D.
Q	квартал (1-4)
MM	месяц (1-12)
RM	римский номер месяца (I-XII)
MONTH	имя месяца
MON	сокращенное имя месяца
WW	неделя года (1-53)
IWW	неделя года (1-52 или 1-53) по ISO
W	неделя месяца (1-5)
DDD	день года (1-366)
DD	день месяца (1-31)
D	день недели (1-7)
DAY	имя дня
DY	сокращенное имя дня
J	юлианский день (число дней с 1 января 4712 г. до н.э.)
AM,PM	индикатор полудня
A.M.,P.M.	индикатор полудня с точками
HH,HH12	час дня (1-12)
HH24	час суток (0-23)
MI	минута (0-59)
SS	секунда (0-59)
SSSSS	секунд после полуночи (0-86399)

Если формат опущен, подразумевается, что строка str задана в умалчиваемом формате даты. Если аргумент fmt имеет значение 'J' (юлианский день), то число num должно быть целым.

TO_CHAR для дат
function TO_CHAR (dte DATE [, fmt VARCHAR2])
return VARCHAR2

Преобразует дату dte в символьную строку типа VARCHAR2 в формате, заданном моделью формата fmt. (Допустимые модели формата приведены в описании функции TO_DATE.) Если опустить fmt, подразумевается умалчиваемый формат даты. Например функция TO_DATE(Dat_Rojd,’DD Month yyyy’) возвратит символьное значение ’12 Апрель 1980’, а TO_DATE(Dat_Rojd,’DD-MM-yy’) – ’12-04-80’

USER
function USER return VARCHAR2

Возвращает в верхнем регистре имя текущего пользователя ORACLE. Эта функция не имеет аргументов.

USERENV
function USERENV (str VARCHAR2) return VARCHAR2

Возвращает информацию о текущей сессии, полезную для составления аудиторской таблицы или для определения используемого языка и набора символов.

Символьная строка str может иметь одно из следующих значений:

'ENTRYID' Возвращает идентификатор аудиторской записи.

"LANGUAGE' Возвращает используемые язык, территорию и набор символов базы данных.

'SESSIONID' Возвращает идентификатор аудиторской сессии.

'TERMINAL' Возвращает идентификатор терминала в операционной системе.

8.4 Курсоры

PK/SQL позволяет непосредственно использовать SQL-операторы, изменяющие или удаляющие множества строк. Но как процедурный язык, предназначенный, прежде всего, для задания того, как выполнять те или иные действия, PL/SQL ориентирован на работу с отдельными строками, а не с их множествами. Именно курсоры позволяют произвольно обрабатывать информацию по одной строке. Множество строк, возвращаемых запросом (активное множество), может состоять из нуля, одной или нескольких строк, в зависимости от того, сколько строк удовлетворяют вашим поисковым условиям. Когда запрос возвращает несколько строк, можно явно определить курсор для обработки этих строк.

Курсор определяется в декларативной части блока PL/SQL, подпрограммы или пакета путем задания его имени и специфицирования запроса. После этого осуществляется манипуляция курсором при помощи трех команд: OPEN, FETCH и CLOSE. Работа с курсором очень похожа на работу с файлами в таких языках программирования, как PASCAL или C.

Прежде всего, инициализируется курсор предложением OPEN, которое идентифицирует активное множество. Затем с помощью предложения FETCH извлекается первая строка. Можно повторять FETCH неоднократно, пока не будут извлечены все строки. После обработки последней строки освобождается курсор предложением CLOSE.

Можно обрабатывать параллельно несколько запросов, объявив и открыв несколько курсоров. Специфицируется курсор объявлением:

```
CURSOR имя [ (список параметров курсора) ]
  IS оператор_SELECT [FOR UPDATE]
```

где параметр, в свою очередь, имеет следующий синтаксис

имя_переменной [IN] тип_данных [:=<значение>|DEFAULT <значение>]

Формальные параметры курсора должны иметь режим IN. Открытие курсора предложением OPEN исполняет предложение

SELECT и идентифицирует АКТИВНОЕ МНОЖЕСТВО, т.е. все строки, удовлетворяющие поисковым условиям запроса. Для курсоров, объявленных с фразой FOR UPDATE, предложение OPEN также осуществляет блокировку этих строк. Сфера параметров курсора локальна в этом курсоре, что означает, что к этим параметрам можно обращаться лишь в запросе, который участвует в объявлении курсора. Значения параметров курсора используются ассоциированным запросом в момент открытия курсора.

Пример объявления и открытия курсора, формирующего множество строк таблицы сотрудников определенного отдела:

```
DECLARE
  CURSOR CUR1(n_o number) IS SELECT * FROM Sotr
                                WHERE nom_otd=n_o;
……….
 BEGIN

OPEN CUR1(105);
……..
```

Предложение OPEN не извлекает строк активного множества. Для этого используется предложение FETCH.

```
FETCH имя_курсора INTO локальные объекты;
```

Для каждого значения столбца, извлекаемого запросом, ассоциированного с курсором, в списке INTO должна быть соответствующая переменная, имеющая такой же тип с этим столбцом или допускающая неявное преобразование. Обычно используется не индивидуальные переменные, а запись. Последовательно используя FETCH (обычно в цикле) можно извлечь все строки множества. Если выдается FETCH, но в активном множестве больше нет строк, то значения переменных в списке INTO не определены. Каждый явно объявленный курсор, имеет четыре атрибута: %NOTFOUND, %FOUND, %ROWCOUNT и %ISOPEN. Атрибуты позволяют получать полезную информацию о выполнении многострочного запроса. Для обращения к атрибуту просто присоединяется его имя к имени курсора. Атрибуты явного кур-

сора можно использовать в процедурных предложениях, но не в предложениях SQL.

- **%NOTFOUND** — если последняя операция FETCH вернула строку, %NOTFOUND дает FALSE. Если последняя операция FETCH не смогла вернуть строку (так как активное множество исчерпано), %NOTFOUND дает TRUE. Операция FETCH должна в конце концов исчерпать активное множество, так что, когда это происходит, никакого исключения не возбуждается;
- **%FOUND** — логически противоположен атрибуту %NOTFOUND;
- **%ROWCOUNT** — когда открывается курсор, его атрибут %ROWCOUNT обнуляется. Перед первой операцией FETCH %ROWCOUNT возвращает 0. Впоследствии, %ROWCOUNT возвращает число строк, извлеченных операциями FETCH из активного множества на данный момент. Это число увеличивается, если последняя FETCH вернула строку;
- **%ISOPEN** — дает TRUE, если явный курсор открыт, и FALSE в противном случае.

В следующем примере показан типичный цикл обработки курсора:

```
DECLARE
  CURSOR Cur1 IS SELECT * from Sotr;
  Str_Sotr  Cur1%ROWTYPE;
 BEGIN
   OPEN Cur1;  -- формируется активное множество
LOOP
 FETCH Cur1 INTO Str_Sotr;
   EXIT Cur1%NOTFOUND; -- выход из цикла, если множество
                       --  закончилось
  . . . . . .
  --   обработка данных

END LOOP;
CLOSE Cur1;
END;
```

Рассмотренный выше цикл используется настолько часто, что введен специальный цикл FOR по курсору, чтобы упростить кодирование. Курсорный цикл FOR неявно объявляет свой индекс цикла как запись типа %ROWTYPE, открывает курсор, в цикле извлекает строки из активного множества в поля записи, и закрывает курсор, когда все строки обработаны или когда вы выходите из цикла. Не нужно явно открывать и закрывать курсор и извлекать из него строки. Пример, рассмотренный выше, в цикле по курсору будет выглядеть так:

```
DECLARE
  CURSOR Cur1 IS SELECT * from Sotr;
BEGIN
  FOR St_Sotr IN Cur1
   LOOP
  . . . . . .
  --   обработка данных

END LOOP;
END;
```

Перед каждой итерацией курсорного цикла FOR, PL/SQL извлекает данные в неявно объявленную запись St_Sotr, которая эквивалентна следующей явно объявленной записи:

```
St_Sotr  Cur1%ROWTYPE;
```

Эта запись определена только внутри цикла, в ней нельзя обращаться к ее полям вне цикла. Можно даже не объявлять курсор, а задать его неявно в курсорном операторе FOR:

```
FOR Cur1 in (SELECT * FROM Sotr)
  LOOP  . . . . .
```

Pl/SQL использует неявный курсор при выполнении любого SQL-оператора, который автоматически открывается и закрывается. Этими действиями нельзя управлять, можно лишь узнать значения атрибутов неявного курсора (при этом для ссылки на

него используется имя SQL) по результатам выполнения последнего SQL-оператора.

8.5 Хранимые подпрограммы

Подпрограммы на PL/SQL отдельно компилировать и сохранять в базе данных. Такие скомпилированные и готовые к выполнению подпрограммы называются хранимыми. Они позволяют повысить производительность среды клиент-сервер, позволяют экономить память на сервере при одновременной работе нескольких приложений, повышают целостность приложений и базы и обеспечивают дополнительные возможность и гибкость защиты данных. При вызове хранимая подпрограмма помещается в кэш, где совместно используется различными приложениями. Про обращении к подпрограмме сервер сначала обращается к кэшу и, если подпрограммы в нем нет, считывает подпрограмму с диска, увеличивая таким образом производительность системы. Когда пользователь вызывает подпрограмму (если у него есть привилегии ее выполнять), все доступы к данным выдаются ему на время выполнения подпрограммы, даже если у него нет явного доступа к данным. Например, пользователь вызывает процедуру вставки нового сотрудника. Операторы вставки в таблицу в процедуре будут выполнены даже если у пользователя нет привилегий на выполнение оператора INSERT для таблицы сотрудников Sotr. Как было отмечено выше, производительность повышается и за счет уменьшения объема передаваемого по сети информации. Хранимая подпрограмма является объектом базы данных и имена хранимых подпрограмм должны быть уникальны в пределах одной схемы. Синтаксис создания хранимых подпрограмм аналогичен объявлениям процедур и функций:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
                [(объявления формальных параметров)]
      IS|AS
[ объявления локальных объектов]
  BEGIN
     Исполняемая часть
  [ EXCEPTION  блок обработки исключений]
END [имя_процедуры];
```

```
CREATE [OR REPLACE] FUNCTION имя_функции
                [(список формальных параметров)]
  RETURN тип_возвращаемого_значения
  IS|AS
[ объявления локальных объектов]
  BEGIN
     Испольняемая часть
  [ EXCEPTION  блок обработки исключений]
END [имя_функции];
```

Если подпрограмма впервые создается, опцию OR REPLACE можно опустить, при изменении подпрограммы эта опция обязательна. Вызов хранимых подпрограмм аналогичен вызову локальных подпрограмм. Исходный текст хранимой подпрограммы хранится в словаре базы данных и для вывода текста хранимой подпрограммы можно выполнить SQL-запрос:

```
SELECT text FROM User_Source
  WHERE name='имя подпрограммы в верхнем регистре'
    ORDER BY line
```

При компиляции хранимой подпрограммы могут ошибки, сообщения о которых можно просмотреть из представления User_Errors:

```
SELECT line,position,text FROM User_errors
  WHERE name=' имя подпрограммы в верхнем регистре'
```

Рассмотрим хранимую процедуру перевода сотрудника из одного отдела в другой для которой необходимы два параметра — номер сотрудника и номер нового отдела:

```
CREATE OR REPLACE PROCEDURE Perevod_in_Otdel
        (N_Sotr number, N_otdel number) IS
  Sotr_exist  number:=0;
BEGIN
   SELECT count(*) INTO Sotr_exist FROM Sotr
                    WHERE Nom_Sotr=N_Sotr;
   IF Sotr_exist=0 THEN
      dbms_output.put_line('Сотрудник с таким номером не существует');
      RETURN;
```

```
   END IF;
  UPDATE Sotr SET Nom_Otd=N_Otdel WHERE Nom_Sotr=N_Sotr;
  COMMIT;
END;
```

Для вывода вызов процедуры Dbms_out.Put_Line несколько неудобен, проще создать в своей схеме хранимую процедуру с более коротким именем и вызывать ее для вывода на экран. Тип параметра зададим символьный, поскольку в этот тип неявно преобразуются большинство поддерживаемых ORACLE типов:

```
 CREATE OR REPLACE PROCEDURE Pr(In_par Varchar2) IS
   BEGIN
      Dbms_output.Put_line(In_Par);
   END;
```

8.6 Триггеры баз данных

ТРИГГЕР БАЗЫ ДАННЫХ — это хранимая программная единица PL/SQL, ассоциированная с конкретной таблицей базы данных. ORACLE исполняет (возбуждает) триггер базы данных автоматически каждый раз, когда оператор SQL изменяет данные в этой таблице. В отличие от подпрограмм, которые должны вызываться явно, триггер базы данных вызывается неявно. Триггеры базы данных используются:

- для аудита (отслеживания) модификаций данных;
- автоматической журнализации (регистрации) изменений;
- реализации сложных правил поддержки целостности данных, которые невозможно организовать декларативно при создании таблицы;
- автоматического вычисления значений столбцов;
- осуществления сложных процедур защиты;
- поддержки дублированных таблиц.

С каждой таблицей можно ассоциировать до 12 триггеров базы данных. Триггер базы данных состоит из трех частей: события триггера, необязательного ограничения триггера и действия триггера. Когда происходит событие триггера, триггер базы дан-

ных возбуждается, и анонимный блок PL/SQL выполняет предписанное действие. Триггеры базы данных возбуждаются с привилегиями владельца, а не текущего пользователя. Поэтому владелец должен иметь должный доступ ко всем объектам, вовлекаемым в действие триггера. В несколько сокращенном виде команда создания триггера имеет вид:

```
CREATE [OR REPLACE] TRIGGER  имя_триггера
время_срабатывания     операторы_срабатывания
ON имя таблицы
[FOR EACH ROW [WHEN (условие срабатывания)]]
PL/SQL-блок;
```

время_срабатывания: BEFORE|AFTER — триггер срабатывает до или после оператора, вызвавшего срабатывание триггера;

операторы_срабатывания: DELETE [OR INSERT]

[OR UPDATE [OF имя_столбца1, имя столбца2…]] — здесь задается оператор(ы) действия с таблицей, для оператора UPDATE можно дополнительно задать имена столбцов, которые изменяются ;

FOR EACH ROW — указывает, что триггер является строчным, т.е. срабатывает для каждой строки (при отсутствии этой опции триггер является операторным, т.е. срабатывает только один раз). Для строчного оператора можно указать условие срабатывания, где для каждой строки проверяется условие и, если оно истинно, триггер срабатывает. Если же нет (FALSE или NULL), тело триггера не выполняется. В теле строчного триггера можно указывать корреляционные имена OLD и NEW для строки со значениями столбцов до и после выполнения оператора. Эти имена можно использовать как имя записи с полями, соответствующим столбцам таблицы, перед корреляционными именами ставится двоеточие. Понятно, что для оператора INSERT имеет смысл только корреляционное имя NEW, для DELETE — имя OLD, а для UPDATE — оба имени. Кроме того, в теле триггера, объявленного для нескольких операторов срабатывания, доступны встроенные функции INSERTING, DELETING и UPDATING логического типа, позволяющие определить оператор, вызвавший

триггер. Рассмотрим примеры. Создадим триггер, модифицирующий вводимое значение фамилии таким образом, что бы удалялись начальные пробелы и фамилия переводилась в верхний регистр.

```
CREATE TRIGGER Mod_Famil
  BEFORE  INSERT OR UPDATE
ON Sotr
FOR EACH ROW
 BEGIN
   :NEW.Famil:=LTRIM(UPPER(:NEW.Famil));
 end;
```

Следующий триггер осуществляет дублирование данных таблицы отделов Otd в таблицу Otd_Dubl.

```
CREATE TRIGGER Otd_Dub
AFTER INSERT OR UPDATE OR DELETE ON Otd
DECLARE
Nom_otd1 otd.nom_otd%type;
Name1 otd.name%type;
Nom_tel1 otd.nom_tel%type;
Etaj1 otd.etaj%type;
BEGIN
IF INSERTING THEN
INSERT INTO Otd_Dubl VALUES(:new.nom_otd,:new.name,
                                  :new.nom_tel,:new.etaj);
ELSIF UPDATING THEN
IF :new.nom_otd is not null THEN
Nom_otd1:=:new.nom_otd;
ELSE
Nom_otd1:=:old.nom_otd;
END IF;
IF :new.name is not null THEN
Name1:=:new.name;
ELSE
Name1:=:old.name;
END IF;
IF :new.nom_tel is not null THEN
Nom_tel1:=:new.nom_tel;
ELSE
Nom_tel1:=:old.nom_tel;
```

```
END IF;
IF :new.etaj is not null THEN
Etaj1:=:new.etaj;
ELSE
Nom:=:old.etaj;
END IF;
UPDATE Otd_Dubl SET  nom_otd:=nom_otd1,name=name1,
nom_tel=nom_tel1,etaj=etaj1 WHERE nom_otd=:old.nom_otd;
ELSE
DELETE Otd_Dubl WHERE nom_otd=:old.nom_otd;
END IF;
END;
```

Триггер прерывает выполнение оператора, активировавшего этот триггер, если в теле триггера наблюдается необработанное исключение (пример такого рода рассмотрен в главе обработки исключений). В отличие от хранимой процедуры тело триггера не компилируется при создании, поэтому (при относительно большом коде) желательно часть кода оформлять в виде обычной хранимой процедуры для повышения быстродействия. В теле триггера допустимы все команды SQL, что и в обычном анонимном блоке или хранимой процедуры, кроме операторов управления транзакциями. Понятно, что эти операторы недопустимы и в хранимой процедуре, если она вызывается в теле триггера.

8.7 Обработка исключений

В PL/SQL условие ошибки называется ИСКЛЮЧЕНИЕМ. Исключения могут быть внутренние (именованные и неименованные) или определены пользователем.

Примеры внутренне определенных исключений включают «ДЕЛЕНИЕ НА 0» и «НЕХВАТКУ ПАМЯТИ». Некоторые общие внутренне определенные исключения имеют предопределенные имена, такие как ZERO_DIVIDE и STORAGE_ERROR. Другим внутренне определенным исключениям имена могут быть присвоены. Пользовательские исключения объявляются в декларативной части любого блока, подпрограммы. В отличие от внутренних исключений, пользовательские исключения должны иметь имена.

Когда возникает ошибка, соответствующее исключение возбуждается. Это значит, что нормальное выполнение останавливается, и управление передается на часть обработки исключений блока или подпрограммы PL/SQL. Внутренние исключения возбуждаются неявно (автоматически) системой исполнения; пользовательские исключения возбуждаются явно, посредством предложений RAISE, которые могут также возбуждать предопределенные исключения.

Для обработки возбуждаемых исключений пишутся отдельные части программы, называемые **ОБРАБОТЧИКАМИ ИСКЛЮЧЕНИЙ**. После выполнения обработчика исключений исполнение текущего блока заканчивается, и окружающий блок продолжает свое выполнение со следующего предложения. Если окружающего блока нет (т.е. текущий блок не вложен в другой блок), то управление возвращается в хост-окружение.

Предопределенные исключения

Внутреннее исключение возбуждается неявно всякий раз, когда программа PL/SQL нарушает правило ORACLE или превышает установленный системой лимит. Каждая ошибка ORACLE имеет номер, однако исключения должны обрабатываться по их именам. Поэтому PL/SQL внутренне определяет некоторые распространенные ошибки ORACLE как исключения. Например, предопределенное исключение NO_DATA_FOUND возбуждается, когда предложение SELECT INTO не возвращает ни одной строки.

Для обработки других ошибок ORACLE вы можете использовать общий обработчик OTHERS. Функции сообщений об ошибках SQLCODE и SQLERRM (они имеют смысл только в обработчике исключений) особенно полезны в обработчике OTHERS, так как они возвращают код ошибки ORACLE и текст сообщения об ошибке.

Рассмотрим наиболее распространенные именованные внутренние исключения:

- **DUP_VAL_ON_INDEX** — возбуждается, когда операция INSERT или UPDATE пытается создать повторяющееся значение в столбце, ограниченном опцией UNIQUE;

- **INVALID_CURSOR** — возбуждается, когда вызов PL/SQL специфицирует некорректный курсор (например, при попытке закрыть неоткрытый курсор);
- **INVALID_NUMBER** — возбуждается в предложении SQL, когда преобразование символьной строки в число сбивается из-за того, что строка не содержит правильного представления числа. Например, следующее предложение INSERT возбудит исключение INVALID_NUMBER, когда ORACLE попытается преобразовать 'HALL' в число: INSERT INTO emp (empno,ename,deptno) VALUES ('HALL', 7888, 20); В процедурных предложениях вместо этого исключения возбуждается VALUE_ERROR;
- **NO_DATA_FOUND** — возбуждается, когда предложение SELECT INTO не возвращает ни одной строки, или при обращении к неинициализированной строке таблицы PL/SQL;
- **PROGRAM_ERROR** — возбуждается, когда PL/SQL встретился с внутренней проблемой;
- **STORAGE_ERROR** — возбуждается, когда PL/SQL исчерпал доступную память, или когда память запорчена;
- **TOO_MANY_ROWS** — возбуждается, когда предложение SELECT INTO возвращает больше одной строки;
- **VALUE_ERROR** — возбуждается при возникновении арифметической ошибки, ошибки преобразования, ошибки усечения или ошибки ограничения. Например, VALUE_ERROR возбуждается при усечении строкового значения, присваиваемого переменной PL/SQL. (Однако при усечении строкового значения, присваиваемого хост-переменной, никакого исключения не возбуждается.) В процедурных предложениях VALUE_ERROR возбуждается при ошибке преобразования символьной строки в число;
- **ZERO_DIVIDE** — возбуждается при попытке деления числа на 0.

Другие внутренние исключение не имеют имен, определить их можно по коду, которое возвращает функция SQLCODE. Обрабатываются исключения в обработчике, конструкция которого имеет вид:

```
EXCEPTION
  WHEN имя_исключения [OR имя_исключения [OR …]]   THEN
    ---  обработка исключений
 [ WHEN имя_исключения [OR имя_исключения [OR …]] THEN
  ---  обработка исключений   ]
   ……..
[ WHEN OTHERS  THEN
  --- обработка исключений  ]
```

Когда автоматически возбуждается предопределенное исключений или, с помощью RAISE пользовательское, оно перехватывается обработчиком. В предложении WHEN OTHERS перехватываются исключения, которые отсутствуют в выше объявленных списках исключений. В этом случае, код исключения определяется с помощью функции

```
function SQLCODE return number;
```

возвращающую код возбужденного исключения. Функция

```
function SQLERRM return VARCHAR;
function SQLERRM(code  NUMBER) return VARCHAR;
```

возвращает сообщение об ошибки. Функция SQLERRM без аргумента возвращает сообщение об ошибке с текущим значением SQLCODE. Заметим, что функция SQLCODE и функция SQLERRM без аргумента имеют смысл только в обработчике исключений. Функция SQLCODE вне обработчика всегда возвращает 0, а функция SQLERRM — сообщение о нормальном, успешном завершении.

Пользовательские исключения объявляются в декларативной части анонимного блока, подпрограммы или триггера используя следующий синтаксис:

```
Имя_исключения   EXCEPTION;
```

а возбуждаются оператором RAISE:

```
RAISE имя_исключения.
```

Для примера использования пользовательского исключения, составим хранимую процедуру изменения зарплаты сотрудника, пределы суммарных зарплат отделов хранятся в таблице

Limit_zarpl_otdel(nom_otd, Limit_zarpl)

где первый столбец — номер отдела, а второй — пределы суммарных зарплат. В процедуру будем передавать два параметра, первый параметр — номер сотрудника, второй — новое значение зарплаты:

```
CREATE PROCEDURE Replace_Zarpl(nom_s number, New_zarpl_sotr number)
 IS
  N_O sotr.nom_otd%type;  -- для хранения номера отдела сотрудника
  Zarpl_sotr sotr.zarpl%type;  -- для хранения текущей зарплаты сотрудника
  Sum_zarpl number; -- для хранения суммарной зарплаты отдела
  Sum_zarpl_lim limit_zarpl_otdel.limit_zarpl%type; -- для хранения
-- предельной зарплаты отдела
  Err_zarpl  EXCEPTION;
BEGIN
  SELECT nom_otd INTO N_O  FROM  Sotr WHERE nom_sotr=nom_s;
  SELECT Zarpl INTO Zarpl_sotr  FROM  Sotr WHERE nom_sotr=nom_s;
  SELECT sum(zarpl) INTO Sum_zarpl FROM Sotr WHERE nom_otd=N_O;
  SELECT Limit_zarpl INTO Sum_zarpl_lim FROM Limit_zarpl_otdel
                                          WHERE nom_otd=N_O;
IF Sum_zarpl-Zarpl_sotr+New_zarpl_sotr>Sum_zarpl_lim THEN
     RAISE Err_zarpl;
END IF;
UPDATE Sotr SET Zarpl=New_zarpl_sotr WHERE nom_sotr=nom_s;
EXCEPTION
  WHEN  NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Сотрудника с таким номером не существует');
  WHEN  Err_zarpl THEN
  DBMS_OUTPUT.PUT_LINE('Превышена предельная сумма зарплат отдела');
  WHEN OTHERS THEN
  --  Здесь можно обработать другие исключения
  END;
```

С помощью прагмы в декларативной части пользовательскому исключению можно присвоить код ошибки:

```
PRAGMA  EXCEPTION_INIT(имя_исключения, код ошибки);
```

который можно определить в обработчике с помощью функции SQLCODE.

Как было замечено выше, если в триггере возбудить исключение и не обработать его, можно прервать выполнение оператора, возбудившего триггер. Все модификации данных в триггере также анулируются, поскольку ORACLE перед выполнением оператора выполняет неявную точку сохранения и откатывает все изменения к этой точке. Например, чтобы исключить ввод отрицательного значения зарплаты сотрудника, можно определить такой триггер (хотя такое исключение ввода проще определить при создании таблицы с помощью ограничения CHECK):

```
CREATE TRIGGER Neg_zarpl BEFORE INSERT OR UPDATE OF Zarpl
ON Sotr FOR EACH ROW
BEGIN
  IF :New.Zarpl<0 THEN
    RAISE Zero_Divide;  -- можно возбудить любое исключение
  END IF:
END;
```

Более корректное использование исключений в триггере (с использованием процедуры raise_application_error) будет рассмотрено на практических занятиях.

ЛИТЕРАТУРА

1. Дейт, К. Дж. Введение в системы баз данных: Пер. с англ. — 6-е изд. — К.: Диалектика, 1998. — 784 с.

2. Карпова Т.С. Базы данных: модели, разработка, реализация — СПб.: Питер, 2002. — 304 с.

3. Петров В.Н. Информационные системы — СПб.: Питер, 2002. — 688 с.

4. Кренке Д. Теория и практика построения баз данных. — 8-е изд. — СПб.: Питер, 2003. — 800 с.

5. Шкарина Л. Язык SQL: учебный курс.- СПб.:Питер, 2001.- 592 с.:ил.

6. Джеймс Р. Грофф, Пол Н. Вайнберг SQL: полное руководство: пер. с англ. - К.: Издательская группа BHV, 1998. -608с.