Ю. Ю. ГРОМОВ, О. Г. ИВАНОВА,
А. В. ЯКОВЛЕВ, В. Г. ОДНОЛЬКО

УПРАВЛЕНИЕ ДАННЫМИ

Тамбов
• Издательство ФГБОУ ВПО «ТГТУ» •
2015

Ю. Ю. ГРОМОВ, О. Г. ИВАНОВА, А. В. ЯКОВЛЕВ, В. Г. ОДНОЛЬКО
УПРАВЛЕНИЕ ДАННЫМИ

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Тамбовский государственный технический университет»

**Ю. Ю. ГРОМОВ, О. Г. ИВАНОВА,
А. В. ЯКОВЛЕВ, В. Г. ОДНОЛЬКО**

УПРАВЛЕНИЕ ДАННЫМИ

Допущено Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебника для студентов высших учебных заведений,
обучающихся по направлению подготовки 230400
«Информационные системы и технологии»

Тамбов
Издательство ФГБОУ ВПО «ТГТУ»
2015

ББК з81я73
УДК 004.08(075.8)
У67

Рецензенты:
Доктор физико-математических наук, профессор,
заслуженный деятель науки РФ
В. Ф. Крапивин

Кандидат технических наук, профессор,
зав. кафедрой ИБ ФГБОУ ВПО «ТГТУ»
Ю. Ф. Мартемьянов

Громов, Ю. Ю.

У67 Управление данными : учебник / Ю. Ю. Громов, О. Г. Иванова, А. В. Яковлев, В. Г. Однолько. – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2015. – 192 с. – 100 экз.

ISBN 978-5-8265-1385-9.

Целью учебника является систематическое изложение теоретических основ построения баз данных. Рассматриваются основные понятия баз данных. Даётся характеристика моделей данных, подробно описывается реляционная модель. Излагаются современные подходы к концептуальному проектированию баз данных. Раскрываются принципы организации баз данных в сетях, а также современные направления развития баз данных. По каждой главе приводятся контрольные вопросы и задания.

Предназначен для студентов направления 230400 «Информационные системы и технологий», а также может быть полезно студентам всех направлений и специальностей подготовки в области информационных и коммуникационных технологий.

ББК з81я73
УДК 004.08(075.8)

ISBN 978-5-8265-1385-9

ВВЕДЕНИЕ

Современный этап развития общества характеризуется его глобальной информатизацией и повсеместным использованием средств информационных и коммуникационных технологий. Широкое использование профессионально ориентированных информационных систем (ИС) становится важнейшим аспектом деятельности специалистов любого профиля.

В настоящее время в условиях широкой информатизации общества всё большее распространение получают справочные системы, системы информационной поддержки деятельности учреждений, системы поддержки принятия решений, системы автоматизированного учёта и контроля, системы автоматизированного проектирования и множество других систем на базе средств информационных и коммуникационных технологий.

Основу функционирования информационных систем составляют *базы данных*. В широком понимании база данных представляет совокупность сведений о реальных процессах, событиях или явлениях, относящихся к определённой предметной области, и организованную таким образом, чтобы обеспечить удобное представление этой совокупности как в целом, так и в любой её части.

Истоки технологий баз данных относятся к началу 1960-х гг. К этому времени уже был накоплен некоторый опыт решения задач обработки информации средствами технологий файловых систем, основанных на использовании магнитных лент с последовательным доступом к данным. Появление устройств памяти прямого доступа определило начало становления новых технологий, связанных с созданием, поддержкой и использованием баз данных.

В период 1960–70-х гг. формируются основы методологии построения баз данных. Существенный вклад в технологии баз данных внесла CODASYL (ассоциация представителей крупнейших поставщиков и пользователей средств вычислительной техники), в отчётах которой был впервые проведён систематический анализ разработанного к тому времени программного инструментария и фактически была предложена обобщённая функциональная модель СУБД, впервые сформулированы концепции многоуровневой архитектуры, функционирования систем управления базами данных общего назначения, концепция схемы базы данных и языка определения данных, основополагающие понятия сетевой модели данных. Дальнейшее формирование архитектурной концепции баз данных было продолжено Рабочей группой ANSI/X3/SPARC (была предложена трёхуровневая модель систем баз данных, в значительной степени определившая дальнейшее развитие технологий баз данных).

Одновременно с подходом CODASYL формировался иной подход на основе иерархической структуризации данных, оказавший значительное влияние на разработки иерархических СУБД.

Значительный (в своё время, по существу, революционный) вклад в развитие теории баз данных был сделан американским математиком Э. Ф. Коддом, разработавшим реляционный подход к базам данных. В настоящее время реляционная модель данных не только не утратила своей актуальности, но и получила дальнейшее развитие благодаря объектным технологиям.

Современные технологии баз данных характеризуются объектным подходом, дальнейшим развитием архитектурных принципов «клиент-сервер» и промежуточного слоя, интеграцией неоднородных информационных ресурсов, расширением сферы применения благодаря использованию CASE-средств и др.

Знание основных идей и методов в области проектирования профессионально ориентированных баз данных, владение навыками разработки и внедрения подобных систем становятся важнейшими компонентами системы подготовки специалистов, объектами профессиональной деятельности которых являются информационные процессы, определяемые спецификой предметной области.

1. ОСНОВЫ ПОСТРОЕНИЯ БАЗ ДАННЫХ

1.1. АРХИТЕКТУРА СИСТЕМЫ БАЗ ДАННЫХ

Под *базой данных* (БД) понимается «совокупность данных, организованных по определённым правилам, предусматривающим общие принципы описания, хранения и манипулирования данными, независимая от прикладных программ. Эти данные относятся к определённой предметной области и организованы таким образом, что могут быть использованы для решения многих задач многими пользователями».

Для описания базы данных инициативной группой ANSI/SPARC (Американский национальный институт стандартов/Комитет по требованиям и планированию стандартов) в 1970-х гг. была предложена трёхуровневая архитектура. Архитектура ANSI/SPARC включает следующие уровни: внутренний, концептуальный и внешний (рис. 1.1). Каждый уровень описывается соответствующей схемой (средствами языка описания данных соответствующего уровня), определяющей свойства базы данных в терминах типов содержащихся в БД данных. Уровни связаны механизмами междууровневого отображения данных.

Внешний уровень определяет точку зрения пользователей на базу данных. Отдельного пользователя интересует не вся БД, а только некоторая часть её. Пользовательское представление базы данных

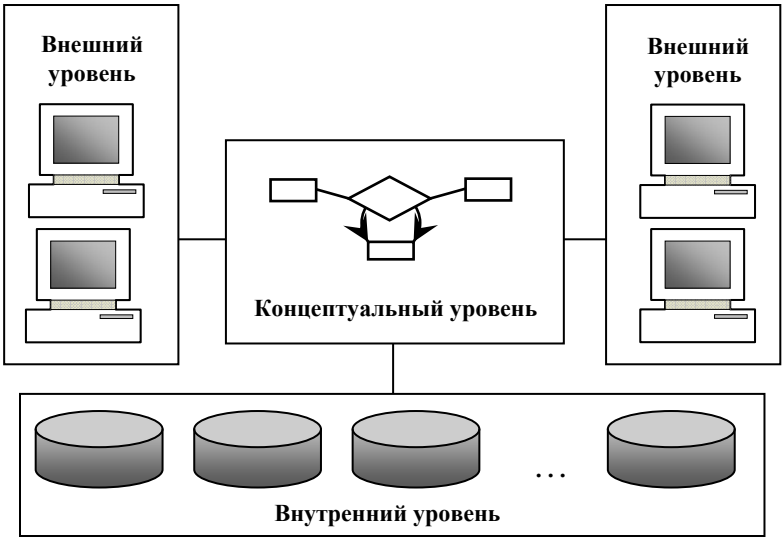


Рис. 1.1. Уровни архитектуры

(*внешнее представление*) будет абстрактным по сравнению с физическим способом хранения данных. Оно отражает представление содержимого базы данных отдельным пользователем. Например, пользователь из отдела кадров может рассматривать базу данных как набор сведений об отделах, о служащих и ничего не знать о клиентах компании или о выпускаемой продукции. Каждый пользователь создаёт своё внешнее представление. Пользовательские представления отображаются в данные концептуального уровня. Это отображение демонстрирует, как различные приложения будут использовать данные. Внешний уровень определяется средствами *внешней схемы*. Приложения внешнего уровня создаются средствами либо одного из распространённых языков программирования (например, C++), либо специального языка запросов (SQL).

Концептуальный уровень отражает обобщённую модель предметной области, для которой создавалась база данных. На этом уровне база данных представлена в наиболее общем виде. *Концептуальное представление* – это представление данных такими, какие «они есть на самом деле», а не то, какими их представляют пользователи, абстрактное представление *всей* базы данных. Может быть несколько внешних представлений, каждое из которых состоит из более или менее абстрактного представления определённой части базы данных, и может быть только одно концептуальное представление, состоящее из абст-

рактного представления базы данных в целом. Концептуальное представление определяется с помощью *концептуальной схемы*.

Внутренний уровень связан со способами хранения и извлечения данных. *Внутреннее представление* отражает низкоуровневую структуру базы данных. Понятие структуры физической базы данных включает: формат хранимой записи, структуру путей доступа и размещение записей на физических устройствах, и т.д. Внутреннее представление описывается средствами *внутренней схемы*, определяющей не только типы хранимых записей, но и индексы, представления памяти, упорядочение полей и т.д. Отображение концептуального уровня во внутренний изолирует концептуальную модель от влияния каких-либо возможных изменений в хранимых данных на физических носителях. При изменении структуры хранимой базы данных изменяется отображение «концептуальный-внутренний» таким образом, чтобы результаты изменений не коснулись концептуального уровня.

Трёхуровневая архитектура позволяет обеспечить *физическую независимость хранимых данных.* Разработчик базы данных может при необходимости изменить физическую модель данных, переписав хранимые данные на другие носители информации и реорганизовав их физическую структуру. Также можно подключить к системе любое число новых пользователей (новых приложений), дополнив, если надо, концептуальную модель. Указанные изменения не будут замечены существующими пользователями системы (окажутся «прозрачными» для них), так же как не будут замечены и новые пользователи. Следовательно, независимость данных обеспечивает возможность развития системы баз данных без разрушения существующих приложений.

Система баз данных представляет собой совокупность следующих взаимосвязанных компонентов: данные (хранимая база данных), программное обеспечение (СУБД, приложения, операционная система и т.д.), аппаратное обеспечение (процессор, оперативная память, магнитные диски для хранения информации и т.д.), пользователи (рис. 1.2).

Рассмотрим подробнее такие компоненты системы баз данных, как пользователи и программное обеспечение.

С базами данных работают различные категории *пользователей*: прикладные программисты, конечные пользователи, администраторы базы данных.

Конечные пользователи – это основная категория пользователей. Конечные пользователи работают с базами данных через рабочую станцию или терминал, используя при этом либо приложения, либо интерфейс СУБД. В зависимости от особенностей создаваемой базы данных круг конечных пользователей может существенно различаться. Конечным пользователем может быть, например, клиент компьютерной фирмы, просматривающий каталог продукции или услуг, или начальник отдела, анализирующий перспективы её развития.

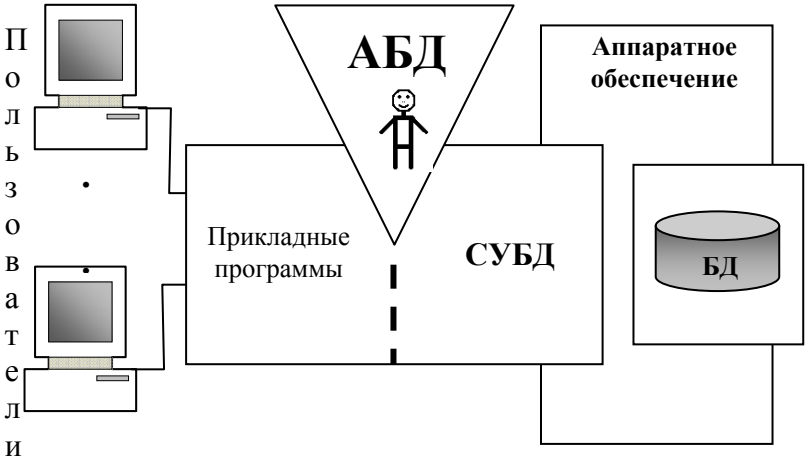


Рис. 1.2. Компоненты системы баз данных

Прикладные программисты разрабатывают внешние приложения, которые позволяют выполнять над данными все стандартные операции: выборку, удаление и изменение существующей информации, вставку новой информации.

Администратор базы данных (АБД) – под этим понятием подразумевается лицо (или группа лиц, возможно, целое штатное подразделение), на которое возложено управление средствами базы данных организации. В его функции входит:

– координировать все действия по проектированию, реализации и ведению базы данных; учитывать перспективные и текущие требования пользователей; следить, чтобы база данных удовлетворяла актуальным информационным потребностям;

– анализировать существующие программные средства и возможность их использования в базе данных, разрабатывать программно-технические мероприятия по развитию базы данных;

– разрабатывать и реализовывать меры по обеспечению защиты данных от некомпетентного их использования, от сбоев технических средств, по обеспечению секретности определённой части данных и разграничению доступа к данным;

– выполнять работы по ведению словаря данных; контролировать избыточность и противоречивость данных, их достоверность;

– следить за тем, чтобы БД отвечала заданным требованиям по производительности, т.е. чтобы обработка запросов выполнялась за приемлемое время; выполнять при необходимости изменения мето-

дов хранения данных, путей доступа к ним, связей между данными, форматов данных; определять степень влияния изменений в данных на всю БД; координировать вопросы технического обеспечения системы аппаратными средствами исходя из требований, предъявляемых БД к оборудованию;

– координировать работы системных программистов, разрабатывающих дополнительное программное обеспечение для улучшения эксплуатационных характеристик системы;

– координировать работы прикладных программистов, разрабатывающих новые приложения и выполнять проверку и включение прикладных программ в состав программного обеспечения системы;

– работать с конечными пользователями, в том числе и по вопросам обучения и консультирования и т.п.

Программное обеспечение включает, прежде всего, систему управления базами данных (СУБД) – комплекс программных средств, «предназначенный для создания и хранения базы данных на основе некоторой модели данных, обеспечения логической и физической целостности содержащихся в ней данных, надёжного и эффективного использования ресурсов (данных, пространства памяти, вычислительных ресурсов), предоставления к ней санкционированного доступа для приложений и конечных пользователей, а также для поддержки функций администратора баз данных».

Рассмотрим функции СУБД подробнее:

– СУБД должна предоставлять пользователям и программам два типа языков: язык описания данных (для описания логической структуры данных) и язык манипулирования данными (для выполнения запросов пользователя на выборку, изменение, удаление данных);

– СУБД должна контролировать пользовательские запросы и следить за выполнением правил безопасности и целостности, определённых АБД;

– В СУБД должен быть предусмотрен механизм восстановления данных;

– СУБД должна обеспечить функцию словаря данных. Словарь данных содержит «данные о данных», так называемые метаданные. Метаданные словаря предоставляются в виде, удобном для восприятия человеком, и характеризуют состав и структуру пользовательских данных в базе данных. Словарь данных предназначен для документирования разработки системы базы данных, справочного обслуживания её пользователей, а также для персонала АБД.

СУБД является важнейшим, но не единственным компонентом программного обеспечения. Другие программные компоненты: средства разработки приложений, средства проектирования, утилиты, генераторы отчётов и т.д.

База данных не существует без данных. Данные должны быть определённым образом организованы.

С учётом вышеизложенного детализированная архитектура баз данных представлена на рис. 1.3.

1.2. ЖИЗНЕННЫЙ ЦИКЛ БАЗЫ ДАННЫХ

Разработка базы данных представляет итеративный, пошаговый процесс. Каждый шаг ведёт к созданию рабочей базы данных; каждая итерация идёт от моделирования – к созданию структуры, а затем обратно в том порядке, в каком это имеет смысл делать. Проектирование

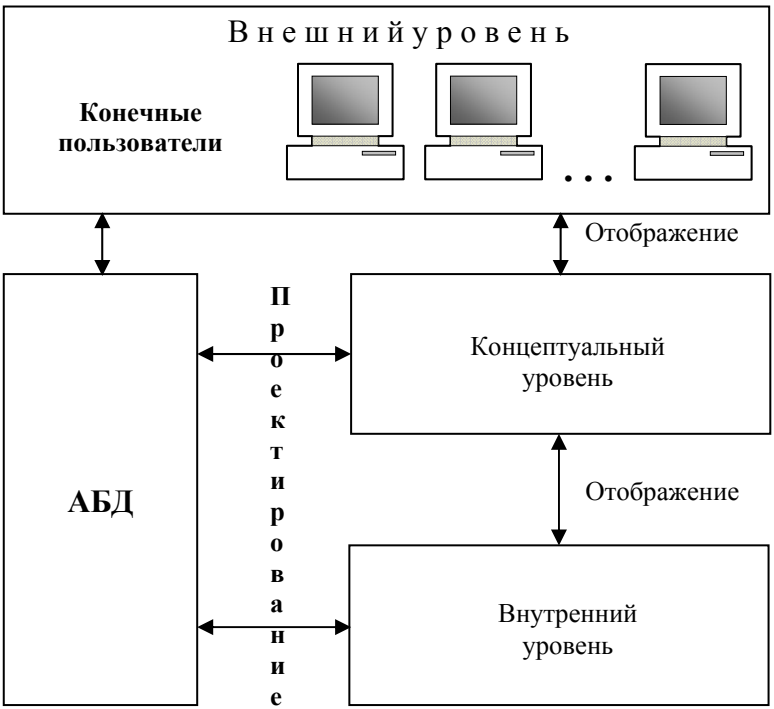


Рис. 1.3. Детализированная архитектура баз данных

базы данных начинается с определения информационных потребностей пользователей, создания модели данных и заканчивается утилизацией базы данных. Процедура создания концептуальной схемы базы данных, определения данных, включаемых в базу данных, создание программ обновления и обработки данных называется *жизненным циклом базы данных* (ЖЦ). Жизненный цикл включает в себя процессы проектирования, реализации и поддержания системы базы данных и состоит из следующих этапов:

- предварительное планирование;
- проверка осуществимости;
- определение требований;
- концептуальное проектирование;
- реализация;
- оценка работы и поддержка базы данных;
- снятие с эксплуатации.

Опишем главные задачи каждого этапа.

Предварительное планирование выполняется в процессе разработки стратегического плана базы данных. Стратегический план предполагает количество и вид баз данных, которые требуется создать для организации. Когда начинается разработка проекта реализации, общая информационная модель, созданная в процессе планирования базы данных, пересматривается и уточняется. Предварительное планирование предполагает определение функций и количества используемых прикладных программ, приложений, находящихся в процессе создания. Эта информация помогает установить связи между текущими приложениями и определить, каким образом используется информация приложений, а также сформулировать будущие требования к системе.

Проверка осуществимости определяет технологическую, операционную и экономическую осуществимость плана создания базы данных. Технологическая осуществимость предполагает определение доступности необходимого оборудования и программного обеспечения, необходимых для работы базы данных: имеются ли в наличии данные ресурсы, или необходимо их приобретение. Операционная осуществимость связана с определением квалификации и опыта специалистов, работающих с БД. Экономическая целесообразность предполагает получение определённой выгоды от внедрения базы данных.

Этап определения требований включает выбор целей базы данных, выяснение информационных потребностей различных подразделений и требований к оборудованию и программному обеспечению. Информационные потребности могут выясняться с помощью анкет, опросов менеджеров и работников компании, а также на основе документов

компании. Общая информационная модель, созданная на этапе планирования базы данных, разделяется на модели для каждого отдела компании, являющиеся основой для создания проекта следующего этапа.

Этап концептуального проектирования включает создание концептуальной схемы базы данных. На этом же этапе создаются подробные модели пользовательских представлений, которые затем преобразуются в концептуальную модель, фиксирующую все элементы данных, которые будет содержать база данных.

В процессе реализации базы данных выбирается и приобретается СУБД. Затем концептуальная модель преобразуется в проект реализации базы данных, создаётся словарь данных, база данных заполняется данными, создаются прикладные программы и обучаются пользователи. Построение словаря данных как центрального хранилища определений структуры данных – ключевой шаг в реализации базы данных. Словарь данных предназначен для системного персонала администрирования данных, прикладных программистов и конечных пользователей.

Оценка базы данных включает опросы пользователей в целях выяснения неучтённых информационных потребностей. При необходимости вносятся изменения, обеспечивается поддержка системы путём добавления новых программ.

Последний этап – снятие с эксплуатации базы данных.

Контрольные вопросы и задания

1. Какие уровни включает архитектура баз данных?
2. Дать определение внутреннего уровня.
3. Указать различие между внешним и концептуальным представлениями базы данных.
4. Какие виды отображений определяются в архитектуре баз данных? Охарактеризовать их.
5. Какие основные компоненты включает система баз данных?
6. Охарактеризовать категории пользователей БД.
7. Перечислить функции администратора баз данных.
8. Нарисовать схему архитектуры баз данных.
9. Перечислить и кратко описать этапы жизненного цикла базы данных.
10. Указать назначение словаря данных.

2. МОДЕЛИ ПРЕДСТАВЛЕНИЯ ДАННЫХ

2.1. КЛАССИФИКАЦИЯ МОДЕЛЕЙ ДАННЫХ

Основная задача построения моделей данных – адекватное описание предметной области для последующего перевода на язык информационных систем. При этом одними из основополагающих в концепции баз данных являются обобщенные категории «данные» и «модель данных».

Категория «данные» тесно связана с понятием «информация». Под *информацией* понимают любые сведения об объектах и явлениях окружающей среды, их параметрах, свойствах и состоянии, которые воспринимают информационные системы (в том числе и живые организмы) в процессе жизнедеятельности и работы. Под *данными* можно понимать информацию, фиксированную в определённой форме, пригодной для последующей обработки, хранения и передачи. Понятие *данные* в концепции баз данных – это набор конкретных значений, параметров, характеризующих объект, условие, ситуацию или любые другие факторы. В энциклопедии технологий баз данных данные определяются как «представление фактов о предметной области системы баз данных или информационной системы в форме, допускающей их хранение и обработку на компьютерах, передачу по каналам связи, а также восприятие человеком». Примеры данных: ткань платьевая «Вечер», $50 и т.д. Для использования данных пользователь должен задать им определённую структуру с учётом смыслового содержания. Поэтому центральным понятием в области баз данных является понятие модели данных. Под моделью понимается представление реальных сущностей, при котором отражаются только некоторые их свойства и которое может материализоваться в различных формах. В теории баз данных используются разные модели: модели предметной области, модели данных, архитектурные модели, модели транзакций. Для их описания используются математический аппарат, графическая нотация, специально разработанные дескриптивные языки и т.д.

Не существует однозначного определения термина «модель данных», у разных авторов эта абстракция определяется с некоторыми различиями.

Модель данных – это некоторая абстракция, которая, будучи приложима к конкретным данным, позволяет пользователям и разработчикам трактовать их как сведения, содержащие не только данные, но и взаимосвязь между ними.

На рисунке 2.1 представлена классификация моделей данных.

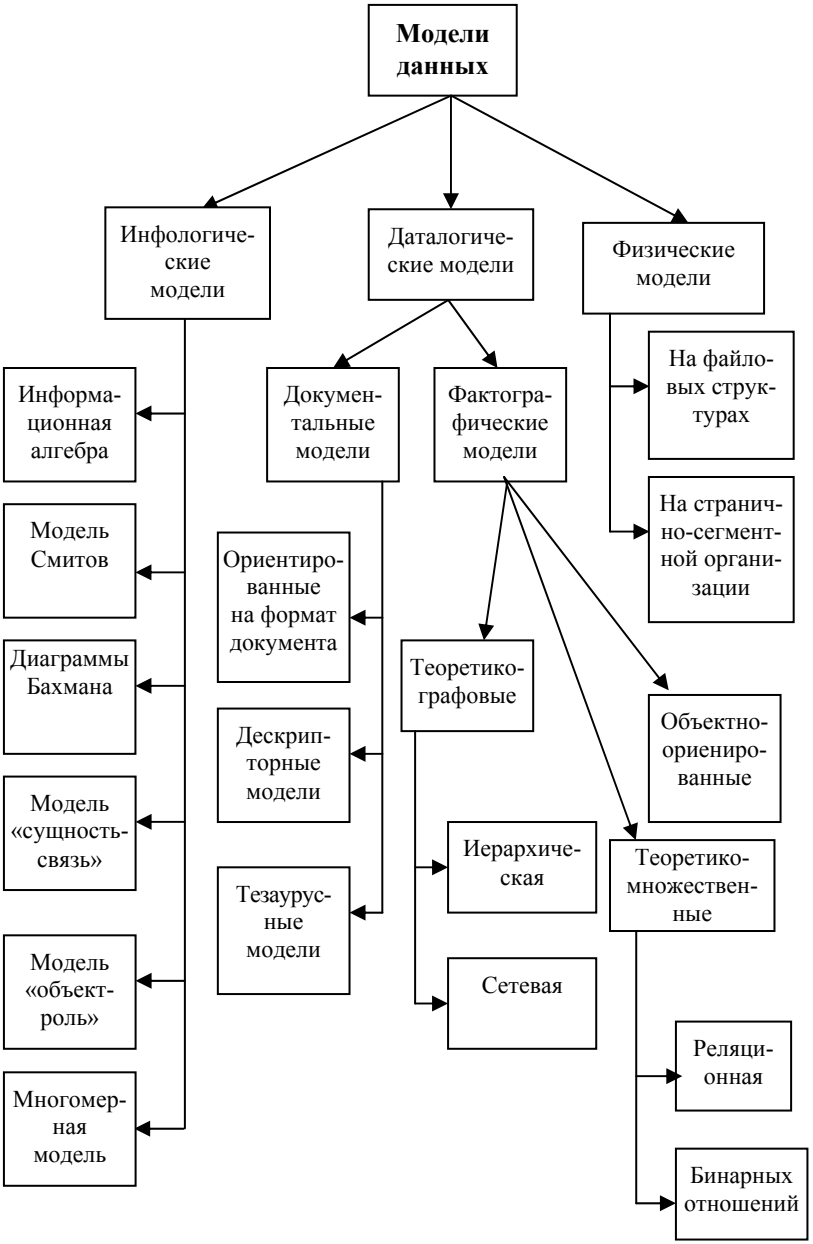


Рис. 2.1. Классификация моделей

В соответствии с рассмотренной ранее трёхуровневой архитектурой ANSI/SPARC можно определить понятие модели данных по отношению к каждому уровню.

Физическая организация данных оказывает основное влияние на эксплуатационные характеристики базы данных. Разработчики пытаются создать наиболее производительные физические модели данных, предлагая пользователям тот или иной инструментарий для поднастройки модели под конкретную СУБД. *Физическая* модель данных оперирует категориями, касающимися организации внешней памяти и структур хранения, используемых в данной операционной среде. В настоящий момент в качестве физических моделей используются различные методы размещения данных, основанные на файловых структурах: это организация файлов прямого и последовательного доступа, индексных файлов и инвертированных файлов, файлов, использующих различные методы хеширования, взаимосвязанных файлов. Кроме того, современные СУБД широко используют страничную организацию данных. Физические модели данных, основанные на страничной организации, являются наиболее перспективными.

Модели данных, используемые на концептуальном уровне, характеризуются большим разнообразием. По отношению к ним внешние модели называются подсхемами и используют те же абстрактные категории, что и концептуальные модели данных.

При проектировании базы данных выделяют ещё один уровень, предшествующий рассмотренным трём уровням. Модель этого уровня должна выражать информацию о предметной области в виде, независимом от используемой СУБД. Эти модели называются *инфологическими*, или *семантическими*. Инфологическая модель отображает реальный мир в некоторые, понятные человеку концепции, полностью независимые от параметров среды хранения данных. Термин «инфологическая модель» означает «концептуальная модель в самом широком смысле», многие авторы вместо него используют термины «абстрактная модель», «концептуальная в широком смысле модель».

Инфологические модели данных используются на ранних стадиях проектирования для описания структур данных в процессе разработки приложений. Инфологическая модель должна быть отображена в компьютерно-ориентированную *даталогическую* модель, поддерживаемую конкретной СУБД. В даталогическом аспекте рассматриваются вопросы представления данных в памяти информационной системы.

Документальные модели данных применяются для представления слабоструктурированной информации, ориентированной в основном на свободные форматы документов, текстов на естественном языке (монографий, публикаций в периодике, текстов законодательных актов и т.п.).

Модели, ориентированные на формат документа, связаны, прежде всего, с *языками разметки документов.* Стандартный обобщённый язык разметки документов SGML (Standard Generalized Markup Language) позволяет отделить аспекты содержания документов от аспектов его представления. Описание документов в этом языке не зависит от программно-аппаратной платформы, на которой осуществляется их обработка. На его основе разработаны программные системы управления документами, а также браузеры для просмотра размеченных документов. На основе языка SGML разработан язык гипертекстовой разметки HTML (Hyper Text Markup Language), применяемый для отображения статистических данных на Web-страницах. Этот язык определяет оформление элементов документа и имеет некий ограниченный набор инструкций – тегов, при помощи которых осуществляется процесс разметки. В качестве элемента гипертекстовой базы данных, описываемой HTML, используется текстовый файл, который может легко передаваться по сети с использованием протокола HTTP. Эта особенность, а также то, что HTML является открытым стандартом и огромное количество пользователей имеет возможность применять возможности этого языка для оформления своих документов, безусловно, повлияли на рост популярности HTML и сделали его сегодня главным механизмом представления информации в Интернете.

Однако HTML сегодня уже не удовлетворяет современным требованиям. И ему на смену был предложен новый язык гипертекстовой разметки XML (Extensible Markup Language). Язык XML стал основой активно развивающейся новой более перспективной и совершенной платформы для среды Web. Он используется в качестве средства для описания грамматики других языков и контроля за правильностью составления документов. Сам по себе XML не содержит никаких тегов, предназначенных для разметки, он просто определяет порядок их создания.

Тезаурусные модели основаны на принципе организации словарей и содержат определённые языковые конструкции и принципы их взаимодействия в заданной грамматике. Эти модели эффективно используются в системах-переводчиках, особенно многоязыковых переводчиках. Принцип хранения информации в этих системах подчиняется тезаурусным моделям.

Дескрипторные модели – самые простые из документальных моделей, они широко использовались на ранних стадиях использования документальных баз данных. В этих моделях каждому документу соответствовал дескриптор – описатель. Этот дескриптор имел жёсткую структуру и описывал документ в соответствии с теми характеристи-

ками, которые требуются для работы с документами в разрабатываемой документальной БД. Например, для БД, содержащей описание патентов, дескриптор содержал название области, к которой относился патент, номер патента, дату выдачи патента и ещё ряд ключевых параметров, которые заполнялись для каждого патента. Обработка информации в таких базах данных велась исключительно по дескрипторам, т.е. по тем параметрам, которые характеризовали патент, а не по самому тексту патента.

Фактографические модели представлены в виде специальным образом организованных совокупностей формализованных записей данных. Основанные на таких моделях фактографические системы используются не только для реализации информацинно-справочных функций (в отличие от документальных систем), но и для решения задач обработки данных. Под обработкой данных понимается специальный класс решаемых на ЭВМ задач, связанных с вводом, хранением, сортировкой, отбором и группировкой данных однородной структуры. Центральным функциональным звеном фактографических систем является СУБД.

2.2. РАЗНОВИДНОСТИ ИНФОЛОГИЧЕСКИХ МОДЕЛЕЙ ДАННЫХ

Информационная алгебра была разработана рабочей группой комитета CODASYL. Целью данной работы являлось создание структуры машинно-независимого языка описания задач, ориентированного на системный уровень обработки данных. Основой концептуальной модели является представление, что информационная система имеет дело с объектами и событиями реального мира, которые представляются в виде данных.

Информационная алгебра оперирует понятиями «сущность» и «свойства». *Сущность* – нечто физически существующее в реальном мире. Сущность имеет *свойства*. Для каждой сущности каждому из её свойств приписывается *значение* из множества значений *свойства*. Также в информационной алгебре вводится ряд понятий: *система координат*, *точка значений*, *пространство свойств*. Вводятся также понятия *комплекса* и *агрегата*.

При описании задач основной операцией является *отображение* одного подмножества пространства свойств на другое. Рассматриваются два типа отображений. Одно соответствует операциям в данном файле (можно, например, определить отображение группы из пяти точек для каждого рабочего в одну новую точку, которая будет содержать итог за неделю). Эта операция называется *агрегированием*. Второй тип отображения соответствует операции обработки файлов, при

которой точки из некоторого числа входных файлов обрабатываются для получения нового выходного файла – *комплексирование данных*.

Создатели информационной алгебры надеялись, что развиваемый подход приведёт к появлению трансляторов, позволяющих переводить реляционные выражения в процедурную форму. Проектировщики должны специфицировать релевантные (существенные для задачи) наборы данных, а также связи и правила их объединения, в соответствии с которыми данные обрабатываются, классифицируются и объединяются в различные подмножества, в том числе в выходные результаты.

Важно заметить, что в модели информационной алгебры выделены основополагающие элементы: *сущности*, *свойства*, *значения свойства*, которые позволяют адекватно описать некоторую предметную область и реально существующие объекты. Выделение этих элементов следует считать важным достижением. Здесь выделены три основные составляющие, присущие природе данных: носитель свойств («сущность»), сами свойства («свойства»), каждому из которых приписывается значение («значение свойства»), также вводится понятие пространства – «пространство свойств».

Однако данная модель не отражает ряд важных моментов, присущих природе данных, в частности не учитывается, что:

– носители свойств могут находиться в определённых отношениях друг с другом и образовывать некоторую структуру;

– сами свойства между собой также могут находиться в некоторых отношениях.

В модели CODASYL введено понятие *значения показателя*, при этом также необходимо указать некоторую характеристику упорядочения, такую как номер наблюдения, дату измерения показателя, время и т.п., чтобы была возможность различать ряд значений определённого объекта по определённому показателю. Данная характеристика, ввиду её природой естественности, в модели CODASYL присутствует неявно и входит в свойства. Выделение характеристики упорядочения позволяет при необходимости исследовать динамику изменения показателей, задать отношения: быть позже, быть раньше, относиться к одному интервалу (к неделе, месяцу, году), отстоять на определённый промежуток времени и др.; ввести операции усреднения по интервалам и подсчёта итоговых сумм за период. То есть использование характеристики упорядочения даёт возможность в значительной мере автоматизировать процесс подготовки данных для последующего анализа, в том числе статистического. Существует целое направление так называе-

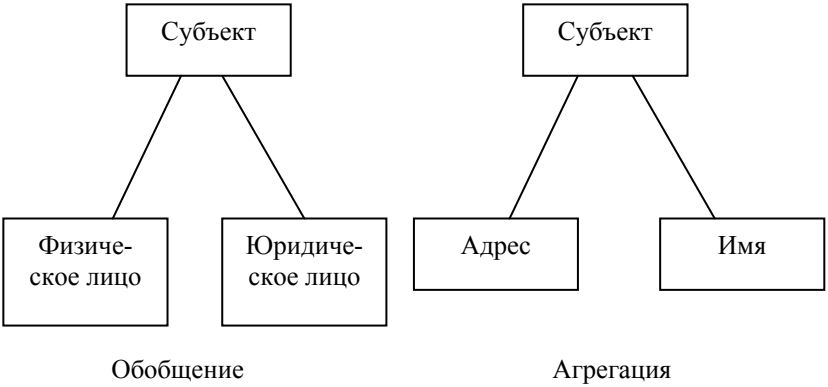


Рис. 2.2. Два вида связей в модели Смитов

мых временных (temporal) баз данных, учитывающих изменения данных во времени.

Модель Смитов. В семантическом моделировании проектируется схема понятий прикладной области в их взаимосвязи. Предлагались и предлагаются различные пути такого моделирования. Вот, например, какие метапонятия рассматривали для концептуального моделирования в конце 1970-х гг. Дж. Смит и Д. Смит.

Исходными базовыми понятиями в трактовке этих двух специалистов являются *объекты* и *связи* между объектами. Связи могут быть двух видов: *обобщение* и *агрегация* (рис. 2.2).

Обобщение интуитивно ясно, и связывает одни объекты с другими, по смыслу более общими. Например, объект «животное» есть обобщение для объектов «собака» и «лошадь». Агрегация связывает разнородные объекты по признаку компонентного вхождения в другие объекты, как например, «колеса» и «кузов» связаны с «автомобилем» тем, что последний состоит из первых. Независимо оба вида связей образуют каждый свою иерархию среди объектов модели.

Кроме этих базовых имеются и другие понятия концептуальной модели, как-то атрибут, отношение, экземпляр, индивид. Самое замечательное в модели Смитов – это относительность перечисленных понятий. Одно и то же явление может быть и объектом, и отношением, и атрибутом, и экземпляром, и индивидом, и всё определяется точкой зрения на явление. Зависимость интерпретации от точки зрения на явление (а точнее – возможность выбора точек зрения с разной интерпретацией) – это очень мощное свойство, придающее концептуальной модели большую гибкость и приспособляемость в описании проекти-

руемой ИС. Это свойство, например, будь оно реализовано, позволило бы в информационной системе смотреть на «адрес» то как на объект реестра адресов, то как на атрибут «лица», то как на отношение, связывающее владельца с остальными жильцами – когда, где и кому как нужно. Наиболее близко к концептуальной в этом отношении подошла (теоретическая) реляционная модель данных, а вот объектный подход с его фиксированной интерпретацией структуры отстоит от реляционного на шаг назад.

В модели Смитов выделяются две иерархии – иерархия агрегаций (отношение разнородных объектов) и иерархия обобщений (по типу «собака, лошадь – животное»), в точках пересечения появляются абстрактные объекты. Вводится также ряд понятий: индивиды, категории, компоненты. Для успешной интеграции понятий существует «принцип относительности объектов», который утверждает, что индивиды, категории, отношения и компоненты – разные способы рассмотрения одних и тех же объектов. Разработана методология спецификаций, основанная на принципах относительности объектов и сохранения индивидов. Отказ от чёткого разграничения ролей объектов является одновременно и сильной и слабой стороной данной модели. Слабые стороны данной модели проявляются в тех случаях, когда можно чётко разделить объекты (носители свойств) и свойства, что характерно для систем статистической обработки. Можно отметить, что не существует формализма, позволяющего отличить объект от свойства, поэтому это должно задаваться извне для построения более конкретной модели.

Модель Бахмана напоминает навигационную модель страниц и ссылок сегодняшнего Интернета, её иногда называют моделью навигации данных. На диаграммах Бахмана изображают типы записей и связи между типами записей.

Следует учитывать, что это одна из первых инфологических моделей. Чарльз Бахман в GE (General Electric) построил прототип системы навигации по данным. За руководство работы инициативной группой DBTG, разработавшей стандартный язык определения данных и манипулирования данными, Бахман получил Тьюринговскую премию. В своей Тьюринговской лекции он описал эволюцию моделей плоских файлов к новому миру, где программы могут осуществлять навигацию между записями, следуя связям между записями. Идеологическая основа работы Бахмана (более «научно» называемая моделью базы данных) IDS, за которую Бахман заслуженно был удостоен в 1973 г. высшей компьютерной награды ACM, получила название «сетевой» (network).

Модель «сущность-связь». Наиболее популярной семантической моделью является модель «сущность-связь» (E/R – Entity/Relationship), предложенная Питером Пин-Шен Ченом в 1976 г. На использовании разновидностей E/R модели основано большинство современных подходов к проектированию баз данных (в основном реляционных). Данная модель имеет графическую природу, в ней используются изображения в виде диаграмм с прямоугольниками и стрелками, представляющие главные элементы данных и их связи. В данной модели выделены *объекты* (объектом называется «предмет, который может быть чётко идентифицирован») и *свойства* объектов. Таким образом, определяются отношения типа «объект-свойство». В связи с наглядностью представления данных модели «сущность-связь» получили широкое распространение в CASE-системах.

Объектная модель – логическая схема объектной БД в одной из общепринятых систем описания (обозначений). Хотя, по выражению К. Дж. Дейта, «не существует общепринятой, абстрактной и формально определённой "объектной модели данных"» и применительно к «объектной "модели"» правильнее говорить об «удобном ярлыке для целой совокупности некоторых взаимосвязанных идей», конкретные CASE-системы, реализующие на свой лад некоторые из этих идей, всё же существуют.

Объектная схема – схема БД конкретной объектной СУБД, для описания модели данных используются основные принципы объектно-ориентированного программирования.

Многомерная схема – схема данных в одной из многомерных систем представления данных. Данные представляются посредством гиперкуба (некоторого куба со множеством измерений).

Информационные системы масштаба предприятия, как правило, содержат приложения, применяемые менеджерами высшего звена и предназначенные для комплексного многомерного анализа данных, их динамики, тенденций и т.п. Такой анализ в конечном итоге призван способствовать принятию решений. Нередко эти системы так и называются – системы поддержки принятия решений (DSS). Указанные приложения обычно обладают средствами предоставления пользователю агрегатных данных для различных выборок из исходного набора в удобном для восприятия и анализа виде. Чаще всего такие агрегатные функции образуют многомерный (а, следовательно, нереляционный) набор данных (нередко называемый гиперкубом или метакубом, оси которого содержат параметры, а ячейки – зависящие от них агрегатные данные). Пример многомерной модели показан на рис. 2.3.

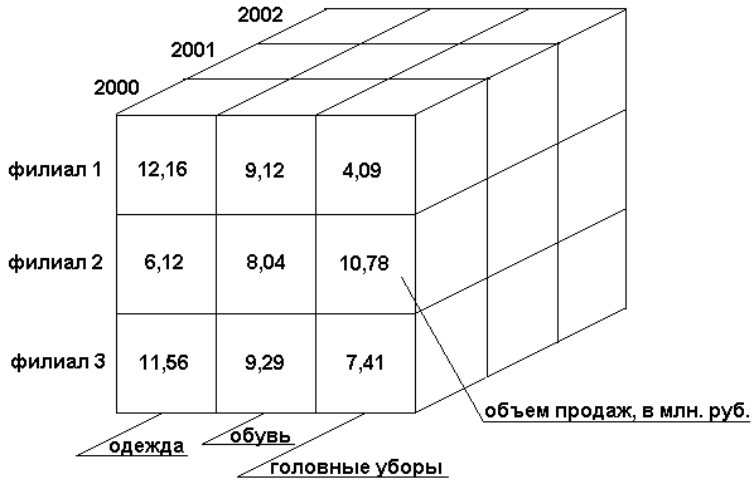


Рис. 2.3. Пример трёхмерной модели

С середины 1990-х гг. интерес к многомерным моделям стал приобретать массовый характер. Многомерные системы позволяют оперативно обрабатывать информацию для проведения анализа и принятия решения. Многомерные СУБД предназначены для интерактивной аналитической обработки. По сравнению с реляционной моделью многомерная организация данных обладает более высокой наглядностью и информативностью.

К основным понятиям многомерных моделей относятся *измерение* и *ячейка*. Измерение образуют множество однотипных данных, образующих одну из граней гиперкуба. Ячейка – это поле, значение которого однозначно определяется фиксированным набором значений. Тип поля определён как цифровой.

Вдоль каждого измерения данные могут быть организованы в виде иерархии, отражающей различные уровни их детализации. Благодаря такой модели данных пользователи могут формулировать сложные запросы, генерировать отчёты, получать подмножества данных.

При использовании более трех измерений представить и изобразить такой куб в рамках 3-мерного пространства, ограниченного высотой, шириной и глубиной, невозможно. В данном случае разработчики применяют специальные методы для отображения неотображаемого, например показ нескольких последовательностей (series) на одном графике. Каждая последовательность закрашивается отдельным цве-

том. Группа последовательностей представляет собой значение одного 4-го измерения.

Технология комплексного многомерного анализа данных получила название OLAP (On-Line Analytical Processing). Концепция OLAP была описана в 1993 г. известным исследователем баз данных и автором реляционной модели данных Э. Ф. Коддом.

Нередко для повышения скорости выполнения запросов пользователей данные кубов вычисляются заранее и хранятся в многомерной базе данных.

Отметим, что многомерный анализ данных может быть осуществлён как в клиентском приложении, так и на сервере баз данных. Все производители ведущих серверных СУБД (IBM, Informix, Microsoft, Oracle, Sybase) производят серверные средства для такого анализа.

Существует мнение, причём вполне обоснованное, что многомерные модели используются не для описания данных, а для их представления, так как «универсализация» отношений приводит к «потере точности» описания, но зато и к удобству восприятия информации конечным пользователем. Таким образом, значение многомерных схем – преимущественно «интерфейсное» (они удобны конечным пользователям), а не описательное.

Объект-роль – модель концептуального описания, принятая в системе Info Modeler фирмы Visio. В этой системе для модели «объект-роль» используется два языка: графический и [условно-] естественный.

В данной модели предполагается отсутствие принципиального различия между объектами и свойствами, в ряде случаев они могут меняться местами, всё зависит от «роли», которой исполняет объект при определённом описании.

Приведённая классификация не является законченной и всеобъёмлющей. Можно выделить ряд «гибридных» моделей, имеющих отношения к разным классам. Например, RM/T – расширенная реляционная модель, предложенная в 1979 г. Коддом для «лучшего учёта семантики» прикладной области. В отличие от реляционной модели, RM/T вообще не получила никакого воплощения в реальных системах, но она также имеет большое методологическое значение.

Контрольные вопросы и задания

1. Определить понятие «модель данных».
2. Привести классификацию моделей данных согласно архитектуре ANSI/SPARC.

3. Описать физические модели данных.
4. Дать характеристику инфологическим моделям.
5. Охарактеризовать документальные модели данных.
6. Какие языки используются для описания моделей, ориентированных на формат данных?
7. Какой принцип положен в основу тезаурусных моделей?
8. Охарактеризовать дескрипторные модели.
9. Каким образом представлены фактографические модели?
10. Какими понятиями оперирует информационная алгебра?
11. В чём различие операций агрегирования и комплексирования данных?
12. Определить особенности модели «объект-роль».
13. В чем заключаются достоинства E/R-модели?
14. Описать модель Смитов.
15. Указать особенности модели Бахмана.
16. За какую работу Ч. Бахман получил Тьюринговскую премию?
17. Кем была разработана модель «сущность-связь»?
18. Привести пример многомерной модели.
19. Охарактеризовать основные понятия многомерной модели.
20. В чём суть OLAP-технологии?

3. ДАТАЛОГИЧЕСКИЕ МОДЕЛИ ДАННЫХ

Хранимые в БД данные описываются различными моделями представления данных. К классическим (традиционным) моделям относятся: сетевая, иерархическая, реляционная.

Сетевая и иерархическая модель являются историческими предшественниками реляционной. Все ранние (дореляционные) системы не основывались на каких-либо абстрактных моделях, понятие *модели данных* фактически появилось в лексиконе специалистов в области БД только вместе с реляционным подходом (в 1970-е гг.). В ранних системах доступ к БД производился на уровне записей. Пользователи этих систем осуществляли явную навигацию в БД, используя языки программирования, расширенные функциями СУБД. Интерактивный доступ к БД поддерживался только путём создания соответствующих прикладных программ с собственным интерфейсом.

После появления реляционных систем большинство ранних систем было оснащено «реляционными» интерфейсами. Однако в большинстве случаев это не сделало их по-настоящему реляционными системами, поскольку оставалась возможность манипулировать данными в естественном для них режиме.

СУБД, основанные на сетевой и иерархической моделях, используются и в настоящее время.

3.1. ИЕРАРХИЧЕСКИЕ МОДЕЛИ

Иерархическая модель данных была создана в 1960-х гг. как отражение потребностей практики. Иерархическая модель состоит из упорядоченного набора экземпляров типа *дерево* (рис. 3.1).

Тип «дерево» является составным. Он может включать в себя подтипы («поддеревья»), также являющиеся типом «дерево». Тип «дерево» состоит из вершины («корневого» типа) и упорядоченного набора из нуля или более типов «поддеревьев». Каждый из элементарных типов, включённых в тип «дерево», является простым или составным типом «запись». Простая «запись» состоит из одного типа, например, символьного, а составная «запись» объединяет некоторую совокупность различных типов, например числовые и символьные. Пример типа «дерево» (схемы иерархической БД) приведён на рис. 3.2.

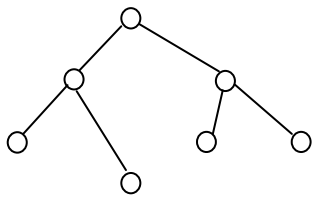


Рис. 3.1. Дерево

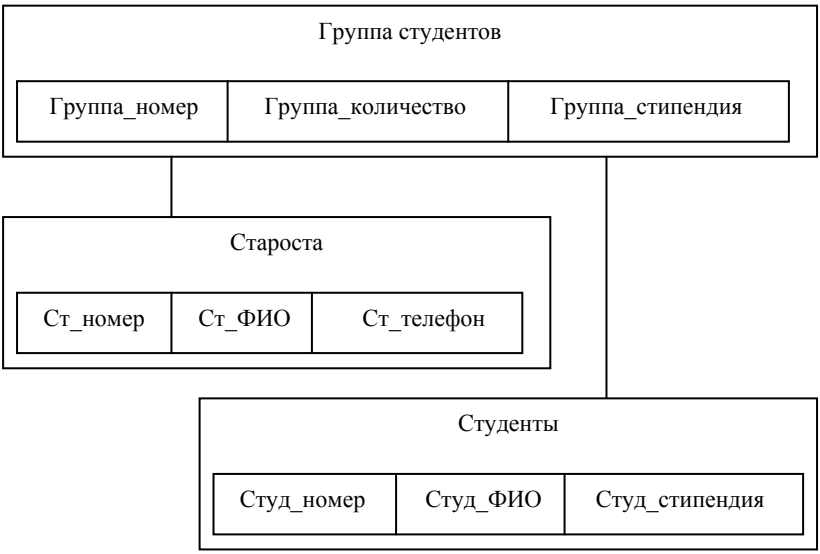


Рис. 3.2. Пример типа «дерево»

Корневой тип имеет подчинённые типы и сам не является подтипом (подчинённым типом). Подтип является *потомком* по отношению к *предку* (родителю).

В примере, приведённом на рис. 3.2, *Группа* является предком для *Староста* и *Студенты*, а *Староста* и *Студенты* – потомки *Группа*. Между типами записи поддерживаются связи.

Тип «дерево» в целом представляет собой иерархически организованный набор типов «запись». База данных представляет совокупность таких деревьев. Пример данных в структуре рис. 3.2 приведён на рис. 3.3.

Потомки одного и того же типа являются *близнецами*. В иерархической модели предусмотрены навигационные операции по структуре базы данных и операции манипулирования данными. Например, могут быть следующие операции:

- найти указанное «дерево» БД (например, Группу 21 ИТ);
- перейти от одного *дерева* к другому;
- перейти от одной записи к другой внутри *дерева* (например, к следующей записи типа *Студенты*);
- перейти от одной записи к другой в порядке обхода иерархии;
- вставить новую запись в указанную позицию;
- удалить текущую запись.

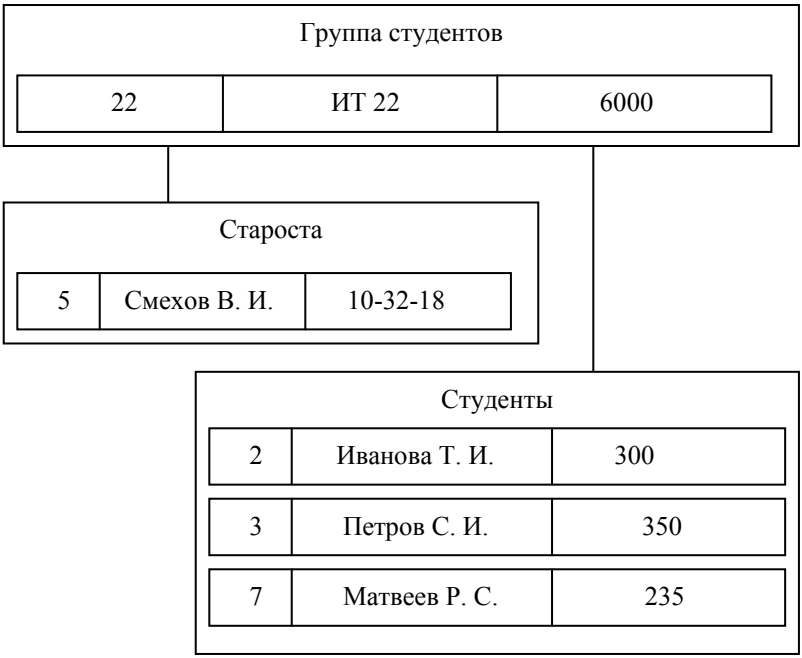


Рис. 3.3. Пример данных в базе данных

Между предками и потомками автоматически поддерживается целостность ссылок. Основное правило: никакой потомок не может существовать без своего родителя, у некоторых родителей не может быть потомков.

Иерархическая модель данных активно использовалась во многих СУБД до появления реляционных систем.

«Живучесть» иерархических моделей обусловлена тем, что многие структуры данных естественным образом иерархичны (например, в области биологии: виды, классы, группы и т.д.).

3.2. СЕТЕВЫЕ МОДЕЛИ

Концепция сетевой модели данных была сформулирована в конце 1960-х гг. в Предложениях группы CODASYL, и с тех пор на неё ссылаются как на модель CODASYL DTBG. Сети представляют естественный способ представления отношений между объектами. Они широко применяются в математике, физике, социологии и других облас-

тях знаний. Сетевой подход к организации данных является расширением иерархического. В иерархических структурах запись-потомок должна иметь одного предка; в сетевой структуре данных потомок может иметь любое число предков. Возможные связи в сетевой модели представлены на рис. 3.4.

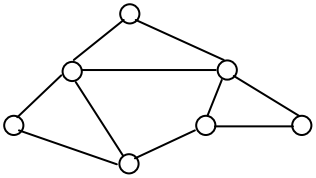


Рис. 3.4. Сети

Сетевая БД состоит из набора записей и набора связей между этими записями. Тип связи определяется для двух типов записи: предка и потомка. Экземпляр типа связи состоит из одного экземпляра типа записи предка и упорядоченного набора экземпляров типа записи потомка. Для данного типа связи С с типом записи предка Р и типом записи потомка РТ должны выполняться следующие два условия:

1) каждый экземпляр типа Р является предком только в одном экземпляре С;

2) каждый экземпляр РТ является потомком не более чем в одном экземпляре С.

На формирование типов связи не накладываются особые ограничения, т.е. возможны, например, следующие ситуации:

– тип записи потомка в одном типе связи может быть типом записи предка в другом типе связи (как в иерархии);

– данный тип записи Р может быть типом записи предка в любом числе типов связи;

– данный тип записи Р может быть типом записи потомка в любом числе типов связи;

– может существовать любое число типов связи с одним и тем же типом записи предка и одним и тем же типом записи потомка; и если L1 и L2 – два типа связи с одним и тем же типом записи предка Р и одним и тем же типом записи потомка С, то правила, по которым образуется родство, в разных связях могут различаться;

– типы записи X и Y могут быть предком и потомком в одной связи и потомком и предком – в другой;

– предок и потомок могут быть одного типа записи.

Пример сетевой схемы БД приведён на рис. 3.5.

Примерный набор операций для манипулирования данными может быть следующим:

– найти конкретную запись в наборе однотипных записей (студента Петрова С. И.);

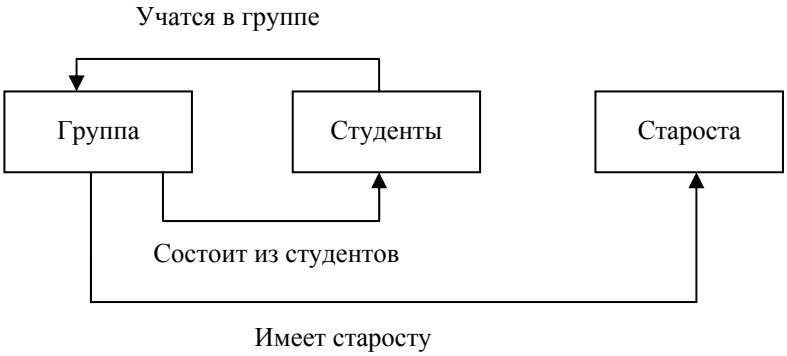


Рис. 3.5. Пример сетевой БД

– перейти от предка к первому потомку по некоторой связи (к первому студенту группы 22 ИТ);

– перейти к следующему потомку в некоторой связи (от Петрова С. И. к Матвееву Р. С.);

– перейти от потомка к предку по некоторой связи (найти группу Ивановой Т. И.);

– создать новую запись;

– удалить запись;

– изменить запись;

– включить в связь;

– исключить из связи;

– переставить в другую связь и т.д.

В принципе поддержание целостности связей не требуется, но иногда требуют целостности по ссылкам (как в иерархической модели).

Системы на основе сетевой модели не получили широкого распространения на практике.

3.3. РЕЛЯЦИОННЫЕ МОДЕЛИ

Реляционный подход формировался в 1970-е гг. Публикация известной статьи Э. Ф. Кодда о реляционной модели данных (в июне 1970 г.) стимулировала целый ряд математических исследований, направленных на изучение свойств реляционных баз данных, впоследствии приведших к созданию теории реляционных баз данных. В своих работах Э. Ф. Кодд описал математические основы реляционной модели. Основная идея Кодда состояла в том, чтобы применить концепцию

математических отношений к моделированию данных. Отношение – это таблица из строк и столбцов. Наиболее важные характеристики реляционной модели заключаются в следующем:

– Описание данных производится в соответствии с их естественной структурой, т.е. не требуется добавления дополнительных структур для машинного представления.

– Обеспечивается независимость данных от их физического представления, от связей между данными и от соображений реализации.

– Модель обеспечивает строгую математическую основу для интерпретации выводимости, избыточности и непротиворечивости отношений.

В это же время компания IBM начала разработку нескольких исследовательских проектов и программных продуктов-прототипов, нацеленных на создание реляционных СУБД. Благодаря успешному созданию исследовательских прототипов, реляционная теория стала активно продвигаться на практике. В 1980-х гг. было создано большое количество коммерческих реляционных продуктов – от компаний IBM, Oracle Corp., Ingres Corp. и других поставщиков. В настоящее время реляционные СУБД доминируют на рынке инструментальных средств разработки систем баз данных.

Реляционные системы продолжают совершенствоваться и их внутренняя природа значительно меняется. В связи с распространением объектного подхода в 1990-е гг. появились объектно-реляционные модели. Расширяется и сфера применения реляционных систем.

Примером развития реляционной модели является постреляционная модель. Классическая реляционная модель предполагает неделимость данных, постреляционная модель снимает ограничение неделимости данных, допускает многозначные поля, состоящие из множества значений. Набор значений многозначных полей считается отдельной таблицей, встроенной в основную таблицу (табл. 3.1).

3.1. Постреляционная таблица

Наименование материала	Тип	Количество
Ткань пальтовая	п/ш	300
	ч/ш	200
Ткань костюмная	п/э	150
	п/ш	500

На длину и количество полей в записях не накладывается требование постоянства, поэтому структура данных и таблиц имеет большую гибкость.

Достоинством постреляционной модели является возможность представления совокупности связанных реляционных таблиц одной постреляционной таблицей. Это обеспечивает высокую наглядность представления информации и повышение эффективности её обработки. К недостаткам можно отнести сложность решения проблемы обеспечения целостности и непротиворечивости хранимых данных.

3.3.1. Основные понятия реляционной модели

Реляционная модель определяется тремя аспектами данных: структурой данных (объектами данных), целостностью данных и обработкой данных. Основными понятиями, описывающими структуру данных в реляционной модели, являются: отношение, кортеж, атрибут, домен, первичный ключ.

Под ***отношением*** будем понимать двумерную таблицу, содержащую некоторые данные о рассматриваемой предметной области.

Кортеж представляет упорядоченный набор элементов (соответствует строке этой таблицы).

Атрибут соответствует столбцу таблицы. Количество атрибутов называется ***степенью*** отношения.

Домен представляет множество всех значений для определённых атрибутов отношения.

Первичный ключ – уникальный идентификатор отношения, однозначно определяющий каждый кортеж.

В реляционной теории определяются свойства отношений:

– Отношение не должно содержать одинаковых кортежей. Важным следствием этого свойства является то, что каждую строку можно однозначно определить с помощью набора атрибутов, составляющих первичный ключ.

– Кортежи не упорядочены сверху вниз.

– Атрибуты не упорядочены слева направо.

– Все значения атрибутов должны быть *атомарными* (простыми), т.е. не допускать группы значений в одном столбце одной строки (не расчленять значения).

Реляционная модель данных была предложена в 1970 г. математиком Эдгаром Коддом (Codd E. F.). РМД является наиболее широко распространённой моделью данных и единственной из трёх основных моделей данных, для которой разработан теоретический базис с использованием теории множеств.

Домен 1	Домен 2	**Домен 3 (ключ)**	Домен 4	Домен 5
Группа	*ФИО студента*	*Номер зачётной книжки*	*Год рождения*	*Размер стипендии*
С-72	Волкова Елена Павловна	С-12298	1991	1550,00
С-91	Белов Сергей Юрьевич	С-12299	1990	1400,00
• • •				
С-72	Фролов Юрий Вадимович	С-14407	1991	0

Рис. 3.6. Пример табличной формы представления отношения

Отношение удобно представлять как таблицу, где строка является кортежем, а столбец соответствует домену (рис. 3.6, отношение *СТУДЕНТЫ*). Количество строк в таблице (кортежей в отношении) называется мощностью отношения, количество столбцов (атрибутов) – арностью.

Отношение имеет имя, которое отличает его от имён всех других отношений. Атрибутам реляционного отношения назначаются имена, уникальные в рамках отношения. Обращение к отношению происходит по его имени, а обращение к атрибуту – по имени отношения и имени атрибута.

Каждый атрибут определён на некотором домене, несколько атрибутов отношения могут быть определены на одном и том же домене (например, номера рабочего и домашнего телефонов). Домен задаётся типом данных, размером и ограничениями целостности: например, пол – это символьное поле длиной 1, которое может принимать значения из множества ('м', 'ж'). В реляционных базах данных поддерживаются такие типы данных, как символьный, числовой, дата и некоторые другие (конкретный перечень типов зависит от СУБД).

Атрибут может быть обязательным и необязательным. Значение обязательного атрибута должно быть определено в момент внесения данных в БД. Если атрибут необязательный, то для таких случаев предусмотрено специальное значение – NULL, которое можно интерпретировать как «неизвестное значение». Значение NULL не привязано к определённому типу данных, т.е. может назначаться данным любых типов.

Перечень атрибутов отношения с их типами данных и размерами определяют схему отношения. Отношения, построенные по одинако-

вой схеме, называют односхемными; по различным схемам – разносхемными.

Ключ отношения – это атрибут (группа атрибутов), значения которого классифицируют или идентифицируют кортеж. Например, значение атрибута *Группа* отношения *СТУДЕНТЫ* позволяет выделить среди всех студентов института студентов конкретной группы. Если ключ состоит из нескольких атрибутов, он называется *составным.* Если значения ключа уникальны в рамках столбца отношения, то такой ключ называется *потенциальным.* Потенциальных ключей может быть несколько (или не быть ни одного), но для отношения выделяется один основной ключ – первичный. Первичный ключ идентифицирует экземпляр сущности, его значение должно быть уникальным (*unique*) и обязательным (*not null*). (На рисунке 3.6 первичный ключ выделен полужирным шрифтом) Неуникальные ключи ещё называют *вторичными.*

РМД не поддерживает групповые отношения (по версии CODASYL). Для связей между отношениями используются внешние ключи. Внешний ключ (*foreign key*) – это атрибут подчинённого (дочернего) отношения, который является копией первичного (*primary key*) или уникального (*unique*) ключа родительского отношения. (Пример – отношение *ОЦЕНКИ*, связанное с отношением *СТУДЕНТЫ* внешним ключом *Номер зачётной книжки*, рис. 3.7)

Номер зачётной книжки	Дисциплина	Оценка
С-12298	Программирование	5
С-12298	Дискретная математика	4
С-14407	Программирование	3

Рис. 3.7. Связь отношений «Оценки» и «Студенты» по внешнему ключу

Если связь необязательная, то значение внешнего ключа может быть неопределённым (*null*).

Фактически внешние ключи логически связывают экземпляры сущностей разных типов (родительской и подчинённой сущностей).

Внешний ключ – это ограничение целостности, в соответствии с которым множество значений внешнего ключа является подмножеством значений первичного или уникального ключа родительской таблицы.

Ограничение целостности по внешнему ключу проверяется в двух случаях:

- при добавлении записи в подчинённую таблицу СУБД проверяет, что в родительской таблице есть запись с таким же значением первичного ключа;
- при удалении записи из родительской таблицы СУБД проверяет, что в подчинённой таблице нет записей с таким же значением внешнего ключа.

Примечание: внешний ключ может ссылаться на первичный ключ этой же таблицы. Это позволяет описывать унарную связь – иерархию однотипных сущностей. Например, если в таблицу *СОТРУДНИКИ* добавить поле *Руководитель* и описать его как внешний ключ на эту же таблицу, то в этом поле будет храниться идентификатор руководителя данного сотрудника (рис. 3.8). Атрибут *Руководитель* является необязательным.

Табельный номер	№ отдела	ФИО	Должность	Руководитель
002	1	Сухов К. А.	Директор	
034	1	Петрова К. В.	Секретарь	002
988	2	Рюмин В. П.	Начальник отдела	002
909	2	Серова Т. В.	Вед. программист	988

Рис. 3.8. Внешний ключ «Руководитель», ссылающийся на первичный ключ этой же таблицы

Все операции над данными в РМД выполняются над отношением и требуют задания имени отношения. Если операция применяется к части отношения, то может потребоваться идентификация кортежа или группы кортежей и задание имён атрибутов. В РМД используются следующие операции:

- *запомнить*: внесение информации в БД (требует формирования значений уникального ключа и обязательных атрибутов кортежа);
- *извлечь*: чтение данных;
- *обновить*: модификация данных – изменение значений атрибутов кортежей;
- *удалить*: физическое или логическое удаление данных (кортежей).

Структуризация данных в РМД существенно отличается от структуризации данных по версии CODASYL (табл. 3.2).

3.2. Сравнение структуризации данных в РМД и по версии CODASYL

Термины версии CODASYL	*Термины* (*и синонимы*) *РМД*
Элемент данных	Атрибут (поле)
Агрегат	–
Запись (группа)	Кортеж (запись, строка)
Совокупность записей одного типа	Отношение (таблица)
Набор (групповое отношение)	–
База данных	База данных

Примечание: в реляционной модели данных набор (групповое отношение) моделируется с помощью внешнего ключа, описывающего связь между двумя таблицами.

Схема отношения (*заголовок отношения*) представляет собой список имен атрибутов. Например, ПРОВАЙДЕРЫ INTERNET (*Название Провайдера, Почасовая Оплата, Скорость, Web-сайт*). Множество собственно кортежей отношения часто называют *содержимым* (*телом*) *отношения.*

Список, в котором указываются имена реляционных таблиц с перечислением их атрибутов (первичные ключи подчёркнуты) и определений внешних ключей, называется *реляционной схемой базы данных*.

Для пользователей информационной системы является важным, чтобы база данных отражала предметную область однозначно и непротиворечиво. Если она обладает такими свойствами, то говорят, что БД удовлетворяет условию целостности. Чтобы добиться выполнения этого условия, на БД накладывают некоторые ограничения, называемые *ограничениями целостности*. Выделяют два основных типа ограничений: *целостность сущностей* и *ссылочная целостность*.

Кортежи реляционной таблицы представляют в модели элементы конкретных объектов реального мира – сущностей. Ограничение первого типа состоит в том, что любой кортеж любого отношения должен отличаться от любого другого кортежа этого отношения, т.е. любое отношение должно обладать первичным ключом. Это требование автоматически удовлетворяется, если в системе не нарушаются базовые свойства отношений. Первичный ключ определяет каждый кортеж, а следовательно, каждый элемент сущности. Для работы с данными каждого кортежа необходимо знать значение ключа. Таким образом, элемент не должен записываться в базу данных до тех пор, пока не определены значения его ключевых атрибутов. То есть никакой ключевой атрибут любой строки не должен быть пустым.

Внешние ключи служат для обеспечения целостности данных, называемой *ссылочной целостностью*. Это означает, что значение внешнего ключа должно быть либо пустым, либо равным одному из текущих значений первичного ключа другой таблицы, иначе каждому значению внешнего ключа должны соответствовать строки в связываемых отношениях. Для нашего примера это означает, что если в отношении ЗАКАЗЫ указан *Код клиента*, то этот клиент должен существовать.

Ограничения целостности должны поддерживаться СУБД. Для соблюдения целостности сущностей достаточно гарантировать отсутствие в отношении кортежей с одним и тем же значением первичного ключа. Со ссылочной целостностью значительно сложнее. Необходимо следить за тем, чтобы не появлялись некорректные значения внешних ключей при обновлении отношений (например, заказы несуществующих клиентов). При удалении кортежа существует три подхода, позволяющие поддерживать ссылочную целостность:

– запрещается производить удаление кортежа, на который существуют ссылки (либо сначала удалить ссылающиеся кортежи, либо изменить значения их внешнего ключа);

– при удалении кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа становится неопределённым;

– при удалении кортежа из отношения, на которое ведётся ссылка, из ссылающегося отношения автоматически удаляются все ссылающиеся кортежи (каскадное удаление).

Большинство современных СУБД способны контролировать соблюдение правила ссылочной целостности. Для этой цели используются различные объекты баз данных (ссылочные ограничения и правила, триггеры и др.).

Уточним ещё раз условия, которым должны удовлетворять данные в реляционных таблицах:

– все строки таблицы должны быть уникальны, т.е. не может быть в таблице двух одинаковых строк;

– имена столбцов таблицы должны быть различны, а значения их атомарными;

– все строки одной таблицы должны иметь одну структуру, соответствующую именам и типам столбцов;

– последовательность размещения строк и столбцов в таблице является несущественной.

База данных включает одну или несколько таблиц, объединённых смысловым содержанием, а также процедурами контроля целостности и обработки информации. Помимо таблиц, база данных содержит ещё

и другие объекты: экранные формы, отчёты, прикладные программы, работающие с информацией базы данных. Кроме того, база данных хранит *словарь* данных (метаданные – «данные о данных»).

Широкое распространение реляционной модели объясняется в первую очередь простотой представления и формирования базы данных, универсальностью и удобством обработки данных, которая осуществляется с помощью декларативного языка запросов SQL (Structured Query Language).

Моделирование предметной области в рамках реляционной модели создаёт некоторые сложности, так как в этой модели нет специальных средств для отображения различных типов связей и агрегатов. Отсутствие агрегатов приводит к тому, что при проектировании реляционной БД приходится проводить нормализацию отношений. После нормализации данные об одной сущности предметной области распределяются по нескольким таблицам, что усложняет работу с БД.

Отсутствие специальных механизмов навигации (как в иерархической или сетевой, моделях), с одной стороны, ведёт к упрощению модели, а с другой, к многократному увеличению времени на извлечение данных, так как во многих случаях требуется просмотреть всё отношение для поиска нужных данных.

В РМД нет понятий «режим включения» и «класс членства». Но с помощью внешних ключей и дополнительных возможностей СУБД их можно эмулировать.

Итак, реляционная модель данных – это модель данных, основанная на представлении данных в виде набора отношений, каждое из которых является подмножеством декартова произведения определённых множеств. Манипулирование данными в РМД осуществляется с помощью операций реляционной алгебры (РА) или реляционного исчисления. Реляционная алгебра основана на теории множеств, а реляционное исчисление базируется на математической логике (вернее, на исчислении предикатов первого порядка). Изучение реляционного исчисления выходит за рамки данного пособия. Мы рассмотрим только операции реляционной алгебры.

3.3.2. Реляционная алгебра

В предыдущем разделе мы рассмотрели два аспекта реляционной модели: структуру данных и целостность данных. Третьим, наиболее важным аспектом реляционной модели, являются предложенные Е. Ф. Коддом реляционные языки обработки данных: реляционная алгебра и реляционное исчисление.

И реляционная алгебра, и реляционное исчисление являются теоретическими языками. Выражения реляционной алгебры и формулы реляционного исчисления выполняются над отношениями реляционных БД, т.е. операнды и результаты всех действий являются отношениями. Языки реляционной алгебры являются *процедурными*, так как отношение, являющееся результатом запроса к реляционной БД, вычисляется при пошаговом выполнении последовательности реляционных операторов, применяемым к отношениям. Операторы состоят из операндов, в роли которых выступают отношения, и реляционных операций. Результатом реляционной операции является отношение.

Языки *исчислений*, в отличие от реляционной алгебры, являются *непроцедурными* (описательными, или декларативными) и позволяют выражать запросы с помощью предиката первого порядка (высказывания в виде функции), которому должны удовлетворять кортежи или домены отношений. Запрос к БД, выполненный с использованием подобного языка, содержит лишь информацию о желаемом результате. Для этих языков характерно наличие наборов правил для записи запросов. Непроцедурные языки формулируют, что нужно делать, а не как этого добиться.

Механизмы реляционной алгебры и реляционного исчисления эквивалентны. Это означает, что для любого допустимого выражения реляционной алгебры можно построить эквивалентную (производящую такой же результат) формулу реляционного исчисления, и наоборот.

Основные операции реляционной алгебры, предложенные Коддом, следующие*:* объединение, разность (вычитание), пересечение, декартово (прямое) произведение (или произведение), выборка (селекция, ограничение), проекция, деление и соединение.

К. Дж. Дейт расширил этот набор, включив операции: переименования атрибутов, образования новых вычисляемых атрибутов, вычисления итоговых функций, построения сложных алгебраических выражений, присвоения, сравнения и т.д.

Операции реляционной алгебры Кодда можно разделить на *две* группы: множественные (объединение, разность, пересечение и произведение) и специальные реляционные (проекция, выборка, деление и соединение).

Операции реляционной алгебры могут выполняться над одним отношением (унарная операция) или над двумя отношениями (бинарная операция). При выполнении бинарной операции отношения, участвующие в операциях, должны быть совместимы по структуре. Для этого, во-первых, отношения должны иметь одинаковое количество

атрибутов. Во-вторых, атрибуты должны иметь одинаковые имена. В-третьих, атрибуты должны быть определены на одном домене. Структура результирующего отношения по определённым правилам наследует свойства структур исходных отношений. В большинстве рассматриваемых бинарных реляционных операций будем считать, что заголовки исходных отношений идентичны. Некоторые отношения не являются совместимыми по структуре, но становятся таковыми после переименования атрибутов. Операция переименования рассмотрена далее.

В нашем пособии мы рассмотрим операции реляционной алгебры, следуя подходу, предложенному К. Дж. Дейтом.

В качестве примера воспользуемся базой данных поставщиков и тканей. В таблице **S (поставщики)** находятся сведения о поставщиках тканей: код поставщика, название, рейтинг, город расположения поставщика. В качестве первичного ключа таблицы выбран код поставщика S#. Таблица **M (материалы)** содержит сведения о поставляемых тканях: код материала; наименование материала, тип материала, поверхностная плотность материала (MS), название города, где производится этот материал. Первичным ключом этой таблицы является M# (код материала). Последняя таблица **SM (поставки**) содержит сведения о поставках тканей, осуществляемых поставщиками, и их количестве. Например, поставщик S1 поставляет ткани M1, M2 и др., поставщик S4 – ткань M4 и т.д. Предполагается, что в одно и то же время не может быть более, чем одной поставки для одного поставщика и для одной ткани. Поэтому в качестве первичного ключа этой таблицы можно выбрать составной ключ S#M#, представляющий уникальную комбинацию для каждой поставки с точки зрения набора текущих поставок в таблице. Каждое из полей S# и M# таблицы SM в отдельности является внешним ключом по отношению к таблице S и M соответственно. Можно также отметить, что данные в табл. 3.3 – 3.5 приведены неполные и упрощённые, реальная база данных содержит значительно больше объектов и сведений.

3.3. Поставщики

S#	Поставщик	Рейтинг	Город_П
S1	ООО «Росмануфактура»	30	Москва
S2	ООО «Сибтекстиль»	15	Тюмень
S3	ТД «Ивановские ткани»	30	Иваново
S4	ЧП Федоров А. К.	20	Москва
S5	ООО «Андромеда»	10	Омск

3.4. Материалы

M#	Наименование материала	Тип	MS	Город_М
M1	Ткань пальтовая «Ассоль»	п/ш	380	Москва
M2	Сукно костюмное «Снежинка»	ч/ш	400	Омск
M3	Ткань пальтовая-мохер «День»	п/ш	600	Улан-Удэ
M4	Ткань костюмная «Вечер»	п/э	280	Тюмень
M5	Ткань костюмно-платьевая «Искра»	п/э	270	Тюмень
M6	Драп пальтовый «Бриз»	ч/ш	750	Москва

3.5. Поставки

S#	M#	Количество
S1	M1	3900
S1	M2	2100
S1	M3	2700
S1	M4	2400
S1	M5	2200
S1	M6	1900
S2	M1	3200
S2	M2	4100
S3	M2	700
S4	M2	1100
S4	M4	900
S4	M5	4300

Операции реляционной алгебры

Объединением двух совместимых отношений R1 и R2 одинаковой размерности (R1 UNION R2) является отношение R, содержащее все кортежи, которые принадлежат исходным отношениям (за исключением повторяющихся).

Пример 3.1. Объединение отношений

Пусть отношением R1 будет множество поставщиков из Москвы, а отношением R2 – множество поставщиков, которые поставляют материал M1. Тогда отношение R обозначает поставщиков, находящихся в Москве, или поставщиков, поставляющих материал M1, либо тех и других (рис. 3.9).

Отношение R1

П#	Поставщик	Статус	Город_П
S1	ООО «Росмануфактура»	30	Москва
S4	ЧП Федоров А. К.	20	Москва

Отношение R2

П#	Поставщик	Статус	Город_П
S1	ООО «Росмануфактура»	30	Москва
S2	ООО «Сибтекстиль»	15	Тюмень

Отношение R

П#	Поставщик	Статус	Город_П
S1	ООО «Росмануфактура»	30	Москва
S4	ЧП Федоров А. К.	20	Москва
S2	ООО «Сибтекстиль»	15	Тюмень

Рис. 3.9. Пример объединения отношений

S4	ЧП Федоров А. К.	20	Москва

Рис. 3.10. Пример вычитания отношений

Вычитанием совместимых отношений R1 и R2 одинаковой размерности (R1 MINUS R2) является отношение, тело которого состоит из множества всех кортежей, принадлежащих отношению R1, но не принадлежащих отношению R2.

Для тех же отношений R1 и R2 из предыдущего примера отношение R будет представлять собой множество поставщиков, находящихся в Москве, но не поставляющих материал M1 (рис. 3.10).

Можно заметить что результат операции вычитания зависит от порядка следования операндов, т.е. R1 MINUS R2 и R2 MINUS R1 – не одно и то же.

Пересечением двух совместимых отношений R1 и R2 одинаковой размерности (R1 INTERSECT R2) является отношение R с телом, включающим в себя кортежи, *одновременно* принадлежащие обоим отношениям.

S1	ООО «Росмануфактура»	30	Москва

Рис. 3.11. Пример пересечения отношений

Для отношений R1 и R2 результирующее отношение R будет означать всех поставщиков из Москвы, поставляющих материал M1. Тело отношения R состоит из единственного элемента (рис. 3.11).

Произведением отношения R1 степени k1 (степень отношения – количество атрибутов в отношении) и отношения R2 степени k2 (R1 TIMES R2), которые не имеют одинаковых имён атрибутов, является такое отношение R степени (k1+k2), заголовок которого представляет сцепление заголовков отношений R1 и R2, а тело – имеет кортежи, такие, что первые k1 элементов кортежей принадлежат множеству R1, а последние k2 элементов – множеству R2. То есть можно отметить, что произведение возвращает отношение, содержащее всевозможные кортежи, которые являются сочетанием двух кортежей, принадлежащих соответственно двум определённым отношениям.

При необходимости получить произведение двух отношений, имеющих одинаковые имена одного или нескольких атрибутов, применяется операция переименования RENAME.

Пример 3.2. Произведение отношений

Отношение R1 представляет множество номеров всех текущих поставщиков {S1, S2, S3, S4, S5}, а отношение R2 – множество номеров всех материалов {M1, M2, M3, M4, M5, M6}. Результат операции – множество пар типа «поставщик-материал», т.е. {(S1, M1), (S1, M2), (S1, M3)…(S5, M6)}.

Можно заметить, что на практике явное использование произведения требуется только для сложных запросов.

Выборка возвращает отношение, содержащее все кортежи из определённого отношения, которые удовлетворяют определённым условиям. Выборка (R WHERE f) отношения R по формуле f представляет собой новое отношение с заголовком отношения R и телом, состоящим из таких кортежей отношения R, которые удовлетворяют истинности логического выражения, заданного формулой f.

Для записи формулы f используются имена атрибутов, константы, логические операции (AND – И, OR – ИЛИ, NOT – НЕ), операции отношения между величинами и скобки.

Пример 3.3. Выборка

1. Логическое выражение: M WHERE MS < 400 (рис. 3.12). Результирующее отношение содержит кортежи таблицы M, содержащие материалы, поверхностная плотность которых меньше 400.

M#	Наименование материала	Тип	MS	Город_М
М1	Ткань пальтовая «Ассоль»	п/ш	380	Москва
М4	Ткань костюмная «Вечер»	п/э	280	Тюмень
М5	Ткань костюмно-платьевая «Искра»	п/э	270	Тюмень

Рис. 3.12. Результат выборки 1

2. Логическое выражение: SM WHERE S# = «S1» AND M# = «M1» (рис. 3.13). Результирующее отношение содержит кортеж таблицы SM, определяющий поставку поставщиком S1 материала М1.

Проекция возвращает отношение, содержащее все кортежи определенного отношения после исключения из него некоторых атрибутов.

Проекция отношения R на атрибуты X, Y, ..., Z (R [X, Y, ..., Z]), где множество (X, Y, ..., Z} является подмножеством полного списка атрибутов заголовка отношения R, представляет собой отношение с заголовком X, Y, ..., Z и телом, содержащим кортежи отношения R, за исключением повторяющихся кортежей. Повторение одинаковых атрибутов в списке X, Y, ..., Z запрещается.

Операция проекции допускает следующие дополнительные варианты записи.

1. Отсутствие списка атрибутов подразумевает указание всех атрибутов (операция тождественной проекции).

2. Выражение вида R[ ] означает *пустую* проекцию, результатом которой является пустое множество.

3. Операция проекции может применяться к произвольному отношению, в том числе и к результату выборки.

S#	M#	Количество
S1	M1	3900

Рис. 3.13. Результат выборки 2

Пример 3.4. Проекция

1. Результат проекции отношения М на атрибуты Тип, Город_М (М[Тип, Город_М]) представлен на рис. 3.14.

Тип	Город_М
п/ш	Москва
ч/ш	Омск
п/ш	Улан-Удэ
п/э	Тюмень
ч/ш	Москва

Рис. 3.14. Результат проекции 1

S#	Город_П
S1	Москва
S4	Москва

Рис. 3.15. Результат проекции 2

2. Результат проекции ((S WHERE Город_П=«Москва»)[S#, Город_П]) (рис. 3.15).

Деление для двух отношений (бинарного и унарного) возвращает отношение, содержащее все значения одного атрибута бинарного отношения, которое соответствует (в другом атрибуте) всем значениям в унарном отношении.

Результатом деления отношения R1 с атрибутами A и B на отношение R2 с атрибутом B (R1 DIVIDEBY R2), где A и B простые или составные атрибуты, причем атрибут B – общий атрибут, определённый на одном и том же домене (множестве доменов составного атрибута), является отношение R с заголовком A и телом, состоящим из кортежей r, таких, что в отношении R1 имеются кортежи (r, s), причём множество значений s включает множество значений атрибута B отношения R2.

Пример 3.5. Деление отношений

R1 – проекция SM[S#, M#], R2 – отношение с заголовком M# и телом M2, M4. Результатом будет отношение R с заголовком S# и телом S1, S4 (рис. 3.16).

Отношения R1, *R2*, *R*

S#	M#
S1	M1
S1	M2
S1	M3
S1	M4
S1	M5
S1	M6
S2	M1
S2	M2
S3	M2
S4	M2
S4	M4
S4	M5

M#
M2
M4

S#
S1
S4

Рис. 3.16. Пример деления отношений

Соединение возвращает отношение, кортежи которого – это сочетание двух кортежей, имеющих общие значения для одного или нескольких общих атрибутов этих двух отношений. Операция соединения имеет несколько разновидностей. Наиболее важным является естественное соединение.

Операция *естественного соединения* (операция JOIN) применяется к двум отношениям, имеющим общий атрибут. Этот атрибут в отношениях имеет одно и то же имя и определён на одном и том же домене.

Пусть отношения R1 и R2 имеют заголовки $\{X_1, X_2, \ldots, X_M, Y_1, Y_2, \ldots, Y_N\}$ и $\{Y_1, Y_2, \ldots, Y_N, Z_1, Z_2, Z_P\}$ соответственно; т.е. атрибуты $Y_1, Y_2, \ldots, Y_N$ – общие атрибуты для двух отношений. Можно рассматривать выражения $\{X_1, X_2, \ldots, X_M\}$, $\{Y_1, Y_2, \ldots, Y_N\}$, $\{Z_1, Z_2, Z_P\}$ как три составных атрибута X, Y, Z соответственно. Тогда *естественным* соединением отношений R1 и R2 (R1 JOIN R2) является отношение с заголовком {X, Y, Z} и телом, содержащим множество всех таких кортежей, для которых в отношении R1 значение атрибута X равно x, атрибута Y равно y, в отношении R2 значение атрибута Y равно y, значение атрибута Z равно z.

Допустим, что атрибуты Город_П и Город_М определены на одном и том же домене: множестве названий городов. Имя этого домена может быть Город. Результат естественного соединения (S JOIN M) приведён на рис. 3.17.

S#	Поставщик	Рейтинг	Город	M#	Наименование материала	Тип	MS
S1	ООО «Росманufактура»	30	Москва	M1	Ткань пальтовая «Ассоль»	п/ш	380
S1	ООО «Росманufактура»	30	Москва	M6	Драп пальтовый «Бриз»	ч/ш	750
S2	ООО «Сибтекстиль»	15	Тюмень	M4	Ткань костюмная «Вечер»	п/э	280
S2	ООО «Сибтекстиль»	15	Тюмень	M5	Ткань костюмно-платьевая «Искра»	п/э	270
S4	ЧП Федоров А. К.	20	Москва	M1	Ткань пальтовая «Ассоль»	п/ш	380
S4	ЧП Федоров А. К.	20	Москва	M6	Драп пальтовый «Бриз»	ч/ш	750
S5	ООО «Андромеда»	10	Омск	M2	Сукно костюмное «Снежинка»	ч/ш	400

Рис. 3.17. Результат естественного соединения

Соединение C_f(R1, R2) отношений R1 и R2 *по условию*, заданному формулой f (θ – соединение), представляет собой отношение R, которое можно получить путём произведения отношений R1 и R2 с последующим применением к результату операции выборки по формуле f. Правила записи формулы f такие же, как и для операции выборки.

Другими словами, соединением отношения R1 по атрибуту A с отношением R2 по атрибуту B (отношения не имеют общих имён атрибутов) является результат выполнения операции вида:

(R1 T1MES R2) WHERE A Θ B,

где Θ – логическое выражение над атрибутами, определёнными на одном (нескольких – для составного атрибута) домене.

Пример 3.6. θ-соединение

Необходимо найти соединение отношений S и P по атрибутам Город_П и Город_М соответственно, причём кортежи результирующего отношения должны удовлетворять отношению «больше».

(S TIMES M) WHERE Город_П > Город_М. Результат операции θ-соединения показан в табл. 3.6.

При рассмотрении операций реляционной алгебры К. Дж. Дейтом вводятся дополнительные операторы: переименования, расширения, подведения итогов, обновления.

3.6. Результат операции соединения

S#	Поставщик	Рейтинг	Город_П	M#	Наименование материала	Тип	MS	Город_М
S2	ООО «Сибтекстиль»	15	Тюмень	M1	Ткань пальтовая «Ассоль»	п/ш	380	Москва
S2	ООО «Сибтекстиль»	15	Тюмень	M2	Сукно «Снежинка»	ч/ш	400	Омск
S2	ООО «Сибтекстиль»	15	Тюмень	M6	Драп пальтовый «Бриз»	ч/ш	750	Москва
S5	ООО «Андромеда»	10	Омск	M1	Ткань пальтовая «Ассоль»	п/ш	380	Москва
S5	ООО «Андромеда»	10	Омск	M6	Драп пальтовый «Бриз»	ч/ш	750	Москва

Оператор *переименования* позволяет изменить имя атрибута отношения и имеет следующий синтаксис:

<отношение> RENAME <исходное имя атрибута>
AS <новое имя атрибута>,

где <отношение> задаётся именем отношения либо выражением реляционной алгебры. В последнем случае выражение заключают в круглые скобки.

Например, **S RENAME Город_П AS**
Город_размещения_Поставщика

Допустим, необходимо изменить несколько имён атрибутов. Для такого случая Дейт вводит оператор *множественного переименования*, синтаксис которого приведён ниже:

<отн.> RENAME <исх_имя_атр.1> AS <нов_имя_атр.1>,
<исх_имя_атр.2> AS <нов_имя_атр.2>,...,
<исх_имя_атр.N> AS <нов_имя_атр.N>.

Оператор *расширения* позволяет добавить в отношение дополнительный атрибут, значения которого вычисляются посредством некоторых скалярных вычислений. Оператор расширения имеет вид

EXTEND <отношение> ADD <выражение>AS<нов_атрибут>,

где к исходному отношению добавляется (ключевое слово ADD) новый атрибут с именем <нов_атрибут>, значения которого подсчитываются по правилам, заданным <выражением>. Исходное отношение может быть задано именем отношения или с помощью выражения реляционной алгебры, заключённого в круглые скобки. При этом имя нового атрибута не должно входить в заголовок исходного отношения и не может использоваться в <выражении>.

Например, **EXTEND S ADD 'Поставщик' AS SNAME**.

Приведенное выражение дополняет отношение S столбцом SNAME, значениями которого является символьная константа 'Поставщик'.

Помимо обычных арифметических операций и операций сравнения в выражении можно использовать итоговые функции, такие как COUNT (количество), SUM (сумма), AVG (среднее), MAX, MIN.

Оператор *множественного расширения* имеет следующую синтаксическую конструкцию:

EXTEND <отношение> ADD <выр_1> AS <атр_1>,
<выр_2> AS <атр_2>,...,
<выр_N>AS <атр_N>.

Операция *подведения итогов* SUMMARIZE выполняет «вертикальные» или групповые вычисления и имеет следующий формат:

SUMMARIZE <отношение> BY (<список атрибутов>) ADD <выражение> AS <новый атрибут>,

где исходное отношение задается именем отношения либо заключённым в круглые скобки выражением реляционной алгебры, <список атрибутов> представляет собой разделенные запятыми имена атрибутов исходного отношения A_1, A_2, ..., A_N, <выражение> – это скалярное выражение, аналогичное выражению операции EXTEND, а <новый атрибут> – имя формируемого атрибута. В списке атрибутов и в выражении не должно использоваться имя нового атрибута.

Результатом операции SUMMARIZE является отношение R с заголовком, состоящим из атрибутов списка, расширенного новым атрибутом. Для получения тела отношения R сначала выполняется проецирование (назовём проекцию R1) исходного отношения на атрибуты A_1, A_2, ..., A_N , после чего каждый кортеж проекции расширяется новым (N + 1)-м атрибутом. Поскольку проецирование, как правило, приводит к сокращению количества кортежей по отношению к исходному отношению (удаляются одинаковые кортежи), то можно считать, что происходит своеобразное группирование кортежей исходного отношения: одному кортежу отношения R1 соответствует один или более (если было дублирование при проецировании) кортежей исходного отношения. Значение (N + 1)-го атрибута каждого кортежа отношения R формируется путем вычисления выражения над соответствующей этому кортежу группой кортежей исходного отношения.

Пример 3.7. Подведение итогов

Пусть требуется вычислить количество поставок по каждому из поставщиков:

SUMMARIZE SP BY (S#) ADD COUNT AS Количество поставок

Результат вычисления показан на рис. 3.18.

S#	Количество поставок
S1	6
S2	2
S3	1
S4	3

Рис. 3.18. Подведение итогов

Отметим, что функция COUNT определяет количество кортежей в каждой из групп исходного отношения.

Рассмотрим ещё один пример. С помощью выражения

SUMMARIZE SP BY (M#) ADD SUM (Количество) AS Общее количество

определяется общее количество по каждому виду материала.

Оператор *множественного подведения итогов*, подобно соответствующим операциям переименования и расширения, выполняет одновременно несколько «вертикальных» вычислений и записывает результаты в отдельные новые атрибуты. Примером данного оператора может служить следующая запись:

SUMMARIZE SP BY (M#) ADD SUM (Количество) AS Общее количество, AVG (Количество) AS (Сред_знач).

Оператор *реляционного присвоения* можно представить следующим образом:

<выражение-цель>:= <выражение-источник>,

где оба выражения представляют совместимые по структуре отношения. Вычисленное значение <выражение-источник> присваивается отношению <выражение-цель>, заменяя его предыдущее значение.

Операция присвоения позволяет обновлять базу данных. С помощью операции присвоения можно не только полностью заменить все значения отношения <выражение-цель>, но и добавить или удалить кортежи. Приведём пример, в котором в отношение S добавляется один кортеж:

S:=S UNION{(<S# :'S6'>, <Поставщик: Донецкая мануфактура>, <Рейтинг:40>, <Город_П: Донецк>)}

Оператор *вставки* имеет следующий вид:

INSERT <выражение-источник> INTO <выражение-цель>,

где оба выражения представляют совместимые по структуре отношения. Выполнение операции заключается в вычислении значения отношения <выражение-источник> и вставке полученных кортежей в отношение, заданное <выражение-цель> (в приведённом примере в отношение T).

INSERT (S WHERE Город_П ='Москва') INTO T

Оператор *обновления* имеет следующий вид:

UPDATE <выражение-цель> <список элементов>,

где <список элементов> представляет собой последовательность разделенных запятыми операций присвоения <атрибут>:= <скалярное выражение>. Результатом выполнения операции обновления является отношение, полученное после присвоения соответствующих значений атрибутам отношения, заданного целевым выражением.

Например, **UPDATE M WHERE Тип='п/ш' Город_П:='Ростов'**. Эта операция предписывает изменить значение атрибута *Город_П* (независимо от того, каким оно было) на новое значение – 'Ростов' таких кортежей отношения M, атрибут *Тип* которых имеет значение 'п/ш'.

Оператор *удаления* имеет следующий вид:

DELETE <выражение-цель>,

где <выражение-цель> представляет собой реляционное выражение. Все кортежи в результирующем отношении удаляются. Например, **DELETE S WHERE Рейтинг <20**.

Операция *реляционного сравнения* может использоваться для прямого сравнения двух отношений. Она имеет следующий синтаксис:

<выражение1> Θ<выражение2>,

где выражения представляют совместимые по структуре отношения, а знак Θ – это один из следующих операторов сравнения: = (равно), ≠ (не равно), ≤ (подмножество), < (собственное подмножество), ≥ (надмножество), > (собственное надмножество).

Например, сравнение: «Совпадают ли города поставщиков и города, в которых производятся ткани» можно записать так:

S[Город_П] = M[Город_M].

Сравнение «Есть ли поставщики, не осуществляющие поставки материалов?» записывается следующим образом:

S[M#]=SP[M#].

Приведём несколько примеров использования реляционных операторов.

1. Получить названия поставщиков, осуществляющих поставки материала M2:

((SP JOIN S) WHERE M#='M2') [Поставщик].

2. Получить названия поставщиков, которые не поставляют материал M2:

((S [S#] MINUS (SP WHERE M#='M2') [S#]) JOIN S) [Поставщик].

3. Получить названия поставщиков, поставляющих все детали:

((SP [S#, M#] DIVIDEBY M [M#] JOIN S [Поставщик].

4. Получить названия поставщиков, которые поставляют по крайней мере один материал типа 'п/ш':

(((M WHERE Тип='п/ш') JOIN SP) [S#] JOIN S) [Поставщик].

В реализациях конкретных реляционных СУБД реляционная алгебра и реляционное исчисление в «чистом» виде не используются. Фактическим стандартом доступа к реляционным данным стал язык SQL (Structured Query Language), представляющий собой смесь операторов реляционной алгебры и выражений реляционного исчисления, использующий синтаксис, близкий к фразам английского языка и расширенный дополнительными возможностями, отсутствующими в реляционной алгебре и реляционном исчислении.

3.3.3. Язык запросов по образцу QBE

Для манипулирования данными в базах данных используются *запросы*. Запрос представляет собой сообщение конечного пользователя или приложения, направляемое СУБД и активизирующее в базе данных следующие действия: выборку, вставку, удаление или обновление указанных в запросе данных. Запросы описываются с помощью языков запросов: QBE (Query By Example) – язык запросов по образцу; SQL (Structured Query Language) – структурированный язык запросов. Оба языка являются непроцедурными, т.е. описывают свойства результата («что надо сделать»), а не алгоритм решения задачи («как это сделать»).

Для манипулирования данными указанные языки имеют практически одинаковые возможности. Главное отличие между ними заключается в способе формирования запросов: язык QBE предполагает ручное или визуальное формирование запроса, в то время как использование SQL означает программирование запроса.

Язык QBE позволяет создавать запросы к БД путём заполнения предлагаемой СУБД запросной формы. Традиционные компьютерные языки являются текстовыми, в них решение задач формулируется в виде символьных строк. QBE является графическим языком, в котором запросы формулируются посредством графического представления таблиц базы данных. Такой способ задания запросов обеспечивает высокую наглядность и не требует знания программирования – достаточно описать образец ожидаемого результата. В каждой из современных реляционных СУБД имеется свой вариант языка QBE, незначительно отличающийся от первого описания QBE, предложенного М. М. Злуфом в 1975 – 1977 гг. Помещая символы в определённые места в столбцах таблицы – шаблона запроса, пользователь может определять условия отбора строк для запроса, группировки данных, формат вывода данных, операции обновления данных.

На языке QBE можно создавать *однотабличные* и *многотабличные* (выбирающие или обрабатывающие данные из нескольких связанных таблиц) запросы.

Запросная форма имеет вид таблицы-шаблона, имя и названия полей которой совпадают с именем и названиями полей соответствующей исходной таблицы. Чтобы узнать имена доступных таблиц БД, в языке QBE предусмотрен запрос на выборку имён таблиц. Названия полей исходной таблицы могут вводиться в шаблон вручную или автоматически.

Для иллюстрации средств и возможностей языков QBE и SQL используем БД небольшой торговой фирмы. В базе данных хранится следующая информация: информация о клиентах, с которыми работает данная фирма; информация о заказах, сделанных клиентами; информация о товарах данной фирмы (табл. 3.7 – 3.9). В таблицах приведены неполные и упрощённые данные.

В таблице CUST (табл. 3.7) хранятся данные о клиентах: номер клиента – CUST_NUM; название фирмы-клиента – CUST_NAME; общий объём заказов, сделанных клиентом за год, – CUST_SUM.

В таблице PROD (табл. 3.8) хранятся сведения о товарах фирмы: идентификатор товара – PROD_ID; наименование товара – PROD_ NAME; цена товара – PRICE; количество единиц товара на складе – STORE.

Таблица ORDERS (табл. 3.9) содержит сведения о заказах: номер заказа – ORDER_NUM; номер клиента, сделавшего заказ, – CUST_NUM; идентификатор заказанного продукта – PROD_ID; количество заказанного продукта – QTY; дата поставки – DATE_ORDER. Для простоты будем считать, что в одном заказе может упоминаться только один товар.

3.7. CUST

CUST_NUM	CUST_NAME	CUST_SUM
3101	ООО «PC-Style»	50000
3102	ООО «Гермес»	70000
3103	ЧП Федоров В. Г.	15000
3104	ООО «Формат»	100000
3105	ЧП Гришин П.В.	20000
3106	ОАО «Энтрон»	125000
3107	ЗАО»IT-COM»	135000
3108	ООО «Омега»	95000
3109	ОАО « PC-Центр»	160000

3.8. PROD

PROD_ID	PROD_NAME	PRICE	STORE
3P	Процессор Celeron 2400	90	1000
4P	Процессор Athlon XP 2600+	110	1200
6P	Процессор Pentium-4 2600	200	800
4MB	Материнская плата GA 81 PE 1000	100	1400
7MB	Материнская плата EPOX 8KRA2+	95	1350
3V	Видеокарта Nvidia GeForce FX5600 128mb	140	770
4V	Видеокарта Ati Radeon 9500 64mb	150	850
1M	ОЗУ DDR 256 MB PC 2700	45	1150
2M	ОЗУ DDR 256 MB PC 3200	50	1250
3M	ОЗУ DDR 512 MB PC 3200	97	1600

3.9. ORDERS

ORDER_NUM	CUST_NUM	PROD_ID	QTY	DATE_ORDER
221	3101	3P	15	20.12.02
222	3105	4MB	30	20.12.02
223	3106	6P	16	21.12.02
224	3101	3V	12	5.01.03
225	3105	4P	27	17.01.03
226	3104	1M	10	14.02.03
227	3103	7MB	18	18.02.03
228	3102	2M	21	1.03.03
229	3103	3P	17	5.03.03
230	3108	3M	16	5.03.03
231	3107	4MB	10	7.03.03

QBE предлагает пользователю для создания запроса заполнение таблиц-шаблонов. В первом столбце таблицы-шаблона выводится имя таблицы, во всех остальных – имена столбцов.

Пример 3.8. Пример заполнения шаблона запроса

Запрос «Вывести названия всех клиентов» можно сформулировать с помощью следующего шаблона (рис. 3.19):

CUST УРЕ	CUST_NUM	CUST_NAME	CUST_SUM
		P.	

Рис. 3.19. Шаблон запроса

В приведённом шаблоне «P.» – это команда вывода, означающая, что выводятся значения заданного столбца. После команды в языке QBE ставится точка.

Результат выполнения приведённого шаблона следующий:

Обратим внимание на то, что результатом запроса является реляционная таблица.

CUST_NAME

ООО «PC-Style»

ООО «Гермес»

ЧП Федоров В.Г.

ООО «Формат»

ЧП Гришин П.В.

ОАО «Энтрон»

ЗАО»IT-COM»

ООО «Омега»

ОАО « PC-Центр»

Рассмотрим несколько вариантов создания запросов.

Пример 3.9. Запрос с простым условием сравнения

Вывести все товары, цена которых не превышает 90 долларов. Шаблон приведён на рис. 3.20.

Результатом запроса является следующий набор данных:

PROD	PROD_ID	PROD_NAME	PRICE	STORE
		P.	< 90	

Рис. 3.20. Запрос с простым условием

PROD_NAME

ОЗУ DDR 256 MB PC 2700

ОЗУ DDR 256 MB PC 3200

3.4. ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

При проектировании реляционной базы данных разработчику необходимо решить вопрос о наиболее эффективной структуре данных. Основная цель проектирования БД – это сокращение избыточности хранимых данных, а следовательно, экономия объёма используемой памяти, уменьшение затрат на многократные операции обновления избыточных копий и устранение возможности возникновения противоречий из-за хранения в разных местах сведений об одном и том же объекте.

Проектирование должно быть эффективным, т.е. обеспечивать минимальное дублирование данных, удобство их обработки и обновления. Для удовлетворения этих требований необходимо определить, из каких отношений должна состоять БД, какие атрибуты должны входить в эти отношения.

Например, рассмотрим отношение R (табл. 3.10).

3.10. Отношение R

Код_материала	Наименование	Тип	Код_модели
101	Ткань пальтовая «Ассоль»	п/ш	312
101	Ткань пальтовая «Ассоль»	п/ш	512
210	Ткань подкладочная «Фея»	п/э	312
210	Ткань подкладочная «Фея»	п/э	512
210	Ткань подкладочная «Фея»	п/э	460
210	Ткань подкладочная «Фея»	п/э	430
315	Драп пальтовый «Бриз»	ч/ш	460

Можно заметить, что таблица спроектирована не совсем удачно. В четырёх кортежах, соответствующих материалу с кодом 210, повторяется одна и та же информация о наименовании и типе ткани. Проблема возникает из-за того, что одна и та же ткань может использоваться в разных моделях. Такое дублирование данных называется *избыточностью* данных. Избыточность данных вызывает нежелательные явления, возникающие в процессе работы с базой данных, называемые *аномалиями*.

Предположим, что *тип* ткани подкладочной «Фея» указан неправильно. Тогда необходимо внести изменения не в один кортеж, а во все четыре (представьте, сколько может быть кортежей в реальной БД!). Такая ситуация, при которой изменение значения одного данного мо-

жет повлечь за собой просмотр и редактирование нескольких строк таблицы, называется *аномалией модификации* (редактирования).

Далее предположим, что ткани подкладочной в течение какого-либо времени не было на складе, и модели, в которых она должна была участвовать, уже пошиты. Если принимается решение об удалении всех сведений о пошитых моделях, то информация о подкладочной ткани тоже утрачивается. Ситуация, заключающаяся в том, что при удалении каких-либо данных из таблицы может исчезнуть и информация, напрямую не связанная с удаляемой, называется *аномалией удаления*.

Возможна и другая ситуация: приобретена ткань, но пока ещё не решено, в какой модели она будет участвовать. Поэтому, если мы хотим избежать пустых (неопределённых) значений в кортежах, информацию о ней мы не можем ввести в таблицу. Такая ситуация, при которой невозможно ввести одни данные из-за отсутствия других данных, называется *аномалией ввода*.

Избавиться от избыточности данных позволяет нормализация базы данных. *Нормализация* – это процесс преобразования отношения путём *декомпозиции* (разбиения) его на два или более отношений, имеющих лучшие свойства. Окончательная цель нормализации сводится к получению такого проекта базы данных, в котором *каждый факт появляется лишь в одном месте*, т.е. исключена избыточность информации.

Нормализация – это классический метод проектирования реляционной базы данных. Исходной точкой здесь является представление предметной области в виде одного или нескольких отношений, и на каждом шаге проектирования производится некоторый набор схем отношений, обладающих лучшими свойствами. Процесс проектирования представляет собой процесс нормализации схем отношений, причём каждая следующая нормальная форма обладает лучшими свойствами по сравнению с предыдущей.

Каждой нормальной форме соответствует некоторый набор ограничений. Отношение находится в определённой нормальной форме, если оно удовлетворяет набору ограничений этой формы. Основные свойства нормальных форм:

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

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

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

Процесс нормализации основан на понятии *функциональной зависимости*. Функциональные зависимости позволяют накладывать определённые ограничения на реляционную схему. Идея состоит в том, что значение одного атрибута в кортеже определяет значение другого атрибута. Например, в каждом кортеже отношения R *Код_материала* определяет *Наименование*; *Код_материала* определяет *Тип* (табл. 3.11). Можно записать функциональные зависимости:

Код_материала → Наименование;

Код_материала → Тип.

Для дальнейшего изложения потребуется несколько определений.

Определение 1. *Функциональная зависимость*

Пусть A и B – атрибуты в отношении R. Атрибут B *функционально зависит* от атрибута A, если в любой момент времени каждому значению атрибута A соответствует в точности одно значение атрибута B. Функциональная зависимость записывается следующим образом: A → B. Данная запись означает, что если два кортежа в таблице R имеют одно и тоже значение атрибута A, то они имеют одно и тоже значение атрибута B. Атрибут в левой части называется детерминантом, так как его значение определяет значение атрибута в правой части. Ключи таблицы являются детерминантами.

Определение 2. *Неключевой атрибут*

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

Определение 3. *Полная функциональная зависимость*

Функциональная зависимость называется *полной*, если неключевой атрибут зависит от всего составного ключа и не зависит от его частей.

Определение 4. *Транзитивная функциональная зависимость*

Если атрибут B функционально зависит от атрибута A (A → B), а атрибут C функционально зависит от атрибута B (B → C), но обратная зависимость отсутствует, то говорят, что атрибут C зависит от A *транзитивно*.

Определение 5. *Взаимно независимые атрибуты*

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

Определение 6. Отношение находится в первой нормальной форме (1NF), если значения его атрибутов атомарны. Исходное отношение строится таким образом, чтобы оно было в 1NF.

Определение 7. Отношение находится во второй нормальной форме (2NF), если выполняются ограничения первой нормальной формы и каждый неключевой атрибут функционально полно зависит от всего первичного ключа.

Определение 8. Отношение находится в третьей нормальной форме (3NF), если выполняются ограничения второй нормальной формы и все неключевые атрибуты взаимно независимы и полностью зависят от первичного ключа (т.е. в отношении отсутствуют транзитивные зависимости неключевых атрибутов от первичного ключа).

Проектирование базы данных рассмотрим на следующем примере. Пусть имеются сведения о поставках некоторой фирмой материалов. Сформируем исходное отношение *Поставки* (табл. 3.11).

3.11. Таблица Поставки

Код_заказа	Код_материала	Код_клиента	Город_ клиента	Коли-чество	Дата_ поставки
1	2	3	4	5	6
110	21	BN	Омск	300	25.07.02
110	30	BN	Омск	200	25.07.02
120	13	BR	Москва	160	12.08.02
120	15	BR	Москва	150	12.08.02
120	16	BR	Москва	80	12.08.02
120	18	BR	Москва	250	12.08.02
130	20	BR	Москва	120	14.08.02
130	55	BR	Москва	200	14.08.02
130	71	BR	Москва	300	14.08.02
140	2	BS	Тюмень	300	26.08.02
140	62	BS	Тюмень	90	26.08.02
150	31	BN	Омск	60	4.09.02
150	70	BN	Омск	20	4.09.02
160	30	AN	Улан-Удэ	50	18.09.02
160	69	AN	Улан-Удэ	10	18.09.02

В качестве первичного ключа выберем составной ключ *Код_заказа-Код_материала*, однозначно определяющий каждый кортеж отношения. Данное отношение находится в 1NF, так как все значения столбцов являются атомарными. Эта таблица содержит избыточные данные. Например, одни и те же сведения о клиенте повторяются в записи о каждом заказанном материале. Результатом избыточности являются следующие аномалии:

- адрес клиента можно ввести в базу данных тогда, когда он заказал хотя бы одну ткань;
- при удалении записи о заказанной ткани одновременно удаляются сведения о заказе и клиенте, его разместившем;
- при смене адреса клиента необходимо обновлять все записи и заказанных им материалах.

Напомним, что отношение находится во второй нормальной форме, если оно находится в первой нормальной форме и его неключевые атрибуты полностью зависят от всего первичного ключа. В таблице *Поставки* неключевые атрибуты *Код_клиента*, *Город_клиента*, *Дата_поставки* зависят от атрибута *Код_заказа*, являющегося частью составного ключа. Поэтому отношение *Поставки* не соответствует второй нормальной форме.

Для того чтобы перейти от первой ко второй нормальной форме, необходимо выполнить следующие шаги:

- определить, на какие части можно разбить первичный ключ так, чтобы некоторые из неключевых атрибутов функционально полно зависели от одной из этих частей;
- создать новое отношение для каждой такой части ключа и группы зависящих от неё атрибутов и переместить их в это отношение (т.е. построить проекции на части составного первичного ключа и атрибуты, зависящие от этих частей). Часть бывшего ключа станет при этом первичным ключом нового отношения;
- удалить из исходной таблицы атрибуты, перемещённые в другие отношения, кроме тех из них, которые станут внешними ключами (построить проекцию без атрибутов, находящихся в частичной функциональной зависимости от первичного ключа).

Для приведения исходной таблицы ко второй нормальной форме поля *Код_заказа*, *Код_клиента*, *Город_клиента*, *Дата_поставки* перемещаются в новую таблицу *Поставки1*, при этом *Код_заказа* – первичный ключ новой таблицы.

Вторая таблица *Заказы* будет содержать составной первичный ключ *Код_заказа-Код_материала* и поле *Количество*.

В результате новые таблицы будут выглядеть следующим образом (табл. 3.12, 3.13):

3.12. Поставки1

Код_ заказа	Код_ клиента	Город_ клиента	Дата_ поставки
110	BN	Омск	25.07. 02
120	BR	Москва	12.08. 02
130	BR	Москва	14.08. 02
140	BS	Тюмень	26.08. 02
150	BN	Омск	4.09.02
160	AN	Улан-Удэ	18.09. 02

3.13. Заказы

Код_ заказа	Код_материала	Количество
110	21	300
110	30	200
120	13	160
120	15	150
120	16	80
120	18	250
130	20	120
130	55	200
130	71	300
140	2	300
140	62	90
150	31	60
150	70	20
160	69	10
160	30	50

Проанализируем полученные таблицы. В таблице *Заказы* не наблюдается явная избыточность данных. Однако для таблицы *Поставки1* можно указать следующие аномалии:

- адрес конкретного клиента может содержаться в базе только тогда, когда есть заказы;
- удаление сведений о заказе в таблице *Поставки1* приведёт к удалению сведений о клиентах;
- при изменении адреса заказчика придётся обновить все кортежи в таблице *Поставки1*.

Устранить эти аномалии позволяет *третья нормальная форма*. Считается, что таблица соответствует третьей нормальной форме, если она находится во второй нормальной форме и её неключевые атрибуты взаимно независимы и зависят только от первичного ключа. В отношении *Поставки1* существует транзитивная зависимость между неключевыми атрибутами Город_клиента и Код_клиента (Код_клиента → Город_клиента).

Для перехода от второй нормальной формы к третьей необходимо исключить транзитивные зависимости. Для этого требуется выполнить следующие шаги:

- определить все атрибуты (или группы атрибутов), от которых зависят другие атрибуты (выявить транзитивные зависимости);
- создать новое отношение для каждого такого атрибута и для группы зависящих от него атрибутов и переместить их в это отношение (т.е. построить проекцию отношения на атрибуты, являющиеся причиной транзитивной зависимости). Атрибут, от которого зависят все остальные перемещённые атрибуты, станет при этом первичным ключом нового отношения;
- удалить перемещённые атрибуты из исходного отношения, оставив лишь те, которые станут внешними ключами.

Для приведения таблицы *Поставки1* к третьей нормальной форме создадим новую таблицу *Клиенты* (табл. 3.14) и переместим в неё атрибуты *Код_клиента* и *Город_клиента*. Атрибут *Город_клиента* из таблицы *Поставки1* удалим, а атрибут *Код_клиента* оставим в качестве внешнего ключа (табл. 3.16). Таблицу *Заказы* оставим без изменения (табл. 3.15).

3.14. Клиенты

Код_клиента	Город_ клиента
BN	Омск
BR	Москва
BS	Тюмень
AN	Улан-Удэ

3.15. Заказы

Код_заказа	Код_материала	Коли-чество
1	2	3
110	21	300
110	30	200
120	13	160
120	15	150
120	16	80
120	18	250
130	20	120
130	55	200
130	71	300
140	2	300
140	62	90
150	31	60
150	70	20
160	69	10
160	30	50

3.16. Поставки 2

Код_заказа	Код_клиента	Дата_поставки
110	BN	25.07.02
120	BR	12.08.02
130	BR	14.08.02
140	BS	26.08.02
150	BN	4.09.02
160	AN	18.09.02

На практике в большинстве случаев процесс проектирования заканчивается построением третьей нормальной формы. Для нашего примера после проведения нормализации можно заметить следующие улучшения:

- сведения о клиенте можно хранить, если клиент не сделал ни одного заказа;
- сведения о заказанном материале можно удалить, не опасаясь удаления данных о клиенте и заказе;
- изменение адреса клиента или даты регистрации заказа теперь требуют изменения только одной записи.

Существуют нормальные формы более высокого порядка.

Подробнее с технологией нормализации можно ознакомиться в литературе по теории реляционных баз данных [5, 8, 10, 14, 18 – 20, 26 – 29, 31].

Контрольные вопросы и задания

1. Привести различия между иерархической и сетевой моделями данных.
2. Охарактеризовать реляционную модель данных.
3. Перечислить составные элементы реляционной модели.
4. Что такое первичный ключ?
5. Перечислить условия, при соблюдении которых таблицу можно считать отношением.
6. В чём суть целостности сущностей?
7. Определить условия ссылочной целостности.
8. Определить различие между первичным и внешним ключами.
9. Каково назначение внешних ключей?
10. В чём различия между реляционной алгеброй и реляционным исчислением?
11. Перечислить операции реляционной алгебры.

12. Назвать и охарактеризовать дополнительные операции реляционной алгебры, предложенные К. Дж. Дейтом.

13. Дать характеристику языку QBE.

14. Перечислить операторы языка SQL.

15. Выполнить сравнение языков QBE и SQL.

16. В чём заключается процесс нормализации?

17. Привести примеры аномалий ввода, модификации, удаления данных.

18. Что такое функциональные зависимости?

19. Перечислить требования нормальных форм.

20. Описать этапы перехода от первой нормальной формы ко второй.

21. Какая последовательность действий необходима для перехода к третьей нормальной форме?

22. Составить алгебраическое выражение (или последовательность реляционных операций), необходимое для выполнения следующих запросов к базе данных *поставщиков и материалов*:

- получить полную информацию обо всех поставках в Москве;
- получить номера материалов, поставляемых поставщиком из Тюмени;
- получить такие пары номеров материалов, которые одновременно поставляются одним поставщиком;
- получить общее количество товаров, поставляемых поставщиком S1;
- получить названия поставщиков, которые поставляют по крайней мере один материал типа п/ш;
- получить типы материалов, поставляемых поставщиком S1.

23. В компании есть несколько отделов, в каждом отделе есть несколько сотрудников, несколько проектов, несколько кабинетов. Каждый сотрудник имеет план работы (несколько заданий). Для таких заданий существует ведомость полученных вознаграждений. В каждом кабинете есть несколько телефонов. В базе данных должна содержаться следующая информация:

- для каждого отдела: номер отдела, бюджет и номер сотрудника, который возглавляет этот отдел;
- для каждого сотрудника: номер сотрудника, номер текущего проекта, номер кабинета, номер телефона, название заданий вместе с датами и размерами всех оплат;
- для каждого проекта: номер проекта и бюджет;
- для каждого кабинета: номер кабинета, площадь, номера всех телефонов, установленных в кабинете.

Составить множество нормализованных отношений для представления этой информации.

24. Создать реляционную схему базы данных предприятия сферы сервиса (парикмахерской, мастерской по ремонту бытовой техники, компьютерной фирмы и т.п.). Схема должна содержать как минимум семь таблиц, приведенных к третьей нормальной форме. Обосновать выбор структур таблиц, их взаимосвязь. Описать процесс нормализации таблиц.

25. Дать определение данных на SQL для базы данных *поставщиков* и *материалов*.

26. Сформулировать на SQL для базы данных *поставщиков* и *материалов* следующий запрос: «Получить названия поставщиков, поставляющих материал М2».

27. Сформулировать на SQL следующее обновление базы данных *поставщиков* и *материалов*: «Изменить тип материала п/ш на ч/ш», «Удалить все проекты, для которых нет поставок».

28. Сформулировать на QBE следующий запрос: «Вывести список клиентов, чья общая сумма годовых заказов превышает 70000».

29. Сформулировать на QBE следующий запрос: «Перечислить название и цену товаров, поставка которых осуществляется после 1.03.03».

30. Сформулировать на QBE следующее изменение базы данных: «Удалить сведения о клиенте с номером 3101».

4. СЕМАНТИЧЕСКОЕ МОДЕЛИРОВАНИЕ

4.1. ОСНОВНЫЕ ПОНЯТИЯ СЕМАНТИЧЕСКОГО МОДЕЛИРОВАНИЯ

Реляционная модель данных (РМД) достаточна для моделирования многих предметных областей. Но она имеет и свои недостатки:

1) в РМД нет достаточных средств для представления смысла (семантики) предметной области;

2) в РМД нет средств для представления зависимостей между данными (отношениями);

3) для многих приложений недостаточно представления данных в виде плоских таблиц;

4) РМД не предлагает никакого аппарата для разделения сущностей и связей.

Указанные недостатки привели к созданию семантических моделей данных (СМД).

На практике СМД используются на первой стадии проектирования БД. При этом в терминах СМД описывается концептуальная (понятийная) схема БД, которая затем преобразуется к реляционной или другой схеме. Этот процесс описан соответствующими методиками.

Семантика реальной предметной области должна независимым от модели способом представляться в сознании её создателя. Поэтому в последнее время получило развитие семантическое моделирование. Основная цель такого подхода – организация интерфейса проектировщика, а также конечного пользователя с информационной системой на уровне представлений в пределах предметной области, а не на уровне структур данных. Наиболее популярной в настоящее время инфологической, или как её ещё называют семантической, моделью является модель «сущность-связь» (entity-relationship, E/R). Сочетая в себе предметную наглядность и теоретическую обоснованность, она является мощным инструментом проектирования баз данных.

Модель «сущность-связь» разработана Ченом (Chen) в 1976 г. в целях упрощения концептуального проектирования баз данных, на использовании которой основано большинство современных подходов к проектированию баз данных.

Модель представляет собой ряд графических диаграмм, включающих небольшое число разнородных компонентов.

Основными компонентами ER-модели являются сущность, связь и атрибут.

Сущность – это реальный или абстрактный объект, информация о котором должна сохраняться.

Считается, что каждая сущность полностью описывается совокупностью её свойств. Таким образом, каждая сущность должна иметь следующие характеристики:

- имя;
- объём (количество сущностей, которые можно описать данным понятием);
- содержание (совокупность свойств).

Свойство (*атрибут*) – это элементарная единица структуры сущности, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности. Атрибут – неотъемлемое свойство сущности или связи. По значениям атрибутов можно идентифицировать сущность. Значения атрибутов составляют основную часть сведений, хранящихся в базе данных.

Например, свойства сущности СТУДЕНТ – фамилия, имя, № зачётки, группа, адрес, возраст, пол и т.п.

Набор свойств сущности в принципе бесконечен, но он зависит от информационных потребностей пользователя и решаемых им задач. При составлении инфологической модели следует как можно более полно описать свойства сущности, учитывая не только текущие задачи, но и возможные будущие потребности пользователей базы данных.

С понятием *свойства* (атрибута) тесно связано понятие *домена*.

Домен – множество значений, которые может принимать атрибут. Домены могут быть бесконечными (хотя и перечислимыми) множествами.

Атрибуты делятся на три типа :

- *Ключевые атрибуты* позволяют различить сущности внутри одного понятия (ключ может состоять из нескольких атрибутов).
- *Дифференциальные атрибуты* содержат смысл сущности (то, что отличает её от других сущностей).
- *Валентные атрибуты* служат для связи между разными сущностями.

Например, «Код студента» – ключевой атрибут, так как он уникален для каждого студента; Фамилия, Имя, Адрес – дифференциальные атрибуты (они отличают эту сущность, например, от сущности ПРЕДПРИЯТИЕ; Группа – валентный атрибут, так как он может применяться для связи с сущностью СПЕЦИАЛЬНОСТЬ.

Для краткого описания сущности используется его схема, которая представляет собой совокупность имён атрибутов сущности: P (S1, S2, ..., Sn). Ключевой атрибут в схеме сущности выделяют подчёркиванием.

Например: СТУДЕНТ (Номер зачётки, ФИО, Адрес, Группа); СПЕЦИАЛЬНОСТЬ (Код, Название, Факультет, Группы).

Атрибуты делятся на:

- простые;
- составные;
- однозначные;
- многозначные;
- производные.

Простой атрибут состоит из одного компонента с независимым существованием.

Составной атрибут состоит из нескольких компонентов, каждый из которых характеризуется независимым существованием.

Однозначный атрибут содержит одно значение для одной сущности.

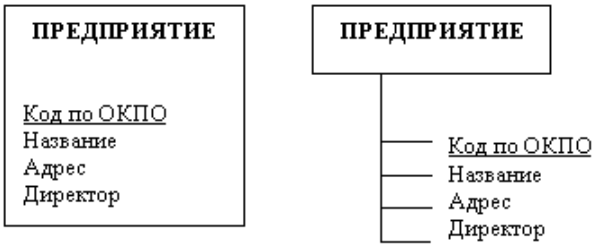
Многозначный атрибут может содержать несколько значений для одной сущности.

Производный атрибут представляет значение, производное (вычисляемое) от значения связанного с ним атрибута или множества атрибутов, принадлежащих некоторой сущности.

На ER-диаграммах свойства сущностей отображаются по-разному в зависимости от разновидности модели.

Например, на рис. 4.1, *а* имена свойств записаны малыми буквами внутри прямоугольника, изображающего сущность, под именем сущности. На рисунке 4.1, *б* свойства вынесены за прямоугольник сущности и связаны с ним линиями, ключевое свойство выделено.

Связь – это ассоциация между двумя понятиями. Она всегда является бинарной и может существовать между двумя разными сущностями или между сущностью и ей же самой (рекурсивная связь). В любой связи выделяются два конца, на каждом из которых указывается имя конца связи, степень конца связи (сколько экземпляров сущностей связывается), обязательность связи (т.е. любая ли сущность должна участвовать в данной связи).



а) *б*)

Рис. 4.1. Варианты изображения свойств сущности

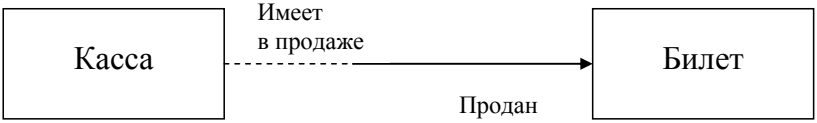


Рис. 4.2. Изображение связи на ER-диаграмме

На диаграмме связь может изображаться в виде линии между сущностями (рис. 4.2). В месте присоединения связи к сущности используется «вилка» или стрелка, если связь может относиться к нескольким экземплярам этой сущности. Одноточечный вход соответствует связи, в которой может участвовать только один экземпляр данной сущности. Обязательный конец связи изображается сплошной линией, а необязательный – прерывистой линией.

Каждый БИЛЕТ продан одной конкретной КАССОЙ; каждая КАССА может иметь в продаже 0, один или более билетов.

Важнейшими характеристиками связей между сущностями, а также между сущностями и их атрибутами являются следующие:

- *время существования* – различают постоянные, долговременные и кратковременные связи. **Например**, значение свойства РАЗМЕР_СТИПЕНДИИ для понятия СТУДЕНТ обновляется раз в семестр, а свойство НОМЕР_ЗАЧЕТКИ постоянно на протяжении обучения. Время существования влияет на то, как отображается свойство или связь в базе данных: будет ли она заложена в структуру БД либо будет реализована алгоритмическим путём (в программе). Кратковременные, легко вычисляемые свойства и связи рекомендуется реализовывать алгоритмически;
- *избирательность* – различают необязательные, возможные, условные и обязательные связи. **Например**, связь понятий СТУДЕНТ-СТИПЕНДИЯ зависит от успеваемости (условная связь), ЛИЧНОСТЬ – ИНОСТРАННЫЙ ЯЗЫК связаны необязательной связью, если предметная область – компрессорный завод, и обязательной, если предметная область – лингвистический институт;
- *ассоциативность* **(**степень, мощность, кардинальность) – различают связи 1:1, 1:M, N:M. Показатель кардинальности указывает количественное соотношение сущностей для каждой связи.

Связь 1:1 (*один-к-одному*) – это такая, при которой каждой сущности понятия A соответствует только одна сущность понятия B (рис. 4.3), **например:** СТУДЕНТ – НОМЕР_ЗАЧЁТКИ

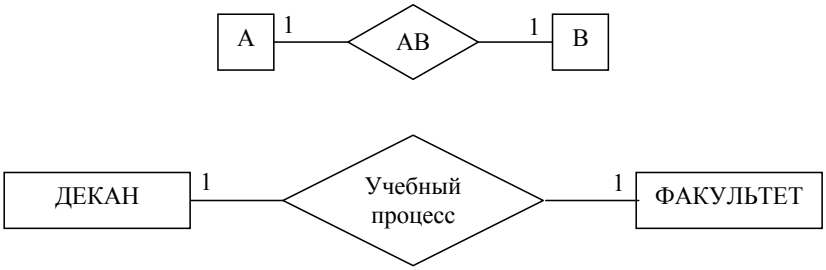


Рис. 4.3. Связь один-к-одному

Связь 1:M (*один-ко-многим*) – это такая, при которой каждой сущности понятия A соответствует несколько (или 0) сущностей понятия B, а одной сущности понятия B соответствует только одна сущность понятия A (рис. 4.4), **например:** ГРУППА -> СТУДЕНТ.

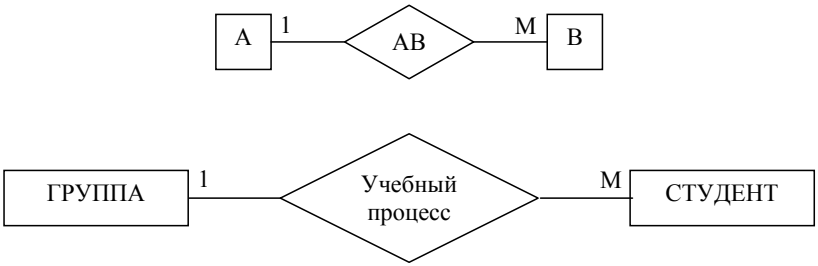


Рис. 4.4. Связь один-ко-многим

Связь N:M (*многие-ко-многим*) – это такая, при которой каждой сущности понятия A соответствует несколько (или 0) сущностей понятия B и наоборот (рис. 4.5), **например:** СТУДЕНТ <-> ПРЕПОДАВАТЕЛЬ.

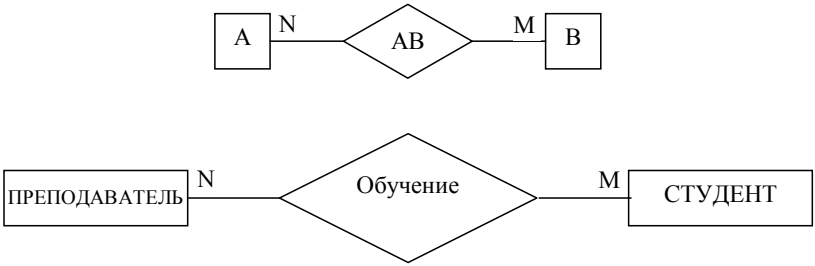


Рис. 4.5. Связь многие-ко-многим

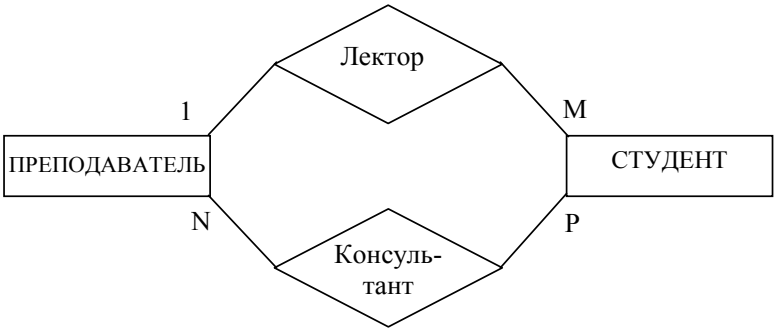


Рис. 4.6. Пример множества связей между понятиями ПРЕПОДАВАТЕЛЬ и СТУДЕНТ

Характер связей между сущностями может оказаться более сложным (рис. 4.6).

Степень связи – количество экземпляров сущности, которые охвачены данной связью (рис. 4.7).

Если связь определена между двумя понятиями, то её степень – 2, а называется такая связь *бинарной*. Связь между тремя понятиями называется *тернарной*, четырьмя понятиями – *кватернарной* и т.д. В общем случае связь между *n* понятиями называется *n*-арной.

Рекурсивная связь – связь, в которой одни и те же сущности участвуют несколько раз в разных ролях.

Рекурсивная связь часто называется *унарной*. **Например:** каждый студент из понятия СТУДЕНТ может исполнять обязанности дежурного по отношению к другим студентам того же понятия.

Связи (отношения) между сущностями возникают в результате осмысления человеком задач реального мира. Основной механизм познания мира – это абстрагирование, которое состоит в выделении существенных и игнорировании несущественных свойств и связей. Оно позволяет разбить задачу на более простые подзадачи и упорядочивает наши знания о реальном мире.

Различают три основных *типа абстрагирования* и соответственно три вида *операций над сущностями*: агрегация, обобщение, ассоциация. Соответственно обратные логические операции над сущностями – это декомпозиция, специализация и индивидуализация.

Агрегация – это метод абстрагирования, при котором сущность – агрегат связана с другими сущностями, как целое связано с частями. *Декомпозиция* – это деление сложной сущности на компоненты.

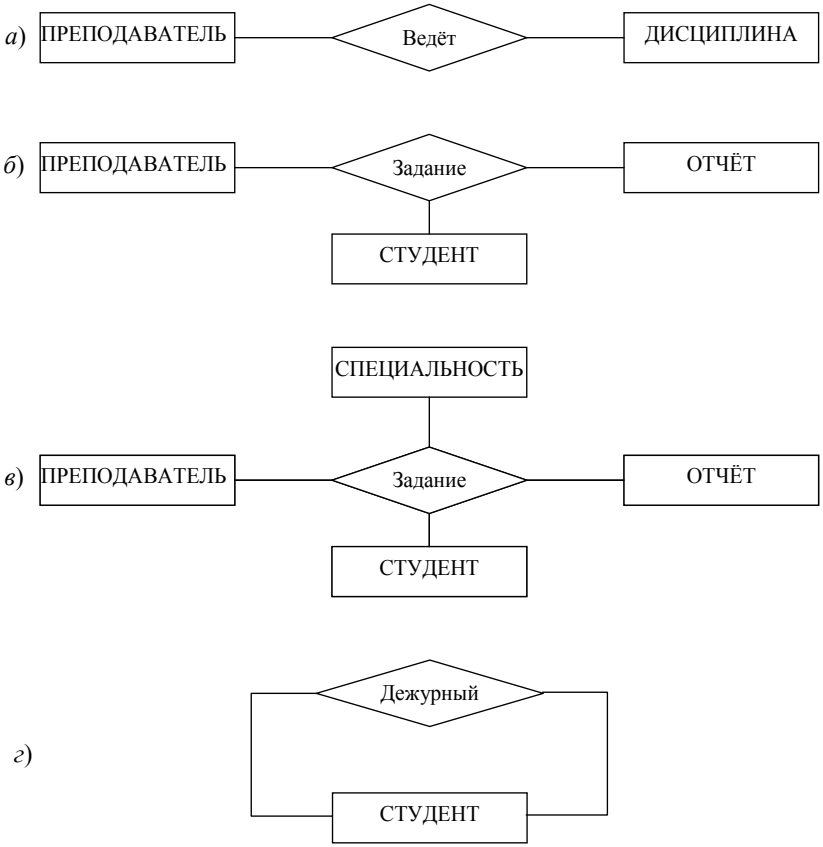


Рис. 4.7. Примеры связей:
а – бинарная; *б* – тернарная; *в* – кватернарная; *г* – унарная (рекурсивная)

Свойства агрегата – это совокупность свойств компонентов. Обычно составные части агрегата имеют совершенно разный набор свойств. **Например**, понятия ОБОРУДОВАНИЕ, АУДИТОРИЯ, СОТРУДНИКИ, ДИСЦИПЛИНЫ являются компонентами сущности КАФЕДРА. КАФЕДРА представляет собой сущность – агрегат.

На ER-диаграмме компоненты и агрегат изображаются прямоугольниками. Агрегат связан с компонентами связями типа «имеет/входит в» (рис. 4.8).

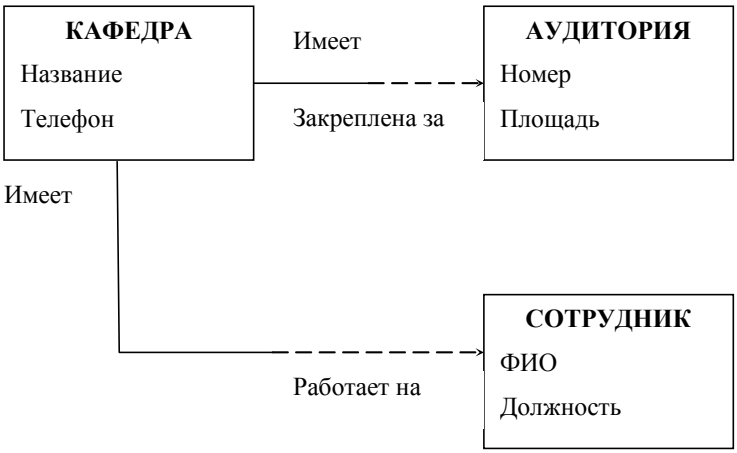


Рис. 4.8. Графическое представление агрегата
Агрегат КАФЕДРА состоит из двух компонентов:
АУДИТОРИЯ и СОТРУДНИКИ

Обобщение – это метод абстрагирования, при котором обобщенная сущность связана с другими сущностями отношением «род-вид», *специализация* – операция, обратная обобщению. **Например**, обобщённая сущность УЧАЩИЕСЯ может быть разделена на категории СТУДЕНТЫ, ШКОЛЬНИКИ, АСПИРАНТЫ.

Категории обычно имеют много общих свойств, но некоторые свойства у них различны. Свойства обобщённого понятия представляют собой перечень общих свойств категорий, но могут включать и дополнительные свойства по сравнению с исходными, чаще всего это такие свойства, как Среднее значение, Сумма и т.п. На ER-диаграмме категории изображаются треугольниками, а обобщённое понятие – прямоугольником, который соединён с категориями связями типа «является видом… (is kind of..)» (рис. 4.9).

Ассоциация – это такая логическая операция, которая устанавливает связи между конкретными экземплярами разных сущностей. *Индивидуализация* – это независимое рассмотрение связанных ассоциацией сущностей. Обычно ассоциативные связи возникают в ходе реальных бизнес-процессов. **Например**, сущности СТУДЕНТ, ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА ассоциируются с сущностью ЧТЕНИЕ ЛЕКЦИЙ или СДАЧА ЭКЗАМЕНА. Сущности ТОВАР, МАГАЗИН, ПРОДАВЕЦ связаны отношением ПРОДАЖИ.

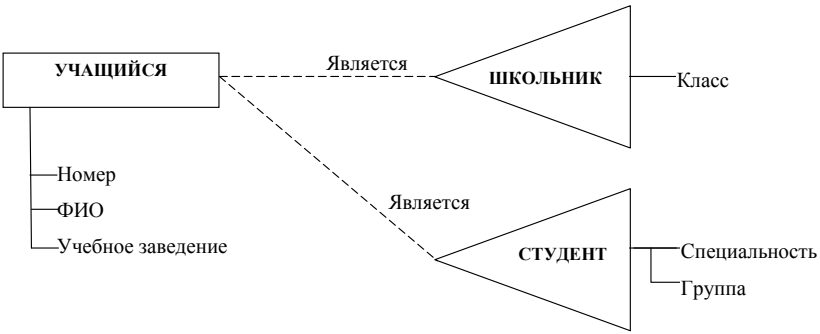


Рис. 4.9. Графическое представление обобщенного понятия

На ER-диаграмме ассоциация может отображаться по-разному (рис. 4.10):

а) в виде понятия, которое изображается ромбом, который соединен линиями со связанными понятиями;

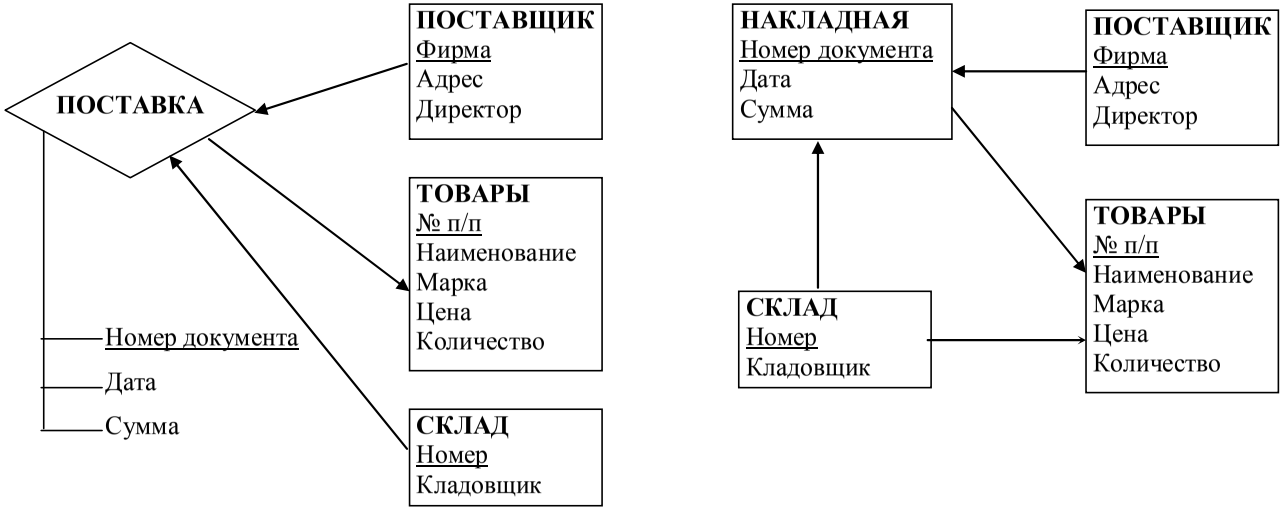
б) в виде множества бинарных связей между простыми понятиями.

Вариант а) применяется обычно в двух случаях: если ассоциация соответствует процессу, который сопровождается составлением документов, реквизиты которых не отображены в виде свойств других сущностей, либо если процесс соединяет более двух сущностей.

ER-модели широко применяются как при ручном, так и при автоматизированном проектировании баз данных. При этом в некоторых системах, например в системе IDEF, отсутствуют сложные понятия (агрегаты, ассоциации, обобщённые понятия), они заменяются связями. В разных системах применяются различные обозначения мощности связей и классов членства (обязательности связи) и т.д.

В заключение приведем рекомендуемый порядок построения ER-модели:

1. В каждом внешнем представлении выявить сущности и их свойства исходя из анализа экономических документов. Их должно быть не более 6–7 в каждом представлении.
2. Обозначить сущности именами, которые должны быть краткими, отражать смысл и быть привычными для пользователя.
3. Выбрать ключевое свойство (свойства). Если такого нет, то вводят условное свойство «Порядковый №».
4. Выявить связи между сущностями.
5. Найти общие свойства сущностей (обобщение).
6. Рассмотреть все пары сущностей и определить ассоциативные связи.
7. Определить степень каждой связи.



а) *б*)

Рис. 4.10. Графическое представление ассоциации

8. Записать формулы расчёта, связывающие вычисляемые свойства.

9. Объединить модели, построенные для разных внешних представлений. Объединение ведут по 2 – 4 модели за один шаг, получая более сложные объекты и устраняя противоречия (например, разные наименования одних свойств и одинаковые – для разных понятий).

Задача построения модели ПО осложняется тем обстоятельством, что деление информации на свойства, сущности и связи условно, и один и тот же элемент информации можно описать по-разному. **Например**, информацию о преподавателе можно рассматривать как сущность, а можно и как свойство сущности ДИСЦИПЛИНА; любую ассоциированную сущность можно рассматривать как связь между несколькими сущностями. Принятие конкретного решения зависит от специфики предметной области и применяемых средств автоматизации проектирования.

Рассмотрим пример построения ER-модели. В качестве предметной области примем видеотеку. Детализируя диаграмму потоков данных, для каждого из четырёх основных процессов мы получили четыре хранилища данных: Члены видеотеки, Фильмы, Аренда и Поставщики. В ER-модели им будут соответствовать четыре одноименных сущности. Определим структуру каждой сущности, принимая во внимание реквизиты документов, которые обрабатываются в видеотеке: учётные карточки членов видеотеки, карточки учёта фильмов, справочник поставщиков фильмов и журнал учёта аренды фильмов. Свойства понятий сведём в табл. 4.1.

Рассмотрим ассоциативные связи между сущностями, которые соответствуют информационным процессам.

Таблица 4.1

Сущность	Свойства
Члены видеотеки	ФИО, Адрес, Телефон, Интересы
Поставщики	Фирма, Адрес, Телефон, Расчётный счёт, К кому обращаться
Фильмы	№ по каталогу, Название, Тип (тематика), Страна-производитель, Год выпуска, Анонс, Цена, Количество дисков
Аренда	№ п/п, Дата выдачи, Отметка о возврате, Фильм, ФИО клиента

а) В процессе учёта членов видеотеки при регистрации нового члена необходимо убедиться, не был ли он записан раньше, не является ли должником. При исключении из членов следует проверить, вернул ли клиент все арендованные фильмы. Данный процесс использует информацию из сущностей Члены и Аренда (рис. 4.11, *а*) Линия со стрелкой отображает связь между сущностями типа «один-комногим», например, каждому члену видеотеки может соответствовать много записей в журнале аренды, но каждой записи в журнале соответствует только один конкретный клиент. При регистрации каждого нового клиента нужно проверить все карточки о членах видеотеки, поэтому на диаграмме показана рекурсивная связь.

б) При выдаче фильмов в аренду регистратор должен проверить, является ли клиент членом видеотеки, не числятся ли за ним несданные фильмы, затем в журнал аренды заносятся сведения об аренде фильма, а количество имеющихся дисков уменьшается на 1. При возврате фильмов следует в журнале внести пометку о возврате, а количество имеющихся в наличии дисков данного фильма увеличить на 1. Кроме того, регистратор может предложить клиентам справки об имеющихся в наличии фильмах. Соответствующая диаграмма показана на рис. 4.11, *б*.

в) Процесс «Учёт поставщиков» соединяет сущности Фильмы и Поставщики (рис. 4.11, *в*) с помощью ассоциированной сущности «Поставки».

г) Управление фильмотекой заключается в периодическом предоставлении руководству отчётов об имеющихся фильмах и о ходе их аренды, поэтому данный процесс соединяет сущности «Фильмы» и «Аренда» (рис. 4.11, *г*). Содержание отчётов отображает простая сущность «Отчёты начальству».

Обратите внимание, что ассоциированные сущности, как правило, имеют временное свойство (Дата поставки, Дата регистрации и т.п.) и ключевое свойство типа «Номер документа», так как они соответствуют информационным процессам, протекающим во времени. В заключение объединим модели, полученные при анализе разных внешних представлений, устранив двусмысленности, дублирование атрибутов и рекурсивную связь (рис. 4.12).

Можно построить другой вариант ER-диаграммы, в которой ассоциированные сущности заменены непосредственными бинарными связями между объектами. Таким образом, варианты модели могут различаться. Следует выбрать тот вариант, который наиболее просто, понятно и полно отображает необходимую информацию о предметной области.

Рис. 4.11. Построение фрагментов ER-модели при анализе информационных процессов:

а – связи при регистрации нового члена и исключении из членов; *б* – связи при учёте аренды фильмов (простое понятие «Аренда» заменено ассоциативным понятием); *в* – связи при учёте поставок фильмов; *г* – связи при выдаче отчётов руководству

Рис. 4.12. Объединённая диаграмма ER-модели предметной области «Видеотека»

В процессе создания инфологической модели могут возникать нежелательные ситуации, которые называются *ловушками соединения*. Причины этих проблем кроются в неправильной интерпретации семантики предметной области, в том числе смысла связей между выделенными понятиями.

Очень важно своевременно выявлять в модели данных ловушки соединения, иначе они могут привести к неадекватному описанию предметной области и необходимости перестройки всей концептуальной модели. Наиболее распространёнными являются два вида ловушек соединения:

1) ловушки разветвления;
2) ловушки разрыва.

Ловушка разветвления имеет место в том случае, когда модель отображает связь между понятиями, но путь между отдельными сущностями этих понятий однозначно не определяется.

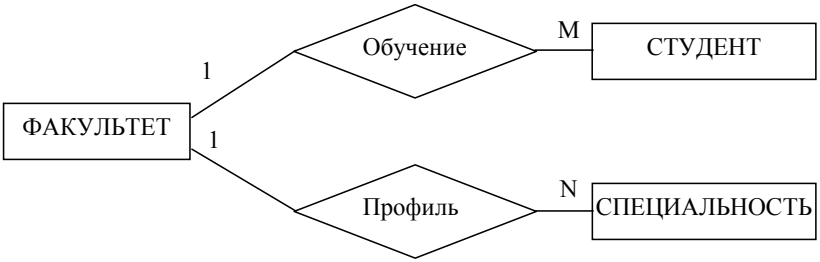


Рис. 4.13. Пример ловушки разветвления

Ловушка разветвления возникает в случае, если две или больше связей один-ко-многим разветвляются из одного понятия. Потенциальная ловушка разветвления показана на рис. 4.13, где две связи типа 1:M выходят из одного понятия ФАКУЛЬТЕТ. Из модели следует, что на одном факультете осуществляется обучение по нескольким специальностям, и на факультете учится множество студентов. Проблема может возникнуть при попытке выяснить, по какой специальности обучается каждый из студентов факультета.

Для обнаружения этой проблемы удобно пользоваться семантическими сетями (рис. 4.14). С помощью семантической модели

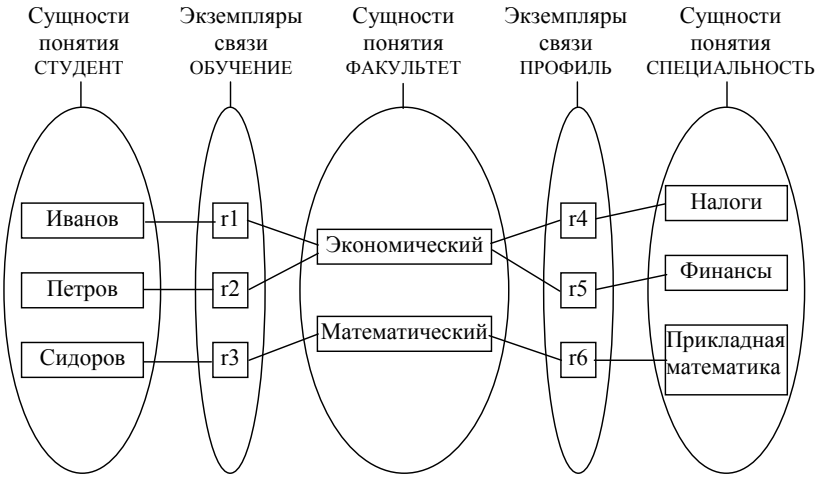


Рис. 4.14. Семантическая сеть ER-модели с ловушкой разветвления

на конкретном примере невозможно дать однозначный ответ на вопрос: «По какой специальности обучается студент Петров?» – это ловушка разветвления. Эта ситуация возникла из-за неправильной трактовки связей между сущностями ФАКУЛЬТЕТ, СПЕЦИАЛЬНОСТЬ, СТУДЕНТ.

Эту проблему можно решить только путём перестройки исходной модели.

Результат адекватного преобразования модели представлен на рис. 4.15. На этой модели можно определить, что студент Петров учится на экономическом факультете по специальности «Налоги».

Если проверить полученную модель на уровне отдельных сущностей (как показано на рис. 4.16), то можно убедиться, что по преобразованной модели можно дать ответ на поставленный вопрос.

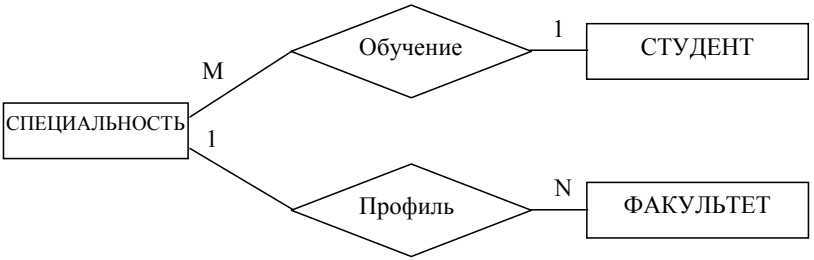


Рис. 4.15. Преобразованная ER-модель

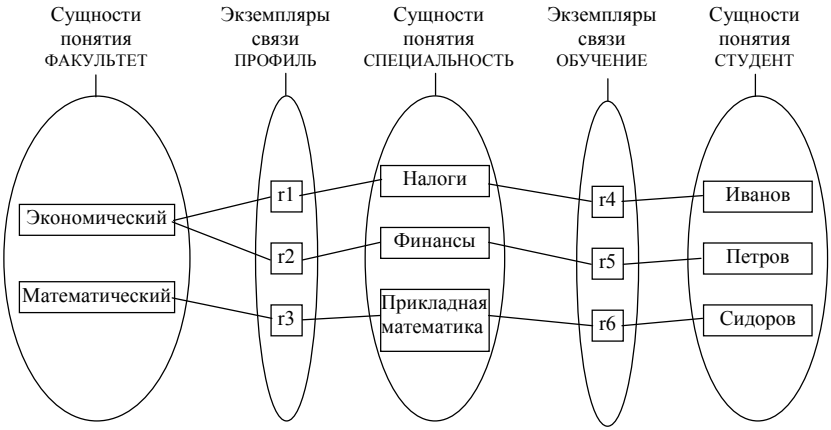


Рис. 4.16. Семантическая сеть преобразованной ER-модели

Ловушка разрыва появляется в том случае, если в модели предполагается наличие связи между сущностями, но не существует пути между отдельными экземплярами этих сущностей.

Ловушка разрыва возникает при наличии связи, образующей часть пути между связанными сущностями. Потенциальная ловушка разрыва показана на примере связей между сущностями ОБЩЕЖИТИЕ, СТУДЕНТ, КОМНАТА (рис. 4.17).

С помощью семантической сети ER-модели (рис. 4.18), невозможно дать ответ на вопрос: «В каком общежитии находится комната с номером 534?». Это возникает из-за неправильной интерпретации связей между сущностями ОБЩЕЖИТИЕ, СТУДЕНТ и КОМНАТА.

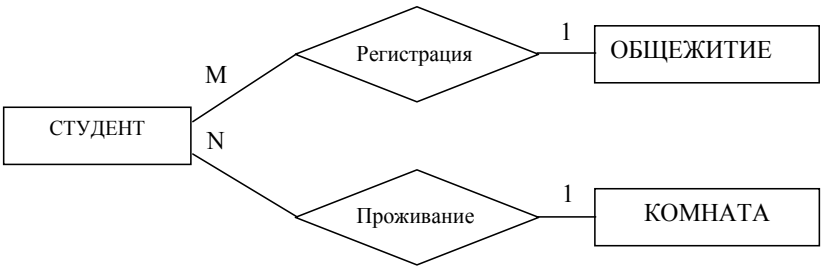


Рис. 4.17. Пример ловушки разрыва

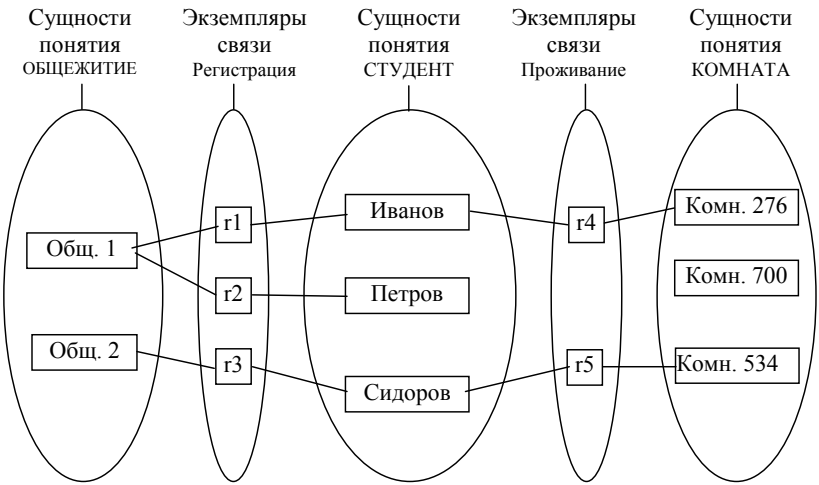


Рис. 4.18. Семантическая сеть ER-модели с ловушкой разветвления

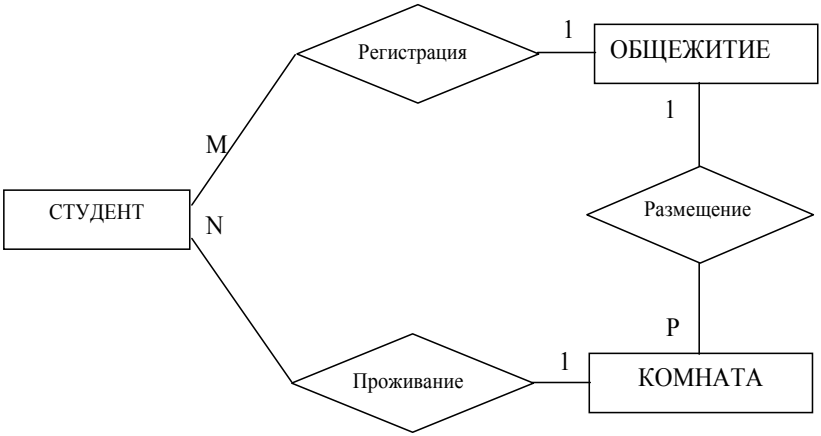


Рис. 4.19. Преобразованная ER-модель

Устранить эту проблему можно только путём перестройки ER-модели для представления правильного взаимоотношения между этими сущностями. Преобразованная ER-модель показана на рис. 4.19. В модель добавлена связь Размещение между сущностями ОБЩЕЖИТИЕ и КОМНАТА.

Если исследовать новую структуру на уровне отдельных сущностей (рис. 4.20), то можно ответить на поставленный вопрос: «Комната с номером 534 находится в общежитии № 2?».

Рассматривая методику построения ER-модели, мы использовали фрагменты различных нотаций. На сегодняшний день наибольшую популярность получили:

1. Методология Питера Чена.

В данной нотации сущность представляет собой множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, идей, предметов и т.п.), обладающих общими атрибутами или характеристиками. Любой объект системы может быть представлен только одной сущностью, которая должна быть уникально идентифицирована. При этом имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (например, Аэропорт, а не Внуково).

Отношение в самом общем виде представляет собой связь между двумя и более сущностями. Именование отношения осуществляется с помощью грамматического оборота глагола (имеет, определяет, может владеть и т.п.).

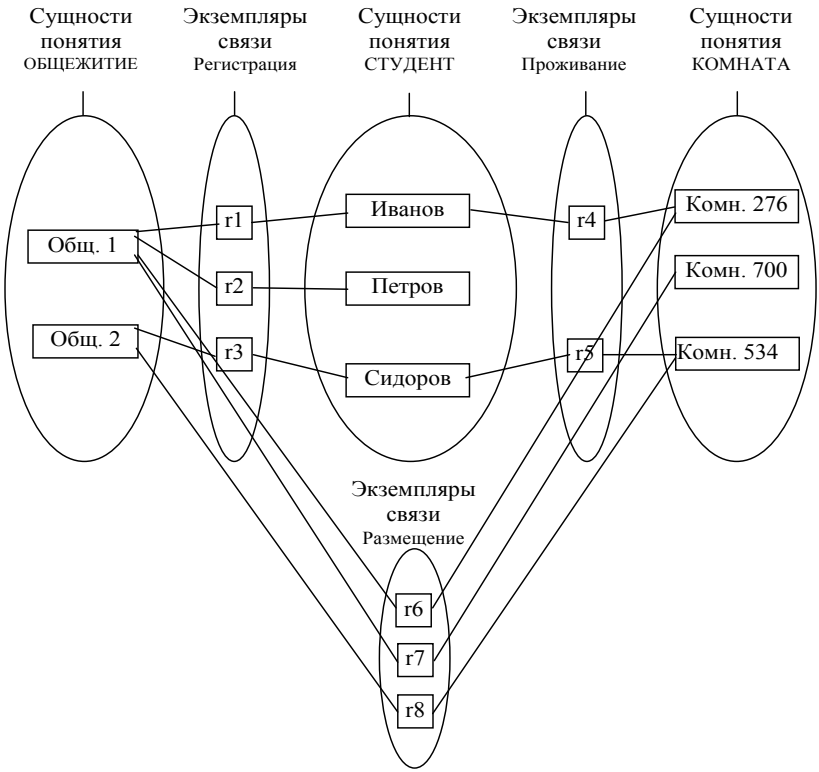


Рис. 4.20. Семантическая сеть преобразованной ER-модели

Другими словами, сущности представляют собой базовые типы информации, хранимой в базе данных, а отношения показывают, как эти типы данных взаимоувязаны друг с другом. Введение подобных отношений преследует две основополагающие цели:

1) обеспечение хранения информации в единственном месте (даже если она используется в различных комбинациях);

2) использование этой информации различными приложениями.

Основные элементы, используемые при построении ИЛМ (информационно-логической модели) по нотации Чена, представлены в табл. 4.2.

Связь соединяется с ассоциируемыми сущностями линиями. Возле каждой сущности на линии, соединяющей ее со связью, цифрами указывается класс принадлежности (рис. 4.21).

4.2. Основные элементы в нотации Чена

Элемент диаграммы	Обозначает
имя	Независимая сущность
имя	Зависимая сущность
имя	Родительская сущность в иерархической связи
имя	Связь
имя	Идентифицирующая связь
имя	Атрибут
имя	Первичный ключ (внешний ключ с одной чертой)

2. Методология IDEF1. Используется в CASE-средствах ERwin, Design/IDEF. В методологии используются следующие соглашения:

– каждому классу объектов присваивается уникальное имя и номер;

– обязательная связь отображается сплошной линией, необязательная – пунктирной;

– мощность связи «один» отображается линией, «много» – точкой;

– связь может дополнительно определяться с помощью указания мощности (типа) связи. Мощность может принимать следующие значения: N – ноль, один или более (принимается по умолчанию); Z – ноль или один, P – один или более;

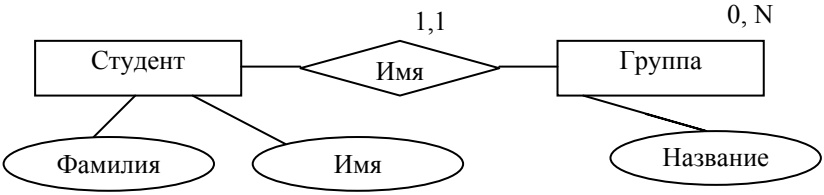


Рис. 4.21. ER-модель по нотации Чена

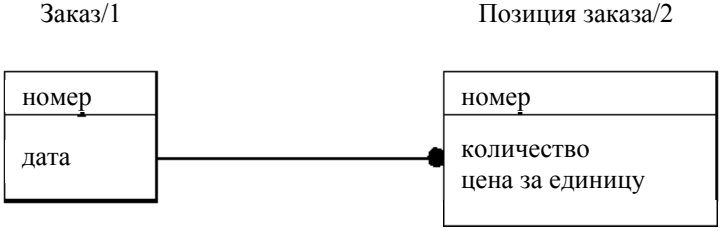


Рис. 4.22. Пример представления ER-диаграммы в методологии IDEF1

– свойства класса объектов отображаются в виде списка имен внутри блока, отображающего класс объектов;

– атрибуты первичного ключа изображаются вверху и отделяются от других.

Пример представления ER-диаграммы в методологии IDEF1 приведён на рис. 4.22.

На рисунке 4.22 отображена та же ситуация в предметной области, что и на рис. 4.21.

3. Методология Ричарда Баркера. Используемые в методологии элементы: класс объектов, свойство класса объектов, уникальные идентификаторы, опциональность свойств, связи, мощность (тип), опциональность и переносимость связей, уникальность объекта из связи, супертипы, подтипы, арки.

В нотации Баркера используется только один тип диаграмм – ER. Сущность представляется прямоугольником любого размера, содержащим внутри себя имя сущности, список имён атрибутов (возможно, неполный) и указатели ключевых атрибутов (знак «#» перед именем атрибута).

Все связи являются бинарными и представляются линиями с двумя концами (соединяющими сущности), для которых должно быть определено имя, степень множественности и степень обязательности. Для множественной связи линия присоединяется к прямоугольнику сущности в трёх точках, а для одиночной связи – в одной точке. При обязательной связи рисуется непрерывная линия до середины связи, при необязательной – пунктирная линия.

Читается связь отдельно для каждого конца, показывая, как сущность 1 связывается с сущностью 2, и наоборот. Пример оформления показан на рис. 4.23.

Также для построения диаграмм можно использовать нотацию Мартина, основные элементы которой приведены в табл. 4.3.

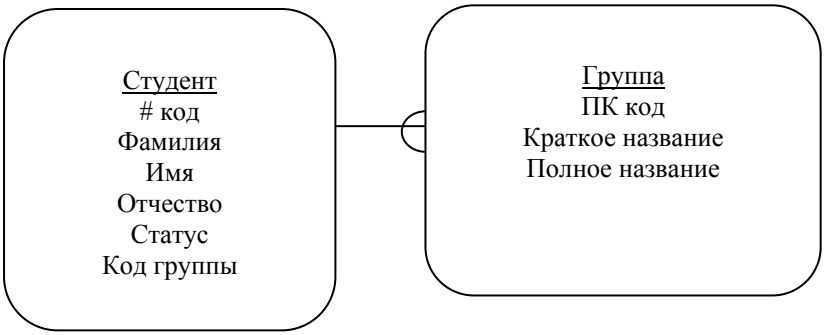


Рис. 4.23. Пример оформления диаграммы по нотации Баркера

Список атрибутов приводится внутри прямоугольника, обозначающего сущность. Ключевые атрибуты подчёркиваются. Связи изображаются линиями, соединяющими сущности, вид линии в месте соединения с сущностью определяет кардинальность связи (табл. 4.4).

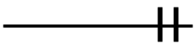
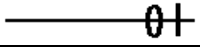
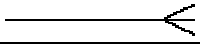
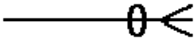
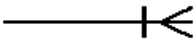
Имя связи указывается на линии, её обозначающей. Пример диаграммы, выполненной по нотации Мартина, приведён на рис. 4.24.

Помимо вышеперечисленных нотаций могут использоваться и другие: OMT, SSADM, нотация Гейна–Сарсона, Йордона–Де Марко и т.д. Все они обладают практически одинаковой функциональностью и различаются лишь в деталях. Например, в нотации Гейна–Сарсона для обозначения функций используются прямоугольники с закруглёнными углами, а также не рассматриваются управляющие потоки данных. В остальном эти системы обозначений эквивалентны.

4.23. Основные элементы нотаций Мартина

Элемент диаграммы	Обозначает
имя	Независимая сущность
имя	Зависимая сущность
имя	Родительская сущность в иерархической связи

4.4. Кардинальность связи в нотациях Мартина

Обозначение	Кардинальность
	Нет
	1,1
	0,1
	M,N
	0,N
	1,N

Диаграммы «сущность-связь» предназначены для разработки моделей данных и обеспечивают стандартный способ определения данных и отношений между ними. Фактически с помощью ER-диаграмм осуществляется детализация хранилищ данных проектируемой системы, а также документируются сущности системы и способы их взаимодействия, включая идентификацию объектов, важных для предметной области (сущностей), свойств этих объектов (атрибутов) и их отношений с другими объектами (связей).

На основе созданной ER-схемы строится реляционная модель базы данных. Порядок построения реляционной модели из ER-схемы следующий:

1. Каждая простая сущность отображается в таблицу, при этом имя сущности становится именем таблицы.

2. Каждый атрибут сущности становится столбцом таблицы с тем же именем, при этом выбирается более точный формат значения атрибута. Ключевые свойства преобразуются в первичные ключи таблицы.

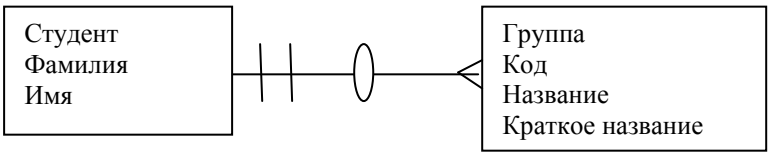


Рис. 4.24. Диаграмма, выполненная по нотации Мартина

3. Столбцы, соответствующие необязательным атрибутам, могут содержать неопределённые значения, обязательные столбцы не могут.

4. Если в соответствующие ключевые свойства входят связи, к числу столбцов первичного ключа добавляется копия ключевого свойства сущности, находящейся на дальнем конце связи.

5. Связи «один-ко-одному», «один-ко-многим» преобразуются во внешние ключи. Делается копия уникального идентификатора с конца связи «один», которая мигрирует в таблицу на конце связи «много», где и становится внешним ключом.

Правила генерации таблиц из ER-диаграмм опираются на два основных фактора – тип связи и класс принадлежности сущности.

Правило 1. Если *связь типа 1:1 и класс принадлежности обеих сущностей является обязательным*, то необходима только одна таблица. Первичным ключом этой таблицы может быть первичный ключ любой из двух сущностей.

Правило 2. Если *связь типа 1:1 и класс принадлежности одной сущности является обязательным*, *а другой – необязательным*, то необходимо построить таблицу для каждой сущности. Первичный ключ сущности должен быть первичным ключом соответствующей таблицы. Первичный ключ сущности, для которой класс принадлежности является необязательным, добавляется как атрибут в таблицу для сущности с обязательным классом принадлежности.

Сущность с необязательным классом принадлежности именуется *родительской,* а с обязательным – *дочерней.* Первичный ключ родительской сущности (НФ), помещаемый в таблицу, представляющую дочернюю сущность, называется *внешним* ключом родительской сущности. Связь между указанными таблицами устанавливается путем связи первичного и внешнего ключей. Если внешний ключ представляет связь 1:1, то должны быть запрещены его дублирующие значения.

Правило 3. Если *связь типа 1:1 и класс принадлежности обеих сущностей является необязательным*, то необходимо построить три таблицы – по одной для каждой сущности и одну для связи. Первичный ключ сущности должен быть первичным ключом соответствующей таблицы. Таблица для связи среди своих атрибутов должна иметь ключи обеих сущностей.

Итак, для связи типа 1:1 существуют три отдельных правила формирования предварительных таблиц из ER-диаграмм.

Для связи типа 1:M существуют только два правила. Выбор одного из них зависит от класса принадлежности сущности на стороне M. Класс принадлежности сущности на стороне 1 не влияет на выбор.

Правило 4. Если *связь типа 1:M и класс принадлежности сущности на стороне M является обязательным*, то необходимо построить таблицу для каждой сущности. Первичный ключ сущности должен быть первичным ключом соответствующей таблицы. Первичный ключ сущности на стороне 1 добавляется как атрибут в таблицу для сущности на стороне M. Если внешний ключ представляет связь 1:M, то должны быть разрешены его дублирующие значения.

Правило 5. Если *связь типа 1:M* и *класс принадлежности сущности на стороне M является необязательным*, то необходимо построить три таблицы – по одной для каждой сущности и одну для связи. Первичный ключ сущности должен быть первичным ключом соответствующей таблицы. Таблица для связи среди своих атрибутов должна иметь ключи обеих сущностей.

Для связи типа M:N класс принадлежности сущности не имеет значения.

Правило 6. Если *связь типа M:N*, то необходимо построить три таблицы – по одной для каждой сущности и одну для связи. Первичный ключ сущности должен быть первичным ключом соответствующей таблицы. Таблица для связи среди своих атрибутов должна иметь ключи обеих сущностей.

Контрольные вопросы и задания

1. Перечислить недостатки реляционной модели данных.
2. Основные понятия ER-модели. Привести пример ER-схемы.
3. Какие вы знаете операции над сущностями? Привести примеры.
4. Каков порядок построения ER-модели?
5. Каков порядок преобразования ER-модели в реляционную базу данных?
6. Построить ER-схему и на её основании реляционную базу данных для предметной области поставки товаров в магазин.
7. Построить ER-схему и на её основании реляционную базу данных для предметной области ШКОЛА, при этом требуется хранить информацию о школьниках, преподавателях, предметах, оценках.
8. Предложить свою предметную область и для неё построить ER-схему и на её основании реляционную базу данных.

5. БАЗЫ ДАННЫХ В СЕТЯХ

Повсеместное внедрение и развитие сетевых технологий, тенденция к распределению вычислительных ресурсов накладывают отпечаток на принципы построения СУБД. Применение сетевых технологий позволило значительно укрупнить информационные системы, увеличить число пользователей, обеспечить целостность в рамках всей системы.

В основе построения сетевых систем управления данными лежат два подхода: централизованный и децентрализованный. При централизованном подходе систему баз данных можно рассматривать как структуру, состоящую из двух частей – сервера и набора клиентов. Под сервером в данном контексте понимается СУБД, а клиенты – это различные приложения, инициирующие запросы на услуги к серверу. Данный вариант архитектуры называется архитектурой «клиент-сервер». Разделение системы баз данных на две части определяет возможность обработки этих частей на разных машинах. Поэтому можно говорить о распределённой обработке информации. Распределённая обработка может быть разнообразной и осуществляться на разных уровнях.

Децентрализованный подход предусматривает возможность определённого узла сети выступать как в роли клиента, так и в роли сервера данных. Очевидно, что в этом случае значительно усложняются задачи поддержки транзакций и обеспечения целостности.

В этой главе мы рассмотрим некоторые варианты распределённой обработки данных.

5.1. АРХИТЕКТУРА «КЛИЕНТ-СЕРВЕР»

Определим основные понятия *сервер* и *клиент*. Под сервером в информационных системах понимается программа (компьютер), предоставляющая услуги по запросам других программ (компьютеров), называемых клиентами. В контексте баз данных вводится понятие *сервер баз данных*:

1. СУБД в архитектуре «клиент-сервер».
2. Компьютер в сети, на котором поддерживается база данных и осуществляется обработка пользовательских запросов.

Сервер баз данных осуществляет целый комплекс действий по управлению данными. Перечислим его основные функции:

- выполнение пользовательских запросов на выбор и модификацию данных и метаданных;
- хранение и резервное копирование данных;

- поддержка ссылочной целостности данных согласно правилам, определённым в базе данных;
- обеспечение авторизованного доступа к данным на основе проверки прав и привилегий пользователей;
- протоколирование операций и ведение журнала транзакций (сущность транзакции будет рассмотрена ниже).

Клиентские узлы поддерживают пользовательские интерфейсы и функциональность приложений.

Основой архитектуры «клиент-сервер» является принцип централизации хранения и обработки данных. Централизованная база данных физически сосредоточена в одном месте и управляется одним компьютером-сервером. Классическая двухзвенная схема «клиент-сервер» представлена на рис. 5.1.

Архитектура «клиент-сервер» допускает различные варианты реализации. В первоначальном (централизованном) варианте архитектуры «клиент-сервер» приложение (пользовательская программа) не разбивалось на части, использовались ресурсы только одного мощного компьютера. СУБД, данные и приложения хранились на одном мощном мини-компьютере или мейнфрейме, принимающим входную информацию с пользовательского терминала и отображающего на нём данные (разделение функций было на процессном уровне – один процесс выполнял функции клиента, другой – сервера). С появлением персональных компьютеров и локальных вычислительных сетей появилась возможность распределения ресурсов по всем компьютерам

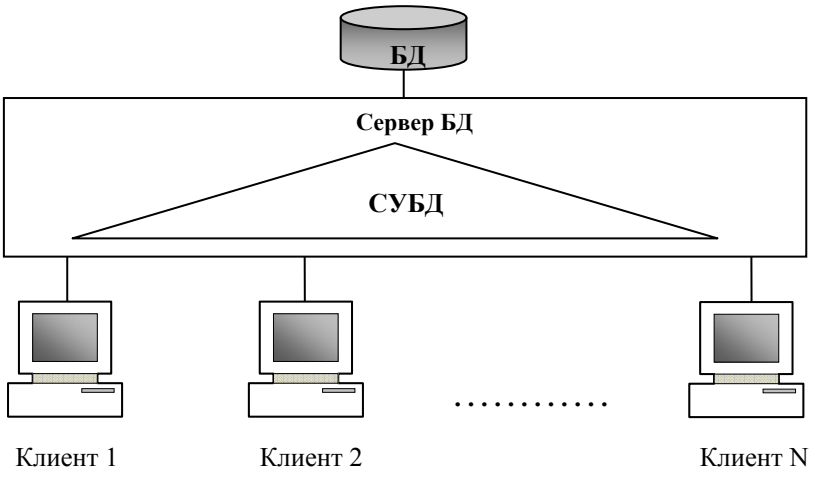


Рис. 5.1. Архитектура «клиент-сервер»

сети с позиции максимального использования их ресурсов. Основным принципом разбиения приложения на части является разделение на группы функций стандартного интерактивного приложения:

1. Функции ввода и отображения данных (Presentation Logic – презентационная логика). К этой функции относятся все интерфейсные экранные формы, с которыми работает пользователь, а также отображаемая на экране результативная и справочная информация.

2. Прикладные функции, определяющие основные алгоритмы решения задач (Business Logic – бизнес-логика).

3. Функции обработки данных внутри приложения (Database Logic – логика обработки данных).

4. Функции управления информационными ресурсами (Database Manager Logic). Это собственно СУБД, обеспечивающая хранение и управление базами данных.

5. Служебные функции, играющие роль связок между функциями первых четырёх групп.

В децентрализованной архитектуре «клиент-сервер» существуют различные варианты распределения функций между компьютером-сервером и компьютером-клиентом в двухзвенной модели: от мощного сервера, на котором производится практически вся работа, до мощного клиента, когда большая часть функций выполняется компьютером-клиентом, а сервер обрабатывает поступающие к нему по сети SQL-запросы. Рассмотрим наиболее популярные из них.

В модели *удалённого доступа к данным* (Remote Data Access, RDA) на сервере хранится база данных и ядро СУБД. На клиенте располагается презентационная логика и бизнес-логика. Клиент обращается к серверу с запросом на языке SQL, либо посредством средств пользовательского интерфейса (API). Данную модель поддерживает большое число серверных СУБД, имеющих SQL-интерфейсы, и инструментальных средств, обеспечивающих создание клиентской части.

Модель *сервера БД* (Database Server – DBS) характеризуется тем, что функции компьютера-клиента ограничиваются презентационной логикой. Большая часть бизнес-логики переложена на сервер. Иногда такую модель называют моделью с «тонким клиентом». Эта модель является более технологичной, чем RDA и применяется в таких СУБД, как Informix, Ingres, Sybase, Oracle, SQL MS Server.

Существуют и более сложные варианты реализации архитектуры «клиент-сервер», например, трёхзвенные информационные системы с использованием серверов приложений, реализующих бизнес-логику информационной системы (модель сервера приложений – Application Server (AS)) (рис. 5.2).

Центральным звеном модели является сервер приложений, на котором реализуется несколько прикладных функций. Серверов приложений может быть несколько, и каждый из них представляет свой вид сервиса.

Программные средства сервера приложений относятся к категории *программного обеспечения промежуточного слоя,* которое определяется как «некоторый набор процедур или функций, обеспечивающих взаимодействие двух разнородных программ». Программные средства этой категории применимы к компьютерным сервисам практически любого вида, включая управление базами данных и информацией. Многие компании-поставщики программного обеспечения выпускают программные продукты, основанные на стандартах IDAPI, ODBC, DRDA или других стандартах промежуточного слоя и предоставляющие интерфейсные возможности для клиента и сервера.

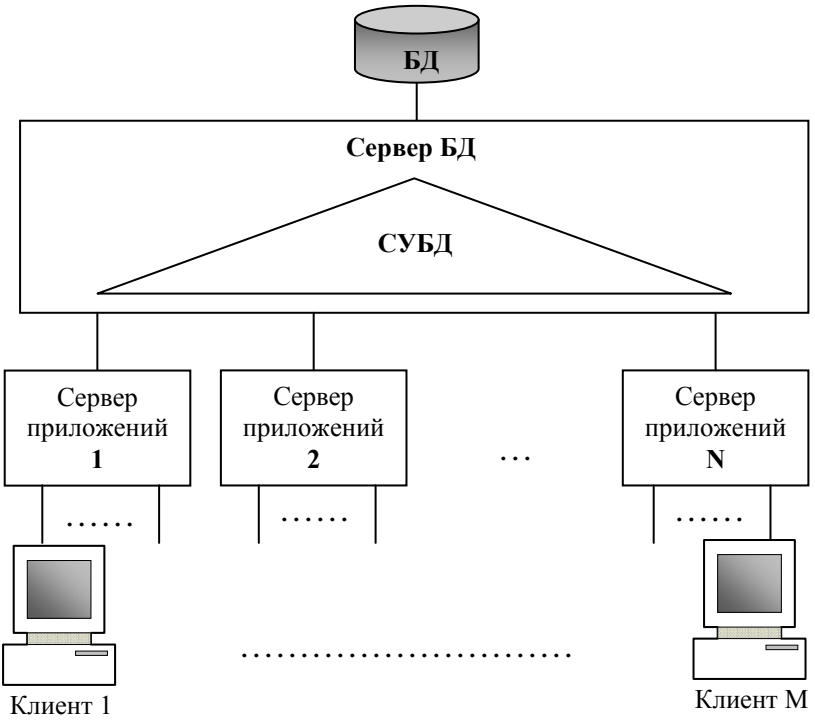


Рис. 5.2. Трёхзвенная модель архитектуры «клиент-сервер»

Затрагивая вопрос о категории промежуточного слоя, необходимо упомянуть о программных системах промежуточного слоя – мониторах транзакций.

Транзакцией называется совокупность операций манипулирования данными (вставкой, удалением, выборкой, обновлением) в системах баз данных, которая переводит базу данных из одного целостного (непротиворечивого) состояния в другое целостное состояние. Транзакция рассматривается как некоторое неделимое с точки зрения пользователя действие над базой данных. Например, транзакцией может быть последовательность операций по формированию заказа на сборку компьютера в компьютерной фирме: ввод нового заказа с реквизитами заказчика, формирование платёжного документа, запрос на комплектующие. С точки зрения сотрудника компьютерной фирмы, это единая операция: если она будет прервана, то база данных потеряет своё целостное состояние. Восстановление данных в СУБД является составной часть управления данными и подразумевает восстановление самой базы данных, т.е. возвращение базы данных в *правильное* состояние в случае изменения данных в результате сбоя. Восстановление данных реализуется через механизм транзакций. *Завершение* транзакции означает, что все операции, входящие в состав транзакций, успешно завершены и результат их работы сохранён в базе данных. *Откат* транзакции означает, что все уже выполненные операции отменяются и все объекты базы данных, затронутые этими операциями, возвращены в исходное состояние. Для реализации возможности отката транзакций большинство СУБД поддерживают запись в журналы транзакций, позволяющие сохранять промежуточные состояния и восстанавливать исходные данные при откате. Существуют различные модели транзакций.

Под монитором обработки транзакций (Transaction Processing Monitor – TPM) понимаются средства программного обеспечения промежуточного слоя, предназначенные для обеспечения эффективного управления информационными и вычислительными ресурсами в распределённых системах при обработке транзакций. Понятие транзакции в TPM шире, чем транзакция в СУБД. Основными функциями TPM являются: аутентификация пользователей и проверка полномочий доступа, управление коммуникациями, необходимыми для выполнения транзакций, собственно управление транзакциями, включая управление блокировкой ресурсов, журнализацию, фиксацию, откаты и восстановление транзакций.

Основное достоинство архитектуры «клиент-сервер» заключается в снижении сетевого трафика при выполнении запросов и распределе-

нии процесса загрузки базы данных. SQL обеспечивает определённый интерфейс между клиентской и серверной системами, эффективно передавая запросы к базе данных. Преимущества данной архитектуры обеспечили ей большую популярность. Многие известные компании-поставщики программного обеспечения предлагают серверные СУБД и инструментальные средства разработки клиентских приложений.

5.2. РАСПРЕДЕЛЁННЫЕ БАЗЫ ДАННЫХ

В настоящее время существует два класса СУБД: настольные и серверные. Работа с небольшой базой данных, расположенной на персональном компьютере, не подключённом к локальной сети (настольный вариант СУБД), становится уже нехарактерной для настоящего времени.

Компьютеры объединяются в локальные сети.

Даже если разрабатывается БД для небольшой фирмы, всегда есть территориально-удалённые специфичные пользователи.

Распределённая база данных состоит из составных частей, размещенных на разных узлах сети в соответствии с каким-либо критерием (рис. 5.3).

Распределённые базы данных (РаБД) управляются распределенными системами управления базами данных (РаСУБД). Существуют различные модели распределения данных. РаБД могут работать под управлением одинаковых и неодинаковых СУБД. В первом случае говорят об *однородных распределённых системах*, во втором – о *неоднородных*.

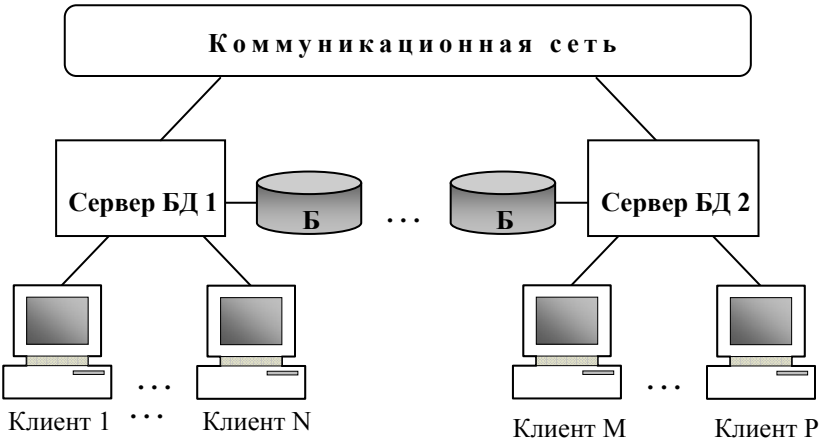


Рис. 5.3. Распределённая база данных

Однородные распределённые системы баз имеют в своей основе один продукт СУБД, обычно с единственным языком баз данных (например, SQL с расширениями для управления распределёнными данными). Существует множество различных вариантов построения однородной системы РаБД. Например, на некотором узле вычислительной сети может располагаться одна глобально доступная «главная машина СУБД», с которой связаны компоненты для доступа к данным локальных баз данных, размещённые совместно с самими этими базами данных в пределах всей компании (или отдельного её подразделения в зависимости от масштаба распределения). Более сложные модели могут допускать распределённость самой СУБД, когда каждый её компонент «на равных правах» имеет доступ к данным любого другого узла.

Неоднородные системы включают два или более существенно различающихся продукта управления данными. Неоднородные системы баз данных можно, в свою очередь, также подразделить на классы в широком диапазоне – от федеративных систем до различных типов систем мультибаз данных.

Однородные распределённые системы баз данных обычно проектируются методом «сверху вниз», в целом аналогично проектированию централизованных баз данных (создание концептуальной, логической, физической – моделей данных).

Неоднородные же, напротив, чаще всего строятся «снизу вверх» с целью создать общую среду управления над существовавшими ранее разрозненными базами данных. Основной проблемой при этом становится объединение схем баз данных таким образом, чтобы предоставить как новым, так и прежним приложениям доступ и к новым, и к прежним ресурсам данных. Процесс создания системы *мультибаз* данных технически сложен и нетривиален.

При создании однородных распределённых баз данных используется два метода распределения данных: фрагментация и тиражирование.

Фрагментация означает декомпозицию объектов базы данных, таких, как реляционные таблицы, на две или более частей, которые размещаются на разных компьютерных системах.

Проиллюстрируем это понятие примером. В БД имеется таблица, содержащая данные о сотрудниках, работающих в различных филиалах, или о заказах на продажу. Таблица может быть разделена на фрагменты по географическому или другому характеристическому признаку.

При *горизонтальной* фрагментации делаются горизонтальные «срезы» в соответствии со значением какого-либо столбца таблицы. Строки данных о сотрудниках могут разбиваться на подмножества,

соответствующие филиалам. Данные о продажах фрагментируются по магазинам, где эти продажи производились.

При *вертикальной* фрагментации разбиение таблицы осуществляется не по строкам, а по столбцам. В этом случае некоторая часть информации о каждом сотруднике хранится в одном месте, а другая часть (относящаяся к той же таблице) – в другом.

Независимо от вида фрагментации, поддерживается глобальная схема, представляющая единое описание всех составляющих её локальных баз данных, позволяющая воссоздать из имеющихся фрагментов логически централизованную таблицу или другую структуру базы данных.

Основным достоинством модели фрагментации является то, что пользователи всех узлов (при исправных коммуникационных средствах) получают информацию с учётом всех последних изменений. Второе достоинство состоит в экономном использовании внешней памяти компьютеров, что позволяет организовывать БД больших объёмов.

К недостаткам модели распределённой БД, основанной на фрагментации данных, относится следующее: жёсткие требования к производительности и надёжности каналов связи, а также большие затраты коммуникационных и вычислительных ресурсов из-за их связывания на всё время выполнения транзакций. При интенсивных обращениях к распределённой БД, большом числе взаимодействующих узлов, низкоскоростных и ненадёжных каналах связи обработка запросов по этой схеме становится практически невозможной.

Тиражирование (или *репликация*) означает создание копий некоторых фрагментов базы данных в целях приближения данных к месту их использования. Основное достоинство метода заключается в сокращении сетевого трафика и увеличении производительности системы. *Репликаторы* представляют множество различных физических копий некоторого объекта базы данных (обычно таблицы), для которых в соответствии с определёнными в базе данных правилами поддерживается синхронизация (идентичность) с некоторой «главной» копией. Теоретически значения всех данных в тиражированных объектах должны автоматически и незамедлительно синхронизироваться друг с другом. (На практике это правило обычно несколько ослабляется.) В некоторых системах копии используются исключительно в режиме чтения и обновляются в соответствии с заданным расписанием. В других средах допускается модификация отдельных значений в копиях, и эти изменения распространяются в соответствии с процедурами планирования и координации.

Основной *недостаток* метода тиражирования БД заключается в том, что на некотором интервале времени возможно «расхождение» копий БД.

Различные поставщики предлагают множество программных продуктов, позволяющих управлять распределёнными ресурсами. При этом в результате общего соглашения определились некоторые характеристики «идеальной» РаСУБД:

- *Прозрачность относительно местоположения*. Для пользователя не должно быть разницы, где физически находятся данные, данные должны «выглядеть» так, если бы они находились на локальном компьютере.
- *Гетерогенные системы.* СУБД должна работать с данными, которые хранятся на системах с различной архитектурой и с разной производительностью.
- *Прозрачность относительно сети.* СУБД должны работать одинаково в разнородных сетях: от высокоскоростных ЛВС до телефонных линий.
- *Распределённые запросы.* У пользователя должна быть возможность строить запросы из любых таблиц, вне зависимости от их реального физического расположения.
- *Распределённые изменения*. У пользователя должна быть возможность изменять данные в любой таблице, независимо от реального физического расположения таблиц и самого пользователя.
- *Распределённые транзакции.* СУБД должна выполнять транзакции, выходящие за границы одной вычислительной системы, и при этом поддерживать целостность данных при появлении отказов как в отдельных системах, так и в сети в целом.
- *Безопасность.* СУБД должна обеспечить защиту всей распределённой базы от несанкционированного доступа.
- *Универсальный доступ*. СУБД должна обеспечивать единую методику доступа ко всей корпоративной базе данных.

Следует отметить, что в силу ряда причин ни одно из существующих распределённых СУБД не соответствует «идеалу» в полной мере. Распределенные базы данных являются одним из стратегических направлений развития вычислительной техники, поэтому решение вышеперечисленных задач следует искать на пути разработки новых подходов к обработке распределённых данных.

Доступ к данным в условиях распределённости представляет сложную задачу, требует выделения значительных вычислительных ресурсов и затрат на передачу информации. В рамках решения данной комплексной задачи специалистами IBM была предложена четырёх-

уровневая схема доступа к распределённым данным, такое расчленение проблемы обеспечивает хорошую основу для понимания принципов управления распределёнными ресурсами.

Уровень 1. У*даленный запрос*.

На данном уровне пользователь может выполнить инструкцию SQL, которая производит доступ или модификацию информации в одной удалённой базе данных (рис. 5.4).

Пользователь может таким образом обращаться к разным базам данных, но при этом СУБД не поддерживает транзакции, состоящие из нескольких инструкций (согласно терминологии IBM, транзакция – «удалённая единица работы»).

Очевидно, что возможности удалённых запросов ограничены, такие запросы полезны в условиях однократного или кратковременного доступа к данным, хранящимся на удалённой станции. При этом обработка данных производится в основном средствами станции, инициирующей запрос.

Уровень 2. *Удалённая транзакция*. На данном уровне обеспечиваются более сложные транзакции, состоящие из нескольких инструкций SQL (рис. 5.5).

У пользователя появляется возможность задать СУБД последовательность инструкций SQL, осуществляющих доступ и модификацию данных в удалённой базе, а затем выполнить или отменить всю последовательность целиком, как одну транзакцию, которая может быть успешно завершена или целиком отвергнута. При этом все инструкции, составляющие транзакцию, должны быть адресованы к одной базе данных.

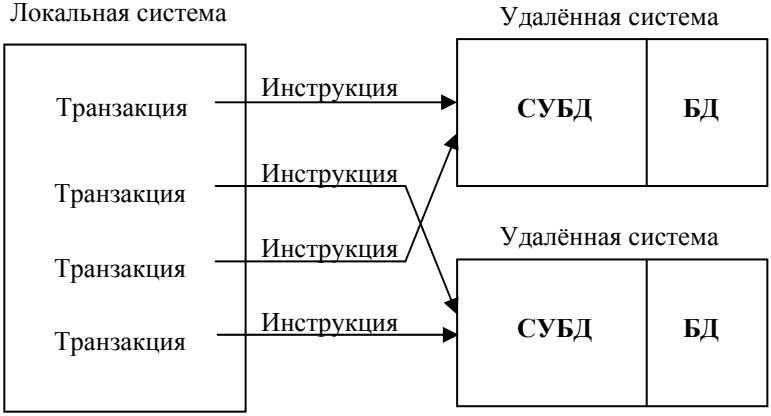


Рис. 5.4. Доступ к распределённым данным: удалённые запросы

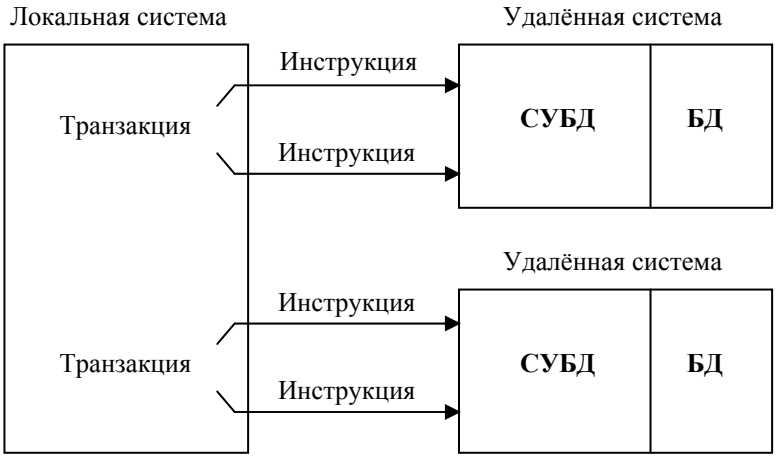


Рис. 5.5. Доступ к распределённым данным: удалённые транзакции

Для обеспечения связи СУБД различных типов при удалённых транзакциях часто используются так называемые шлюзовые программы. Некоторые шлюзовые программы не только обеспечивают выполнение удалённых транзакций, но и дают пользователю возможность объединять в одном запросе таблицы из локальной базы данных с таблицами из удалённой базы данных, управляемой СУБД другого типа. Однако эти программы не обеспечивают (и не могут обеспечить) выполнение транзакций, необходимых для реализации более высоких уровней доступа к распределённым данным. Например, шлюзовая программа может обеспечить по отдельности целостность локальной базы данных и целостность удаленной базы данных, но не гарантирует полного выполнения транзакций в разных узлах сети.

Уровень 3. *Распределённая транзакция* (рис. 5.6). На данном уровне каждый отдельный пользователь посредством SQL может обращаться с запросом на выборку или модификацию данных к одной базе данных в одной удалённой вычислительной системе. Транзакция, представляющая собой последовательность операторов SQL, может обращаться к двум или более базам данных, которые могут быть расположены в различных системах. При этом СУБД гарантирует целостность транзакций, т.е. то, что все части транзакции во всех участвующих системах будут завершены или целиком отменены. Очевидно, что уровень сложности системы, поддерживающей подобные транзакции, должен быть на порядок выше, чем для предыдущих.

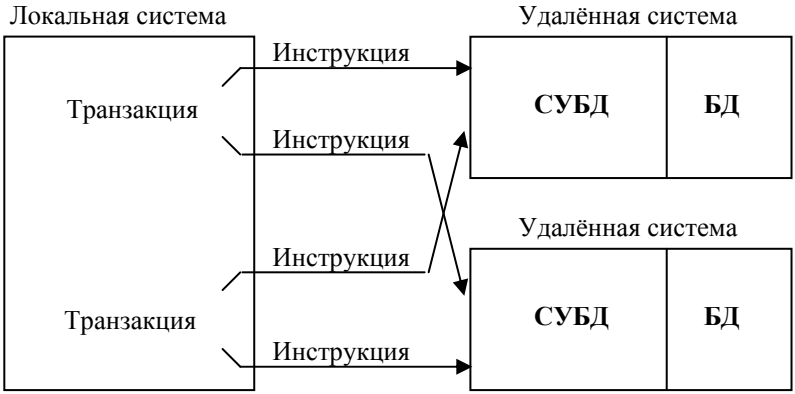


Рис. 5.6. Доступ к распределённым данным: распределённые транзакции

Чтобы обеспечить выполнение распределённой транзакции, необходимо активное взаимодействие отдельных СУБД, участвующих в транзакции. Данное взаимодействие обычно осуществляется при помощи специального протокола, называемого протоколом *двухфазного выполнения*, основное назначение которого – поддерживать целостность распределенных транзакций без участия пользователя.

Уровень 4. *Распределённый запрос* (рис. 5.7). На данном уровне один оператор SQL может обращаться к таблицам из двух или более баз данных, которые расположены в различных вычислительных системах. При этом СУБД отвечает за автоматическое выполнение оператора во всех системах, участвующих в запросе. Транзакция представляет собой последовательность инструкций. Как и на предыдущем уровне СУБД должна обеспечить целостность распределённой транзакции во всех участвующих системах.

На уровне распределённых запросов предъявляются повышенные требования к программам оптимизации инструкций. На данном уровне оптимизирующая программа при оценке альтернативных способов обслуживания оператора SQL должна учитывать скорость передачи данных в сети. Например, если локальной СУБД приходится многократно обращаться к некоторой удалённой таблице (допустим при создании объединения), то эффективнее будет скопировать по сети некоторую часть таблицы целиком посредством одной операции групповой передачи данных, чем многократно считывать по сети отдельные записи.

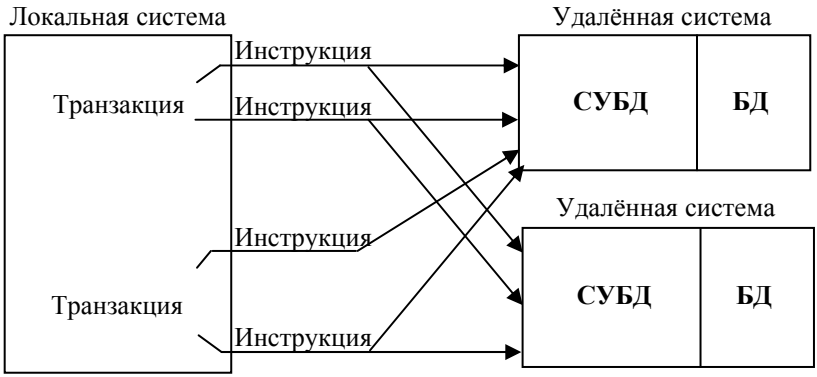


Рис. 5.7. Доступ к распределённым данным: распределённые запросы

Оптимизирующая программа должна также определить, какая СУБД будет эффективнее управлять выполнением оператора. Например, если большая часть таблиц находится в удалённой системе, то, вероятно, было бы эффективнее, чтобы обработку выполняла удалённая СУБД этой системы. Однако при чрезмерной загрузке удалённой системы такое решение может оказаться малоэффективным. В целом задача оптимизирующей программы становится и более сложной, и более важной. В силу своей сложности в настоящее время распределённые запросы не поддерживаются полностью ни в одной коммерческой СУБД, при этом работы в этом направлении ведутся активно и следует ожидать прогресса в данном направлении в самое ближайшее время.

Ряд новых направлений открывается в связи с развитием объектно-ориентированных систем, в частности сред, основанных на объектно-реляционном подходе. Пожалуй, наиболее важная область в управлении распределёнными объектами – это брокеры объектных запросов (Object Request Broker, ORB). Концепция распределения объектов распространилась не только на среды объектно-ориентированных СУБД, но и почти на любые мыслимые среды информационных систем главным образом благодаря работам, проводимым под эгидой Object Management Group (OMG), которая разработала стандарт общей архитектуры брокера объектных запросов – CORBA (Common Object Request Broker Architecture).

Сильным аргументом в пользу подхода к распределению данных, основанного на серверах (централизованный подход), является то преимущество, которое ему обеспечивает более высокая степень контроля поддерживающих ресурсов данных и реализаций. Этот аргумент может быть усилен в объектно-ориентированной области благодаря под-

ходу, предусматривающему использование брокеров объектных запросов. Иначе говоря, базовая распределённая серверная модель открывает дорогу для семантически более богатой модели с инкапсуляцией методов средствами объектной модели.

Благодаря технологиям CORBA среды управления данными могут быть интегрированы одна с другой в значительной степени таким же образом, как и при серверном подходе. Отличие состоит, однако, в том, что введение с помощью ORB объектно-ориентированного уровня обеспечивает псевдообъектно-ориентированную обстановку не только для погружённых в неё ООСУБД, но и для РСУБД, иерархических систем, сред плоских файлов и других архитектур управления ресурсами данных. В действительности синтезированная над ORB среда и серверы методов поддерживают в системе парадигму управления распределёнными объектами независимо от внутренних реализаций.

5.3. БАЗЫ ДАННЫХ В ИНТЕРНЕТ

С появлением Интернет получила дальнейшее развитие архитектура «клиент-сервер». С подключением локальных сетей к Интернет, созданием внутрикорпоративных сетей появляется возможность с любого рабочего места в организации получить доступ к информационным ресурсам сети.

Архитектура «клиент-сервер» на базе Интернет имеет трёхуровневую организацию. Пользовательским интерфейсом является Web-браузер, расположенный на персональном компьютере – «тонком» клиенте. Web-браузер взаимодействует с Web-сервером, последний в свою очередь является клиентом для сервера баз данных (рис. 5.8).

Наиболее часто в качестве сервера БД используется SQL-сервер.

Различают следующие механизмы доступа к БД: на стороне Web-сервера и на стороне Web-клиента.

В первом случае обращение к серверу БД производится следующим путём. Web-клиент заполняет специальную форму запроса к БД и пересылает её Web-серверу. Программами Web-сервера вызываются внешние по отношению к ним программы в соответствии с соглашениями одного из интерфейсов: CGI или API. Внешние программы пишутся на языках программирования типа C++, Perl, PHP. Программы, написанные в соответствии с интерфейсом CGI, называются CGI-сценариями. Внешние программы взаимодействуют с сервером БД на языке SQL, преобразуя текст запроса в HTML-форме в SQL-запрос. После получения результатов запроса внешняя программа формирует требуемую HTML-страницу, передаёт её Web-серверу, который пересылает её Web-клиенту.

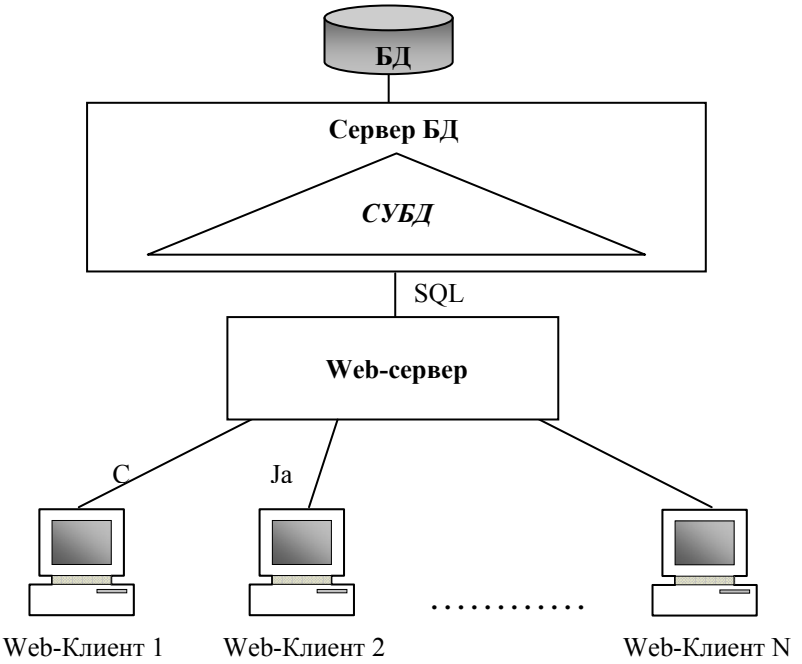


Рис. 5.8. Архитектура «клиент-сервер» на базе Web

При доступе к БД на стороне клиента основным средством реализации механизма взаимодействия Web-клиента и сервера БД является язык JAVA, на котором пишутся программы (апплеты). JAVA-программы хранятся на Web-сервере. В тексте HTML-документа ставятся в нужных местах ссылки на соответствующие апплеты, при обнаружении которых в процессе работы с гипертекстом происходит автоматическая пересылка JAVA-программы с сервера в среду выполнения браузера и загрузка на выполнение. В диалоговом режиме с пользователем уточняются характеристики запроса. Получив управление, JAVA-апплет взаимодействует с сервером БД, в результате чего полученная из БД информация предоставляется пользователю. Для обращения к серверам баз данных разработан стандарт JDBC.

Из двух рассмотренных механизмов явного предпочтения отдать тому или иному варианту нельзя. Всё зависит от целей и условий разработки клиент-серверных программ: зависимость от операционной системы, вида Web-сервера и т.п.

Развитие интернет-технологий стимулирует появление новых направлений в сфере сервиса. Уже никого не удивляют интернет-

библиотеки, интернет-магазины, интернет-справочники. Интересным перспективным направлением в сфере сервиса является проект «интеллектуальная квартира». Например, одной из задач программы менеджера такой квартиры является следить за наличием в холодильнике заданного набора продуктов; при необходимости менеджер посылает заказ в интернет-магазин, который впоследствии выполняется.

Ввиду постоянного увеличения числа пользователей Интернет большие перспективы развития имеют интернет-СУБД бронирования авиабилетов, билетов на концерты, столиков в ресторанах. В настоящее время уже не выглядит полной фантастикой интерактивная справочно-информационная система Голливуда, содержащая фильмы, информацию об актёрах и режиссёрах, поддерживающая интерактивные форумы поклонников, позволяющая участвовать во всевозможных виртуальных шоу.

Контрольные вопросы и задания

1. Определить основные понятия архитектуры «клиент-сервер».
2. Перечислить функции сервера баз данных.
3. Какой принцип является основой архитектуры «клиент-сервер»?
4. Нарисовать двухзвенную схему модели «клиент-сервер».
5. Охарактеризовать функции стандартного интерактивного приложения.
6. Назвать варианты разделения функций между компьютером-сервером и компьютером-клиентом для двухзвенной модели.
7. Охарактеризовать модель удалённого доступа к данным (RAD).
8. Назвать достоинства и недостатки модели сервера баз данных.
9. Какие средства относятся к категории программного обеспечения *промежуточного слоя*?
10. Определить назначение сервера приложений.
11. Что такое транзакция?
12. Указать функции монитора транзакций.
13. Нарисовать схему распределённой базы данных.
14. Охарактеризовать однородные распределённые системы.
15. Каким методом проектируются неоднородные распределённые системы?
16. Указать достоинства и недостатки метода фрагментации данных.
17. Описать метод тиражирования данных.
18. Перечислить характеристики «идеальной» РаСУБД.
19. Охарактеризовать уровни доступа к распределенным данным на примере четырёхуровневой схемы.
20. Нарисовать модель доступа к данным на базе Интернет.
21. Указать основные механизмы доступа к БД в сети Интернет.

6. СТРУКТУРИРОВАННЫЙ ЯЗЫК ЗАПРОСОВ SQL

6.1. ОБЩИЕ СВЕДЕНИЯ ОБ SQL

SQL (Structured Query Language) – структурированный язык запросов – является инструментом, предназначенным для выборки и обработки информации, содержащейся в компьютерной базе данных. SQL является языком программирования, применяемым для организации взаимодействия пользователя с базой данных (рис. 6.1). SQL работает только с реляционными базами данных и предоставляет пользователю следующие функциональные возможности:

- изменение структуры представления данных;
- выборка данных из базы данных;
- обработка базы данных, т.е. добавление новых данных, изменение, удаление имеющихся данных;
- управление доступом к базе данных;
- совместное использование базы данных пользователями, работающими параллельно;
- обеспечение целостности базы данных.

Изначально под «запросом» подразумевалась операция выборки данных или манипулирования данными (вставка, обновление, изменение строк). На самом деле уже при его создании SQL являлся полным языком баз данных, позволявшим выполнять весь спектр операций с базой данных: создание объектов БД (таких как таблицы, представления, последовательности и т.п.), изменение структуры объектов БД, добавление ограничений целостности, удаление объектов БД и т.д.

SQL представляет собой непроцедурный язык, используемый для управления данными реляционных СУБД. Термин «непроцедурный» означает, что на данном языке можно сформулировать, что нужно сделать с данными, но нельзя задать конкретный алгоритм, как именно это следует сделать. Иными словами, в этом языке отсутствуют алгоритмические конструкции, такие как присваивания, операторы цикла, разветвления, переключатели и др.

SQL появился после создания реляционной алгебры в середине 1970-х гг., он был разработан фирмой IBM в рамках проекта экспериментальной реляционной СУБД System R. Но этот язык был настолько прост и удобен, что получил широкое распространение и постепенно стал стандартом «де-факто» (фактическим стандартом) для языков манипулирования данными в реляционных СУБД. Все СУБД, претендующие на название «реляционные», реализовали тот или иной диалект SQL. Изначально эти диалекты могли существенно различаться между собой, что затрудняло переносимость приложе-

ний между различными СУБД. Чтобы решить эту проблему, в дальнейшем были разработаны юридические стандарты SQL. Такого рода стандарты разрабатываются специальными международными организациями по стандартизации. Каждый такой стандарт представляет собой объёмный документ, тщательным образом описывающий все команды и функции языка.

Первый международный стандарт SQL был принят в 1989 г. Американским Национальным Институтом Стандартов (ANSI – American National Standards Institute), как ANSI, X3.135-1989 или ANSI SQL/89. Этот стандарт, помимо ANSI, был также одобрен Международной Организацией Стандартов (ISO – International Standards Organization) в документе ISO 9075-1989. Иногда этот стандарт ещё называют SQL1. Дальнейшее развитие информационных технологий, связанных с базами данных, потребовало расширения и доработки первого стандарта SQL. Так, в конце 1992 г. был принят новый международный стандарт языка SQL – SQL/92 или SQL2, а в 1999 г. – SQL3.

Нужно заметить, что в настоящее время ни одна СУБД не реализует стандарт SQL в полном объёме. Кроме того, во всех диалектах языка имеются возможности, не являющиеся стандартными. Таким образом, можно сказать, что каждый диалект – это надмножество некоторого подмножества стандарта SQL.

Язык SQL оперирует терминами, несколько отличающимися от терминов реляционной теории, например, вместо «отношений» используются «таблицы», вместо «кортежей» – «строки», вместо «атрибутов» – «колонки» или «столбцы». Язык SQL является реляционно полным. Это означает, что любой оператор реляционной алгебры может быть выражен подходящим оператором SQL.

Стандарт языка SQL, хотя и основан на реляционной теории, но во многих местах отходит от неё. Например, отношение в реляционной модели данных не допускает наличия одинаковых кортежей, а таблицы в терминологии SQL могут иметь одинаковые строки. Имеются и другие отличия.

Структурированный язык запросов SQL позволяет выполнять операции над таблицами (создание, удаление, изменение структуры) и над данными таблиц (выборка, изменение, добавление и удаление), а также производить некоторые сопутствующие операции. SQL основан на реляционном языке исчисления кортежей и является непроцедурным языком, т.е. не содержит операторов управления, организации подпрограмм, ввода-вывода и т.п. В связи с этим SQL автономно не используется, обычно он погружён в среду встроенного языка программирования СУБД. В различных СУБД состав языка может несколько меняться.

Состав SQL

Выделяют пять составляющих языка:

1) Data Definition Language (DDL – язык описания данных) состоит из команд, создающих объекты БД: таблицы, поля, первичные и вторичные ключи, индексы и т.д.;

2) Data Manipulation Language (DML – язык манипулирования данными) состоит из команд, позволяющих изменять информацию в БД – добавлять, удалять, изменять;

3) Data Control Language (DCL – язык управления данными) – команды, определяющие права доступа на получение и модификацию информации в БД, проведение транзакций;

4) Transaction Control Language (TCL – язык управления транзакциями) – команды предназначены для управления транзакциями;

5) Data Retrieval Language (DRL – язык получения данных) – вывод данных.

Интерактивный (*встроенный*) *и динамический SQL*

Выделяют два подмножества: интерактивный (ISQL) и динамический (DSQL) SQL. Первый используется в диалоговом режиме на уровне: ввод операторов – выполнение. Второй – для использования языка SQL внутри других языков программирования, называют вложенным.

Встроенный SQL

Основная проблема встраивания SQL в язык программирования состояла в том, что SQL – реляционный язык, т.е. его операторы большей частью работают со множествами, в то время как в языках программирования основными являются скалярные операции. Решение SQL состоит в том, что в язык дополнительно включаются операторы, обеспечивающие покортежный доступ к результату запроса к БД.

Для этого в язык вводится понятие курсора, с которым связывается оператор выборки. Над определённым курсором можно выполнять оператор OPEN, означающий материализацию отношения – результата запроса, оператор FETCH, позволяющий выбрать очередной кортеж результирующего отношения в память программы, и оператор CLOSE, означающий конец работы с данным курсором.

Дополнительную гибкость при создании прикладных программ со встроенным SQL обеспечивает возможность параметризации операторов SQL значениями переменных включающей программы.

Динамический SQL

Для упрощения создания интерактивных SQL-ориентированных систем в SQL System R были включены операторы, позволяющие во время выполнения транзакции откомпилировать и выполнить любой оператор SQL.

Оператор PREPARE вызывает динамическую компиляцию оператора SQL, текст которого содержится в указанной переменной сим-

вольной строке включающей программы. Текст может быть помещён в переменную при выполнении программы любым допустимым способом, например, введён с терминала.

Оператор DESCRIBE служит для получения информации об указанном операторе SQL, ранее подготовленном с помощью оператора PREPARE. С помощью этого оператора можно узнать, во-первых, является ли подготовленный оператор оператором выборки, и, во-вторых, если это оператор выборки, получить полную информацию о числе и типах столбцов результирующего отношения.

Для выполнения ранее подготовленного оператора SQL, не являющегося оператором выборки, служит оператор EXECUTE. Для выполнения динамически подготовленного оператора выборки используется аппарат курсоров с некоторыми отличиями по части задания адресов переменных включающей программы, в которые должны быть помещены значения столбцов текущего кортежа результата.

Основные операторы SQL представлены в табл. 6.1

6.1. Операторы SQL

Вид	Название	Назначение
DDL	CREAT TABLE	Создание таблицы
	DROP TABLE	Удаление таблицы
	ALTER TABLE	Изменение структуры таблицы
	CREAT INDEX	Создание индекса
	DROP INDEX	Удаление индекса
	CREAT VIEW	Создание представления
	DROP VIEW	Удаление представления
	CREAT DOMAIN	Создание домена
	ALTER DOMAIN	Изменение домена
	DROP DOMAIN	Удаление домена
	RENAME	Переименование
DRL	SELECT	Выборка записей
	TRANSFORM	Создание перекрёстного запроса
	UNION	Создание запроса на объединение
DML	UPDATE	Изменение записей
	INSERT	Вставка новых записей
	DELET	Удаление записей
TCL	START TRANSACTION	Начало транзакции
	COMMIT	Успешное завершение транзакции
	ROLLBACK	Откат транзакции
DCL	GRANT	Передача привилегий
	REVOKE	Изъятие привилегий

Запросы DDL предназначены для определения структуры данных, тем самым позволяют создавать домены, таблицы, индексы, представления, а также позволяют обеспечить целостность данных посредством задания ограничителей целостности.

Запросы DRL (Data Retrieval Language) – вывод данных Основа – SELECT.

Запросы DML различаются на запросы действий и на специальные запросы. Выходной набор может быть редактируемым, т.е. изменения, сделанные в выходном наборе, будут произведены и в таблицах, на которых основан запрос, если соблюдены определённые (различные для каждой СУБД) требования. Специальные запросы, к которым относятся запрос на объединение, запрос на создание перекрёстной таблицы и некоторые другие, также возвращают таблицу как результат своего действия. Запросы действий, напротив, оказывают лишь некоторое действие над данными. Результат этого действия можно просмотреть, открыв соответствующие таблицы.

Запросы TCL предназначены для управления транзакциями.

Запросы DCL предназначены для обеспечения управления доступа к данным.

Запросы SQL представляют собой инструкции, состоящие из фраз (предложений). Инструкции называются по имени оператора, определяющего суть инструкции. Такой оператор практически всегда следует первым. Фразы, в свою очередь, также называются по имени ключевого слова, с которого они начинаются.

Язык SQL имеет определённые специальные термины, которые используются для описания самого языка. Среди них такие слова, как запрос, предложение и предикат, которые являются важнейшими в описании и понимании языка, но не означают чего-либо самостоятельного для SQL.

Команды являются инструкциями, с которыми Пользователь обращается к SQL-базе данных. Команды состоят из одной или более логических частей, называемых предложениями. Предложение начинается ключевым словом, которое даёт имя предложению, и состоит из ключевых слов и аргументов. Например, предложения “FROM Salespeople“ и “WHERE city = ‘London’”. Аргументы завершают или изменяют значение предложения. В примерах выше Salespeople – аргумент, а FROM – ключевое слово предложения FROM. Аналогично, “city = ‘London’“ – аргумент предложения WHERE.

Объекты – структуры в базе данных, которым даны имена и сохраняются в памяти. Они включают в себя базовые таблицы, представления (второй тип таблиц) и индексы.

Условные обозначения синтаксиса:

Обозначение	Объект или термин	Пример
Прописные буквы	Команды, функции, ключевые слова	SELECT * FROM таблица
Строчные буквы	Синтаксические переменные, т.е. обозначения, вместо которых надо подставить конкретные значения	SELECT * FROM таблица
Вертикальная черта	Разделяет альтернативные элементы синтаксиса	OFF|ON
Квадратные скобки	Список необязательных элементов синтаксиса	[OFF|ON]
Фигурные скобки	Список обязательных элементов синтаксиса	{OFF|ON}
Подчеркивание	Значение по умолчанию	{OFF|ON}
Многоточие	Означает, что выражение перед многоточием может быть повторено несколько раз	SELECT столбец, … FROM таблица

6.2. ИСПОЛЬЗОВАНИЕ SQL ДЛЯ ИЗВЛЕЧЕНИЯ ИНФОРМАЦИИ ИЗ ТАБЛИЦ (КОМАНДА SELECT)

Запрос – команда, которая передаётся программе управления базой данных и сообщает ей, чтобы она вывела определённую информацию из таблиц в памяти. Эта информация обычно посылается непосредственно на экран компьютера или терминала, хотя в большинстве случаев её можно также послать принтеру, сохранить в файле (как объект в памяти компьютера) или представить как вводную информацию для другой команды или процесса.

Запросы обычно рассматриваются как часть языка DML (Языка Манипулирования Данными). Так как запрос не меняет информацию в таблицах, а просто показывает её пользователю, лучше рассматривать запросы как самостоятельную категорию среди команд DML, которые часто производят действие, а не просто показывают содержание базы данных. Все запросы в SQL состоят из одиночной команды. Структура этой команды обманчиво проста, потому что обычно её приходится расширять так, чтобы выполнить высоко сложные оценки и обработки данных. Эта команда называется SELECT (ВЫБОР).

В самой простой форме команда SELECT просто инструктирует СУБД, чтобы извлечь информацию из таблицы. Например, можно вывести таблицу Salespeople, напечатав следующее:

```
SELECT snum, sname, sity, comm
FROM Salespeople;
```

Вывод для этого запроса:

snum	sname	city	comm
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Здесь

snum – уникальный номер, назначенный каждому продавцу;

sname – имя продавца;

city – расположение продавца (город);

comm – комиссионные продавцов в десятичной форме.

Другими словами, эта команда просто выводит данные из таблицы. Команда состоит из следующих частей:

SELECT – ключевое слово, которое сообщает базе данных, что эта команда запрос. Все запросы начинаются этим словом, сопровождаемым пробелом.

snum, sname, city, comm – это список столбцов из таблицы, которые выбираются запросом. Любые столбцы, не перечисленные здесь, не будут включены в вывод команды. Это, конечно, не значит, что они будут удалены или их информация будет стёрта из таблиц, потому что запрос не воздействует на информацию в таблицах.

FROM – ключевое слово, подобное SELECT, которое должно быть представлено в запросе данного вида. Оно сопровождается пробелом и затем именем таблицы, используемой в качестве источника информации.

Salespeople – в данном случае это имя таблицы Salespeople.

Точка с запятой используется во всех интерактивных командах SQL как знак окончания команды.

Строки выводятся в том порядке, в котором они найдены в таблице.

Для вывода всех столбцов таблицы применяется знак звездочка (*), который может применяться для вывода полного списка столбцов следующим образом:

```
SELECT *
FROM Salespeople;
```

Столбцы выводятся в том порядке, в котором они указаны.

Для удаления избыточных данных из-за двойных значений применяется аргумент DISTINCT (ОТЛИЧИЕ). Двойные значения могут появиться даже в хорошо спроектированной базе просто вследствие запроса на вывод столбцов, не включающих ключа отношения.

Для получения списка без дубликатов следует ввести следующее:

```
SELECT DISTINCT city
FROM Salespeople;
```

Аргумент DISTINCT – полезный способ избежать избыточности данных, но не рекомендуется безоглядно использовать DISTINCT, потому что это может скрыть неточность в интерпретации данных. Например, можно предположить, что имена всех Продавцов различны. Если другой пользователь помещает второго Axelrod'а в таблицу Salespeople, а используются SELECT DISTINCT sname, вы не будете даже знать о существовании двойника.

Аргумент DISTINCT может указываться только один раз в данном предложении, даже если предложение SELECT выбирает несколько полей. Аргумент DISTINCT опускает строки в тех случаях, когда все выбранные поля идентичны. Строки, в которых некоторые значения одинаковы, а некоторые различны, будут сохранены. Исключение возникает только тогда, когда DISTINCT используется внутри агрегатных функций, как описано далее.

Вместо аргумента DISTINCT можно указать аргумент ALL, который имеет противоположный эффект, а именно дублирование строк вывода сохранится. Так как это тот же самый случай, когда не указывается ни DISTINCT, ни ALL, то ALL по существу скорее пояснительный, а не действующий аргумент.

Таблицы имеют тенденцию становиться очень большими, поскольку с течением времени в таблицу могут добавляться строки. Язык SQL даёт возможность устанавливать критерии, чтобы определить, какие строки будут выбраны для вывода.

Предложение WHERE – предложение команды SELECT, которое позволяет устанавливать предикаты, которые могут быть или истинными или неистинными для некоторой строки таблицы. Команда извлекает только те строки из таблицы, для которой такое утверждение верно. Например, необходимо увидеть имена и комиссионные всех продавцов в Лондоне. Можно ввести такую команду:

```
SELECT sname, city
FROM Salespeople
WHERE city = 'LONDON';
```

Когда предложение WHERE представлено, СУБД просматривает всю таблицу по одной строке и исследует каждую строку, чтобы определить, верно ли утверждение.

Оператор отношения – математический символ, который указывает на определённый тип сравнения между двумя значениями.

Операторы отношения, которыми располагает язык SQL:

= Равный
> Больше чем
< Меньше чем
>= Больше чем или равно
<= Меньше чем или равно
<> Не равно

Основные Булевы операторы также включены в SQL. Булевы операторы связывают одно или более истинных/неистинных значений и производят единственное истинное/неистинное значение. Стандартными операторами Буля, распознаваемыми в SQL, являются AND, OR, и NOT.

Пример.

```
SELECT *
FROM Salespeople
WHERE comm<=0.11;
```

Включая в предикаты операторы Буля, можно значительно увеличить их возможности. Например:

```
SELECT *
FROM Salespeople
WHERE city='London'
AND comm<=0.11;
```

Оператор NOT должен предшествовать предикату и не должен помещаться перед оператором отношения. Например, неправильно

```
AND comm NOT <=0.11;
```

Круглые скобки в языке SQL имеют обычную интерпретацию, т.е. то, что внутри них оценивается первым и обрабатывается как единое выражение.

В дополнение к операторам отношения и булевским операторам язык SQL использует специальные операторы IN, BETWEEN, LIKE и IS NULL .

Оператор IN определяет набор значений, в которое данное значение может или не может быть включено. В соответствии с приведёнными ранее примерами, если требуется найти всех продавцов, которые размещены в Barcelona или в London, необходимо использовать следующий запрос:

```
SELECT *
FROM Salespeople
WHERE city = 'Barcelona'
OR city = 'London';
```

Однако имеется и более простой способ получить ту же информацию:

```
SELECT *
FROM Salespeople
WHERE city IN('Barcelona', 'London');
```

Как можно видеть, оператор IN определяет набор значений в виде членов набора, заключённых в круглые скобки и отделённых запятыми. Программа СУБД проверяет различные значения указанного поля, пытаясь найти совпадение со значениями из набора. Если это случается, то предикат верен.

Оператор BETWEEN похож на оператор IN. В отличие от определения по значениям из набора, как это делает оператор IN, оператор BETWEEN определяет диапазон значений, в котором истинен предикат. Необходимо ввести ключевое слово BETWEEN с начальным значением, ключевое AND и конечное значение. В отличие от оператора IN, оператор BETWEEN чувствителен к порядку, и первое значение в предложении должно быть первым по алфавитному или числовому порядку.

Следующий запрос будет извлекать из таблицы Salespeople всех продавцов с комиссионными между 0.10 и 0.12:

```
SELECT *
FROM Salespeople
WHERE comm BETWEEN .10 AND .12;
```

Язык SQL не делает непосредственной поддержки невключения границ. Необходимо или определить ваши граничные значения так, чтобы включающая интерпретация была приемлема, или сделать что-нибудь типа следующего:

```
SELECT *
FROM Salespeople
WHERE(comm BETWEEN .10, AND .12)
AND NOT comm IN(.10, .12);
```

Так же, как и операторы отношения, BETWEEN может работать с символьными полями. Это означает, что можно использовать BETWEEN, чтобы выбирать ряд значений из упорядоченных по алфавиту значений.

Следующий запрос выбирает всех продавцов, чьи имена попали в определённый алфавитный диапазон:

```
SELECT *
FROM Salespeople
WHERE sname BETWEEN 'A' AND 'G';
```

Оператор LIKE применим только к полям типа CHAR или VARCHAR, с которыми он используется, чтобы находить подстроки. В качестве условия оператор LIKE использует групповые символы (wildcards).

Имеются два типа групповых символов, используемых с оператором LIKE:

1) символ подчёркивания (_) замещает любой одиночный символ.

Например, 'b_t' будет соответствовать словам 'bat' или 'bit', но не будет соответствовать 'brat';

2) знак процента (%) замещает последовательность любого числа символов. Например '%p%t' будет соответствовать словам 'put', 'posit' или 'opt', но не 'spite'.

Оператор LIKE может быть удобен, если вы ищете имя или другое значение, и если вы не помните, как они точно пишутся. Предположим, что вы не уверены, как записано по буквам имя одного из ваших продавцов Peal или Peel. Можно просто использовать ту часть, которую вы знаете, и групповые символы, чтобы находить все возможные пары (вывод этого запроса показывается также):

```
SELECT *
FROM Salespeople
WHERE sname LIKE 'P _ _ l%';
```

Групповые символы подчёркивания, каждый из которых представляет один символ, добавят только два символа к уже существующим 'P' и 'l', поэтому имя наподобие Prettel не может быть показано. Групповой символ '%' в конце строки необходим в большинстве реализаций, если длина поля sname больше чем число символов в имени Peel (потому что некоторые другие значения sname длиннее, чем 4 символа). В таком случае значение поля sname, фактически сохраняемое как имя Peel, сопровождается рядом пробелов. Следовательно, символ 'l' не будет рассматриваться концом строки. Групповой символ '%' просто соответствует этим пробелам. Это необязательно, если поле sname имеет тип VARCHAR.

А что же делать, если нужно искать знак процента или знак подчёркивания в строке? В LIKE-предикате можно добавить предложение ESCAPE и определить любой одиночный символ как символ ESC, т.е. как символ «исключения». Символ ESC используется сразу перед процентом или подчёркиванием в предикате и означает, что процент или подчёркивание будет интерпретироваться как символ, а не как групповой символ.

Например, для поиска имени с символом ‘/’ следует ввести:

```
SELECT *
FROM Salespeople
WHERE sname LIKE ‘%/_%’ESCAPE’/’;
```

Символ ESC применяется только к одиночному символу сразу после него. Сам символ ESC, который может появиться в LIKE-строке, сопровождается вторым знаком ESCAPE. Символ ESC лучше всего понимать как «взять следующий символ буквально как символ».

Например, чтобы искать местонахождение строки ‘_/’ в sname столбце, следует ввести:

```
SELECT *
FROM Salespeople
WHERE sname LIKE ‘% /_ / / %’ESCAPE’/’;
```

Часто в таблице могут появляться записи, в которых нет никаких значений в одном, нескольких или во всех полях, например, потому что информации в этом поле и не должно быть, или потому что это поле просто не заполнялось. Язык SQL учитывает такой вариант, позволяя вводить значение NULL (ПУСТОЙ) в поле. Когда значение поля равно NULL, это означает, что программа базы данных специально промаркировала это поле, как не имеющее смысла для этой строки (или записи).

Маркировка NULL отличается от просто назначения полю значения нуля или пробела, которые база данных будет обрабатывать так же, как и любое другое значение. Точно так же, как NULL не является техническим значением, оно не имеет и типа данных. Оно может помещаться в любой тип поля. Тем не менее NULL в SQL часто упоминается как нуль.

Так как NULL указывает на отсутствие значения, нельзя знать, каков будет результат любого сравнения с использованием NULL. Когда NULL сравнивается с любым значением, даже с другим таким же NULL, результат будет ни истинным, ни неистинным, он неизвестен.

Неизвестное значение предиката ведёт себя так же, как неистинная строка, которая, произведя неистинное значение в предикате, не

будет выбрана запросом. Следует иметь в виду, что в то время как NOT от неистинного равняется истинно, NOT от неизвестного равняется неизвестно. Следовательно, выражение типа ‘city = NULL’ или ‘city IN (NULL)’ будет неизвестно независимо от значения city.

Чтобы различать неистинное и неизвестное, язык SQL предоставляет специальный оператор IS, который используется с ключевым словом NULL для указания на значение NULL.

Специальным операторам, о которых ранее шла речь, может предшествовать NOT. В этом случае NOT изменяет значение предиката на противоположное. Например,

WHERE city NOT NULL эквивалентно WHERE NOT city IS NULL;

WHERE city NOT IN(‘London’, ‘San Jose’) эквивалентно

WHERE NOT city IN(‘London’, ‘ San Jose’).

Аналогично можно использовать NOT BETWEEN и NOT LIKE.

6.3. ОБОБЩЕНИЕ ДАННЫХ С ПОМОЩЬЮ АГРЕГАТНЫХ ФУНКЦИЙ

Агрегатные, или общие, функции в языке SQL берут группы значений из полей и сводят их к одиночному значению.

Список агрегатных функций:

COUNT	Определяет число строк
SUM	Вычисляет арифметическую сумму не NULL-значений заданного в запросе поля
AVG	Вычисляет среднее среди не NULL-значений заданного поля
MAX	Находит наибольшее из не NULL-значений заданного поля
MIN	Находит наименьшее из не NULL-значений заданного поля

Агрегатные функции используют имена полей как аргументы в предложении SELECT-запроса.

Фунции SUM и AVG могут использовать только числовые поля, а функции COUNT, MAX и MIN могут использовать и числовые, и символьные поля. Когда функции MAX и MIN используются с символьными полями, они транслируют значение в код ASCII или в расширенный ASCII и сравнивают значения.

Пример записи запроса:

SELECT SUM (amt) FROM Orders;

Агрегатная функция возвращает одиночное значение, независимо от того, сколько строк находится в таблице. Поэтому агрегатные функции и поля не могут выбираться одновременно, пока не будет использовано предложение GROUP BY, описанное далее.

Функция COUNT несколько отличается от других. Она считает число не NULL-значений в данном столбце или общее число строк в таблице. Часто она используется с DISTINCT для того, чтобы подсчитывать отличающиеся значения в данном поле. Пример использования функции COUNT с аргументом DISTINCT:

```
SELECT COUNT(DISTINCT snum) FROM Orders;
```

Формально аргумент DISTINCT может использоваться с любой агрегатной функцией, но применять его с другими функциями, кроме COUNT, просто неэффективно.

Для подсчёта общего числа строк в таблице используется функция COUNT со звёздочкой вместо имени поля, как например:

```
SELECT COUNT (*) FROM Customers;
```

Функция COUNT со звездочкой обрабатывает и NULL, и дубликаты. По этой причине аргумент DISTINCT в этом случае не используется.

DISTINCT не применим с COUNT (*), потому что он не имеет никакого действия в хорошо разработанной и поддерживаемой базе данных. Если же всё-таки имеются полностью пустые или дублированные строки, то COUNT покажет эту информацию.

В большинстве реализаций агрегатные функции могут также использовать аргумент ALL, который помещается перед именем поля, подобно DISTINCT, но означает противоположное, т.е. включать дубликаты. Стандарт ANSI технически не позволяет этого для COUNT, но многие реализации ослабляют это ограничение.

Основное различие между ALL и *, когда они используются с COUNT, состоит в том, что ALL не может подсчитать значения NULL. Аргумент * является единственным, который включает NULL-значения, и он используется только с COUNT; функции, отличные от COUNT игнорируют значения NULL в любом случае.

Следующая команда SELECT подсчитает число не-NULL-значений в поле rating в таблице Customers (Заказчиков), включая повторения:

```
SELECT COUNT(ALL rating) FROM Customers;
```

Агрегатные функции можно также использовать с аргументами, которые состоят из скалярных выражений, включающих одно или бо-

лее полей. В этом случае не разрешается DISTINCT. Например, следующий запрос находит максимальную сумму значений в двух полях:

```
SELECT MAX(blnc + amt) FROM Orders;
```

Предложение GROUP BY позволяет определять подмножество значений в одном поле в терминах другого поля, и применять функцию агрегата к подмножеству. Это даёт возможность объединять поля и агрегатные функции в едином предложении SELECT.

Например, в следующем запросе для каждого значения поля snum находится максимальное значение в поле amt в таблице Orders.

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum;
```

Предложение GROUP BY можно использовать с несколькими полями. Например,

```
SELECT snum, odate, MAX (amt)
FROM Orders
GROUP BY snum, odate;
```

Предложение HAVING определяет критерии, используемые для того, чтобы удалять определённые группы из вывода, точно так же как предложение WHERE делает это для индивидуальных строк.

Например,

```
SELECT snum, odate, MAX (amt)
FROM Orders
GROUP BY snum, odate
HAVING MAX (amt) > 3000.00;
```

выведет данные только в тех случаях, когда максимальные значения в группах по значениям полей snum, odate превышают 3000.00.

Аргументы в предложении HAVING следуют тем же самым правилам, что и в предложении SELECT, состоящей из команд, использующих GROUP BY. Они должны иметь одно значение на группу вывода. Следующая команда будет запрещена:

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING odate = 10/03/1988;
```

Поле odate не может быть вызвано предложением HAVING, потому что оно может иметь больше чем одно значение на группу вывода. Чтобы избегать такой ситуации, предложение HAVING должно ссылаться только на агрегаты и поля выбранные GROUP BY.

Имеется правильный способ сделать вышеупомянутый запрос:

```
SELECT snum, MAX (amt)
FROM Orders
WHERE odate = 10/03/2014
GROUP BY snum;
```

В предложении HAVING можно использовать поля, выбранные с помощью GROUP BY. Например,

```
SELECT snum, MAX (amt)
FROM Orders
GROUP BY snum
HAVING snum IN (1002,1007);
```

В строгой интерпретации ANSI SQL нельзя использовать агрегат агрегата. Например, может быть отвергнут запрос

```
SELECT odate, MAX(SUM (amt))
FROM Orders
GROUP BY odate;
```

Некоторые реализации не предписывают этого ограничения, которое является выгодным, потому что вложенные агрегаты могут быть очень полезны, даже если они и несколько проблематичны.

Язык SQL позволяет помещать скалярные выражения, константы, символы, текст и комментарии среди выбранных полей. Эти выражения могут дополнять или замещать поля в предложениях SELECT, и могут включать в себя одно или более выбранных полей. Например, можно послать запрос

```
SELECT snum, sname, city, 'have', comm * 100, '%'
FROM Salespeople;
```

и получить в дополнительном столбце на основе столбца comm значения в процентах.

6.4. УПОРЯДОЧЕНИЕ ВЫВОДА ПОЛЕЙ

Язык SQL использует команду ORDER BY для того, чтобы упорядочивать вывод. Многочисленные столбцы упорядочиваются один внутри другого и можно определять возрастание (ASC) или убывание (DESC) для каждого столбца. По умолчанию установлено возрастание. Например,

```
SELECT *
FROM Orders
ORDER BY cnum DESC;
```

Можно также упорядочивать таблицу с помощью другого столбца, например с помощью поля amt, внутри упорядочения поля cnum:

```
SELECT *
FROM Orders
ORDER BY cnum DESC, amt DESC;
```

Согласно требованиям ANSI столбцы, которые упорядочиваются, должны быть указаны в выборе SELECT.

Предложение ORDER BY может использоваться с GROUP BY для упорядочения агрегатных групп. В этом случае предложение ORDER BY должно быть последним.

Вместо имени столбца можно использовать его порядковый номер. Эти номера могут ссылаться не на порядок столбцов в таблице, а на их порядок в выводе. Другими словами, поле, упомянутое в предложении SELECT первым, для предложения ORDER BY – это поле 1, независимо от того, каким по порядку оно стоит в таблице. Например, можно использовать следующую команду, чтобы увидеть определённые поля таблицы Salespeople, упорядоченными в порядке убывания к наименьшему значению комиссионных:

```
SELECT sname, comm
FROM Salespeople
ORDER BY 2 DESC;
```

Основная цель этой возможности ORDER BY – дать возможность использовать GROUP BY со столбцами вывода так же, как и со столбцами таблицы. Столбцы, созданные агрегатной функцией, а также константы или выражения в предложении SELECT-запроса пригодны для использования с предложением ORDER BY, если оно ссылается к ним с помощью номера.

Например,

```
SELECT snum, COUNT(DISTINCT onum)
FROM Orders
GROUP BY snum
ORDER BY 2 DESC;
```

Согласно стандарту ANSI NULL-значения в том поле, которое используется для упорядочивания вывода, могут произвольно располагаться относительно других значений.

6.5. ОБЪЕДИНЕНИЯ НЕСКОЛЬКИХ ТАБЛИЦ В ЗАПРОСЕ

Одна из наиболее важных особенностей языка SQL – это способность задавать связи между многочисленными таблицами и выводить информацию из них в терминах этих связей, используя одну команду. Этот вид операции называется объединением.

В целях объединения таблицы перечисляются списком в предложении FROM-запроса и отделяются друг от друга запятыми. Предикат запроса может ссылаться к любому столбцу любой связанной таблицы и, следовательно, может использоваться для связи между ними. Обычно в предикате сравниваются значения в столбцах различных таблиц, чтобы определить, удовлетворяет ли WHERE установленному условию.

Полное имя столбца таблицы фактически состоит из имени таблицы, сопровождаемого точкой, и затем имени столбца.

Например,

```
Salespeople.snum
Salespeople.city
Orders.odate
```

До этого вы могли опускать имена таблиц, потому что вы запрашивали только одну таблицу одновременно, а SQL достаточно интелектуален, чтобы присвоить соответствующий префикс имени таблицы. Даже когда вы делаете запрос многочисленных таблиц, вы ещё можете опускать имена таблиц, если все её столбцы имеют различные имена. Но это не всегда так бывает. Например, мы можем иметь две типовые таблицы со столбцами, называемыми city. Если мы должны связать эти столбцы (кратковременно), мы будем должны указать их с именами Salespeople.city или Customers.city, чтобы SQL мог их различать.

Предположим, что требуется поставить в соответствие продавцам заказчиков в том городе, в котором они живут, и увидеть все комбинации продавцов и заказчиков для этого города. Вы будете брать каждого продавца и искать в таблице Заказчиков всех заказчиков того же самого города. Вы могли бы сделать это, введя следующую команду:

```
SELECT Customers.cname, Salespeople.sname,
Salespeople.city
FROM Salespeople, Customers
WHERE Salespeople.city = Customers.city;
```

Так как поле city имеется и в таблице Salespeople, и в таблице Customers, имена таблиц должны использоваться как префиксы. Хотя это необходимо только тогда, когда два или более полей имеют одно и то же имя, в любом случае это хорошая идея включать имя таблицы в обьединение для лучшего понимания и непротиворечивости.

В предыдущем примере мы установили связь между двумя таблицами в объединении. Но эти таблицы уже были соединены через snum поле. Эта связь называется состоянием ссылочной целостности. Используя объединение, можно извлекать данные в терминах этой связи. Например, чтобы показать имена всех заказчиков соответствующих продавцам, которые их обслуживают, мы будем использовать такой запрос:

```
SELECT Customers.cname, Salespeople.sname
FROM Customers, Salespeople
WHERE Salespeople.snum = Customers.snum;
```

Это пример объединения, в котором столбцы используются для определения предиката запроса, и в этом случае snum-столбцы из обеих таблиц удалены из вывода. Вывод показывает, какие заказчики каким продавцом обслуживаются; значения поля snum, которые устанавливают связь, отсутствуют.

Объединения, которые используют предикаты, основанные на равенствах, называются объединениями по равенству. Все предыдущие примеры относились именно к этой категории, потому что все условия в предложениях WHERE базировались на математических выражениях, использующих знак равно (=). Строки ‘city = ‘London’ и ‘Salespeople.snum = Orders.snum‘ – примеры таких типов равенств, найденных в предикатах.

Объединения по равенству – это наиболее общий вид объединения, но имеются и другие. Здесь показан пример другого вида объединения:

```
SELECT sname, cname
FROM Salespeople, Customers
WHERE sname < cname AND rating < 200;
```

Такая команда не часто бывает полезна. Она воспроизводит все комбинации имени продавца и имени заказчика так, что первый предшествует последнему в алфавитном порядке, а последний имеет оценку меньше чем 200.

Возможно также создавать запросы, объединяющие более двух таблиц. Предположим, что мы хотим найти все платежи заказчиков, не находящихся в тех городах, где находятся их продавцы. Для этого необходимо связать все три наши типовые таблицы:

```
SELECT onum, cname, Orders.cnum, Orders.snum
FROM Salespeople, Customers, Orders
WHERE Customers.city < > Salespeople.city
AND Orders.cnum = Customers.cnum
AND Orders.snum = Salespeople.snum;
```

Методика объединения таблиц может использоваться для того, чтобы объединять вместе две копии одиночной таблицы.

При объединении таблицы с собой можно комбинировать каждую строку таблицы и с собой, и с каждой другой строкой таблицы. Затем каждая комбинация оценивается в терминах предиката так же, как при объединении мультитаблиц.

Синтаксис команды для объединения таблицы с собой тот же, что и для объединения нескольких таблиц. Чтобы ссылаться к повторяемым именам столбцов, необходимо иметь два различных имени для этой таблицы. Это достигается с помощью определения временных имён, называемых переменными диапазона, переменными корреляции или просто псевдонимами; они определяются в предложении FROM запроса.

Например, найдём все пары заказчиков, имеющих один и тот же рейтинг и живущих в одном городе:

```
SELECT first.cname, second.cname, first.rating, first.city
FROM Customers first, Customers second
WHERE first.rating = second.rating AND first.city=second.city;
```

Обратите внимание, что псевдонимы использованы уже в предложении SELECT до определения их в предложении FROM.

Псевдоним существует только на время выполнения команды!

Для устранения избыточности в выводе необходимо ввести дополнительный предикат.

Рассматриваемая особенность SQL используется для проверки определённых видов ошибок. При просмотре таблицы Приобретений можно видеть, что поля cnum и snum должны иметь постоянную связь. Так каждый заказчик должен быть сопоставлен одному и тому же продавцу каждый раз, когда определённый номер заказчика появляется в таблице Приобретений. Следующая команда будет определять любые несогласованности в этой области:

```
SELECT first.onum, tirst.cnum, first.snum,
second.onum, second.cnum,second.snum
FROM Orders first, Orders second
WHERE first.cnum = second.cnum AND first.snum < > second.snum;
```

Псевдонимы можно использовать в любое время, при необходимости создать альтернативные имена для таблиц в команде. Например, если таблицы имеют очень длинные и сложные имена, можно определить простые односимвольные псевдонимы типа a и b и использовать их вместо имён таблицы в предложении SELECT и предикате. При этом можно использовать любое число псевдонимов для одной таблицы в запросе.

Также возможно создать объединение, которое включает и различные таблицы, и псевдонимы одиночной таблицы.

Пусть задана следующая реляционная база данных:

Таблица Salespeople (Продавцы)

SNUM	SNAME	CITY	COMM
1001	Peel	London	0.12
1002	Serres	San Jose	0.13
1004	Motika	London	0.11
1007	Rifkin	Barcelona	0.15
1003	Axelrod	New York	0.10

Таблица Customers (Заказчики)

CNUM	CNAME	CITY	RATING	SNUM
2001	Hoffman	London	100	1001
2002	Giovanni	Rome	200	1003
2003	Liu	SanJose	200	1002
2004	Grass	Berlin	300	1002
2006	Clemens	London	100	1001
2008	Cisneros	SanJose	300	1007
2007	Pereira	Rome	100	1004

Таблица Orders (Приобретения)

ONUM	AMT	ODATE	CNUM	SNUM
3001	18.69	10/03/2014	2008	1007
3003	767.19	10/03/2014	2001	1001
3002	1900.10	10/03/2014	2007	1004
3005	5160.45	10/03/2014	2003	1002
3006	1098.16	10/03/2014	2008	1007
3009	1713.23	10/04/2014	2002	1003
3007	75.75	10/04/2014	2004	1002
3008	4723.00	10/05/2014	2006	1001
3010	1309.95	10/06/2014	2004	1002
3011	9891.88	10/06/2014	2006	1001

onum – уникальный номер, присвоенный каждому приобретению.

amt – значение суммы приобретений.

odate – дата приобретения.

cnum – номер заказчика, делающего приобретение (из таблицы Заказчиков).

snum – номер продавца, продающего приобретение (из таблицы Продавцов).

6.6. ПОДЗАПРОСЫ

С помощью SQL можно вкладывать запросы внутрь друга друга. Обычно внутренний запрос генерирует значение, которое проверяется в предикате внешнего запроса, определяющего, истинно оно или нет. Простой подзапрос выполняется только один раз.

Например, предположим, что мы знаем имя продавца: Motika, но не знаем значение его поля snum, и хотим извлечь все приобретения из таблицы Приобретений. Запрос имеет следующую форму:

```
SELECT * FROM Orders
WHERE snum =( SELECT snum FROM Salespeople
WHERE sname = 'Motika');
```

Чтобы оценить внешний (основной) запрос, SQL сначала должен оценить внутренний запрос (или подзапрос) внутри предложения WHERE. Конечно же, подзапрос должен выбирать один и только один столбец, а тип данных этого столбца должен совпадать с тем значением, с которым он будет сравниваться в предикате. Часто, как показано выше, выбранное поле и его значение будут иметь одинаковые имена (в этом случае snum), но это необязательно.

Подзапрос должен возвращать одно и только одно значение. Если в предыдущем подзапросе назначить условие snum "WHERE city = "London" вместо "WHERE sname = 'Motika", то можно получить несколько различных значений. Это может сделать уравнение в предикате основного запроса невозможным для оценки истинности или неистинности, и команда выдаст ошибку.

При использовании подзапросов в предикатах, основанных на реляционных операторах, необходимо убедиться, что использовали подзапрос, который будет выдавать одну и только одну строку вывода. Если используется подзапрос, который не выводит никаких значений вообще, команда не потерпит неудачи; но основной запрос не выведет никаких значений.

Подзапросы, которые не производят никакого вывода (или нулевой вывод), вынуждают рассматривать предикат ни как истинный, ни как неистинный, а как неизвестный. Однако неизвестный предикат имеет тот же самый эффект, что и неистинный: никакие строки не выбираются основным запросом.

Можно использовать DISTINCT, чтобы вынудить подзапрос генерировать одиночное значение. Предположим, что мы хотим найти все приобретения для тех продавцов, которые обслуживают Hoffmana (cnum = 2001).

Имеется один способ, чтобы сделать это:

```
SELECT *
FROM Orders
WHERE snum =
(SELECT DISTINCT snum
FROM Orders
WHERE cnum = 2001);
```

Подзапрос установил, что значение поля snum совпало с Hoffman – 1001, и затем основной запрос выделил все приобретения с этим значением snum из таблицы Приобретений (не разбирая, относятся они к Hoffman или нет). Так как каждый заказчик назначен к одному и только этому продавцу, мы знаем, что каждая строка в таблице Приобретений с данным значением cnum должна иметь такое же значение snum. Однако, так как там может быть любое число таких строк, подзапрос мог бы вывести много (хотя и идентичных) значений snum для данного поля cnum. Аргумент DISTINCT предотвращает это. Если наш подзапрос возвратит более одного значения, это будет указывать на ошибку в наших данных.

Следует обратить внимание, что предикаты, включающие подзапросы, используют выражение типа < скалярная форма > < оператор > < подзапрос >, а не < подзапрос > < оператор > < скалярное выражение > или < подзапрос > < оператор > < подзапрос >.

Другими словами, вы не должны записывать предыдущий пример так:

```
SELECT *
FROM Orders
WHERE (SELECT DISTINCT snum
FROM Orders
WHERE cnum = 2001)
= snum;
```

Один тип функций, который автоматически может производить одиночное значение для любого числа строк, конечно же, – агрегатная функция.

Любой запрос, использующий одиночную функцию агрегата без предложения GROUP BY, будет выбирать одиночное значение для использования в основном предикате. Например, требуется увидеть все приобретения, имеющие сумму выше средней на 4 октября:

```
SELECT *
FROM Orders
WHERE amt > (SELECT AVG (amt) FROM Orders
WHERE odate = 10/04/2014);
```

Имейте в виду, что сгруппированные агрегатные функции, которые являются агрегатными функциями, определёнными в терминах предложения GROUP BY, могут производить многочисленные значения. Они, следовательно, не позволительны в подзапросах такого характера. Даже если GROUP BY и HAVING используются таким способом, что только одна группа выводится с помощью подзапроса, команда будет отклонена в принципе. Необходимо использовать одиночную агрегатную функцию с предложением WHERE, что устранит нежелательные группы.

Например, следующий запрос, который должен найти среднее значение комиссионных продавца в Лондоне:

```
SELECT AVG (comm)
FROM Salespeople
GROUP BY city HAVlNG city = “London”;
```

не может использоваться в подзапросе!

Другим способом может быть:

```
SELECT AVG (comm)
FROM Salespeople
WHERE city = “London”;
```

Можно использовать подзапросы, которые производят любое число строк, если используется специальный оператор IN (операторы BETWEEN, LIKE и IS NULL не могут использоваться с подзапросами). Оператор IN определяет набор значений, одно из которых должно совпадать с другим термом уравнения предиката, чтобы предикат был истинным.

При использовании IN с подзапросом SQL просто формирует этот набор из вывода подзапроса. Следовательно, можно использовать IN, чтобы выполнить такой же подзапрос, который не будет работать с реляционным оператором, и найти все атрибуты таблицы Приобретений для продавца в Лондоне:

```
SELECT *
FROM Orders
WHERE snum IN
(SELECT snum
FROM Salespeople
WHERE city = “LONDON”);
```

В ситуации, подобно этой, подзапрос более прост для пользователя, чтобы понимать его, и более прост для компьютера, чтобы его выполнить, чем если бы было использовано объединение:

```
SELECT onum, amt, odate, cnum, Orders.snum
FROM Orders, Salespeople
WHERE Orders.snum = Salespeople.snum
AND Salespeople.city = “London”;
```

Хотя это и произведёт тот же самый вывод, что и в примере с подзапросом.

Конечно, можно также использовать оператор IN, даже когда вы уверены, что подзапрос произведёт одиночное значение. В любой ситуации, где можно использовать реляционный оператор сравнения (=), можно использовать IN. В отличие от реляционных операторов, IN не может заставить команду потерпеть неудачу, если больше чем одно значение выбрано подзапросом. Это может быть или преимуществом, или недостатком.

Смысл всех ранее обсуждённых подзапросов тот, что все они выбирают одиночный столбец. Это обязательно, поскольку выбранный вывод сравнивается с одиночным значением. Подтверждением этому является то, что SELECT * не может использоваться в подзапросе. Имеется исключение из этого, когда подзапросы используются с оператором EXISTS.

В предложении SELECT-подзапроса можно использовать выражение, основанное на столбце, а не просто сам столбец. Это может быть выполнено или с помощью реляционных операторов, или с IN. Например, следующий запрос использует реляционный оператор = :

```
SELECT *
FROM Customers
WHERE cnum =
(SELECT snum + 1000
FROM Salespeople
WHERE sname = ‘Serres’);
```

Он находит всех заказчиков, для которых значение поля cnum на 1000 больше поля snum Serres. Мы предполагаем, что столбец sname не имеет никаких двойных значений (это может быть предписано ограничением UNIQUE;), иначе подзапрос может произвести многочисленные значения. Когда поля snum и cnum не имеют такого простого функционального значения, как например, первичный ключ, что не всегда хорошо, запрос типа вышеупомянутого невероятно полезен.

Также можно использовать подзапросы внутри предложения HAVING. Подзапросы в предложении HAVING выполняются по одному разу для очередного значения из поля в предложении GROUP BY. Эти подзапросы могут использовать свои собственные агрегатные функции, если они не производят многочисленных значений, а также

использовать GROUP BY или HAVING. Следующий запрос является этому примером:

```
SELECT rating, COUNT(DISTINCT cnum)
FROM Customers
GROUP BY rating
HAVING rating >
(SELECT AVG (rating)
FROM Customers
WHERE city = ‘San Jose’);
```

Эта команда подсчитывает заказчиков с оценками выше среднего в San Jose.

Когда используются подзапросы в SQL, можно использовать во внутреннем запросе таблицу, указанную в предложении внешнего запроса FROM, сформировав так называемый соотнесённый подзапрос. Соотнесённый подзапрос выполняется неоднократно, по одному разу для каждой строки таблицы основного запроса.

Например, имеется способ найти всех заказчиков в приобретениях на 3 октября:

```
SELECT *
FROM Customers outer
WHERE 10/03/2014 IN
(SELECT odate
FROM Orders inner
WHERE outer.cnum = inner.cnum);
```

В вышеупомянутом примере «внутренний» (inner) и «внешний» (outer) – это псевдонимы. Эти имена отсылают к значениям внутренних и внешних запросов соответственно. Так как значение в поле cnum внешнего запроса меняется, внутренний запрос должен выполняться отдельно для каждой строки внешнего запроса. Строка внешнего запроса, для которого внутренний запрос каждый раз будет выполнен, называется текущей строкой-кандидатом. Следовательно, процедура оценки, выполняемая соотнесённым подзапросом, – это:

1. Выбрать строку из таблицы, именованной во внешнем запросе. Это будет текущая строка-кандидат.

2. Сохранить значения из этой строки-кандидата в псевдониме таблицы из предложения FROM внешнего запроса.

3. Выполнить подзапрос. Везде, где псевдоним, данный для внешнего запроса, найден (в этом случае «внешний»), использовать значение для текущей строки-кандидата. Использование значения из строки-кандидата внешнего запроса в подзапросе называется внешней ссылкой.

4. Оценить предикат внешнего запроса на основе результатов подзапроса, выполняемого на шаге 3. Он определяет, выбирается ли строка-кандидат для вывода.

5. Повторить процедуру для следующей строки-кандидата таблицы, и так далее, пока все строки таблицы не будут проверены.

В вышеупомянутом примере SQL осуществляет следующую процедуру:

1. Он выбирает строку Hoffman из таблицы Заказчиков.

2. Сохраняет эту строку как текущую строку-кандидат под псевдонимом «внешний».

3. Затем он выполняет подзапрос. Подзапрос просматривает всю таблицу Приобретений, чтобы найти строки, где значение cnum – поле такое же, как значение outer.cnum, которое в настоящее время равно 2001, – поле cnum строки Hoffmana. Затем он извлекает поле odate из каждой строки таблицы Приобретений, для которой это истинно, и формирует набор значений поля odate.

4. Получив набор всех значений поля odate, для поля cnum = 2001, он проверяет предикат основного запроса, чтобы видеть, имеется ли значение на 3 октября в этом наборе. Если это так (а это так), то он выбирает строку Hoffmana для вывода её из основного запроса.

5. Он повторяет всю процедуру, используя строку Giovanni как строку-кандидат, и затем сохраняет повторно, пока каждая строка таблицы Заказчиков не будет проверена.

Конечно, ту же самую проблему можно решить, используя объединение следующего вида:

```
SELECT *
FROM Customers first, Orders second
WHERE first.cnum = second.cnum
AND second.odate = 10/03/2014;
```

При отсутствии имени таблицы или префикса псевдонима, SQL может для начала принять, что любое поле выводится из таблицы с именем указанным в предложении FROM текущего запроса. Если поле с этим именем отсутствует (в нашем случае – snum) в той таблице, SQL будет проверять внешние запросы. Именно поэтому префикс имени таблицы обычно необходим в соотнесённых подзапросах – для отмены этого предположения. Псевдонимы также часто запрашиваются, чтобы давать возможность ссылаться к той же самой таблице во внутреннем и внешнем запросах без какой-либо неоднозначности.

Так же как предложение HAVING может брать подзапросы, оно может брать и соотнесённые подзапросы. Когда используется соотнесённый подзапрос в предложении HAVING, необходимо ограничивать

внешние ссылки к позициям, которые могли бы непосредственно использоваться в самом предложении HAVING. Можно вспомнить, что предложение HAVING может использовать только агрегатные функции, которые указаны в предложении SELECT, или поля, используемые в предложении GROUP BY.

Они являются только внешними ссылками, которые можно делать. Всё это потому, что предикат предложения HAVING оценивается для каждой группы из внешнего запроса, а не для каждой строки.

Следовательно, подзапрос будет выполняться один раз для каждой группы выведенной из внешнего запроса, а не для каждой строки.

Предположим что требуется суммировать значения сумм приобретений покупок из таблицы Приобретений, сгруппировав их по датам, удалив все даты, где бы SUM не был по крайней мере на 2000.00 выше максимальной (MAX) суммы:

```
SELECT odate, SUM (amt)
FROM Orders a
GROUP BY odate
HAVING SUM (amt) >
( SELECT 2000.00 + MAX (amt)
FROM Orders b
WHERE a.odate = b.odate);
```

Подзапрос вычисляет значение MAX для всех строк с той же самой датой, что и у текущей агрегатной группы основного запроса. Это должно быть выполнено, как и ранее, с использованием предложения WHERE. Сам подзапрос не должен использовать предложения GROUP BY или HAVING.

Соотнесённые подзапросы по природе близки к объединениям – они оба включают проверку каждой строки одной таблицы с каждой строкой другой (или псевдонимом из той же) таблицы. Большинство операций, которые могут выполняться с одним из них, будут также работать и с другим. Однако имеются некоторые различия между ними, такое как вышеупомянутая потребность в использовании DISTINCT с объединением и его необязательность с подзапросом. Также подзапросы могут использовать агрегатную функцию в предикате, делая возможным выполнение операций типа нашего предыдущего примера, в котором мы извлекли приобретения, усреднённые для их заказчиков. Объединения, с другой стороны, могут выводить строки из обеих сравниваемых таблиц, в то время как вывод подзапросов используется только в предикатах внешних запросов. Как правило, форма запроса, которая кажется наиболее интуитивной, будет вероятно

лучшей в использовании, но при этом хорошо бы знать обе техники для тех ситуаций, когда та или иная могут не работать.

Теперь мы можем говорить о некоторых специальных операторах которые всегда используют подзапросы как аргументы.

Оператор EXISTS используется, чтобы указать предикату, производить ли стандартному или (обычно) соотнесённому подзапросу вывод или нет.

EXISTS – это оператор, который производит истинное или неистинное значение. Это означает, что он может работать автономно в предикате или в комбинации с другими выражениями, использующими булевы операторы AND, OR и NOT. Он берёт подзапрос как аргумент и оценивает его как истинный, если тот производит любой вывод или как неистинный, если тот не делает этого. Он отличается от других операторов тем, что он не может быть неизвестным.

Например, мы можем решить, извлекать ли нам некоторые данные из таблицы Заказчиков, если и только если один или более заказчиков в этой таблице находятся в San Jose:

```
SELECT cnum, cname, city
FROM Customers
WHERE EXISTS
( SELECT * FROM Customers WHERE city = 'San Jose');
```

Внутренний запрос выбирает все данные для всех заказчиков в San Jose. Оператор EXISTS во внешнем предикате отмечает, что некоторый вывод был произведён подзапросом, и поскольку выражение EXISTS было истинным, делает предикат истинным.

Подзапрос (не соотнесенный) был выполнен только один раз для всего внешнего запроса, и, следовательно, имеет одно значение во всех случаях. Поэтому EXISTS, когда используется этим способом, делает предикат истинным или неистинным для всех строк сразу, что не так уж полезно для извлечения определённой информации.

В вышеупомянутом примере EXISTS должен быть установлен так, чтобы легко выбрать один столбец, вместо того, чтобы выбирать все столбцы, используя в выборе звезду (SELECT *). В этом состоит его отличие от подзапроса, который мог выбрать только один столбец.

Однако в принципе он мало отличается при выборе EXISTS столбцов, или когда выбираются все столбцы, потому что он просто замечает – выполняется или нет вывод из подзапроса, а не использует выведенные значения.

В соотнесённом подзапросе предложение EXISTS оценивается отдельно для каждой строки таблицы, имя которой указано во внешнем запросе точно так же, как и другие операторы предиката, когда

используется соотнесённый подзапрос. Это даёт возможность использовать EXISTS как истинный предикат, который генерирует различные ответы для каждой строки таблицы, указанной в основном запросе. Следовательно, информация из внутреннего запроса будет сохранена, если выведена непосредственно, когда используются EXISTS таким способом.

Например, мы можем вывести продавцов, которые имеют многочисленных заказчиков:

```
SELECT DISTINCT snum
FROM Customers outer
WHERE EXISTS
( SELECT *
FROM Customers inner
WHERE inner.snum = outer.snum
AND inner.cnum < > outer.cnum);
```

Для каждой строки-кандидата внешнего запроса (представляющей заказчика, проверяемого в настоящее время) внутренний запрос находит строки, которые совпадают со значением поля snum (которое имел продавец), но не со значением поля cnum (соответствующего другим заказчикам).

Если любые такие строки найдены внутренним запросом, это означает, что имеются два разных заказчика, обслуживаемых текущим продавцом (т.е. продавцом заказчика в текущей строке-кандидате из внешнего запроса).

Предикат EXISTS поэтому верен для текущей строки, и номер продавца поля (snum) таблицы, указанной во внешнем запросе, будет выведен. Если DISTINCT был не указан, каждый из этих продавцов будет выбран один раз для каждого заказчика, к которому он назначен.

Однако для нас может быть полезнее вывести больше информации об этих продавцах, а не только их номера. Мы можем сделать это, объединив таблицу Customers с таблицей Salespeople:

```
SELECT DISTINCT first.snum, sname, first.city
FROM Salespeople first, Customers second
WHERE EXISTS
( SELECT *
FROM Customers third
WHERE second.snum = third.snum
AND second.cnum < > third.cnum)
AND first.snum = second.snum;
```

Внутренний запрос здесь, как и в предыдущем варианте, фактически сообщает, что псевдоним был изменён. Внешний запрос – это объ-

единение таблицы Salespeople с таблицей Customers, наподобие того что мы видели прежде. Новое предложение основного предиката (AND first.snum = second.snum), естественно, оценивается на том же самом уровне, что и предложение EXISTS. Это функциональный предикат самого объединения, сравнивающий две таблицы из внешнего запроса в терминах поля snum, которое является для них общим. Из-за булева оператора AND оба условия основного предиката должны быть истинны для истинности предиката.

Следовательно, результаты подзапроса имеют смысл только в тех случаях, когда вторая часть запроса истинна, а объединение выполняемо. Таким образом, комбинация объединения и подзапроса может стать очень мощным способом обработки данных.

Предыдущий пример дал понять, что EXISTS может работать в комбинации с операторами Буля. Конечно, то, что является самым простым способом для использования и вероятно наиболее часто используется с EXISTS – это оператор NOT.

Один из способов, которым мы могли бы найти всех продавцов только с одним заказчиком, будет состоять в том, чтобы инвертировать наш предыдущий пример.

```
SELECT DISTINCT snum
FROM Customers outer
WHERE NOT EXISTS
( SELECT *
FROM Customers inner
WHERE inner.snum = outer.snum
AND inner.cnum < > outer.cnum);
```

Одна вещь, которую EXISTS не может сделать, – использовать функцию агрегата в подзапросе. Если функция агрегата находит любые строки для операций с ними, EXISTS верен, невзирая на то, что это – значение функции, если же агрегатная функция не находит никаких строк, EXISTS неправилен.

В языке SQL есть три специальных оператора, ориентированных на подзапросы.

Операторы ALL (все), ANY (любой) и SOME (некоторый) напоминают EXISTS, который воспринимает подзапрос как аргумент; однако они отличаются от EXISTS тем, что используются совместно с реляционными операторами. В этом отношении они напоминают оператор IN, когда тот используется с подзапросами; они берут все значения, выведенные подзапросом, и обрабатывают их вместе. Однако, в отличие от IN, они могут использоваться только с подзапросами.

Операторы SOME и ANY взаимозаменяемы везде. Различие в терминологии состоит в том, чтобы позволить людям использовать тот термин, который наиболее подходит.

Покажем ещё один способ нахождения продавцов с заказчиками, размещёнными в их городах:

```
SELECT *
FROM Salespeople
WHERE city = ANY
(SELECT city FROM Customers);
```

Оператор ANY берёт все значения, выведенные подзапросом (для этого случая – это все значения city в таблице Заказчиков), и оценивает их как истинные, если любой (ANY) из них равняется значению города текущей строки внешнего запроса.

В операторе ANY подзапрос должен выбирать значения такого же типа, как и те, которые сравниваются в основном предикате. В этом его отличие от EXISTS, который просто определяет, производит ли подзапрос результаты или нет, и фактически не использует эти результаты.

Оператор ANY может использовать другие реляционные операторы, кроме равняется (=), и таким образом делать сравнения, которые являются выше возможностей IN. Например, можно найти всех продавцов, имена которых предшествуют именам заказчиков:

```
SELECT *
FROM Salespeople
WHERE sname < ANY
( SELECT cname FROM Customers);
```

В случае использования оператора ALL предикат является истинным, если каждое значение, выбранное подзапросом, удовлетворяет условию в предикате внешнего запроса.

Например, выведем только тех заказчиков, чьи оценки выше, чем у заказчиков в Риме:

```
SELECT *
FROM Customers
WHERE rating > ALL
(SELECT rating FROM Customers WHERE city = ‘Rome’):
```

ALL используется в основном с неравенствами, а не с равенствами, так как значение «равный для всех» может быть результатом подзапроса, только если все результаты идентичны.

В SQL выражение “< > ALL” соответствует «не равен любому» результату подзапроса. Другими словами, предикат верен, если данное значение не найдено среди результатов подзапроса. Например,

```
SELECT *
FROM Customers
WHERE rating < > ALL
(SELECT rating FROM Customers WHERE city = ‘San Jose’);
```

Здесь подзапрос выбирает оценки для города San Jose: 200 (для Liu) и 300 (для Cisneros). Затем основной запрос выбирает все строки с оценкой, не совпадающей ни с одной из них, другими словами, все строки с оценкой 100.

Значительное различие между ALL и ANY – способ действия в ситуации, когда подзапрос не возвращает никаких значений. Всякий раз, когда допустимый подзапрос не в состоянии сделать вывод, ALL автоматически верен, а ANY автоматически неправилен.

Вывод многих запросов можно объединить, используя предложение UNION. Предложение UNION объединяет вывод двух или более SQL-запросов в единый набор строк и столбцов. Например, чтобы получить всех продавцов и заказчиков, размещённых в Лондоне, и вывести их как единое целое, вы могли бы ввести:

```
SELECT snum, sname
FROM Salespeople
WHERE city = 'London'
 UNION
SELECT cnum, cname
FROM Customers
WHERE city = 'London';
```

Когда два (или более) запроса подвергаются объединению, их столбцы вывода должны быть совместимы для объединения. Это означает, что каждый запрос должен указывать одинаковое число столбцов, и каждый должен иметь тип, совместимый с каждым. Символьные поля должны иметь одинаковое число символов. Типы, не определённые ANSI, такие как DATA и BINARY, обычно должны совпадать с другими столбцами такого же нестандартного типа.

Другое ограничение на совместимость – если пустые значения (NULL) запрещены в каком-либо столбце объединения, эти значения необходимо запретить и для всех соответствующих столбцов в других запросах объединения.

Кроме того, мы не можем использовать UNION в подзапросах, а также не можем использовать агрегатные функции в предложении SELECT-запроса в объединении. UNION автоматически исключает дубликаты строк из вывода.

Также можно использовать предложение ORDER BY, чтобы упорядочить вывод из объединения, точно так же как это делается в индивидуальных запросах.

Операция, которая бывает часто полезна, – это объединение из двух запросов, в котором второй запрос выбирает строки, исключённые первым. Это называется *внешним объединением*.

Предположим, что некоторые из ваших заказчиков ещё не были назначены к продавцам. Можно захотеть увидеть имена и города всех ваших заказчиков с именами их продавцов, не учитывая тех, кто ещё не был назначен. Можно достичь этого, формируя объединение из двух запросов, один из которых выполняет основной вывод, а другой выбирает заказчиков с пустыми (NULL) значениями поля snum. Этот последний запрос должен вставлять пробелы в поля, соответствующие полю sname в первом запросе.

Использование этой методики во внешнем объединении даёт возможность использовать предикаты для классификации, а не для исключения.

Мы использовали пример нахождения продавцов с заказчиками, размещёнными в их городах и раньше. Однако вместо просто выбора только этих строк вы возможно захотите, чтобы ваш вывод перечислял всех продавцов, и указывал тех, кто не имел заказчиков в их городах, и кто имел.

Следующий запрос выполнит это:

```
SELECT Salespeople.snum, sname, cname, comm
FROM Salespeople, Customers
WHERE Salespeople.city = Customers.city.

UNION

SELECT snum, sname, ' NO MATCH ', comm
FROM (Salespeople
WHERE NOT city = ANY(SELECT city FROM Customers)
ORDER BY 2 DESC;
```

Строка 'NO MATCH' была дополнена пробелами, чтобы получить совпадение поля cname по длине (это не обязательно во всех реализациях SQL).

Второй запрос выбирает даже те строки, которые исключил первый.

Всякий раз, когда вы выполняете объединение более чем двух запросов, можно использовать круглые скобки, чтобы определить порядок оценки. Другими словами, вместо просто

query X UNION query Y UNION query Z;

можно указать, или

(query X UNION query Y) UNION query Z;

или

query X UNION (query Y UNION query Z);

6.7. ВВОД, УДАЛЕНИЕ И ИЗМЕНЕНИЕ ЗНАЧЕНИЙ ПОЛЕЙ

Значения могут быть помещены и удалены из полей тремя командами языка DML (Язык Манипулирования Данными):

INSERT (ВСТАВИТЬ),
UPDATE (МОДИФИЦИРОВАТЬ),
DELETE (УДАЛИТЬ).

Все строки в SQL вводятся с использованием команды модификации INSERT. В самой простой форме INSERT использует следующий синтаксис:

INSERT INTO <table name>
VALUES (<value>, <value> . . .);

Так, например, чтобы ввести строку в таблицу Salespeople (Продавцов), можно использовать следующее условие:

INSERT INTO Salespeople
VALUES (1001, 'Peel', 'London', .12);

Имя таблицы (в нашем случае – Salespeople (Продавцы)) должно быть предварительно определено в команде CREATE TABLE, а каждое значение, пронумерованное в предложении значений, должно совпадать с типом данных столбца, в который оно вставляется. Значения вводятся в таблицу в поимённом порядке, поэтому первое значение автоматически попадает в столбец 1, второе в столбец 2 и т.д.

Если вам нужно ввести пустое значение (NULL), вы вводите его точно так же, как и обычное значение. Предположим, что ещё не имелось поля city для господина Peel. Можно вставить его строку со значением NULL в это поле следующим образом:

INSERT INTO Salespeople
VALUES (1001, 'Peel', NULL, .12);

Так как значение NULL – это специальный маркер, а не просто символьное значение, он не заключается в одиночные кавычки.

Также можно указывать столбцы, куда требуется вставить значение имени. Это позволяет вам вставлять имена в любом порядке.

Предположим, что вы берёте значения для таблицы Заказчиков из отчёта, выводимого на принтер, который помещает их в таком порядке:

city, cname, и cnum,

и для упрощения требуется ввести значения в том же порядке:

```
INSERT INTO Customers (city, cname, cnum)
VALUES ('London', 'Honman', 2001);
```

Обратите внимание, что столбцы rating и snum отсутствуют. Это значит, что эти строки автоматически установлены в значение по умолчанию.

По умолчанию может быть введено или значение NULL, или другое значение, определяемое по умолчанию. Если ограничение запрещает использование значения NULL в данном столбце, и этот столбец не установлен по умолчанию, этот столбец должен быть обеспечен значением для любой команды INSERT, которая относится к таблице.

Можно также использовать команду INSERT, чтобы получать или выбирать значения из одной таблицы и помещать их в другую, чтобы использовать их вместе с запросом. Чтобы сделать это, вы просто заменяете предложение VALUES (из предыдущего примера) на соответствующий запрос:

```
INSERT INTO Londonstaff
SELECT *
FROM Salespeople
WHERE city = 'London';
```

Здесь выбираются все значения, произведённые запросом, т.е. все строки из таблицы Salespeople со значениями city «London», и помещаются в таблицу, называемую Londonstaff. Чтобы это работало, таблица Londonstaff должна отвечать следующим условиям:

– она должна уже быть создана командой CREATE TABLE;

– она должна иметь четыре столбца, которые совпадают с таблицей Продавцов в терминах типа данных, т.е. первый, второй, и т.д. столбцы каждой таблицы должны иметь одинаковый тип данных (причём они не должны иметь одинаковых имён).

Londonstaff – это теперь независимая таблица, которая получила некоторые значения из таблицы Продавцов (Salespeople). Однако, если значения в таблице Продавцов будут вдруг изменены, это никак не отразится на таблице Londonstaff.

Так как или запрос, или команда INSERT могут указывать столбцы по имени, можно переместить выбранные запросом столбцы, а также переупорядочить те столбцы, которые вы выбрали.

Предположим, например, что вы решили сформировать новую таблицу с именем Daytotals, которая просто будет следить за общим количеством сумм приобретений, упорядоченных на каждый день.

Можно потом вводить эти данные независимо от таблицы Приобретений, но сначала необходимо заполнить таблицу Daytotals информацией, ранее представленной в таблице Приобретений:

```
INSERT INTO Daytotals (date, total)
SELECT odate, SUM (amt)
FROM Orders
GROUP BY odate;
```

Обратите внимание, что, как указано ранее, имена столбцов таблицы Приобретений и таблицы Daytotals не должны быть одинаковыми. Кроме того, если дата приобретения и общее количество – это единственные столбцы в таблице, и они находятся в данном порядке, их имена могут быть исключены из вывода.

Подзапросы можно использовать внутри любого запроса, который генерирует значения для команды INSERT тем же самым способом, которым вы делали это для других запросов – т.е. внутри предиката или предложения HAVING.

Строки из таблицы можно удалять командой DELETE. Она может удалять только введённые строки, а не индивидуальные значения полей.

Чтобы удалить все содержание таблицы Продавцов, можно ввести следующее условие:

```
DELETE FROM Salespeople;
```

Теперь, когда таблица пуста, её можно окончательно удалить командой DROP TABLE.

Обычно нужно удалить только некоторые определённые строки из таблицы. Чтобы определить, какие строки будут удалены, используется предикат так же, как это делается для запросов. Например, чтобы удалить продавца Axelrod из таблицы, можно ввести

```
DELETE FROM Salespeople
WHERE snum = 1003;
```

Мы использовали поле snum вместо поля sname, потому что лучшая тактика – использование первичных ключей, когда требуется, чтобы действию подвергалась одна и только одна строка.

Конечно, можно также использовать DELETE с предикатом, который выбирает группу строк, как показано в этом примере:

```
DELETE FROM Salespeople
WHERE city = 'London';
```

Можно также использовать подзапросы в предикате команды DELETE.

Чтобы изменять некоторые или все значения в существующей строке, используется команда UPDATE.

Эта команда содержит предложение UPDATE, в котором указано имя используемой таблицы и предложение SET, которое указывает на изменение, которое нужно сделать для определённого столбца.

Например, чтобы изменить оценки всех заказчиков на 200, можно ввести

```
UPDATE Customers
SET rating = 200;
```

Конечно, вы не всегда захотите указывать все строки таблицы для изменения единственного значения, так что UPDATE наподобие DELETE может использовать предикаты. Вот как, например, можно выполнить изменение для всех заказчиков продавца Peel (имеющего snum = 1001):

```
UPDATE Customers
SET rating = 200
WHERE snum = 1001;
```

Предложение SET может назначать любое число столбцов, отделяемых запятыми. Все указанные назначения могут быть сделаны для любой табличной строки, но только для одной в каждый момент времени. Предположим, что продавец Motika ушёл на пенсию, и мы хотим переназначить его номер новому продавцу:

```
UPDATE Salespeople
SET sname = 'Gibson', city = 'Boston', comm = .10
WHERE snum = 1004;
```

Эта команда передаст новому продавцу Gibson всех текущих заказчиков бывшего продавца Motika и приобретения в том виде, в котором они были скомпонованы для Motika.

Вы не можете, однако, модифицировать сразу много таблиц в одной команде, частично потому, что вы не можете использовать префиксы таблицы со столбцами, изменёнными предложением SET. Другими словами, вы не можете сказать – «SET Salespeople.sname = Gibson» в команде UPDATE, можно сказать только так – «SET sname = Gibson».

В предложении SET команды UPDATE можно использовать скалярные выражения, однако, включив его в выражение поля, которое будет изменено. В этом отличие от предложения VALUES команды INSERT, в котором выражения не могут использоваться.

Предположим, что вы решили удвоить комиссионные всем вашим продавцам. Можно использовать следующее выражение:

```
UPDATE Salespeople
SET comm = comm * 2;
```

Всякий раз, когда вы ссылаетесь к указанному значению столбца в предложении SET, используемое значение получается из текущей строки, прежде чем в ней будут сделаны изменения с помощью команды UPDATE.

Предложение SET – это не предикат. Он может вводить пустые (NULL) значения так же, как он вводит непустые значения. Так что, если требуется установить все оценки заказчиков в Лондоне в NULL, можно ввести следующее предложение:

```
UPDATE customers
SET rating = NULL
WHERE city = 'London';
```

что обнулит все оценки заказчиков в Лондоне.

Команда UPDATE использует подзапросы тем же самым способом, что и команда DELETE.

6.8. СОЗДАНИЕ ТАБЛИЦ

Этот раздел вводит нас в область SQL, называемую – DDL (*Язык Определения Данных*), где создаются объекты данных SQL.

Таблицы создаются командой CREATE TABLE. Эта команда создает пустую таблицу – таблицу без строк. Значения вводятся с помощью DML команды INSERT.

Команда CREATE TABLE в основном определяет имя таблицы, описание набора имён столбцов, указанных в определённом порядке. Она также определяет типы данных и размеры столбцов. Каждая таблица должна иметь по крайней мере один столбец.

Синтаксис команды CREATE TABLE:

```
CREATE TABLE <table-name >
(<column name > <data type>[(<size>)],
 <column name > <data type> [(<size>)] ...);
```

Так как пробелы используются для разделения частей команды SQL, они не могут быть частью имени таблицы (или любого другого объекта). Подчёркивание (_) обычно используется для разделения слов в именах таблиц.

Значение аргумента размера зависит от типа данных. Если вы его не указываете, ваша система сама будет назначать значение автоматически. Для числовых значений это лучший выход, потому что в этом

случае все ваши поля такого типа получат один и тот же размер, что освобождает вас от проблем их общей совместимости.

Тип данных, для которого вы в основном должны назначать размер, – тип CHAR. Фактически число символов поля может быть от нуля (если поле NULL) до значения аргумента размера. По умолчанию аргумент размера равен 1, что означает, что поле может содержать только одну букву.

Таблицы принадлежат пользователю, который их создал, и имена всех таблиц, принадлежащих данному пользователю должны отличаться друга от друга, как и имена всех столбцов внутри данной таблицы. Отдельные таблицы могут использовать одинаковые имена столбцов, даже если они принадлежат одному и тому же пользователю.

Пользователи, не являющиеся владельцами таблиц, могут ссылаться к этим таблицам с помощью имени владельца этих таблиц, сопровождаемого точкой; например, таблица Employees, созданая Smith, будет называться Smith.Employees, когда она упоминается каким-то другим пользователем. Мы понимаем, что Smith – это *Идентификатор Разрешения* (ID), сообщаемый пользователем (ваш разрешённый ID – это ваше имя в SQL).

Следующая команда будет создавать таблицу Продавцов:

```
CREATE TABLE Saleepeople
(snum integer,
sname char (10),
city char (10),
comm declmal);
```

Порядок столбцов в таблице определяется порядком, в котором они указаны. Имя столбца не должно разделяться при переносе строки (что сделано для удобочитаемости), но отделяется запятыми.

Таблицы могут иметь большое количество строк, а так как строки не находятся в каком-нибудь определённом порядке, на их поиск по указанному значению может потребоваться большое время.

Команда ALTER TABLE используется для того, чтобы изменить определение существующей таблицы. Обычно она добавляет столбцы к таблице. Иногда она может удалять столбцы или изменять их размеры, а также в некоторых программах добавлять или удалять ограничения.

Типичный синтаксис предложения для добавления столбца к таблице:

```
ALTER TABLE <table name> ADD <column name>
<data type> <size>;
```

Столбец будет добавлен со значением NULL для всех строк таблицы.

Новый столбец станет последним по порядку столбцом таблицы. Вообще то, можно добавить сразу несколько новых столбцов, отделив их запятыми, в одной команде. Имеется возможность удалять или изменять столбцы. Наиболее часто изменением столбца может быть просто увеличение его размера, или добавление (удаление) ограничения.

Команда ALTER TABLE опасна, так как изменение может стереть всех пользователей, имеющих разрешение обращаться к таблице. Необходимо разрабатывать таблицы так, чтобы использовать ALTER TABLE только в крайнем случае.

Необходимо быть собственником (т.е. быть создателем) таблицы, чтобы иметь возможность удалить её. Поэтому не волнуйтесь о случайном разрушении ваших данных, SQL сначала потребует, чтобы вы очистили таблицу прежде, чем удалит её из базы данных. Таблица с находящимися в ней строками не может быть удалена. Синтаксис для удаления вашей таблицы, если конечно она является пустой, следующий:

DROP TABLE < table name >;

При подаче этой команды имя таблицы больше не распознаётся, и нет такой команды, которая могла быть дана этому объекту. Необходимо убедиться, что эта таблица не ссылается внешним ключом к другой таблице и что она не используется в определении Представления.

6.9. ИНДЕКСЫ

Индекс – это упорядоченный (буквенный или числовой) список строк или групп строк в таблице.

Индексный адрес – это способ объединения всех значений в группы из одной или больше строк, которые отличаются одна от другой. Дальше мы будем описывать более непосредственный способ, который заставит ваши значения быть уникальными. Но этот метод не существует в ранних версиях SQL. Так как уникальность часто необходима, индексы и использовались с этой целью.

Когда вы создаёте индекс в поле, ваша база данных запоминает соответствующий порядок всех значений этого поля в области памяти. Предположим, что наша таблица Заказчиков имеет тысячи входов, а требуется найти заказчика с номером 2999. Если строки не упорядочены, ваша программа будет просматривать всю таблицу, строку за строкой, проверяя каждый раз значение поля cnum на равенство значению 2999.

Однако, если бы имелся индекс в поле cnum, то программа могла бы выйти на номер 2999 прямо по индексу и дать информацию о том, как найти правильную строку таблицы.

В то время как индекс значительно улучшает эффективность запросов, использование индекса несколько замедляет операции модификации DML (такие как INSERT и DELETE), а сам индекс занимает объём памяти. Следовательно, каждый раз, когда вы создаёте таблицу, необходимо принять решение, индексировать её или нет.

Индексы могут состоять из многочисленных полей. Если больше чем одно поле указывается для одного индекса, второе упорядочивается внутри первого, третье внутри второго и т.д.

Синтаксис для создания индекса обычно следующий:

CREATE INDEX <index name> ON <table name>
(<column name> [,<column name>]...);

Таблица, конечно, должна уже быть создана и должна содержать имя столбца. Имя индекса не может быть использовано для чего-то другого в базе данных (любым пользователем). Однажды созданный, индекс будет невидим пользователю. SQL сам решает, когда он необходим, чтобы ссылаться на него и делает это автоматически.

Если, например, таблица Заказчиков будет наиболее часто упоминаемой в запросах продавцов к их собственной клиентуре, было бы правильно создать такой индекс в поле snum таблицы Заказчиков.

CREATE INDEX Clientgroup ON Customers (snum);

Индексу в предыдущем примере не предписывается уникальность, несмотря на наше замечание, что это является одним из назначений индекса. Данный продавец может иметь любое число заказчиков.

Уникальность задаётся ключевым словом UNIQUE перед ключевым словом INDEX. Поле cnum в качестве первичного ключа станет первым кандидатом для уникального индекса:

CREATE UNIQUE INDEX Custid ON Customers (cnum);

Эта команда будет отклонена, если уже имеются идентичные значения в поле cnum. Лучше создавать индекс сразу после того, как таблица создана, и прежде, чем введены любые значения.

Для уникального индекса более чем одного поля используется комбинация значений, каждое из которых может и не быть уникальным.

Главным признаком индекса является его имя – поэтому он может быть удалён. Обычно пользователи не знают о существовании индекса. SQL автоматически определяет, позволено ли пользователю использовать индекс, и если да, то разрешает использовать его. Однако, если требуется удалить индекс, необходимо знать его имя. Следующий синтаксис используется для удаления индекса:

DROP INDEX <Index name>;

Удаление индекса не воздействует на содержание полей.

6.10. ОГРАНИЧЕНИЕ ЗНАЧЕНИЙ ДАННЫХ

Ограничения – это часть определений таблицы, которая ограничивает значения, которые можно вводить в столбцы.

Значение по умолчанию – это значение, которое вставляется автоматически в любой столбец таблицы, когда значение для этого столбца отсутствует в команде INSERT для этой таблицы. NULL – это наиболее широко используемое значение по умолчанию.

Когда вы создаёте таблицу (или, когда вы её изменяете), можно помещать ограничение на значения, которые могут быть введены в поля. Если вы это сделали, SQL будет отклонять любые значения, которые нарушают критерии, которые вы определили.

Имеются два основных типа ограничений – ограничение столбца и ограничение таблицы. Различие между ними в том, что ограничение столбца применяется только к индивидуальным столбцам, в то время как ограничение таблицы применяется к группам из одного и более столбцов.

Ограничение столбца вставляется в конец имени столбца после типа данных и перед запятой. Ограничение таблицы помещается в конец имени таблицы после последнего имени столбца, но перед заключительной круглой скобкой.

Далее показан синтаксис для команды CREATE TABLE, расширенной для включения в неё ограничения:

```
CREATE TABLE < table name >
(< column name > <data type > < column constraint >,
< column name > < data type > < column constraint > ...
< table constraint >(< column name >
[, < column name >])...);
```

Для краткости мы опустили аргумент размера, который иногда используется с типом данных.

Поля, данные в круглых скобках после ограничения таблицы – это поля, к которым применено это ограничение. Ограничение столбца, естественно, применяется к столбцам, после чьих имён оно следует.

Команду CREATE TABLE можно использовать, чтобы предохранить поле от разрешения в нём пустых (NULL) указателей с помощью ограничения NOT NULL.

Это ограничение накладывается только для отдельных столбцов.

Очевидно, что первичные ключи никогда не должны быть пустыми, поскольку это будет подрывать их функциональные возможности. Кроме того, такие поля, как имена, требуют в большинстве случаев, определённых значений.

Например, вы вероятно захотите иметь имя для каждого заказчика в таблице Заказчиков. Если вы поместите ключевые слова NOT NULL сразу после типа данных (включая размер) столбца, любая попытка поместить значение NULL в это поле будет отклонена. В противном случае SQL понимает, что NULL разрешён.

Например, давайте улучшим наше определение таблицы Продавцов, не позволяя помещать NULL-значения в столбцы snum или sname :

```
CREATE TABLE Salespeople
(snum integer NOT NULL,
sname char (10) NOT NULL,
city char (10),
comm decimal);
```

Если ваша система поддерживает использование ALTER TABLE чтобы добавлять новые столбцы к уже существующей таблице, можно вероятно помещать ограничение столбцов типа NOT NULL для этих новых столбцов. Однако, если вы предписываете новому столбцу значение NOT NULL, текущая таблица должна быть пустой.

Если вы помещаете ограничение столбца UNIQUE в поле при создании таблицы, база данных отклонит любую попытку ввода в это поле значения, которое уже представлено в другой строке.

Это ограничение может применяться только к полям, которые были объявлены как непустые (NOT NULL), так как не имеет смысла позволить одной строке таблицы иметь значение NULL, а затем исключать другие строки с NULL-значениями как дубликаты.

Усовершенствуем нашу команду создания таблицы Продавцов:

```
CREATE TABLE Salespeople
(snum integer NOT NULL UNIQUE,
sname char (10) NOT NULL UNIQUE,
city char (10),
comm decimal);
```

Когда вы объявляете поле sname уникальным, убедитесь, что две Mary Smith будут введены различными способами – например, Mary Smith и M. Smith. В то же время это не так уж необходимо с функциональной точки зрения, потому что поле snum в качестве первичного ключа всё равно обеспечит отличие этих двух строк; это проще для людей, использующих данные в таблицах, чем помнить, что эти Smith не идентичны.

Столбцы, чьи значения требуют уникальности, называются ключами-кандидатами, или уникальными ключами.

Можно также определить группу полей как уникальную с помощью команды ограничения таблицы UNIQUE. Объявление группы полей уникальной отличается от объявления уникальными индивидуальных полей, так как это комбинация значений, а не просто индивидуальное значение, которое обязано быть уникальным.

Наша база данных сделана так, чтобы каждый заказчик был назначен одному и только одному продавцу. Это означает, что каждая комбинация номера заказчика (cnum) и номера продавца (snum) в таблице Заказчиков должна быть уникальной. Можно ввести это ограничение, создав таблицу Заказчиков таким способом:

```
CREATE TABLE Customers
(cnum integer NOT NULL,
cname char (10) NOT NULL,
city char (10),
rating integer,
snum integer NOT NULL,
UNIQUE (cnum, snum));
```

Обратите внимание, что оба поля в ограничении таблицы UNIQUE используют ограничение столбца NOT NULL.

SQL поддерживает первичные ключи непосредственно с ограничением Первичный Ключ (PRIMARE KEY). PRIMARY KEY может ограничивать таблицы или их столбцы. Это ограничение работает так же, как и ограничение UNIQUE, за исключением того, что только один первичный ключ может быть определён для данной таблицы. Синтаксис и определение уникальности первичного ключа те же, что и для ограничения UNIQUE.

Первичные ключи не могут позволять значений NULL. Это означает, что, подобно полям в ограничении UNIQUE, любое поле, используемое в ограничении PRIMARY KEY, должно уже быть объявлено NOT NULL.

Имеется улучшенный вариант создания нашей таблицы Продавцов:

```
CREATE TABLE Salestotal
(snum integer NOT NULL PRIMARY KEY,
sname char (10) NOT NULL UNIQUE,
city char (10),
comm decimal);
```

Ограничение PRIMARY KEY может также быть применено для многочисленных полей, составляющих уникальную комбинацию значений.

Предположим, что ваш первичный ключ – это имя, и вы имеете первое имя и последнее имя сохранёнными в двух различных полях (так что можно организовывать данные с помощью любого из них). Мы можем применить ограничение таблицы PRIMARY KEY для пар:

```
CREATE TABLE Namefield
(firstname char (10) NOT NULL,
lastname char (10) NOT NULL
city char (10),
PRIMARY KEY (firstname, lastname));
```

Имеется много ограничений, которые можно устанавливать для данных, вводимых в таблицы, чтобы видеть, например, находятся ли данные в соответствующем диапазоне или правильном формате, о чём SQL, естественно, не может знать заранее. По этой причине SQL оперирует ограничением CHECK, которое позволяет установить условие, которому должно удовлетворять значение, вводимое в таблицу, прежде чем оно будет принято. Ограничение CHECK состоит из ключевого слова CHECK, сопровождаемого предложением предиката, который использует указанное поле. Любая попытка модифицировать или вставить значение поля, которое могло бы сделать этот предикат неверным, будет отклонена.

Давайте рассмотрим ещё раз таблицу Продавцов. Столбец комиссионных выражается десятичным числом и поэтому может быть умножен непосредственно на сумму приобретений, в результате чего будет получена сумма комиссионных (в долларах) продавца с установленным справа значком доллар ($). Кто-то может использовать понятие процента, однако ведь можно об этом и не знать. Если человек введёт по ошибке 14 вместо .14, чтобы указать в процентах свои комиссионные, это будет расценено как 14.0, что является законным десятичным значением, и будет нормально воспринято системой. Чтобы предотвратить эту ошибку, мы можем наложить ограничение столбца CHECK, чтобы убедиться, что вводимое значение меньше, чем 1.

```
CREATE TABLE Salespeople
(snum integer NOT NULL PRIMARY KEY,
sname char (10) NOT NULL UNIQUE,
city char (10),
comm decimal CHECK(comm < 1));
```

Мы можем также использовать ограничение CHECK, чтобы защитить от ввода в поле определённых значений, и таким образом предотвратить ошибку. Например, предположим, что единственными городами, в которых мы имели ведомства сбыта, являются Лондон, Бар-

селона, Сан-Хосе, и Нью-Йорк. Если вам известны все продавцы, работающие в каждом из этих ведомств, нет необходимости позволять ввод других значений. Если же нет, использование ограничения может предотвратить опечатки и другие ошибки.

```
CREATE TABLE Salespeople
(snum integer NOT NULL UNIQUE,
sname char (10) NOT NULL UNIQUE,
city char (10) CHECK
(city IN ('London', 'New York', 'San Jose', 'Barselona')),
comm decimal CHECK (comm < 1));
```

Можно также использовать CHECK в качестве табличного ограничения. Это полезно в тех случаях, когда требуется включить более одного поля строки в условие. Предположим, что комиссионные .15 и выше будут разрешены только для продавца из Барселоны. Можно указать это со следующим табличным ограничением CHECK:

```
CREATE TABLE Salespeople
(snum integer NOT NULL UNIQUE,
sname char (10) NOT NULL UNIQUE,
city char (10),
comm decimal,
CHECK (comm < .15 OR city = 'Barcelona'));
```

Как можно видеть, два различных поля должны быть проверены, чтобы определить, верен предикат или нет. Имейте в виду, что проверяются два разных поля одной и той же строки. Хотя можно использовать многочисленные поля, SQL не может проверить более одной строки одновременно. Вы не можете, например, использовать ограничение CHECK, чтобы удостовериться, что все комиссионные в данном городе одинаковы.

Значение по умолчанию DEFAULT (ПО УМОЛЧАНИЮ) указывается в команде CREATE TABLE тем же способом что и ограничение столбца.

Предположим, что вы работаете в офисе Нью-Йорка, и подавляющее большинство ваших продавцов живут в Нью-Йорке. Можно указать Нью-Йорк в качестве значения поля city по умолчанию для таблицы Продавцов:

```
CREATE TABLE Salespeople
(snum integer NOT NULL UNIQUE,
sname char (10) NOT NULL UNIQUE,
city char (10) DEFAULT = 'New York',
comm decimal CHECK (comm < 1);
```

Длинные числовые значения более расположены к ошибке, поэтому если подавляющее большинство (или все) ваших порядков должны иметь ваш собственный конторский номер, желательно устанавливать для них значение по умолчанию.

Другой способ использовать значение по умолчанию – это использовать его как альтернативу для NULL. Так как NULL (фактически) неверен при любом сравнении, ином чем IS NULL, он может быть исключён с помощью большинства предикатов. Иногда вам нужно видеть пустые значения ваших полей, не обрабатывая их каким-то определённым образом. Для этого можно установить значение по умолчанию, типа нуль или пробел, которые функционально меньше по значению, чем просто не установленное значение – пустое значение (NULL). Различие между ними и обычным NULL в том, что SQL будет обрабатывать их так же, как и любое другое значение.

Можно также использовать ограничения UNIQUE или PRIMARY KEY в этом поле. Если вы сделаете это, то имейте в виду, что только одна строка одновременно может иметь значение по умолчанию. Любую строку, которая содержит значение по умолчанию, нужно будет модифицировать прежде, чем другая строка с установкой по умолчанию будет вставлена. Это неудобно, поэтому ограничения UNIQUE и PRIMARY KEY (особенно последнее) обычно не устанавливаются для строк со значениями по умолчанию.

Ограничения FOREIGN KEY или REFERENCES очень похожи на рассмотренные, за исключением того, что они связывают группу из одного или более полей с другой группой, и таким образом сразу воздействуют на значения, которые могут быть введены в любую из этих групп.

6.11. ПОДДЕРЖКА ЦЕЛОСТНОСТИ ДАННЫХ

Ранее мы указывали на определённые связи, которые существуют между некоторыми полями наших типовых таблиц. Поле snum таблицы Заказчиков, например, соответствует полю snum в таблице Продавцов и таблице Приобретений. Поле cnum таблицы Заказчиков также соответствует полю cnum таблицы Приобретений. Мы назвали этот тип связи справочной целостностью.

Когда все значения в одном поле таблицы представлены в поле другой таблицы, мы говорим что первое поле ссылается на второе. Например, каждый из заказчиков в таблице Заказчиков имеет поле snum, которое указывает на продавца, назначенного в таблице Продавцов.

Когда одно поле в таблице ссылается на другое, оно называется внешним ключом (foreign key); а поле, на которое оно ссылается, называется родительским ключом. Так что поле snum таблицы Заказчиков – это внешний ключ, а поле snum, на которое оно ссылается в таблице Продавцов, – это родительский ключ.

Имена внешнего и родительского ключей не обязательно должны быть одинаковыми. Внешний ключ не обязательно состоит только из одного поля. Внешний и родительский ключи должны иметь одинаковый размер и тип поля. Каждое значение во внешнем ключе должно быть представлено один, и только один раз, в родительском ключе.

SQL поддерживает справочную целостность с ограничением FOREIGN KEY. Эта функция ограничивает значения, которые можно ввести в базу данных.

Одно из действий ограничения FOREIGN KEY – это отбрасывание значений, которые ещё не представлены в родительском ключе. Это ограничение также воздействует на возможность изменять или удалять значения родительского ключа.

ВНЕШНИЙ КЛЮЧ КАК ОГРАНИЧЕНИЕ ТАБЛИЦЫ

Синтаксис ограничения таблицы FOREIGN KEY:

```
FOREIGN KEY <column list> REFERENCES
<pktable> [<column list>]
```

Первый список столбцов – это список из одного или более столбцов таблицы, которые отделены запятыми и будут созданы или изменены этой командой. Pktable – это таблица, содержащая родительский ключ. Она может быть таблицей, которая создаётся или изменяется текущей командой. Второй список столбцов – это список столбцов, которые будут составлять родительский ключ.

Списки двух столбцов должны быть совместимы, т.е.:

– они должны иметь одинаковое число столбцов;

– в данной последовательности первый, второй, третий и т.д. столбцы списка столбцов внешнего ключа должны иметь одинаковые типы данных и размеры, что и первый, второй, третий и т.д. столбцы списка столбцов родительского ключа.

Создадим таблицу Заказчиков с полем snum, определенным в качестве внешнего ключа, ссылающегося на таблицу Продавцов:

```
CREATE TABLE Customers
(cnum integer NOT NULL PRIMARY KEY
cname char (10),
city char (10),
snum integer,
FOREIGN KEY (snum) REFERENCES Salespeople (snum);
```

Имейте в виду, что при использовании ALTER TABLE вместо CREATE TABLE для применения ограничения FOREIGN KEY значения, которые Вы указываете во внешнем ключе и родительском ключе, должны быть в состоянии справочной целостности.

Вариант ограничения столбца ограничением FOREIGN KEY называется ссылочным ограничением (REFERENCES), так как он фактически не содержит в себе слов FOREIGN KEY, а просто использует слово REFERENCES и далее имя родительского ключа подобно этому:

```
CREATE TABLE Customers
(cnum integer NOT NULL PRIMARY KEY,
cname char (10),
city char (10),
snum integer REFERENCES Salespeople (snum));
```

Вышеупомянутое определяет Customers.snum как внешний ключ, у которого родительский ключ – это Salespeople.snum. Это эквивалентно такому ограничению таблицы:

```
FOREIGN KEY (snum) REGERENCES Salespeople (snum)
```

Используя ограничение FOREIGN KEY таблицы или столбца, можно не указывать список столбцов родительского ключа, если родительский ключ имеет ограничение PRIMARY KEY. Например, если мы поместили ограничение PRIMARY KEY в поле snum таблицы Продавцов, мы могли бы использовать его как внешний ключ в таблице Заказчиков (подобно предыдущему примеру) в этой команде:

```
CREATE TABLE Customers
(cnum integer NOT NULL PRIMARY KEY,
cname char (10),
city char (10),
snum integer REFERENCES Salespeople);
```

Поддержание справочной целостности требует некоторых ограничений на значения, которые могут быть представлены в полях, объявленных как внешний и родительский ключи. Это означает, что родительский ключ должен быть уникальным и не содержать никаких пустых значений (NULL). Следовательно, необходимо убедиться, что все поля, которые используются как родительские ключи, имеют или ограничение PRIMARY KEY, или ограничение UNIQUE наподобие ограничения NOT NULL.

Как ограничения воздействуют на возможность и невозможность использовать команды модификации DML? Для полей, определённых как внешние ключи, ответ довольно простой: любые значения, кото-

рые вы помещаете в эти поля с командой INSERT или UPDATE, должны уже быть представлены в их родительских ключах. Можно помещать пустые (NULL) значения в эти поля, несмотря на то что значения NULL не позволительны в родительских ключах, если они имеют ограничение NOT NULL. Можно удалять (DELETE) любые строки с внешними ключами, не используя родительские ключи вообще.

Любое значение родительского ключа, ссылаемого с помощью значения внешнего ключа, не может быть удалено или изменено. Это означает, например, что вы не можете удалить заказчика из таблицы Заказчиков, пока он ещё имеет приобретения в таблице Приобретений.

Смысл этой системы ограничения в том, что создатель таблицы Приобретений, используя таблицу Заказчиков и таблицу Продавцов как родительские ключи, может наложить значительные ограничения на действия в этих таблицах. По этой причине вы не сможете использовать таблицу, которой вы не распоряжаетесь (т.е. не вы её создавали и не вы являетесь её владельцем), пока владелец (создатель) этой таблицы специально не передаст вам на это право.

Если требуется изменить или удалить текущее ссылочное значение родительского ключа, имеется по существу три возможности:

– можно ограничить, или запретить, изменение (способом ANSI), обозначив, что изменения в родительском ключе ограничены – стратегия ограниченных (RESTRICTED) изменений;

– можно сделать изменение в родительском ключе и назначить изменения во внешнем ключе Каскадируемые (CASCADES) изменения автоматически, что называется каскадным изменением, – стратегия каскадируемых (CASCADES) изменений;

– можно сделать изменение в родительском ключе, и установить внешний ключ в NULL автоматически (полагая, что NULL разрешён во внешнем ключе), что называется пустым изменением внешнего ключа.

В качестве иллюстрации мы покажем несколько примеров того, что можно делать с полным набором эффектов модификации и удаления.

Позволим себе предположить, что вы имеете причину изменить поле snum таблицы Продавцов. Когда вы изменяете номер продавца, требуется, чтобы были сохранены все его заказчики. Однако, если этот продавец покидает свою фирму или компанию, можно не захотеть удалить его заказчиков при удалении его самого из базы данных. Взамен вы захотите убедиться, что заказчики назначены кому-нибудь ещё. Чтобы сделать, это необходимо указать UPDATE с Каскадируемым эффектом и DELETE с Ограниченным эффектом.

```
CREATE TABLE Customers
(cnum integer NOT NULL PRIMARY KEY,
cname char (10) NOT NULL,
city char (10),
rating integer,
snum integer REFERENCES Salespeople,
UPDATE OF Salespeople CASCADES,
DELETE OF Salespeople RESTRICTED);
```

Если вы теперь попробуете удалить Peel из таблицы Продавцов, команда будет не допустима, пока вы не измените значение поля snum заказчиков Hoffman и Clemens для другого назначенного продавца.

С другой стороны, можно изменить значение поля snum для Peel на 1009, и Hoffman и Clemens будут также автоматически изменены.

Третий эффект – Пустые (NULL) изменения. Бывает, что когда продавцы оставляют компанию, их текущие приобретения не передаются другому продавцу. С другой стороны, требуется отменить все приобретения автоматически для заказчиков, чьи счета вы удалите. Изменив номера продавца или заказчика, можно просто передать их ему. Пример ниже показывает, как можно создать таблицу Приобретений с использованием этих эффектов.

```
CREATE TABLE Orders
(onum integer NOT NULL PRIMARY KEY,
amt decimal,
odate date NOT NULL
cnum integer NOT NULL REFERENCES Customers
snum integer REFERENCES Salespeople,
UPDATE OF Customers CASCADES,
DELETE OF Customers CASCADES,
UPDATE OF Salespeople CASCADES,
DELETE OF Salespeople NULLS);
```

Конечно, в команде DELETE с эффектом Пустого изменения в таблице Продавцов ограничение NOT NULL должно быть удалено из поля snum.

6.12. ПРЕДСТАВЛЕНИЕ (VIEW)

Типы таблиц, с которыми вы имели дело до сих пор, назывались базовыми таблицами. Это таблицы, которые содержат данные. Однако имеется другой вид таблиц – представления. Представления – это таблицы, чьё содержание выбирается или получается из других таблиц.

Они работают в запросах и операторах DML точно так же, как и основные таблицы, но не содержат никаких собственных данных.

Представления – подобны окнам, через которые вы просматриваете информацию (как она есть, или в другой форме), которая фактически хранится в базовой таблице. Представление – это запрос, который выполняется всякий раз, когда представление становится темой команды. Вывод запроса при этом в каждый момент становится содержанием представления.

Представление создаётся командой CREATE VIEW. Она состоит из слов CREATE VIEW (СОЗДАТЬ ПРЕДСТАВЛЕНИЕ), имени представления, которое нужно создать, слова AS (КАК), и далее запроса, как в следующем примере:

```
CREATE VIEW Londonstaff
AS SELECT *
FROM Salespeople
WHERE city = 'London';
```

Теперь вы имеете представление, называемое Londonstaff. Можно использовать это представление точно так же, как и любую другую таблицу. Она может быть запрошена, модифицирована, вставлена в, удалена из, и соединена с, другими таблицами и представлениями.

Давайте сделаем запрос такого представления:

```
SELECT *
FROM Londonstaff;
```

Преимущество использования представления по сравнению с основной таблицей в том, что представление будет модифицировано автоматически всякий раз, когда таблица, лежащая в его основе, изменяется. Содержание представления не фиксировано и переназначается каждый раз, когда вы ссылаетесь на представление в команде.

Представление может изменяться командами модификации DML, но модификация не будет воздействовать на само представление. Команды будут на самом деле перенаправлены к базовой таблице.

В нашем примере поля наших представлений имеют свои имена, полученые прямо из имён полей основной таблицы. Однако иногда вам нужно снабжать ваши столбцы новыми именами:

– когда некоторые столбцы являются выводимыми, и поэтому не имеющими имён;

– когда два или более столбцов в объединении имеют те же имена, что в их базовой таблице.

Имена, которые могут стать именами полей, даются в круглых скобках () после имени таблиц. Они не будут использованы, если сов-

падают с именами полей запрашиваемой таблицы. Тип данных и размер этих полей могут отличаться от запрашиваемых полей, которые «передаются» в них. Обычно вы не указываете новых имён полей, но если вы всё-таки сделали это, необходимо сделать это для каждого поля в представлении.

Групповые представления – это представления, которые содержат предложение GROUP BY, или которые основываются на других групповых представлениях.

Групповые представления могут стать превосходным способом обрабатывать полученную информацию непрерывно.

Предположим, что каждый день необходимо следить за номерами заказчиков, номерами продавцов, номерами приобретений, средним от приобретений и общей суммой приобретений. Чем конструировать каждый раз сложный запрос, можно создать следующее представление:

```
CREATE VIEW Totalforday
AS SELECT odate, COUNT (DISTINCT cnum), COUNT
(DISTINCT snum), COUNT (onum), AVG (amt), SUM (amt)
FROM Orders
GROUP BY odate;
```

Теперь вы сможете увидеть всю эту информацию с помощью простого запроса:

```
SELECT *
FROM Totalforday;
```

Представления доступны только для чтения.

Имеются также некоторые виды запросов, которые не допустимы в определениях представлений.

Одиночное представление должно основываться на одиночном запросе. ОБЪЕДИНЕНИЕ (UNION) и ОБЪЕДИНЕНИЕ ВСЕГО (UNION ALL) не разрешаются. УПОРЯДОЧЕНИЕ ПО (ORDER BY) никогда не используется в определении представлений.

Синтаксис удаления представления из базы данных подобен синтаксису удаления базовых таблиц:

```
DROP VIEW < view name >
```

Нет необходимости сначала удалять всё содержание, как это делается с базовой таблицей, потому что содержание представления сохраняется только в течение определённой команды.

Следует помнить, что необходимо являться владельцем представления, чтобы иметь возможность удалить его.

7. ЗАЩИТА ДАННЫХ В БАЗАХ ДАННЫХ

Защита данных – это организационные, программные и технические методы и средства, направленные на удовлетворение ограничений, установленных для типов данных или экземпляров типов данных в СОД.

Защита данных включает предупреждение случайного или несанкционированного доступа к данным, их изменения или разрушения со стороны пользователей или при сбоях аппаратуры. Реализация защиты включает:

- контроль достоверности данных с помощью ограничений целостности;
- обеспечение безопасности данных (физической целостности данных);
- обеспечение секретности данных.

7.1. ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ ДАННЫХ

Обеспечение целостности данных касается защиты от внесения непреднамеренных ошибок и предотвращения последних. Оно достигается за счёт проверки ограничений целостности – условий, которым должны удовлетворять значения данных.

Рассмотрим различные типы ограничений целостности в языке SQL:

1. Уникальность значения первичного ключа (PRIMARY KEY).
2. Уникальность ключевого поля или комбинации значений ключевых полей:

UNIQUE (A) ,

где A – один или несколько атрибутов, указанных через запятую. (1, 2 – явные структурные ограничения целостности.)

3. Обязательность/необязательность значения (NOT NULL/NULL).
4. Задание диапазона значений атрибута Field:

CHECK(field BETWEEN min_value AND max_value)

5. Задание взаимоотношений между значениями атрибутов Field1 и Field2:

CHECK (field1 @ field2), где @ – оператор отношения (например, знак «<>»).

6. Задание списка возможных значений (констант) для атрибута Field:

CHECK (field IN (valuel, value2,.., valueN)).

7. Определение формата атрибута Field (даты, числа и др.). Например:

CHECK (field LIKE ’ _____ - ____-____ ’) -- формат телефонного номера.

8. Определение домена атрибута на основе значений другого атрибута: множество значений некоторого атрибута отношения является подмножеством значений другого атрибута этого или другого отношения (внешний ключ, FOREIGN KEY). (3 – 8 – явные ограничения целостности на значения данных.)

9. Ограничения на обновление данных (например, каждое следующее значение атрибута должно быть больше предыдущего). В SQL напрямую не реализуются, требуют использования специальных возможностей СУБД (триггеров). Ограничения на параллельное выполнение операций (механизм транзакций) и проверка ограничений целостности после окончания внесения взаимосвязанных изменений.

Реализация ограничений целостности возлагается на СУБД или выполняется с помощью специальных программных модулей.

СУБД проводит проверку выполнения ограничений целостности для команд DDL до выполнения команды, а для команд DML либо сразу после выполнения команды, либо после выполнения всей транзакции. (По стандарту ISO этим можно управлять; по умолчанию проверка проводится после каждой операции DML.)

7.2. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ДАННЫХ

Под функцией безопасности (или физической защиты) данных подразумевается предотвращение разрушения или искажения данных в результате программного или аппаратного сбоя. Обеспечение безопасности является внутренней задачей СУБД, поскольку связано с её нормальным функционированием, и решается на уровне СУБД. Цель восстановления базы данных после сбоя – обеспечить, чтобы результаты всех подтверждённых транзакций были отражены в восстановленной БД, и вернуться к нормальному продолжению работы как можно быстрее, изолируя пользователей от проблем, вызванных сбоем.

Виды сбоев

В СУБД предусмотрены специальные механизмы, призванные нивелировать последствия сбоев в работе базы данных. Рассмотрим наиболее типичные сбои и способы защиты от них:

1. Сбой предложения.

Сбой происходит при логической ошибке предложения во время его обработки (например, предложение нарушает ограничение целостности таблицы). Когда возникает сбой предложения, СУБД автоматически откатывает результаты этого предложения, генерирует сообщение об ошибке и возвращает управление пользователю (приложению пользователя).

2. Сбой пользовательского процесса.

Это ошибка в процессе (приложении), работающем с БД, например аварийное разъединение или прекращение процесса. Сбившийся

процесс пользователя не может продолжать работу, тогда как СУБД и процессы других пользователей могут. Система автоматически откатывает неподтверждённые транзакции сбившегося пользовательского процесса и освобождает все ресурсы, занятые этим процессом.

3. Сбой процесса сервера.

Такой сбой вызван проблемой, препятствующей продолжению работы сервера. Это может быть аппаратная проблема, такая как отказ питания, или программная проблема, такая как сбой операционной системы. Восстановление после сбоя процесса сервера может потребовать перезагрузки БД, при этом автоматически происходит откат всех незавершённых транзакций.

4. Сбой носителя (диска).

Эта ошибка может возникнуть при попытке записи или чтения файла, необходимого для работы базы данных (файла БД, файла журнала транзакций и пр.). Типичным примером является отказ дисковой головки, который приводит к потере всех файлов на данном устройстве. В этой ситуации сервер БД не может продолжать работу, и для восстановления базы данных требуется участие человека (обычно администратора базы данных, АБД).

5. Ошибка пользователя.

Например, пользователь может случайно удалить нужные записи или таблицы. Ошибки пользователей могут потребовать участия человека (АБД) для восстановления базы данных в состояние на момент возникновения ошибки.

Таким образом, после некоторых сбоев система может восстановить БД автоматически, а ошибка пользователя или сбой диска требуют участия в восстановлении человека.

Средства физической защиты данных

В качестве средств физической защиты данных чаще всего применяются резервное копирование и журналы транзакций.

Резервное копирование означает периодическое сохранение файлов БД на внешнем запоминающем устройстве. Оно выполняется тогда, когда состояние файлов БД является непротиворечивым. Резервная копия (РК) не должна создаваться на том же диске, на котором находится сама БД, так как при аварии диска базу невозможно будет восстановить. В случае сбоя (или аварии диска) БД восстанавливается на основе последней копии.

Полная резервная копия включает всю базу данных (все файлы БД, в том числе вспомогательные, состав которых зависит от СУБД). Частичная резервная копия включает часть БД, определённую пользователем. Резервная копия может быть инкрементной: она состоит только из тех блоков (страниц памяти), которые изменились со времени последнего резервного копирования. Создание инкрементной копии

происходит быстрее, чем полной, но оно возможно только после создания полной резервной копии.

Создание частичной и инкрементной РК выполняется средствами СУБД, а создание полной РК – средствами СУБД или ОС (например, с помощью команды copy). В резервную копию, созданную средствами СУБД, обычно включаются только те блоки памяти, которые реально содержат данные (т.е. пустые блоки, выделенные под объекты БД, в резервную копию не входят).

Периодичность резервного копирования определяется администратором системы и зависит от многих факторов: объём БД, интенсивность запросов к БД, интенсивность обновления данных и др. Как правило, технология проведения резервного копирования такова:

- раз в неделю (день, месяц) осуществляется полное копирование;
- раз в день (час, неделю) – частичное или инкрементное копирование.

Все изменения, произведённые в данных после последнего резервного копирования, утрачиваются; но при наличии архива журнала транзакций их можно выполнить ещё раз, обеспечив полное восстановление БД на момент возникновения сбоя. Дело в том, что журнал транзакций содержит сведения только о текущих транзакциях. После завершения транзакции информация о ней может быть перезаписана. Для того чтобы в случае сбоя обеспечить возможность полного восстановления БД, необходимо вести архив журнала транзакций, т.е. сохранять копии файлов журнала транзакций вместе с резервной копией базы данных.

Восстановление базы данных

В том случае, если нельзя восстановить БД после сбоя автоматически, восстановление БД выполняется в два этапа:

1) перенос на рабочий диск резервной копии базы данных (или той её части, которая была повреждена);

2) перезапуск сервера БД с повторным проведением всех транзакций, зафиксированных после создания резервной копии и до момента возникновения сбоя.

Если в системе есть архив транзакций, то повторное проведение транзакций может проходить автоматически или под управлением пользователя.

Если произошёл сбой процесса сервера, то требуется перезагрузка сервера для восстановления БД. При перезагрузке СУБД может по содержимому системных файлов узнать, что произошёл сбой, и выполнить восстановление автоматически (если это возможно). Восстановление БД в этой ситуации означает приведение всех данных в БД в согласованное состояние, т.е. откат незавершённых транзакций и проверку того, что все изменения, внесённые завершёнными транзакциями, попали на диск.

Для оптимизации регистрации изменений некоторые СУБД могут записывать в журнал информацию о незавершённых транзакциях, предвидя их завершение. Более того, не дожидаясь подтверждения транзакции, СУБД переписывает на диск модифицированные блоки (при формировании контрольной точки). Поэтому в каждый момент времени в журнале транзакций и в БД может находиться небольшое число записей, модифицированных незавершёнными транзакциями. Эти записи помечаются соответствующим образом. С другой стороны, т.к. изменения сначала попадают в журнал транзакций и только потом в файл базы данных, в любой момент времени БД может не содержать блоков данных, модифицированных подтверждёнными транзакциями. Поэтому в результате сбоя могут возникнуть две потенциальные ситуации:

- Блоки, содержащие подтверждённые модификации, не были записаны в файлы данных, так что эти изменения отражены лишь в журнале транзакций. Следовательно, журнал транзакций содержит подтверждённые данные, которые должны быть переписаны в файлы данных.
- Журнал транзакций и блоки данных содержат изменения, которые не были подтверждены. Изменения, внесённые неподтверждёнными транзакциями, во время восстановления БД должны быть удалены из файлов данных.

Для того чтобы разрешить эти ситуации, СУБД автоматически выполняет два этапа при восстановлении после сбоев: прокрутку вперёд и прокрутку назад.

1. *Прокрутка вперёд* заключается в применении к файлам данных всех изменений, зарегистрированных в журнале транзакций. После прокрутки вперёд файлы данных содержат все как подтверждённые, так и неподтверждённые изменения, которые были зарегистрированы в журнале транзакций.

2. *Прокрутка назад* заключается в отмене всех изменений, которые не были подтверждены. Для этого используются журнал транзакций и сегменты отката, информация из которых позволяет определить и отменить те транзакции, которые не были подтверждены, хотя и попали на диск в файлы БД.

После выполнения этих этапов восстановления БД находится в согласованном состоянии и с ней можно работать.

7.3. ЗАЩИТА ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА

Под функцией секретности данных понимается защита данных от преднамеренного искажения и/или доступа пользователей или посторонних лиц. Для этого вся информация делится на общедоступные данные и конфиденциальные, доступ к которым разрешён только для отдельных групп лиц. Решение этого вопроса относится к компетен-

ции юридических органов или администрации предприятия, для которого создаётся БД, и является внешней функцией по отношению к БД.

Рассмотрим техническую сторону обеспечения защиты данных в БД от несанкционированного доступа. Общий принцип управления доступом к базе данных такой: СУБД не должна разрешать пользователю выполнение какой-либо операции над данными, если он не получил на это права. Санкционирование доступа к данным осуществляется администратором БД. В обязанности администратора БД входит:

- назначение отдельным группам пользователей прав доступа (привилегий) к отдельным группам данных в соответствии с правилами ПО;
- организация системы контроля доступа к данным;
- тестирование вновь создаваемых средств защиты данных;
- периодическое проведение проверок правильности работы системы защиты, исследование и предотвращение сбоев в её работе.

Примечание: администратор БД обычно назначает права доступа в соответствии с проектом БД, который должен включать перечень групп пользователей и их привилегии.

Для каждого пользователя система поддерживает паспорт пользователя, содержащий его идентификатор, имя процедуры подтверждения подлинности и перечень разрешённых операций. Подтверждение подлинности заключается в доказательстве того, что пользователь является именно тем человеком, за которого себя выдаёт. Чаще всего подтверждение подлинности выполняется путём парольной идентификации. Перечень операций обычно определяется той группой, к которой принадлежит пользователь. В реальных системах иногда предусматривается возможность очень ограниченного доступа к данным постороннего человека, не требующая идентификации (доступ типа «гость»).

Парольная идентификация заключается в присвоении каждому пользователю двух параметров: имени (login) и пароля (password). При входе в систему она запрашивает у пользователя его имя, а для подтверждения того, что это имя ввёл его владелец, система запрашивает пароль. Имя выдаётся пользователю при регистрации администратором, пароль пользователь устанавливает сам. При задании пароля желательно соблюдать следующие требования:

- длина пароля должна быть не менее шести символов;
- пароль должен содержать комбинацию букв и цифр или специальных знаков, пароль не может содержать пробелы;
- пароли должны часто меняться.

Для контроля выполнения этих требований обычно применяются специальные программы.

Управление доступом к данным осуществляется через СУБД, которая и обеспечивает защиту данных. Но такие данные вне СУБД становятся общедоступны. Если известен формат БД, можно осуществить

к ней доступ с помощью другой программы (СУБД), и никакие ограничения при этом не помешают. Для таких случаев предусмотрено кодирование данных. Используются различные методы кодирования: перекомпоновка символов в кортеже, замена одних символов (групп символов) другими символами (группами символов) и т.д. Кодирование может быть применено не ко всему кортежу, а только к ключевым полям. Декодирование производится непосредственно в процессе обработки, что, естественно, увеличивает время доступа к данным. Поэтому к кодированию прибегают только в случае высоких требований к конфиденциальности данных.

Предоставление прав доступа (привилегий) в системах, поддерживающих язык SQL, осуществляется с помощью двух команд:

1. GRANT – предоставление одной или нескольких привилегий пользователю (или группе пользователей):

GRANT {<список привилегий> | ALL PRIVILEGES}
ON <имя объекта>
TO {<список пользователей> | PUBLIC}
[WITH GRANT OPTION];

где <список привилегий> – набор прав, которые необходимо предоставить, или ALL PRIVILEGES – все права на данный объект; <имя объекта> – имя объекта БД, к которому предоставляется доступ; <список пользователей> – перечень пользователей (или *ролей*, см. дальше), которым будут предоставлены указанные права; PUBLIC – предопределённый пользователь, привилегии которого доступны всем пользователям БД; WITH GRANT OPTION – ключевые слова, дающие возможность пользователям из списка пользователей предоставлять назначенные права другим пользователям (т.е. передавать эти права).

Права, подразумеваемые под словами ALL PRIVILEGES, зависят от типа объекта. Примерный перечень прав в зависимости от типа объекта БД приведён в табл. 7.1.

2. REVOKE – отмена привилегий:

REVOKE [GRANT OPTION FOR]
{<список привилегий> | ALL PRIVILEGES}
ON <имя объекта>
FROM {<список пользователей> | PUBLIC}
{RESTRICT | CASCADE};

где [GRANT OPTION FOR] – отмена права передачи привилегий; CASCADE – при отмене привилегий у пользователя отменяются все привилегии, которые он передавал другим пользователям; RESTRICT – если при отмене привилегий у пользователя необходимо отменить переданные другим пользователям привилегии, то операция завершается с ошибкой.

Привилегия	Операции	Таблицы	Представления	Процедурные объекты
ALTER	Изменение определения объекта	+	+	+
DELETE	Удаление данных	+	+	
EXECUTE	Выполнение объекта			+
INSERT	Добавление данных	+	+	
SELECT	Чтение данных	+	+	
UPDATE	Изменение данных	+	+	

Примечание. Процедурные объекты – это хранимые процедуры и функции.

Другие ключевые слова имеют то же значение, что и в команде GRANT.

Для того чтобы упростить процесс управления доступом, многие СУБД предоставляют возможность объединять пользователей в группы или определять роли. Роль – это совокупность привилегий, предоставляемых пользователю и/или другим ролям. Такой подход позволяет предоставить конкретному пользователю определённую роль или отнести его к определённой группе пользователей, обладающей набором прав в соответствии с задачами, которые на неё возложены.

Кроме привилегий на доступ к объектам СУБД ещё может поддерживать так называемые *системные привилегии*: это права пользователя на создание/изменение/удаление (create/alter/drop) объектов различных типов. В некоторых системах такими привилегиями обладают только пользователи, включённые в группу АБД. Другие СУБД предоставляют возможность назначения дифференцированных системных привилегий любому пользователю в случае такой необходимости. Например, в СУБД Oracle права на создание таблиц и представлений пользователю manager можно предоставить с помощью той же команды GRANT, только без указания объекта:

GRANT create table, create view TO manager;

8. ЭЛЕМЕНТЫ ПРОЕКТИРОВАНИЯ БАЗ ДАННЫХ

Проектирование базы данных (БД) – одна из наиболее сложных и ответственных задач, связанных с созданием информационных систем (ИС).

В первую очередь ИС должна обеспечивать ведение БД: запись, чтение, модификацию данных, удаление неактуальных данных (возможно, в архив) и защиту данных. Взаимодействие конечных пользователей с БД обычно осуществляется с помощью интерфейсного приложения, входящего в состав ИС. Если пользователей ИС можно разделить на группы по характеру решаемых задач, то приложений может быть несколько (по количеству задач или групп пользователей).

В результате проектирования БД должны быть определены состав базы данных, эффективный для всех её будущих пользователей способ организации данных, и инструментальные средства управления данными.

8.1. ТРЕБОВАНИЯ К ПРОЕКТУ БАЗЫ ДАННЫХ

Основные требования, которым должен удовлетворять проект БД:

1. Корректность схемы БД.

База данных должна быть гомоморфным образом моделируемой предметной области, т.е. каждой сущности ПО должны соответствовать данные в памяти ЭВМ, а каждому процессу – адекватные процедуры обработки данных. Корректность подразумевает также логическую непротиворечивость базы данных, которая поддерживается автоматически с помощью средств СУБД.

2. Обеспечение ограничений на ресурсы вычислительной системы.

В первую очередь имеются в виду ограничения на объёмы внешней и оперативной памяти, которые потребуются для функционирования БД.

3. Эффективность функционирования.

База данных должна быть спроектирована таким образом, чтобы при её эксплуатации соблюдались ограничения на время реакции системы на запросы и модификацию данных.

4. Защита данных.

Проект БД должен включать описание защиты данных от несанкционированного доступа. Защита от сбоев является внутренней функцией СУБД, но требования к настройке механизмов защиты также выдвигаются на этапе проектирования БД, так как определяются предметной областью.

5. Гибкость.

Под этим подразумевается возможность развития и адаптации БД к изменениям предметной области и/или требований пользователей.

Конечно, нельзя предусмотреть все возможные варианты использования и изменения базы данных. Но в большинстве предметных областей основные сущности и их взаимосвязи относительно стабильны. Меняются только информационные требования, т.е. способы использования данных для решения задач.

6. Простота и удобство эксплуатации.

Под этим подразумевается соблюдение привычного для пользователя алгоритма работы с данными. От этого не в последнюю очередь зависит количество ошибок пользователя.

Удовлетворение первых четырёх требований обязательно для принятия проекта.

8.2. ЭТАПЫ ПРОЕКТИРОВАНИЯ БАЗЫ ДАННЫХ

В создании автоматизированной информационной системы, включающей базу данных, можно выделить три этапа:

I. Предпроектная подготовка.

II. Проектирование БД.

III. Реализация (создание БД и ППО).

Общая схема жизненного цикла приложения баз данных приведена на рис. 8.1. Как видно из этого рисунка, проектирование носит итерационный характер. Например, если на каком-либо этапе выясняется, что ранее сформулированные требования или решения не могут быть реализованы, то разработчики должны вернуться на более ранний этап и внести соответствующие изменения. Эти изменения, естественно, могут потребовать корректировки ранее выполненных этапов.

Рассмотрим основные задачи, которые решаются на каждом из этапов.

I.1. Проектирование начинается обычно с планирования, что позволяет:

- разбить задачу на небольшие, независимые, управляемые шаги;
- поставить краткосрочные и долговременные цели, которые служат для оценки фактических результатов проектирования и сравнения их с планом;
- определить временные зависимости между задачами, т.е. определить, какие задачи должны быть решены раньше других (составить сетевой план-график работ);
- выявить узкие места, т.е. ресурсы, от которых план зависит сильнее всего;
- спрогнозировать потребности в кадрах для проекта.

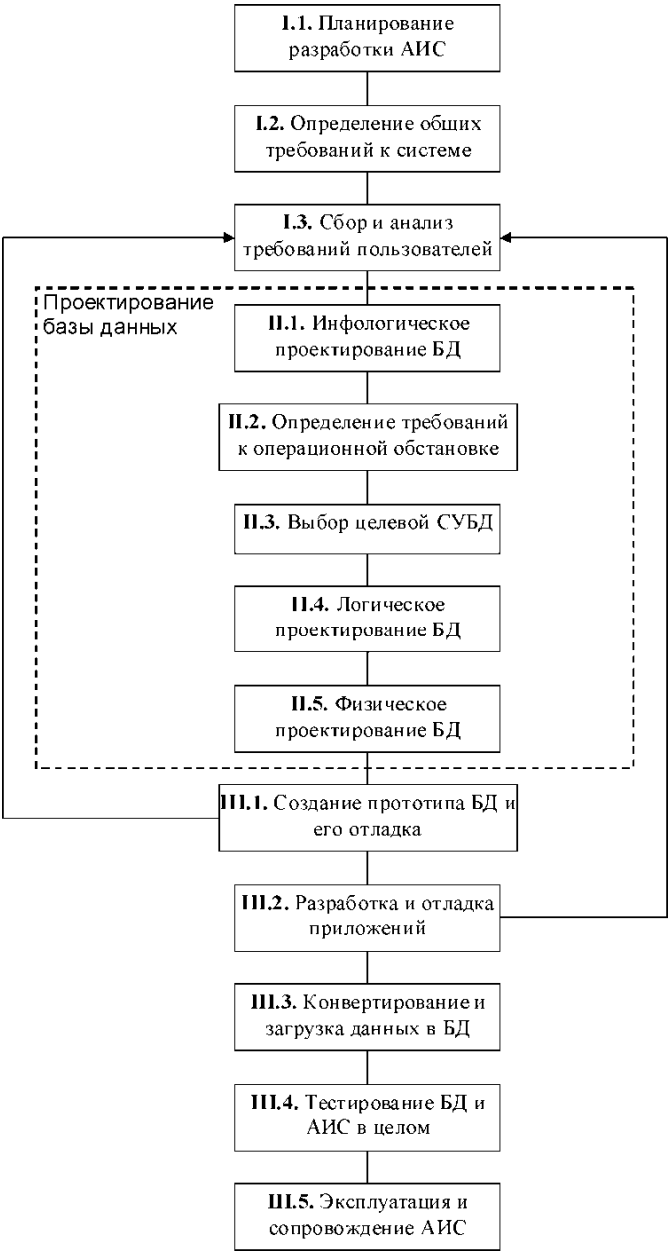


Рис. 8.1. Схема жизненного цикла приложения баз данных

Работа по созданию БД начинается с подбора кадров. Требуется определить, какие специалисты необходимы для выполнения этой работы. В общем случае это должны быть следующие категории:

- Аналитики. Это специалисты исследуемой предметной области, которые в идеале должны быть знакомы с основами создания баз данных. В их задачу входит постановка задачи проектирования: анализ ПО, выявление бизнес- процессов и бизнес-правил, определение требований к БД.
- Пользователи – те работники, для которых создаётся АИС. Именно они обладают знаниями о технологии работы с информацией.
- Проектировщики. Это сотрудники, которые будут заниматься собственно разработкой проекта БД.
- Администраторы. В том случае, если система небольшая, администратор БД может быть один. Если же система большая и территориально распределённая, то помимо АБД потребуется ещё администратор системы, и, возможно, не один. АБД должен появиться не тогда, когда система уже спроектирована, а на этапе проектирования БД. Это необходимо хотя бы потому, что при проектировании для отладки и тестирования обязательно создаётся рабочий прототип БД, и желательно, чтобы за общее обеспечение функционирования этого прототипа отвечал отдельный специалист.
- Разработчики программного обеспечения. Любая БД требует помимо СУБД создания некоторого прикладного программного обеспечения (ППО). Если сложность этого ППО невелика, то обычно его созданием занимаются сами проектировщики. В противном случае необходимо набрать программистов (или выделить из имеющихся), которые будут этим заниматься.

При определении потребности в кадрах может возникнуть ситуация, когда уже на этом этапе станет очевидна невозможность подбора нужного количества специалистов определённого профиля и квалификации. Тогда это может привести к пересмотру объёма задач, стоящих перед базой данных.

I.2. Определение общих требований к системе подразумевает:

1. Предварительный анализ ПО.

Включает в себя сбор документов, характеризующих ПО, укрупнённое описание ПО (не детализированное) и общую постановку задачи.

В процессе анализа и проектирования желательно ранжировать планируемые функции системы по степени важности. Один из возможных вариантов классификации – MoSCoW-анализ (терминология Клегга и Баркера):

Must have – необходимые функции;

Should have – желательные функции;

Could have – возможные функции;

Won't have – отсутствующие функции.

Необходимые функции обеспечивают возможности, которые являются критическими для успешной работы системы. Реализация *желательных* и *возможных* функций системы ограничена временными и/или финансовыми рамками. *Отсутствующие* функции – это те функции, которые реально существуют, но не будут реализованы в этом проекте по различным причинам.

Примечание: если применяются средства автоматизации проектирования (CASE-средства), то задачу последовательного внесения изменений берёт на себя это CASE-средство.

2. Рассмотрение и принятие результатов анализа.

Эта задача обычно решается итеративно во взаимодействии проектировщиков и заказчиков (или аналитиков). На этом этапе важно определить, что проектировщики правильно понимают описание предметной области и задачи, поставленные перед ними аналитиками. Для этого обычно проводятся совместные семинары, на которых проверяется адекватность модели и ПО.

3. Определение критических факторов успеха.

В данном случае под термином *критические факторы* подразумеваются как «жизненно важные для приёмки и успешной реализации проекта», так и «критические с точки зрения функционирования системы». Очевидно, что если не учесть хотя бы один из таких факторов, то существование и успешное функционирование проекта будет поставлено под вопрос.

4. Оценка системных ограничений.

В качестве часто встречающихся ограничений можно отметить следующие:

- финансовые;
- временные;
- технические (например, использование определённой аппаратуры);
- программные (например, выбор определённого программного обеспечения);
- ограничения, определяемые наличием существующих систем, с которыми необходимо обеспечить совместимость.

5. Определение целевой архитектуры.

Под целевой архитектурой в данном случае понимается архитектура с точки зрения СУБД (однозадачная или многозадачная архитектура клиент-сервер, параллельный сервер). Выбор архитектуры по-

влияет в дальнейшем на перечень требуемых аппаратных и программных средств.

6. Определение требований к производительности.

Необходимо примерно оценить количество транзакций в единицу времени и объём обрабатываемых этими транзакциями данных. Требования к производительности зависят от режима, в котором будет функционировать система:

1) *Интерактивный режим*. Для этого режима устанавливается время, в течение которого пользователь должен получить ответ на свой запрос. Обычно время реакции системы не должно превышать нескольких секунд.

2) *Пакетный режим*. Здесь требования к производительности обычно не такие жёсткие, как для интерактивного режима, и выражаются в минутах или часах, требующихся на получение конечного результата вычислений.

3) *Режим реального времени*. Этот режим является самым сложно реализуемым. В настоящее время существует только одна СУБД, которая в полной мере отвечает требованиям режима реального времени: СУБД ЛИНТЕР – единственная СУБД отечественного производства (компания РЕЛЭКС, г. Воронеж).

7. Согласование стандартов проектирования, в частности:

- правил именования объектов;
- стандарта проектной документации;
- правил введения общих типов и т.п.

8. Выбор программных средств для проектирования и реализации системы (имеются в виду вспомогательные средства типа CASE и др.).

I.3. Определение требований пользователей. Учёт этих требований особенно важен тогда, когда проект БД создаётся в развитие существующей информационной системы, так как в этом случае есть определённый опыт работы (традиции, привычки и вместе с тем пожелания). Всё это желательно учитывать для того, чтобы обеспечить преемственность и не вызвать негативного отношения к системе, например, из-за непривычного интерфейса.

Собственно процесс проектирования БД включает в себя следующие основные этапы:

II.1. Информационно-логическое (инфологическое) проектирование.

II.2. Определение требований к операционной обстановке, в которой будет функционировать информационная система.

II.3. Выбор СУБД и других инструментальных программных средств.

II.4. Логическое проектирование БД. (Иногда этот этап называется даталогическим проектированием)

II.5. Физическое проектирование БД.

Эти этапы подробно рассмотрены в следующем разделе.

После того, как проект базы данных создан, наступает этап реализации проекта. Он разбивается на следующие шаги:

III.1. Создание прототипа БД и его отладка. Отладка подразумевает проверку правильности функционирования процедурных объектов БД (триггеры, процедуры, функции). Прототип позволяет определить жизнеспособность проекта БД и выявить его недостатки, что может потребовать внесения изменений в проект. Прототип также нужен как база для разработчиков приложений. Для этого БД наполняется реальными или тестовыми данными.

III.2. Разработка и отладка приложений. Выполняется разработчиками программного обеспечения на основе функциональных требований, которые были выявлены на этапах I.2, I.3, и спецификации БД (схемы БД).

III.3. Конвертирование и загрузка данных в БД. Этот этап выполняется в том случае, если данные в БД загружаются из ранее существовавшей системы.

III.4. Тестирование работы базы данных и АИС в целом. Различают такие виды тестов, как:

- *автономные* – тесты отдельных модулей;
- *тесты связей* – тесты между модулями;
- *регрессивные* – тесты на проверку уже протестированных модулей в связи с подключением новых модулей (функций), которые могут нарушить работу ранее созданных модулей;
- *нагрузочные* – тесты на проверку времени реакции системы в рабочем режиме или определение производительности системы;
- *системные* – тесты на проверку функционирования системы в целом;
- *приёмо-сдаточные* – тесты, которые проводятся при сдаче системы (АИС) в эксплуатацию.

Этап III.4 включает нагрузочные, системные и приёмо-сдаточные тесты.

III.5. Эксплуатация и сопровождение АИС. Здесь можно выделить ряд задач:

- В процессе эксплуатации АИС может возникнуть необходимость внесения изменений в систему. Это может быть вызвано изме-

нениями предметной области, появлением новых задач или выявлением существенных недостатков в АИС. Нельзя забывать о том, что все вносимые изменения должны быть документированы.

- Необходимо выполнять резервное копирование данных, чтобы предотвратить их потерю в случае серьёзного сбоя или ошибки пользователя.
- Сопровождение АИС обычно включает периодические проверки выполнения системных ограничений (на объём данных и время реакции системы). В результате этих проверок удаляются устаревшие данные (если не предусмотрено автоматическое архивирование данных). Улучшение показателей производительности системы может быть достигнуто за счёт настройки СУБД, которая выполняется администратором базы данных. Теперь перейдём к более подробному обсуждению этапов проектирования БД.

8.3. ИНФОЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ

Инфологический подход не содержит формальных способов моделирования реальности, но он закладывает основы методологии проектирования БД.

Первой задачей инфологического проектирования является определение *предметной области* (ПО) системы, позволяющее изучить информационные потребности будущих пользователей. Другая задача этого этапа – анализ ПО, который призван сформировать взгляд на неё с позиций сообщества будущих пользователей БД, т.е. *инфологической модели* ПО.

Анализ ПО выполняется проектировщиком БД с помощью специалистов в данной ПО. В основе анализа лежат документы, используемые в работе предприятия (организации), и технология работы с данными.

Инфологическая модель ПО включает описание структуры и динамики ПО, характера информационных потребностей пользователей системы. Описание выполняется в терминах, понятных пользователю и независимых от реализации системы. Обратите внимание: инфологическая модель ПО не должна зависеть от модели данных, которая будет использована при создании БД.

Обычно описание ПО выражается в терминах не отдельных сущностей и связей между ними, а их типов, связанных с ними ограничений целостности и тех процессов, которые приводят к переходу ПО из одного состояния в другое. Такое описание может быть представлено любым способом, допускающим однозначную интерпретацию.

В простых случаях описание ПО представляется на естественном языке. В более сложных случаях используется также математический аппарат: таблицы, диаграммы, графы и т.п. Если анализ ПО выполняется несколькими специалистами, то они должны принять соглашения, которые касаются:

– используемых методов анализа предметной области;

– правил именования и обозначения сущностей ПО, атрибутов и связей;

– содержания и формата создаваемых ими документов.

Этап инфологического проектирования начинается с моделирования ПО. Проектировщик разбивает ПО на ряд локальных областей (локальных представлений), каждая из которых (в идеале) включает в себя информацию, достаточную для обеспечения информационных потребностей одной группы будущих пользователей или решения отдельной задачи. Каждое локальное представление моделируется отдельно, а затем выполняется их объединение.

Выбор локального представления зависит от масштабов ПО. Обычно ПО разбивается на локальные области так, чтобы каждая из них соответствовала отдельному внешнему приложению и содержала 6–7 сущностей (т.е. объектов, о которых в системе будет накапливаться информация). Таким образом, если ПО небольшая, то разбиение на локальные представления не требуется и моделирование выполняется для ПО в целом.

Существуют разные подходы к инфологическому проектированию. Рассмотрим основные из них.

1. Функциональный подход к проектированию БД.

Этот метод реализует принцип «от задач» и применяется в том случае, когда известны функции некоторой группы лиц и/или комплекса задач, для обслуживания информационных потребностей которых создаётся рассматриваемая БД.

2. Предметный подход к проектированию БД.

Предметный подход применяется в тех случаях, когда у разработчиков есть чёткое представление о самой ПО и о том, какую именно информацию они хотели бы хранить в БД, а структура запросов не определена или определена не полностью. Тогда основное внимание уделяется исследованию ПО и наиболее адекватному её отображению в БД с учётом самого широкого спектра информационных запросов к ней.

3. Проектирование с использованием метода «сущность-связь».

ER-метод является наиболее распространённым методом проектирования БД.

В предметной области необходимо выделить *сущности, атрибуты* и *связи.* Сущности, существование которых не зависит от существования других сущностей, называются *базовыми*, остальные сущности – *зависимыми*. Например, сущность *ЛЕКЦИЯ* зависит от базовых сущностей *ГРУППА*, *ПРЕПОДАВАТЕЛЬ*, *ДИСЦИПЛИНА.*

Для каждой сущности определяются атрибуты (свойства), которые можно условно классифицировать следующим образом:

1) *Идентифицирующие* и *описательные атрибуты.* Идентифицирующие атрибуты имеют уникальное значение для сущностей данного типа и являются *потенциальными ключами.* Они позволяют однозначно распознавать экземпляры сущности. Из потенциальных ключей выбирается один первичный ключ (ПК). В качестве ПК обычно выбирается потенциальный ключ, по которому чаще происходит обращение к экземплярам сущности. Кроме того, ПК должен включать в свой состав минимально необходимое для идентификации количество атрибутов. Остальные атрибуты называются описательными и заключают в себе интересующие свойства сущности.

2) *Составные* и *простые атрибуты.* Простой атрибут состоит из одного компонента, его значение неделимо. Составной атрибут является комбинацией нескольких компонентов, возможно, принадлежащих разным типам данных (например, ФИО или адрес). Решение о том, использовать составной атрибут или разбивать его на компоненты, зависит от характера его обработки и формата пользовательского представления этого атрибута.

3) *Однозначные* и *многозначные атрибуты* (могут иметь соответственно одно или много значений для каждого экземпляра сущности).

4) *Основные* и *производные атрибуты.* Значение основного атрибута не зависит от других атрибутов. Значение производного атрибута вычисляется на основе значений других атрибутов (например, возраст человека вычисляется на основе даты его рождения и текущей даты).

Спецификация атрибута состоит из его названия, типа данных, размера и описания ограничений целостности – множества значений, которые может принимать данный атрибут.

Далее осуществляется спецификация связей. Под связью понимается осмысленная ассоциация между сущностями, например, *СТУДЕНТ учится* в *ГРУППЕ, ВОДИТЕЛЬ выполняет РЕЙС* и т.п. Выявляются все связи между сущностями внутри локального представления. Каждая связь именуется, для неё определяются степень, кардинальность и обязательность.

Кроме спецификации связей типа «сущность-сущность», выполняется спецификация связей типа «сущность-атрибут» и «атрибут-

атрибут» внутри одной сущности. Для этого надо определить зависимости между экземплярами сущностей и атрибутами, а также между атрибутами, относящимися к одному экземпляру сущности. Например, атрибут *Телефоны* сущности *СОТРУДНИК* может быть многозначным и необязательным, т.е. связь *СОТРУДНИК -> Телефоны* имеет тип 1:n и является необязательной для сотрудника. А если рассмотреть атрибуты *Маршрут* и *Стоимость* сущности *БИЛЕТ*, то между ними есть связь 1:1, так как стоимость билета зависит от маршрута (пункт отправления – пункт назначения).

После выявления сущностей и связей ПО строят ER-диаграмму, которая является наглядным отображением модели ПО.

При небольшом количестве локальных областей (не более пяти) объединение локальных представлений выполняется за один шаг. В противном случае обычно выполняют бинарное объединение. При этом проектировщик может формировать конструкции, производные по отношению к тем, которые были использованы в локальных представлениях. Цель введения подобных абстракций:

– объединение в единое целое фрагментарных представлений о различных свойствах одной и той же сущности;

– введение абстрактных понятий, удобных для решения задач системы, установление их связи с более конкретными понятиями модели;

– образование классов и подклассов подобных сущностей (например, класс «изделие» и подклассы типов изделий, производимых на предприятии).

При объединении локальных представлений используют три основополагающие концепции:

1. Идентичность. Два или более элементов модели идентичны, если они имеют одинаковое семантическое значение. Например, *СОТРУДНИК* для отдела кадров и *СОТРУДНИК* для отдела закупок – это один и тот же тип сущности, возможно, с разным набором атрибутов. (Наборы атрибутов исходных сущностей при этом объединяются.)

2. Агрегация. Позволяет рассматривать связь между элементами как новый элемент. Например, связь *экзаменовать* между сущностями *ДИСЦИПЛИНА*, *ПРЕПОДАВАТЕЛЬ*, *СТУДЕНТ* может быть представлена агрегированной сущностью *ЭКЗАМЕН* с атрибутами *Название дисциплины*, *Фамилия преподавателя*, *Фамилия студента*, *Оценка.*

3. Обобщение. Позволяет образовывать многоуровневую иерархию обобщений. Например, в объединяемых представлениях присутствуют следующие сущности:

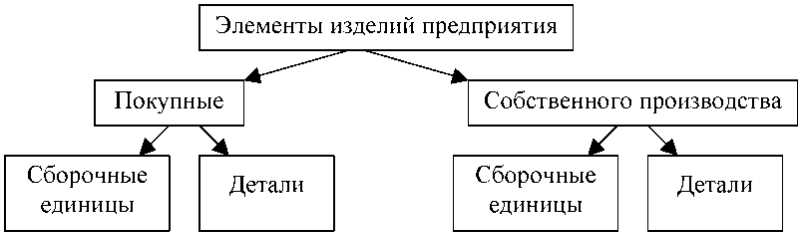


Рис. 8.2. Использование обобщений при объединении

ДЕТАЛИ СОБСТВЕННОГО ПРОИЗВОДСТВА, *ДЕТАЛИ ПОКУПНЫЕ СБОРОЧНЫЕ*, *ЕДИНИЦЫ ПОКУПНЫЕ СБОРОЧНЫЕ*, *ЕДИНИЦЫ СОБСТВЕННОГО ПРОИЗВОДСТВА*. Их можно объединить так, как показано на рис. 8.2.

Это позволит упростить формализацию процессов обработки данных. Например, оформление заказа на покупные элементы изделий в данном примере может быть описано один раз (для второго уровня иерархии).

На этапе объединения необходимо выявить и устранить все противоречия. Например, изменить одинаковые названия семантически различных сущностей или связей или несогласованные ограничения целостности на одни и те же атрибуты в разных приложениях. Устранение противоречий вызывает необходимость возврата к этапу моделирования локальных представлений с целью внесения в них соответствующих изменений.

По завершении объединения результаты проектирования представляют собой концептуальную инфологическую модель ПО. Она фиксируется в виде общей ER-диаграммы предметной области. Модели локальных представлений – это внешние инфологические модели (внешние схемы).

На этапе анализа ПО также решаются следующие задачи:

1) Определение правил (ограничений целостности), которым должны удовлетворять сущности ПО, атрибуты сущностей и связи между ними. Часть этих правил реализуется в схеме базы данных. Возможности реализации ограничений целостности в схеме БД определяются моделью данных той СУБД, которая будет выбрана для реализации проекта. Остальные правила реализуются с помощью программного обеспечения.

2) Выделение групп пользователей системы. Каждая группа выполняет определённые задачи и обладает разными правами доступа к системе.

3) Создание внешней спецификации тех функций (процессов), которые эта система будет выполнять. Например, для той же библиотечной системы это задачи поиска книг (по определённым критериям), выдачи/приёма книг, определение списка должников и т.д. Эта спецификация является основой для разработки приложений и выдаётся программистам в качестве задания.

8.4. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К ОПЕРАЦИОННОЙ ОБСТАНОВКЕ

На этом этапе производится оценка требований к вычислительным ресурсам, необходимым для функционирования системы, выбор типа и конфигурации ЭВМ, типа и версии операционной системы (ОС).

Выбор зависит от таких показателей, как:

- примерный объём данных в БД;
- динамика роста объёма данных;
- характер запросов к данным (извлечение и обновление отдельных записей, обработка групп записей, обработка отдельных отношений или соединение отношений);
- интенсивность запросов к данным по типам запросов;
- требования ко времени отклика системы по типам запросов;
- режим работы (интерактивный, пакетный или режим реального времени).

Эта информация позволяет определить системные требования к объёму оперативной и дисковой памяти, а также функциональным возможностям ОС.

8.5. ВЫБОР СУБД И ИНСТРУМЕНТАЛЬНЫХ ПРОГРАММНЫХ СРЕДСТВ

Выбор СУБД является одним из важнейших моментов в разработке проекта БД, так как он принципиальным образом влияет на процесс проектирования БД и реализации информационной системы.

Теоретически при осуществлении этого выбора нужно принимать во внимание десятки факторов. Но на практике разработчики руководствуются лишь собственной интуицией и несколькими наиболее важными критериями, к которым относятся:

– тип модели данных, которую поддерживает данная СУБД, адекватность модели данных структуре рассматриваемой ПО;

– характеристики производительности СУБД;

– запас функциональных возможностей для дальнейшего развития информационной системы;

– степень оснащённости СУБД инструментарием для персонала администрирования данными;

– удобство и надёжность СУБД в эксплуатации;

– наличие специалистов по работе с конкретной СУБД;

– стоимость СУБД и дополнительного программного обеспечения.

По результатам предыдущего этапа определены основные характеристики БД, такие как объём памяти и необходимая производительность. В зависимости от этого выбираются 2–3 СУБД, которые соответствуют выявленным требованиям. Например, если объём БД не превысит 100M, большинство запросов выбирает от 1 до 20 записей и время реакции системы не должно превышать 10 секунд, то следует остановить выбор на системах среднего класса, таких как Firebird, PostgreSQL, FoxPro. Для меньших по объёму БД можно выбрать Access или MySQL, а такие серьёзные СУБД, как Oracle, DB/2 или Informix следует рассматривать в тех случаях, когда велик объём данных или имеются высокие требования к производительности системы.

Выбранные СУБД оцениваются по степени соответствия выявленным требованиям к БД, и выбирается та система, которая лучше им соответствует.

8.6. ЛОГИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БД

На этапе логического проектирования инфологическая модель ПО, представленная в виде ER-диаграммы, преобразуется в логическую (концептуальную) схему БД. Решение этой задачи существенно зависит от модели данных, поддерживаемой выбранной СУБД.

Результатом выполнения этапа логического проектирования являются схемы БД концептуального и внешнего уровней архитектуры, составленные на языке определения данных (DDL, Data Definition Language) выбранной СУБД.

8.7. ФИЗИЧЕСКОЕ ПРОЕКТИРОВАНИЕ БД

Основой для физического проектирования является схема БД, полученная на предыдущем этапе. Физическое проектирование заключается в увязке логической структуры БД и физической среды

хранения в целях наиболее эффективного размещения данных. Решается вопрос размещения хранимых данных в пространстве памяти и выбора эффективных методов доступа к различным компонентам «физической» БД. Результаты этого этапа документируются в форме схемы хранения на языке определения данных. Принятые на этом этапе решения оказывают определяющее влияние на производительность системы.

Для реляционной БД на этом этапе определяются параметры распределения памяти для объектов БД, строятся индексы, определяется целесообразность использования хеширования и кластеризации.

Фактически проектирование БД имеет итерационный характер. В процессе функционирования системы становится возможным измерение её реальных характеристик, выявление «узких» мест. И если система не отвечает предъявляемым к ней требованиям, то обычно она подвергается реорганизации, т.е. модификации первоначально созданного проекта.

8.8. АВТОМАТИЗАЦИЯ ПРОЕКТИРОВАНИЯ БД

Функциональное ядро систем автоматизированного проектирования (САПР) БД строится как совокупность взаимосвязанных модулей инфологического моделирования, проектирования схем и физической организации БД.

Существующие в настоящее время САПР БД строятся как человекомашинные экспертные системы. В первую очередь это определяется слабо поддающимся формализации процессом синтеза инфологического описания ПО, т.е. преобразования неформальных представлений реального мира в формальные категории. Этот процесс выполняется экспертом – специалистом в той или иной ПО. Поэтому все проблемы, которые характерны для формирования базы знаний экспертной системы, возникают и в случае САПР БД.

Характерной особенностью САПР БД является её ориентация на коллективное творчество и продолжительность самого процесса проектирования, предполагающего множество итераций. Это находит своё отражение в наличии журнала проектирования и других средств, обеспечивающих ведение и коллективное использование исходных данных, промежуточных и окончательных результатов проектирования. Общая структура САПР БД приведена на рис. 8.3.

Рис. 8.3. Общая структура САПР БД

В настоящее время создан ряд САПР БД, которые называются CASE-средствами. В качестве примеров таких систем можно привести ERWin, BPWin, Designer (Oracle) и др.

8.9. ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ РЕЛЯЦИОННЫХ БД

Проектирование реляционной базы данных проходит в том же порядке, что и проектирование БД других моделей данных, но имеет свои особенности, которые в первую очередь касаются этапа логического проектирования.

На этапе логического проектирования реляционной базы данных также необходимо решить следующие задачи:

1. Преобразовать ER-диаграмму в схему БД.
2. Выявить нереализуемые и необычные конструкции данных.
3. Определить все первичные ключи (ПК).
4. Определить типы данных для полей таблиц.

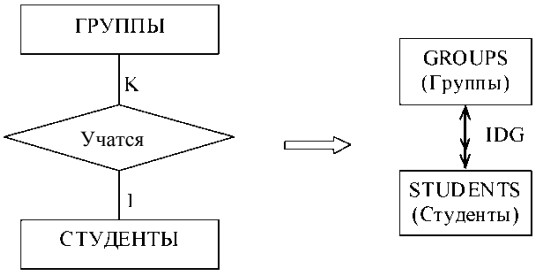


Рис. 8.4. Преобразование бинарной связи 1:n между сущностями разных типов

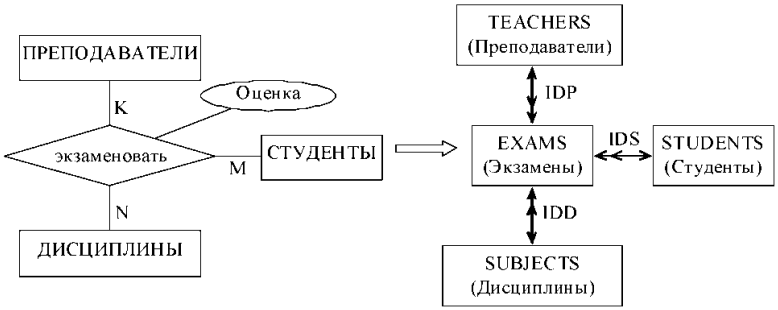


Рис. 8.5. Преобразование связи с атрибутами

Примечание: внешний ключ на схеме отражается двунаправленной стрелкой.

5. Каждая связь со степенью больше двух и связь, имеющая атрибуты, преобразуется в таблицу БД (рис. 8.5).

6. Описать все ограничения целостности.

Правила преобразование ER-диаграммы в схему БД следующие:

1. Каждый тип сущности преобразуется в таблицу БД. В таблицу вносятся все атрибуты, относящиеся к данному типу сущности.

2. Бинарная связь 1:n (между сущностями разных типов) реализуется с помощью внешнего ключа между двумя таблицами (рис. 8.4). Например, *ОТДЕЛЫ* и *СОТРУДНИКИ*, *ГРУППЫ* и *СТУДЕНТЫ* и т.п. *Номер группы* в таблице *ГРУППЫ* является первичным ключом, а *Номер группы* в таблице *СТУДЕНТЫ* – внешним ключом. Это самый часто встречающийся вид связи.

3. Связь 1:1 реализуется в рамках одной таблицы. Исключение из этого правила составляют ситуации, когда связанные сущности существуют независимо друг от друга. Например, связь между сущностями *ВОДИТЕЛИ* и *ТРАНПОРТНЫЕ СРЕДСТВА* при условии, что за каждым транспортным средством закреплён один водитель. Эта схема будет включать две таблицы, а связь между ними можно реализовать с помощью уникального (возможно, необязательного) внешнего ключа в той таблице, которая будет считаться подчинённой.

4. Унарная связь 1:n (между сущностями одного типа) реализуется с помощью внешнего ключа, определённого в той же таблице, что и первичный ключ. Например, для отражения в таблице *СОТРУДНИКИ* связи *руководить* нужно добавить в неё поле *Руководитель.* Это поле будет внешним ключом, ссылающимся на первичный ключ этой же таблицы. Такой ключ позволяет отразить иерархию сотрудников, когда у каждого сотрудника может быть только один непосредственный руководитель, а у директора поле *Руководитель* будет неопределённым (null).

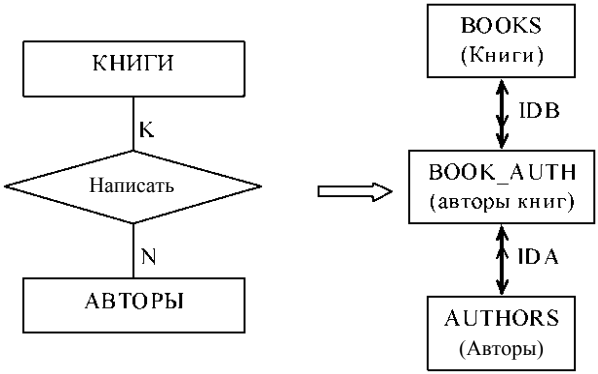


Рис. 8.6. Преобразование бинарной связи 1:n между сущностями разных типов

5. Бинарная связь типа n:m реализуется с помощью промежуточной таблицы. Например, для сущностей *КНИГИ* и *АВТОРЫ* и связи *написать* промежуточная таблица будет содержать два внешних ключа: идентификатор книги и идентификатор автора, написавшего эту книгу (рис. 8.6). В эту промежуточную таблицу также вносятся те атрибуты, которые характеризуют эту связь (например, номер автора в списке авторов этой книги).

6. Унарная связь n:m реализуется с помощью промежуточной таблицы. Например, для отражения связи *ассоциируется* между терминами таблицы *КЛЮЧЕВЫЕ СЛОВА* нужно добавить таблицу *АССОЦИАЦИИ*, в которой будут два внешних ключа на таблицу *КЛЮЧЕВЫЕ СЛОВА* (рис. 8.7).

7. К нереализуемым относятся связи кардинальностью 1:n или n:m, обязательные в обе стороны. Например, связь *заказы-строки заказов:* заказ не может быть пустым, и заказанный товар должен входить в определённый заказ. То есть нельзя добавить заказ, пока в нём

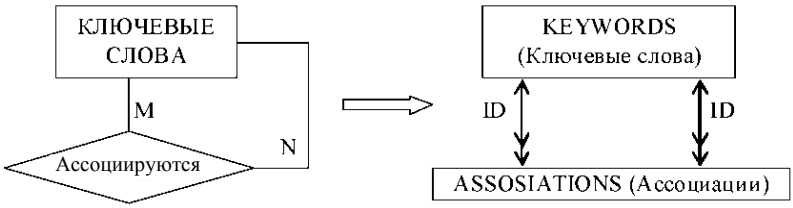


Рис. 8.7. Преобразование унарной связи кардинальности n:m

нет ни одной строки, и нельзя добавить строку в несуществующий заказ. Эта проблема обычно решается так: связь делается необязательной со стороны первичного ключа, а внешний ключ остаётся обязательным. При этом в приложении необходимо предусмотреть правило обработки пустых заказов (например, их удаление).

К необычным конструкциям данных можно отнести так называемые *взаимоисключающие связи*, когда подчинённая сущность связана с одной из двух родительских сущностей. Например, счёт в банке может принадлежать либо физическому лицу, либо юридическому, и не может принадлежать и тому, и другому либо не принадлежать никому. Такую связь можно реализовать по-разному, например, введением в таблицу счетов двух внешних ключей (номер_физ_лица и номер_юр_лица) и следующего ограничения целостности: (*номерфизлица IS NULL AND номерюрлица IS NOT NULL*) *OR* (*номерфизлица IS NOT NULL AND номерюрлица IS NULL*).

Определение первичных ключей

В принципе можно создать таблицу и без первичного ключа. Но наличие у каждой таблицы первичного ключа – хороший стиль проектирования БД. Кроме того, если эта таблица является родительской для какой-либо другой таблицы, то определить первичный или уникальный ключ необходимо, чтобы можно было определить внешний ключ в подчинённой таблице. В качестве ПК следует брать тот уникальный атрибут сущности, по которому чаще всего происходит обращение к данным. Например, для БД налоговой инспекции это ИНН – индивидуальный номер налогоплательщика, а для БД ФОМС (фонда обязательного медицинского страхования) – номер медицинского полиса.

Если у сущности нет уникальных атрибутов, можно рассмотреть уникальные комбинации атрибутов. Но первичный ключ не должен быть длинным, так как ссылающийся на него внешний ключ будет занимать много памяти. Поэтому при отсутствии подходящих атрибутов нужно вводить суррогатный первичный ключ, который не несёт смысловой нагрузки и служит только для идентификации записей. (Некоторые СУБД позволяют определять значения такого ключа, как AUTOINCREMENT, т.е. числовое поле, значение которого начинается с 1 и автоматически увеличивается на 1 при добавлении новой записи)

Определение типов данных атрибутов

Определение типов данных для полей таблиц зависит от требований ПО. Но можно дать следующие общие рекомендации по выбору типов данных:

1. Для коротких символьных значений и символьных строк фиксированной длины следует выбирать тип CHAR. Например, для поля

«единица измерения» со значениями 'кг', 'шт.', 'уп.' (char (3)), для поля «пол» (char (1)) и т.п.

2. Для символьных строк переменной длины нужно выбирать тип VARCHAR с указанием максимально возможной длины хранимого значения. Если при добавлении данных длина строки превысит указанное ограничение, система не сможет добавить данные и вернёт сообщение об ошибке.

3. Для числовых атрибутов, не участвующих в сложных расчётах, нужно использовать основной числовой тип реляционных СУБД – тип NUMBER, указывая реально необходимое количество разрядов. Например, для атрибута *Номер сотрудника* это может быть NUMBER (4) (до 10 000 человек), а для зарплаты – NUMBER (8, 2) (до 999 999.99 рублей).

4. Для числовых атрибутов, которые участвуют в сложных расчётах, следует использовать такие числовые типы, которые хранят данные в машинном (двоичном) представлении. Это ускорит выполнение расчётов.

5. Для числовых атрибутов, имеющих ведущие нули, следует выбирать тип CHAR, а не числовой тип, иначе ведущие нули будут потеряны. Например, для серии и номера паспорта (char (10)).

6. Для хранения дат нужно выбирать тип DATE или его варианты (DATETIME, например). Это позволит использовать арифметику дат и не заботиться о правильности вводимых данных: СУБД сама проверит допустимость даты.

7. Для хранения больших объектов (графических, звуковых и т.п.) следует выбирать специальные типы данных, перечень которых зависит от выбранной СУБД. Это могут быть типы LONG, CLOB (character large object), BLOB (binary large object) и др.

8. Для семантически одинаковых полей разных таблиц нужно выбирать одинаковые типы данных. Например, ФИО сотрудника и ФИО клиента. Во многих СУБД для упрощения типизации данных можно создать специальные типы данных (create type) и использовать их в качестве типов полей таблиц.

Описание ограничений целостности

На этапе логического проектирования необходимо описать все ограничения целостности, обусловленные предметной областью.

Если какое-либо ограничение целостности может быть включено в структуру БД (на языке DCL), то его надо реализовать именно так. СУБД проверяет выполнение ограничений целостности при каждой операции модификации данных, если эта операция может нарушить целостность данных. Если ограничения целостности включены в схему

БД, они проверяются автоматически и нельзя внести в базу ошибочные данные. Если же перенести проверку ограничений целостности в программу, то гарантировать их соблюдение нельзя. Программа, во-первых, может содержать ошибки, во-вторых, её можно «обойти», обратившись к БД напрямую с помощью команд языка DML.

Необходимо обратить особое внимание на поля таблиц, для которых домен определён как список возможных значений. Это ограничение целостности можно реализовать в виде: CHECK(<поле> IN (<список значений>)). Но такой подход имеет следующий недостаток: добавление нового значения в список потребует изменения схемы отношения (команда ALTER TABLE). Можно поступить по-другому: вынести этот список значений в отдельное отношение. Например, список типов образования (начальное, неполное среднее, среднее, среднеспециальное, незаконченное высшее, высшее) для таблицы *СОТРУДНИКИ*. Таблица *ТИПЫ ОБРАЗОВАНИЯ* будет состоять из одного поля *Название типа*, определённого как первичный ключ. Тогда поле *Образование* таблицы *СОТРУДНИКИ* станет внешним ключом.

Определение списка значений позволяет гарантировать правильность вводимых данных и правильность поиска. Если не ограничивать значения поля, то оператор может ввести данные произвольным образом, например: 'незаконченное высшее', 'незаконч. высшее', 'н. высш.' и т.д. Человек понимает, что это одно и то же, а для СУБД это разные значения, и учесть все возможные комбинации в условии поиска очень сложно.

Если какое-либо ограничение целостности (ОЦ) нельзя реализовать средствами DCL, то возможны следующие способы его реализации:

1. С помощью процедурных объектов БД. Чаще всего для этой цели используются триггеры (trigger). Триггер – это процедура БД, которая привязана к конкретной таблице и вызывается автоматически при наступлении определённого события (добавления, удаления или модификации данных этой таблицы). Процедура триггера пишется на том языке, который поддерживается выбранной СУБД (например, PL/SQL для Oracle, Visual Basic для MS SQL Server). Триггер пишется программистом и выполняет те действия, которые обусловлены предметной областью. Например, триггер может осуществлять проверку «возраст принимаемого на работу сотрудника не может быть менее 16-ти лет» или присваивать полю «Дата заказа» текущую дату при добавлении нового заказа. Если триггер диагностирует нарушение ограничений целостности, он выдаст сообщение об ошибке и команда модификации данных не будет выполнена (произойдёт автоматический откат, rollback).

2. Программно (т.е. через приложение). Для большей гарантии соблюдения ОЦ желательно проектировать программу так, чтобы внесение изменений в данные и проверка ОЦ выполнялись в одном единственном месте.

3. Вручную. Ручная процедура обязательно должна быть описана в документации (в руководстве пользователя).

Из всех вышеперечисленных способов самым надёжным является использование триггеров, так как триггеры запускаются автоматически и при внесении изменений в данные вручную, и при программной обработке. Но триггеры сильно замедляют работу БД. Для увеличения эффективности работы можно комбинировать указанные методы реализации ОЦ.

Аномалии модификации данных

После составления концептуальной (логической) схемы БД необходимо проверить её на отсутствие аномалий модификации данных. Дело в том, что при неправильно спроектированной схеме БД могут возникнуть аномалии выполнения операций модификации данных. Эти аномалии обусловлены ограниченностью структуры РМД (отсутствием агрегатов и пр.).

Рассмотрим эти аномалии на примере отношения со следующими атрибутами (атрибуты, входящие в ключ, выделены подчёркиванием):

ПОСТАВКИ (*Номер_поставки*, *Название_товара*, *Цена товара*, *Количество*, *Дата поставки*, *Название поставщика*, *Адрес поставщика*)

Различают три вида аномалий: аномалии обновления, удаления и добавления. Аномалия обновления может возникнуть в том случае, когда информация дублируется. Другие аномалии возникают тогда, когда две и более сущности объединены в одно отношение. Например:

1. Аномалия обновления: в отношении *ПОСТАВКИ* она может возникнуть, если у какого-либо поставщика изменился адрес. Изменения должны быть внесены во все кортежи, соответствующие поставкам этого поставщика; в противном случае данные будут противоречивы.

2. Аномалия удаления: при удалении записей обо всех поставках определённого поставщика все данные об этом поставщике будут утеряны.

3. Аномалия добавления: в нашем примере она возникнет, если с поставщиком заключён договор, но поставок от него ещё не было. Сведения о таком поставщике нельзя внести в таблицу *ПОСТАВКИ*, так как для него не определён ключ (номер поставки и название товара) и другие обязательные атрибуты.

Для решения проблемы аномалии модификации данных при проектировании реляционной БД проводится нормализация отношений.

ЗАКЛЮЧЕНИЕ

Рассмотренные в настоящем учебнике вопросы, относящиеся к проектированию и разработке баз данных, далеко не исчерпывают весь перечень проблем и направлений развития этой отрасли информатики. В настоящее время осуществляются многочисленные исследования в области баз данных, СУБД и построении на их основе информационных систем. Активно разрабатываются новые средства описания и манипулирования данными, а также алгоритмы выполнения операций в СУБД. Требуют новых эффективных решений задачи обеспечения информационной безопасности баз данных, без чего невозможна информационная безопасность и конкретного владельца информации, и организации, и страны в целом.

Дальнейший прогресс в направлении внедрения информационных систем во все сферы деятельности невозможен без разработки и использования распределённых баз данных, совершенствования способов обмена информацией между различными приложениями в глобальных компьютерных сетях. Ведущими мировыми компаниями – разработчиками баз данных предлагаются новые версии своих систем, реализующих всё более широкий спектр функций. Осуществляется разработка и принятие новых стандартов в области СУБД.

Однако следует отметить, что все представленные направления развития теории и практики создания баз данных используют в качестве фундаментальной основы знание основных принципов баз данных. Формирование таких знаний и соответствующих практических умений и являлось основной целью данного учебника.

Авторы надеются, что представленный в данном пособии учебный материал был доступным для понимания и интересным при его изучении.

СПИСОК ЛИТЕРАТУРЫ

1. **Гурвиц, Г. А.** Microsoft Access 2010. Разработка приложений на реальном примере. +CD. / Г. А. Гурвиц. – Москва : BHV, 2010. – 496 с.

2. **Илюшечкин, В. М.** Основы использования и проектирования баз данных / В. М. Илюшечкин. – Москва : Высшее образование, 2011 – 213 с.

3. **Цыганов, А. А.** Управление данными / А. А. Цыганов, А. В. Кузовкин, Б. А. Щукин. – Москва : Academia (Академпресс), 2010. – 256 с.

4. **Коннолли, Т.** Базы данных: проектирование, реализация, сопровождение. Теория и практика / Т. Коннолли, К. Бегг. – Москва : Изд. дом «Вильямс», 2003. – 1440 с.

5. **Карпова, И. П.** Базы данных. Курс лекций и материалы для практических занятий : учебное пособие / И. П. Карпова. – Санкт-Петербург : Питер, 2013. – 240 с.

6. **Попов, В. Б.** Основы информационных и телекоммуникационных технологий. Системы управления базами данных : учебное пособие / В. Б. Попов. – Москва : Финансы и статистика, 2005. – 176 с.

7. **Хомоненко, А. Д.** Базы данных: учебник для вузов / А. Д. Хомоненко, В. М. Цыганков, М. Г. Мальцев. – Санкт-Петербург : КОРОНА принт, 2004. – 672 с.

8. **ГОСТ 20886–85.** Организация данных в системах обработки данных. Термины и определения.

9. **ГОСТ 34.320–96.** Информационные технологии. Система стандартов по базам данных. Концепции и терминология для концептуальной схемы и информационной базы. – Межгосударственный стандарт. Дата введения 01.07.2001.

ОГЛАВЛЕНИЕ

Учебное издание

ГРОМОВ Юрий Юрьевич,
ИВАНОВА Ольга Геннадьевна,
ЯКОВЛЕВ Алексей Вячеславович,
ОДНОЛЬКО Валерий Григорьевич

УПРАВЛЕНИЕ ДАННЫМИ

Учебник

Редактор Л. В. К о м б а р о в а

Инженер по компьютерному макетированию И. В. Е в с е е в а

ISBN 978-5-8265-1385-9

Подписано к изданию 22.01.2015.
Формат 60 × 84 / 16. 11,16 усл. печ. л.
Тираж 100 экз. (1-й з-д 50). Заказ № 33

Издательско-полиграфический центр
ФГБОУ ВПО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел. 8(4752) 63-81-08;
E-mail: izdatelstvo@admin.tstu.ru