

ВВЕДЕНИЕ

Прикладные системы, использующие базы данных, находят применение во всех сферах деятельности и обычно эксплуатируются в течение длительного промежутка времени.

Подобные системы проектировались для управления большим потоком транзакций, каждая из которых сопровождалась внесением каких-либо изменений в оперативные данные предприятия; это так называемые OLTP-системы (Online Transaction Processing). Размер базы данных в таких системах исчисляется: для малых систем — несколькими мегабайтами; для средних — гигабайтами и для очень больших — терабайтами.

В процессе эксплуатации прикладных систем приходится решать разные задачи — обеспечение текущей работы, включение в систему новых возможностей и др. В каких-то случаях возникает необходимость создать новую систему, использующую базу данных, или существенно реорганизовать существующую, с тем чтобы она отвечала требованиям предприятия.

Все это требует знания теории и практики баз данных, включающих в себя как общие концептуальные вопросы создания информационной модели предприятия, так и особенности использования в конечном итоге конкретного продукта — конкретной СУБД.

В данном издании сделана попытка осветить все эти вопросы для OLTP-систем. Рассмотрен обычный жизненный цикл систем с базами данных: проектирование, разработка, реализация и сопровождение.

На этапе проектирования важно рассматривать фундаментальные вопросы разработки информационной модели, не обращая внимания на конкретное физическое представление данных; отсюда рассматриваются инструментальные средства разработки информационной модели — понятие модели данных вообще и модели сущность—связь для описания данных, используемых предприятием, на концептуальном уровне, понятном всем заинтересованным лицам.

Разработка базы данных требует знания реляционной модели, поэтому этой модели уделено существенное внимание.

Реализация базы данных требует рассмотрения вопросов, связанных с организацией доступа к данным на уровне физического хранения; поэтому здесь уделяется определенное внимание и этим вопросам.

Перевод разработанной базы данных на язык реляционной СУБД (РСУБД) требует использования SQL, поэтому в данном издании большое внимание уделяется использованию структурированного

языка запросов как стандартного средства управления объектами и данными реляционной СУБД. В то же время реализация и сопровождение базы данных требуют уже знания конкретной РСУБД. Использование возможностей реляционной базы данных (например, написание различных запросов, разработка процедур и триггеров для реализации необходимого функционального наполнения) также требует применения определенной версии языка SQL. Все учебные примеры, представленные в данном издании, написаны на диалектах языка SQL наиболее распространенных на сегодняшний день поставщиков реляционных СУБД — компаний Microsoft и Oracle, в основе инструментальных средств работы с базами данных которых используется стандарт SQL92.

Эффективная эксплуатация разработанной и реализованной базы данных определяется в том числе и производительностью выполняемых SQL-запросов. В связи с этим рассматриваются принципы работы оптимизатора SQL-запросов, интерпретация плана выполнения SQL-запросов, а также подходы по обеспечению условий для оптимального выполнения SQL-запросов с точки зрения их производительности.

В соответствии с изложенным в настоящем издании можно выделить следующие основные разделы.

- делить следующие основные разделы:

 1. Общее понятие модели данных, концептуальное проектирование, модель данных сущность–связь.
 2. Реляционная модель данных, функциональные зависимости, теория нормализации.
 3. Внутренние структуры хранения, методы доступа к данным, элементы оптимизации.
 4. Языковые средства реляционных СУБД для реализации баз данных и построения SQL-запросов, рассмотрение диалектов языка SQL конкретных РСУБД.
 5. Принципы работы оптимизатора SQL-запроса, интерпретация плана выполнения SQL-запроса, анализ производительности и возможные методы повышения эффективности выполнения SQL-запроса.

ПРЕДИСЛОВИЕ

Данный учебник предназначен для студентов, обучающихся по направлению 230100.62 «Информатика и вычислительная техника», специализация – высокопроизводительные компьютерные системы и технологии. Дисциплина включена в базовую часть профессионального цикла дисциплин ФГОС ВПО (квалификация бакалавр). Для освоения дисциплины необходимы компетенции, сформированные у обучающихся в результате изучения дисциплин математического цикла (в частности, дисциплины раздела «Информатика») и дисциплин профессионального цикла раздела «Программирование». Сформированные при изучении данной дисциплины компетенции необходимы для освоения дисциплины общенаучного цикла «Архитектура информационных систем». Книга может быть использована и по направлению 230401.62 «Прикладная математика».

Жизненный цикл современных информационных систем в большинстве случаев предполагает использование, в том числе, и баз данных. В настоящее время (на момент написания данного издания) существует достаточно большое количество СУБД, в основе которых используются различные модели данных, такие как реляционные, объектные, объектно-реляционные, сетевые, иерархические и т.п. Однако, несмотря на многообразие существующих методологий (разработки) баз данных, подавляющее большинство информационных систем, в том числе и масштаба предприятия, в своей основе используют *реляционные базы данных*. В настоящем издании рассматриваются как концептуальные, так и практические аспекты всех этапов жизненного цикла разработки и использования реляционных баз данных.

Первые семь глав посвящены фундаментальным принципам построения и использования моделей данных, являющихся основой систем баз данных. Рассмотрение моделей данных начинается с понятия *информация* и заканчивается подробным описанием этапов проектирования реляционной модели данных с учетом возможных семантических зависимостей и принципов нормализации.

Процесс реализации модели данных предполагает использование языка SQL как средства описания и манипулирования данными, поэтому далее рассматривается язык SQL в рамках стандарта SQL92, а также его интерпретация в условиях использования конкретных СУБД: Microsoft SQL Server и Oracle Database. Описание языка SQL сопровождается большим количеством примеров.

Эффективность использования существующей базы данных определяется в том числе и тем, насколько рационально используются ресурсы системы. Поэтому в главах 10, 11 рассматриваются основные методы доступа к данным современных структур индексирования, а также принципы работы SQL-оптимизатора и интерпретация плана выполнения SQL-запросов.

ГЛАВА 1

ОСНОВНЫЕ ПОНЯТИЯ

1.1. ПОНЯТИЕ ДАННЫХ

Информационные системы предназначены для хранения, выборки и модификации постоянно существующей информации

Восприятие реального мира можно соотнести с последовательностью разных (часто взаимосвязанных) явлений. Эти явления всегда стремились описать (даже если не всегда понятна причина явления).

Такие описания внешних явлений образуют данные — они хранятся и обрабатываются.

Описание внешних явлений (данных) включает два элемента:

- разрозненные факты, хранящиеся в ЭВМ (значения);
 - смысл данных (интерпретация данных, их семантика).

Часто используется как синоним термина **данные** термин **информация**.

Описание данных требует использование некоторого языка. Описание данных на естественном языке позволяет значения данных (факты) и их семантику фиксировать вместе, так как естественный язык достаточно гибок для этих целей. Например, можно сказать: «Его рост 185 см». Здесь 185 — значение данных, фраза «Его рост ... см» — семантика данных. В данной фразе значение связано с семантикой, и становится понятен и смысл данных, и что с ними можно делать.

Нередко данные и их семантика разделены, и такое разделение можно встретить и в нашей обычной жизни. Например, расписание движения пассажирских поездов может быть представлено в виде таблицы, заголовок которой определяет семантику данных, а расположенные далее строки — значения данных (табл. 1.1).

Таблица 1.1

Расписание движения пассажирских поездов

Номер поезда	Станция назначения	Категория поезда	Дни отправления	Время отправления	Время в пути	Время прибытия
121	Вологда	Пассажирский	По четным	19.40	12	07.40
...

Если такая таблица большая, строки в ее нижней части уже тяжело читать — забывается смысл колонок, т.е. семантика данных. Если же семантика отсутствует (включена где-то в другом, недоступном в настоящий момент месте), такие данные трудно или вообще невозможно понять.

Применение ЭВМ для хранения и обработки данных приводит к еще большему разделению значений данных и их семантики. Компьютеры чаще всего имеют дело со значениями данных; большая часть их семантики как таковая вообще не фиксируется в ЭВМ. Например, если решается некоторая математическая задача моделирования (дифференциальное уравнение, описывающее некоторый реальный процесс), интерпретация полученных в результате решения значений возлагается на пользователя.

На разных этапах развития информационных систем механизмы интерпретации данных определяются подходами в реализации компонентов хранения и обработки данных.

1.2. ФАЙЛОВЫЕ СИСТЕМЫ

В разных системах (файловых) данные и семантика разделены.

- семантика отражается в приложениях, обрабатывающих данные;
 - никаких способов осмысленного доступа к данным, кроме как через приложение, нет.

Файловые системы можно определить как набор программ, которые выполняют для пользователя некоторые операции, связанные с обработкой данных. При этом каждая программа определяет свои собственные данные и управляет ими.

Ограничивающие присущие таким системам:

- разделение и изоляция данных;
 - дублирование данных;
 - зависимость от файлов;
 - несовместимость данных.

1. Определение данных содержится внутри приложений, а не хранится отдельно и независимо от них.
 2. Помимо приложений, не предусмотрено никаких других инструментов доступа к данным и их обработки.

Глобальная причина кроется в разрыве между значениями данных и их семантикой.

1.3. СИСТЕМЫ БАЗ ДАННЫХ

Информационные системы, использующие базы данных, относятся к многопользовательским системам и построены в соответствии с архитектурой клиент—сервер. В соответствии с этой архитектурой сервер управляет некоторым ресурсом (в нашем случае базой данных, т.е. некоторым набором занесенных во внешнюю память компьютера данных), а клиент обращается к серверу за предоставлением некоторых услуг, связанных с организацией доступа к этим данным. В соответствии с этой архитектурой на сервере размещаются сама база данных и некоторая система, управляющая доступом к данным, — система управления базами данных (СУБД). На клиенте реализуется интерфейс, с помощью которого формируются запросы к серверу и отображаются полученные результаты. Прикладная логика может быть реализована как на сервере, так и на клиенте. Кроме того, часто используются так называемые трехзвенные системы, в соответствии с которыми прикладная логика реализуется в виде отдельного компонента, выполняющего функции сервера (сервера приложения) по отношению к клиенту и клиента по отношению к серверу базы данных.

Нас будет интересовать в первую очередь организация баз данных и доступа к данным, т.е. то, что реализуется сервером базы данных.

Введем некоторые определения.

Определение. *База данных — это совместно используемый набор логически связанных данных и их описаний, предназначенный для удовлетворения информационных потребностей организаций.*

Определение. Система управления базами данных (СУБД) — это программное обеспечение, осуществляющее управление базами данных.

Определение. Система баз данных — это компьютеризированная система хранения записей, т.е. это базы данных, СУБД, аппаратура и люди.

Упрощенная схема системы базовых трансляторов изображена на рис. 1.1.

Упрощенная схема системы баз данных приведена на рис. 1.1. В общем случае с системами баз данных работают пользователи разных категорий, каждая из которых обладает своими возможностями. К таким категориям обычно относят администратора данных и базы данных, разработчиков базы данных, прикладных программистов и конечных пользователей.

В системах с базами данных обязанности между пользователями разных категорий обычно распределяются следующим образом.

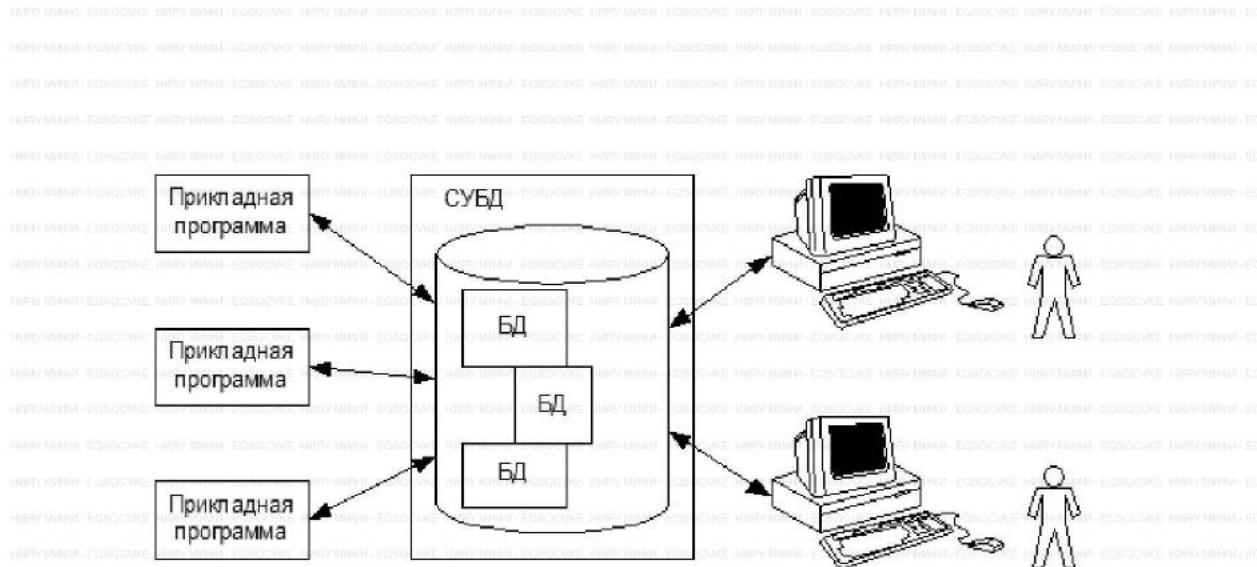


Рис. 1.1. Упрощенная схема баз данных

1. Администраторы данных и баз данных

Администратор данных (АД) отвечает за управление данными, включая планирование базы данных, разработку и сопровождение стандартов, бизнес-правил и деловых процедур, за концептуальное и логическое проектирование базы данных.

Администратор базы данных (АБД) отвечает за физическую реализацию базы данных, сопровождение, обеспечение безопасности и целостности данных; требуется знание СУБД и ОС.

2. Разработчики базы данных

Разработчики базы данных осуществляют разработку логической структуры данных, имеющихся ограничений; должны четко понимать структуру организации и ее бизнес-правила (бизнес-правила описывают основные характеристики данных с точки зрения организации).

3. Прикладные программисты

Прикладные программисты занимаются разработкой приложений, предоставляющих конечным пользователям необходимые функциональные возможности.

4. Конечные пользователи

Выполняют действия, определенные их должностными обязанностями.

1.4. ИСТОРИЯ РАЗВИТИЯ СУБД

Ниже приводится краткое описание истории развития баз данных и СУБД.

1. Начало 1960-х гг. — файловые системы.

- 10 1. Начало 1960-х гг. — файловые системы.

Ниже приведены основные этапы развития СУБД:

- 1. Середина 1960-х гг. — сетевые СУБД. В 1965 г. на конференции CODASYL (Conference on Data System Languages) создана рабочая группа, которая должна была определить спецификации среды, которая допускала бы разработку баз данных и управление данными. Отчет этой группы опубликован в 1971 г. Были определены три компонента:

- сетевая схема (организация базы данных в целом);
- подсхема (часть базы данных, как она видится пользователям и приложениям);
- язык управления данными; сюда были включены язык описания данных (ЯОД, или DDL — Data definition Language) и язык манипулирования данными (ЯМД, или DML — Data Manipulation Language).

Системы на основе CODASYL — это СУБД 1-го поколения, использующие сетевые и иерархические модели данных.

3. В 1970 г. Э. Кодд (Edgar Frank «Ted» Codd) опубликовал статью о реляционной модели данных, что послужило мощным толчком к развитию реляционных СУБД. Коммерческие СУБД, использующие реляционную модель данных, появились в конце 1970-х — начале 1980-х гг. Особо следует упомянуть СУБД System R (IBM, 1976 г.) — в ней был использован язык SQL. СУБД, использующие реляционную модель данных, — это СУБД 2-го поколения.

4. В 1976 г. П. Чен (Peter Pin-Shan Chen) представил модель «сущность—связь», определив тем самым технологию проектирования баз данных. Появились расширенная реляционная модель данных и семантические модели данных.

5. С ростом сложности приложений стали использоваться объектно-ориентированные модели данных, в результате чего появились объектно-ориентированные и объектно-реляционные СУБД, которые определяются как СУБД 3-го поколения.

1.5. ТРЕХУРОВНЕВАЯ АРХИТЕКТУРА ANSI/SPARC

Первая попытка создания стандартной терминологии и общей архитектуры представления данных была предпринята в 1971 г. по результатам конференции по языкам и системам данных CODASYL. При национальном институте стандартов США был создан комитет планирования стандартов и норм — ANSI/SPARC (ANSI — American National Standard Institute, SPARC — Standards Planning and Requirements Committee). Этот комитет в 1975 г. признал необходимость использования трехуровневого подхода при описании представления данных.

В соответствии с этим подходом используются три уровня абстракции описания элементов данных; они формируют трехуровневую архитектуру.

Ниже минимумы, которые определяют минимальные ограничения на количество и типы данных, необходимые для выполнения определенных функций. Структура данных определяет способ представления информации в базе данных.

Внешний уровень — представление данных с точки зрения конечных пользователей. Этот уровень описывает ту часть базы данных, которая относится к каждому конечному пользователю. Например, некоторое предприятие требует организации хранения данных о его сотрудниках. В одном подразделении данного предприятия (Пользователь 1) информация о сотруднике должна включать в себя следующие данные: фамилия, имя, отчество сотрудника; дата рождения; сумма выплат за год. В другом подразделении (Пользователь 2) требуется хранить фамилию, имя, отчество сотрудника; дату приема на работу; подразделение, где работает сотрудник, и занимаемую должность. Отсюда для каждого подразделения создаются свои представления:

Концептуальный уровень — охватывающий внешний, концептуальный и внутренний уровни (рис. 1.2).



Рис. 1.2. Трехуровневая архитектура ANSI/SPARC

Разделение приводит к тому, что каждый отвечает за свое и знает только свое; тем самым достигается независимость от физического и логического представлений.

Внешний уровень — представление данных с точки зрения конечных пользователей. Этот уровень описывает ту часть базы данных, которая относится к каждому конечному пользователю. Например, некоторое предприятие требует организации хранения данных о его сотрудниках. В одном подразделении данного предприятия (Пользователь 1) информация о сотруднике должна включать в себя следующие данные: фамилия, имя, отчество сотрудника; дата рождения; сумма выплат за год. В другом подразделении (Пользователь 2) требуется хранить фамилию, имя, отчество сотрудника; дату приема на работу; подразделение, где работает сотрудник, и занимаемую должность. Отсюда для каждого подразделения создаются свои представления:

Представление 1:

ФИО сотрудника	Дата рождения	Сумма выплат
----------------	---------------	--------------

Представление 2:

ФИО сотрудника	Дата приема на работу	Подразделение	Должность
----------------	-----------------------	---------------	-----------

Концептуальный уровень – обобщающее представление данных.

Описывает, какие данные хранятся в базе данных, а также связи, существующие между ними. На этом уровне может быть создано общее представление, в котором хранится информация о сотрудниках предприятия, например, в следующем виде:

ФИО сотрудника	Дата рождения	Дата приема на работу	Подразделение	Должность	Сумма выплат
----------------	---------------	-----------------------	---------------	-----------	--------------

Внутренний уровень — физическое представление базы данных в компьютере; описывает, как информация хранится в базе данных. Например, информация о сотрудниках может быть представлена в виде следующей структуры, в которой указывается конкретный способ представления данных:

```
struct Employee {  
    int EmployeeId;  
    char LastName [50], FirstName[50], MiddleName[50];  
    struct Date Birthdate, Hiredate;  
    int DepartmentId, Job;  
    float Payments;  
};
```

Концептуальная схема является «сердцем» базы данных. Она поддерживает все внешние представления, а сама поддерживается средствами внутренней схемы. Именно концептуальная схема призвана быть полным и точным представлением требований к данным организации (предприятия).

Трехуровневый подход используется в основном при проектировании систем баз данных. Схемы базы данных, получаемые на разных этапах проектирования, представляют одну и ту же базу данных на разных уровнях абстракции. В зависимости от сложности системы данный подход может быть использован в полном объеме или частично. Так, в каких-то ситуациях создание внешних представлений может не потребоваться — может быть создано сразу обобщенное, концептуальное представление данных. Но наличие концептуального представления, даже для небольших по объему систем баз данных, существенно облегчает понимание структуры базы данных и организацию работы с данными. Поэтому здесь часто используется концептуальное представление для объяснения тех или иных принятых решений, связанных с представлением данных и их обработкой.

Средством, позволяющим представить данные на определенном уровне абстракции, служат модели данных.

Определение. *Модель данных* — это интегрированный набор понятий для описания данных, связей между ними и ограничений, накладываемых на данные в некоторой организации.

Наиболее распространенные модели данных:

- семантические (к ним можно отнести модели данных «сущность–связь», бинарные, семантические сети, объектно-ориентированные);
 - реляционная,
 - сетевая,
 - иерархическая.

Одно из основных требований к СУБД — обеспечение некоторого уровня абстракции при представлении и обработке данных. Поэтому любая СУБД поддерживает по крайней мере одну из моделей данных.

Вопросы

1. Дайте определение термину *информация*. Приведите примеры.
 2. Укажите достоинства и недостатки систем баз данных.
 3. Сформулируйте основные отличия файловых систем и систем баз данных.
 4. Укажите категории пользователей в системах баз данных и их обязанности.
 5. Опишите основную идею подхода при описании представления данных, определенную комитетом ANSI/SPARC.

ГЛАВА 2

ОБЩАЯ ХАРАКТЕРИСТИКА МОДЕЛЕЙ ДАННЫХ

2.1. ОСНОВНЫЕ ПОНЯТИЯ МОДЕЛИ ДАННЫХ

Все модели данных принято делить на две категории: сильно типизированные и слабо типизированные.

Сильно типизированные модели данных — это модели данных, в которых все данные относятся к конкретным категориям. Если появляются новые данные, они либо подгоняются под уже определенную категорию, либо для них определяется новая категория. В дальнейшем связи и соотношения между данными рассматриваются на уровне связей и соотношений между категориями.

Слабо типизированные модели данных — это модели данных, в которых нет никаких предположений для категоризации; элементы данных относятся к той или иной категории только тогда, когда это необходимо в каждом конкретном случае.

Большинство моделей данных, используемых в информационных системах, относятся к сильно типизированным моделям. Следовательно, в таких моделях данных можно выделить следующие структурные компоненты:

- категория;
 - свойства категории;
 - связи между категориями.

Определение. В конкретном применении модели данных совокупность именованных категорий, их свойств и связей между ними называется *схемой*.

Например, некоторая модель используется для представления транспортного отдела, точнее для представления сведений об автомобилях и управляющих ими водителях. Тогда в ней будут представлены две категории: **ВОДИТЕЛЬ** со свойствами *Имя, Возраст, Стаж работы* и **АВТОМОБИЛЬ** со свойствами *Модель, Гос. номер, Дата приобретения*. Между этими категориями имеет место связь: **ВОДИТЕЛЬ УПРАВЛЯЕТ АВТОМОБИЛЕМ**. Соответствующую схему можно представить так:

ВОДИТЕЛЬ (Имя, Возраст, Стаж работы)
АВТОМОБИЛЬ (Модель, Гос. номер, Дата приобретения)
УПРАВЛЯЕТ (ВОДИТЕЛЬ, АВТОМОБИЛЬ)

Определение. Совокупность данных, структура которых соответствует конкретной схеме, называется *реализацией базы данных*.

которой схеме, называемой **рекурсивной**, база данных?

Таким образом, схема определяет структуру данных. Однако структурные спецификации не обеспечивают возможности полной интерпретации семантики данных и способа их использования. Необходимо еще определить допустимые операции над данными. Обычно операции соотносятся со структурами данных.

Операции, предусмотренные моделью данных, преобразуют одну реализацию базы данных в другую, но все эти реализации имеют одну и ту же структуру и соответствуют одной и той же схеме. В соответствии с этим можно дать следующее определение.

Определение. Под базой данных понимают последовательность реализаций, полученных в результате некоторых преобразований и удовлетворяющих одной и той же схеме.

2.2. ПРЕДСТАВЛЕНИЕ СТАТИЧЕСКИХ И ДИНАМИЧЕСКИХ СВОЙСТВ

Модель данных должна некоторым образом отражать реальный мир: предметная область — часть реального мира, представляющая интерес для данного исследования (использования). Реальный мир обладает и статическими, и динамическими свойствами. Следовательно, модель данных должна как-то представлять и статические, и динамические свойства реального мира.

Отсюда модель данных можно определить как множество правил порождения G (Generate Rules) и множество операций O (Operators). Множество правил порождения представляют статические свойства модели данных и соотносятся с языком описания данных (ЯОД). Множество операций представляют динамические свойства модели данных и соотносятся с языком манипулирования данными (ЯМД).

Средствами ЯОД определяются допустимые структуры данных — объектов (категорий, сущностей) и связей, а также допустимые реализации данных.

Определение структуры данных реализуется посредством спецификации соответствующих категорий, которые должны удовлетворять правилам порождения G. Спецификация категорий определяется в терминах атрибутов (свойств категорий) и типов значений каждого атрибута. Селекция допустимых реализаций объектов и связей задается указанием для каждой категории (типа сущности) дополнительных условий — ограничений целостности, которым должна удовлетворять каждая реализация.

В соответствии с этим в некоторых моделях данных правила порождения делятся на две части:

- правила порождения структуры Gs (Generate Structure);
 - правила порождения ограничений Gc (Generate Constraints).

Ниже приведены основные правила порождения множества схем БД:

- правила порождения G обеспечивают порождение множества схем S , каждая из которых (S_i) определяет конкретную структуру данных и специфицирует ограничения целостности;
- конкретной схеме S_i соответствует множество различных реализаций базы данных D_{i1}, D_{i2}, \dots ;
- множество операций определяют допустимые действия над реализацией базы данных D_{ij} для преобразования ее в другую реализацию D_{ik} .

Таким образом (рис. 2.1):

Модель данных

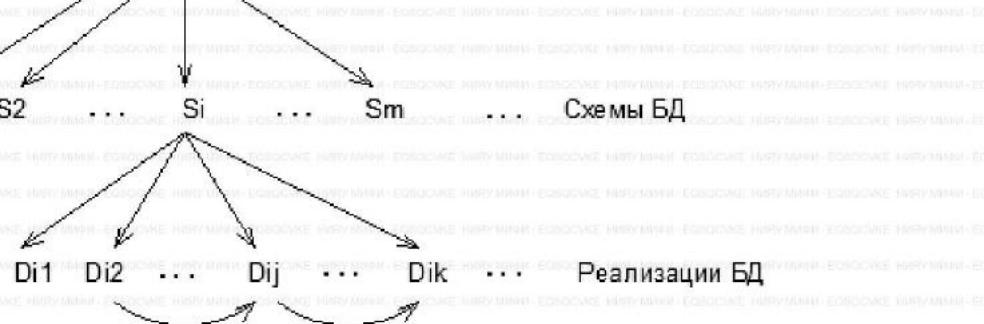


Рис. 2.1. Соотношение между моделями данных, схемами и реализациями базы данных

Следовательно, модель данных определяет структуру, ограничения целостности и допустимые операции.

Рассмотрим общие характеристики и правила, соответствующие представлению этих трех составляющих модели данных.

2.3. ОБЩАЯ ХАРАКТЕРИСТИКА СТРУКТУРНЫХ КОМПОНЕНТОВ. МНОЖЕСТВА: ДОМЕНЫ И АТРИБУТЫ

К структурным компонентам модели данных относятся:

- категории;
- свойства категорий;
- связи между категориями.

Категория представляет собой агрегат свойств. Свойства представляются совокупностью (множеством) значений. Отсюда рассматривается понятие множества.

С точки зрения моделей данных выделяются множества, элементы которых однородны (домены), и множества, построенные на других множествах (отношения).

множествах построены отношения, различаются отношения сущности и отношения связи (рис. 2.2).

множествах построены отношения, различаются отношения сущности и отношения связи (рис. 2.2).

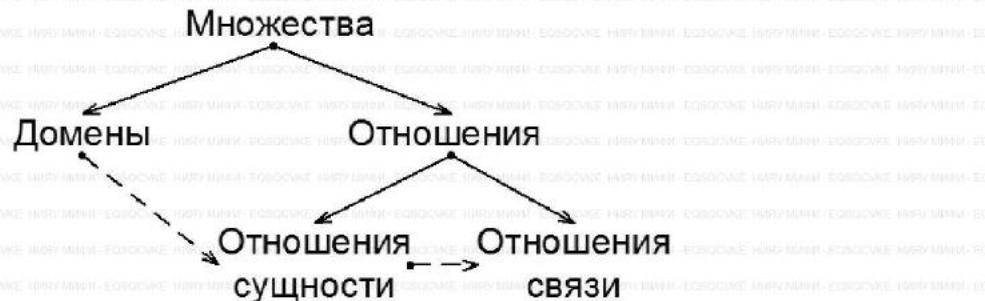


Рис. 2.2. Структурные компоненты модели данных

Начнем рассмотрение с множеств — доменов.

Определение. *Множество* — это собрание правильно идентифицированных объектов, удовлетворяющих некоторому правилу принадлежности.

Правила (условия) принадлежности могут быть определены по-разному, например:

- **ASCII-коды символов,**
или
 - **целые положительные десятичные числа,**
или
 - **вещественные числа в диапазоне от 0 до 1**
и тп.

Правила принадлежности позволяют для каждого элемента определить, относится он к данному множеству или нет. Чтобы определить множество, необходимо определить для него правило принадлежности.

Понятие множества не связано с какой-либо упорядоченностью.

Множество характеризуется двумя важными свойствами: интенционалом (intentional) и экспенсионалом (extensional).

Определение множества (правило принадлежности) задает его интенсионал, например: *целые положительные четные числа*. Интенсионал множества определяет совокупность конкретных множеств, удовлетворяющих правилу принадлежности. Так, для приведенного примера можно определить такую совокупность множеств: {2, 8, 16, 46}, {12, 10, 8, 100, 32}, {2, 4, 8, 16, 32, 64}.

Экстенсиональ множества — это конкретная реализация, удовлетворяющая интенсионалу, например: {2, 4, 8, 16, 32, 64}.

Интенсионал множества соответствует уровню типов, а экстенсионал множества — уровню знаков.

Ниже приведены примеры доменов и атрибутов:

- Домен Целые** — множество целых чисел.
- Атрибут Зарплата** — значение атрибута для домена Целые.
- Атрибут Номер служащего** — значение атрибута для домена Целые.
- Атрибут Возраст** — значение атрибута для домена Целые.

Существуют некоторые множества, значения которых более или менее однородны, например множество целых чисел, множество строк некоторой ограниченной длины и т.п. Такие множества в теории моделей данных получили название **доменов**.

Определение. Домены — это множества, элементы которых более или менее однородны.

Домены можно рассматривать как множества, из которых черпаются значения свойств семантически значимых объектов. Например, если для категории **СЛУЖАЩИЙ** определено свойство **Зарплата**, можно определить домен, например, шестизначных чисел, из которого будут черпаться значения данного свойства.

Таким образом, домен представляет собой множество значений, не имеющих смысловой окраски. Например, из того же домена шестизначных чисел можно черпать значения для свойства **Стоимость** категории **АВТОМОБИЛЬ** или свойства **Вес** категории **ДЕТАЛЬ**. Следовательно, для надлежащего использования значений необходимо связать значения с их семантикой. Отсюда определяется понятие **атрибута**.

Определение. Атрибуты — это именованные домены, представляющие семантически значимые объекты.

Атрибуты определяются на доменах и представляют собой интенсионал именованного домена. Например, атрибут **Зарплата** определен на домене шестизначных чисел. Значения атрибута — это экспенсионалы.

Таким образом, атрибуты и их значения являются интерпретацией объектов реального мира и их свойств. Вводя атрибуты, мы даем интерпретацию абстрактных понятий, таких как числа и строки, а также задаем дополнительные ограничения на операции. Например, для атрибута **Зарплата**, определенного на домене целых шестизначных чисел, определены арифметические операции и все операции отношения — равно, меньше и т.д. Для атрибута **Номер служащего**, который может быть определен на том же домене, определены только операции сравнения — равно и не равно.

Домен можно рассматривать как обобщение атрибутов (рис. 2.3).



Рис. 2.3. Домены и атрибуты

**Атрибуты, определенные на общем домене, наследуют все его
дочерние домены**

Атрибуты, определенные на общем домене, наследуют все его свойства. И наоборот, домен обладает всеми свойствами всех определенных на нем атрибутов.

2.4. ОБЩАЯ ХАРАКТЕРИСТИКА СТРУКТУРНЫХ КОМПОНЕНТОВ. ОТНОШЕНИЯ: СУЩНОСТИ

Атрибуты существуют не сами по себе, а как компоненты других объектов. Посредством агрегации они ассоциируются с другими атрибутами. Например, атрибуты *Имя*, *Адрес*, *Возраст* формируют агрегат **ЛИЧНОСТЬ**. Интерпретация атрибутов и соотношений между ними определяется агрегатами, соответствующими объектам реального мира. Соответствующие агрегаты получили название **отношений**.

Определение. Агрегат, построенный на множествах, определяется как *отношение*.

Отвлекаясь от интерпретации агрегации, получаем агрегат, построенный на множествах. В теории моделей данных широко используется аппарат математической теории множеств.

Определение. Пусть дана некоторая совокупность доменов D_1, D_2, \dots, D_m , не обязательно различных. Отношение, определенное на доменах D_1, D_2, \dots, D_m , есть множество упорядоченных кортежей $\langle d_1, d_2, \dots, d_m \rangle$, таких, что $d_1 \in D_1, d_2 \in D_2, \dots, d_m \in D_m$.

Таким образом, отношение определяет соответствие между множествами.

Поскольку само отношение — тоже множество, как и любое множество, оно характеризуется интенсионалом и экстенсионалом. Интенсионал отношения определяется интенсионалами образующих его множеств. Экстенсионал отношения — конкретная реализация этого отношения.

Рассмотрим пример. Пусть даны следующие множества:

$D1 = \{d1_i \mid d1_i —$ строчная буква английского алфавита} — интенсионал множества, его экстенсионал, например, $\{a, b, c, d, e\}$;

$D_2 = \{d_{2j} \mid d_{2j} — \text{девятчная цифра}\}$ — интенсионал множества, его экстенсионал, например, $\{1, 3, 5\}$.

Определим на этих доменах отношение R :

$R = \{d1_i, d2_j | d1_i \in D1, d2_j \in D2\}$ — интенсионал отношения; задает двухсимвольные кортежи, в которых первый символ — буква, второй — десятичная цифра. Экстенсионалом данного отношения является конкретное множество, например:

$R1 = \{\langle a, 3 \rangle, \langle a, 1 \rangle, \langle c, 1 \rangle\}$.

Отношение можно охарактеризовать степенью и мощностью.

20

**Атрибуты, определенные на общем домене, наследуют все его
дочерние домены**

Атрибуты, определенные на общем домене, наследуют все его свойства. И наоборот, домен обладает всеми свойствами всех определенных на нем атрибутов.

2.4. ОБЩАЯ ХАРАКТЕРИСТИКА СТРУКТУРНЫХ КОМПОНЕНТОВ. ОТНОШЕНИЯ: СУЩНОСТИ

Атрибуты существуют не сами по себе, а как компоненты других объектов. Посредством агрегации они ассоциируются с другими атрибутами. Например, атрибуты *Имя*, *Адрес*, *Возраст* формируют агрегат **ЛИЧНОСТЬ**. Интерпретация атрибутов и соотношений между ними определяется агрегатами, соответствующими объектам реального мира. Соответствующие агрегаты получили название **отношений**.

Определение. Агрегат, построенный на множествах, определяется как *отношение*.

Отвлекаясь от интерпретации агрегации, получаем агрегат, построенный на множествах. В теории моделей данных широко используется аппарат математической теории множеств.

Определение. Пусть дана некоторая совокупность доменов D_1, D_2, \dots, D_m , не обязательно различных. Отношение, определенное на доменах D_1, D_2, \dots, D_m , есть множество упорядоченных кортежей $\langle d_1, d_2, \dots, d_m \rangle$, таких, что $d_1 \in D_1, d_2 \in D_2, \dots, d_m \in D_m$.

Таким образом, отношение определяет соответствие между множествами.

Поскольку само отношение — тоже множество, как и любое множество, оно характеризуется интенсионалом и экстенсионалом. Интенсионал отношения определяется интенсионалами образующих его множеств. Экстенсионал отношения — конкретная реализация этого отношения.

Рассмотрим пример. Пусть даны следующие множества:

$D1 = \{d1_i \mid d1_i —$ строчная буква английского алфавита} — интенсионал множества, его экстенсионал, например, $\{a, b, c, d, e\}$;

$D_2 = \{d_{2j} \mid d_{2j} — \text{девятчная цифра}\}$ — интенсионал множества, его экстенсионал, например, $\{1, 3, 5\}$.

Определим на этих доменах отношение R :

$R = \{d1_i, d2_j | d1_i \in D1, d2_j \in D2\}$ — интенсионал отношения; задает двухсимвольные кортежи, в которых первый символ — буква, второй — десятичная цифра. Экстенсионалом данного отношения является конкретное множество, например:

$R1 = \{\langle a, 3 \rangle, \langle a, 1 \rangle, \langle c, 1 \rangle\}$.

Отношение можно охарактеризовать степенью и мощностью.

20

**Атрибуты, определенные на общем домене, наследуют все его
дочерние домены**

Атрибуты, определенные на общем домене, наследуют все его свойства. И наоборот, домен обладает всеми свойствами всех определенных на нем атрибутов.

2.4. ОБЩАЯ ХАРАКТЕРИСТИКА СТРУКТУРНЫХ КОМПОНЕНТОВ. ОТНОШЕНИЯ: СУЩНОСТИ

Атрибуты существуют не сами по себе, а как компоненты других объектов. Посредством агрегации они ассоциируются с другими атрибутами. Например, атрибуты *Имя*, *Адрес*, *Возраст* формируют агрегат **ЛИЧНОСТЬ**. Интерпретация атрибутов и соотношений между ними определяется агрегатами, соответствующими объектам реального мира. Соответствующие агрегаты получили название **отношений**.

Определение. Агрегат, построенный на множествах, определяется как *отношение*.

Отвлекаясь от интерпретации агрегации, получаем агрегат, построенный на множествах. В теории моделей данных широко используется аппарат математической теории множеств.

Определение. Пусть дана некоторая совокупность доменов D_1, D_2, \dots, D_m , не обязательно различных. Отношение, определенное на доменах D_1, D_2, \dots, D_m , есть множество упорядоченных кортежей $\langle d_1, d_2, \dots, d_m \rangle$, таких, что $d_1 \in D_1, d_2 \in D_2, \dots, d_m \in D_m$.

Таким образом, отношение определяет соответствие между множествами.

Поскольку само отношение — тоже множество, как и любое множество, оно характеризуется интенсионалом и экстенсионалом. Интенсионал отношения определяется интенсионалами образующих его множеств. Экстенсионал отношения — конкретная реализация этого отношения.

Рассмотрим пример. Пусть даны следующие множества:

$D1 = \{d1_i \mid d1_i —$ строчная буква английского алфавита} — интенсионал множества, его экстенсионал, например, $\{a, b, c, d, e\}$;

$D_2 = \{d_{2j} \mid d_{2j} — \text{девятчная цифра}\}$ — интенсионал множества, его экстенсионал, например, $\{1, 3, 5\}$.

Определим на этих доменах отношение R :

$R = \{d1_i, d2_j | d1_i \in D1, d2_j \in D2\}$ — интенсионал отношения; задает двухсимвольные кортежи, в которых первый символ — буква, второй — десятичная цифра. Экстенсионалом данного отношения является конкретное множество, например:

$R1 = \{\langle a, 3 \rangle, \langle a, 1 \rangle, \langle c, 1 \rangle\}$.

Отношение можно охарактеризовать степенью и мощностью.

20

Ниже приведены определения и свойства отображений.

Определение. Отображение $R : S_1 \rightarrow S_2$ называется функцией, если для каждого элемента $s_1 \in S_1$ существует единственный элемент $s_2 \in S_2$, такой что $(s_1, s_2) \in R$.

Свойство 1. Для любых $s_1, s'_1 \in S_1$ и $s_2 \in S_2$ выполняется либо $(s_1, s_2) \in R$, либо $(s'_1, s_2) \in R$, но не оба одновременно.

Свойство 2. Для любых $s_1, s'_1 \in S_1$ и $s_2, s'_2 \in S_2$ выполняется либо $(s_1, s_2) \in R$ и $(s'_1, s'_2) \in R$, либо $(s'_1, s_2) \in R$ и $(s_1, s'_2) \in R$, но не оба одновременно.

Свойство 3. Для любых $s_1, s'_1 \in S_1$ и $s_2, s'_2 \in S_2$ выполняется либо $(s_1, s_2) \in R$ и $(s'_1, s'_2) \in R$, либо $(s'_1, s_2) \in R$ и $(s_1, s'_2) \in R$, либо $(s'_1, s'_2) \in R$ и $(s'_1, s_2) \in R$, но не более трех одновременно.

Для таких агрегатов наряду с интенсионалом и экстенсионалом рассматривается еще одно важное свойство отношений — отображение между отношениями, на которых построен агрегат.

Без потери общности ограничимся рассмотрением бинарных отношений — отношений, построенных на двух множествах. В этом случае отображение определяется как отображение одного множества на другое.

Рассмотрим бинарное отношение связи R , построенное на двух множествах — отношениях сущностей S_1 и S_2 : $R = \{<s_1, s_2> \mid s_1 \in S_1, s_2 \in S_2\}$. Данное отношение связи определяет два отображения:

- прямое, записывается как $R : S_1 \rightarrow S_2$; отображение первого множества в записи (S_1) на второе (S_2);
- обратное, записывается как $R^{-1} : S_2 \rightarrow S_1$; отображение первого множества в записи (S_2) на второе (S_1).

Важной характеристикой отображения является кардинальное число.

Определение. Кардинальное число (или кардинальность) отображения определяется количеством элементов второго множества отображения, связанных с одним элементом первого множества.

Так, для прямого отображения $R : S_1 \rightarrow S_2$, кардинальное число определяется количеством элементов множества S_2 , связанных с одним элементом множества S_1 ; для обратного отображения $R^{-1} : S_2 \rightarrow S_1$ — количеством элементов множества S_1 , связанных с одним элементом множества S_2 .

Так как с разными элементами одного множества может быть связано разное количество элементов другого множества, отображения обычно характеризуются минимальным и максимальным кардинальными числами. Поскольку отношение связи определяет два отображения, используется следующая нотация:

$$R(S_1(m_1, n_1) : S_2(m_2, n_2)).$$

Запись $S_1(m_1, n_1)$ определяет минимальное (m_1) и максимальное (n_1) кардинальные числа отображения $S_2 \rightarrow S_1$. Соответственно запись $S_2(m_2, n_2)$ определяет минимальное (m_2) и максимальное (n_2) кардинальные числа отображения $S_1 \rightarrow S_2$.

Смысл такой записи:

- каждый элемент из S_1 связан минимум с m_2 , максимум с n_2 элементами из S_2 ;
- каждый элемент из S_2 связан минимум с m_1 , максимум с n_1 элементами из S_1 .

Если на отображения не наложены никакие ограничения, считается, что минимальное и максимальное кардинальные числа не определены; в этом случае используется запись $R(S_1(0, \infty) : S_2(0, \infty))$,

или R ($S_1 : S_2$). Это означает, что элемент из S_2 может быть связан с любым количеством элементов из S_1 , и наоборот.

или R ($S_1 : S_2$). Это означает, что элемент из S_2 может быть связан с любым количеством элементов из S_1 , и наоборот.

Наложив те или иные ограничения на минимальное и максимальное кардинальные числа, можно получить различные типы отображений. Пусть, например, определены сущности **СТУДЕНТ** и **КУРС** (курс по выбору). Каждый студент должен выбрать один из курсов, но не более трех. На каждый курс должны быть зачислены не менее 5 и не более 100 студентов. Тогда получаем следующее отношение связей:

ВЫБИРАЕТ (СТУДЕНТ (5, 100) : КУРС (1, 3)).

Рассмотрим некоторые особые типы отображений.

1. Пусть имеем следующее отношение связи: $R(S1(0, \infty) : S2(1, \infty))$. смотрим отображение $S1 \rightarrow S2$. Минимальное кардинальное сло данного отображения равно 1. Это означает, что каждый элемент из $S1$ связан по крайней мере с одним элементом из $S2$ (или ображается по крайней мере одним элементом $S2$). Такое отображение называется полностью определенным на $S1$, а соответствующее ограничение называется ограничением по существованию: для существования объекта в $S1$ необходимо, чтобы он был связан с объектом из $S2$ (рис. 2.4).

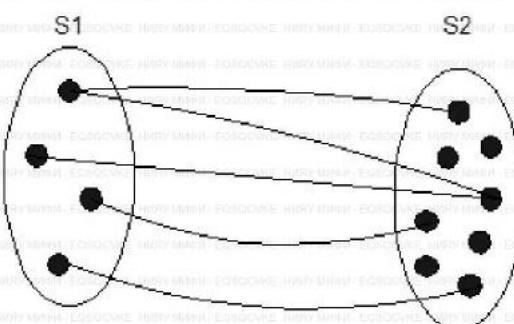


Рис. 2.4. Полноту определенное отображение на S^1

? Пусть имеем следующее отношение связи: $R(S_1(0, \infty) \cdot S_2(0, 1))$.

Рассмотрим отображение $S_1 \rightarrow S_2$. Максимальное кардинальное число данного отображения равно 1. Это означает, что каждый элемент из S_1 связан не более чем с одним элементом из S_2 (или отображается не более чем одним элементом S_2). Такое отображение называется неполным функциональным отображением, так как минимальное кардинальное число отображения равно 0, т.е. не все элементы из S_1 отображаются в S_2 (рис. 2.5).

The diagram shows two sets, S1 and S2, represented as ovals. Set S1 contains 7 solid black dots, while set S2 contains 5 solid black dots. Multiple arrows originate from each dot in S1 and point to different dots in S2, indicating that multiple elements in S1 map to a single element in S2. This visualizes a many-to-one mapping where every element in S1 is associated with at least one element in S2.

Рис. 2.5. Неполное функциональное отображение

3. Пусть имеем следующее отношение связи: $R(S1(0, \infty) : S2(1, 1))$.

Рассмотрим отображение $S_1 \rightarrow S_2$. И минимальное, и максимальное кардинальные числа данного отображения равны 1. Это означает, что каждый элемент из S_1 связан в точности с одним элементом из S_2 (или отображается точно одним элементом S_2). Такое отображение называется полным функциональным отображением (рис. 2.6).

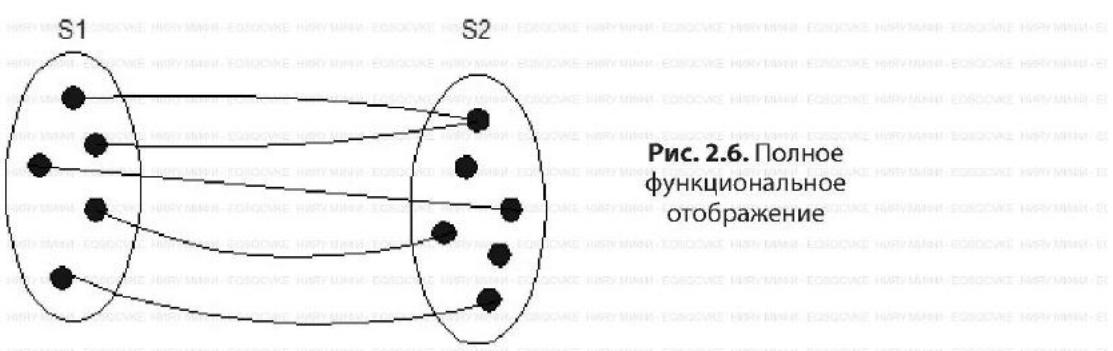


Рис. 2.6. Полное функциональное отображение

Бинарные связи, для которых хотя бы одно отображение является функциональным (полным или неполным), часто называют связями типа 1 : n , или «один ко многим». Бинарные связи, в которых оба отображения не являются функциональными, называют связями типа n : n , или «многие ко многим».

2.6. ОБЩАЯ ХАРАКТЕРИСТИКА ОГРАНИЧЕНИЙ ЦЕЛОСТНОСТИ

Таким образом, структурными компонентами модели данных являются отношения и связи между отношениями. Отношения задаются своими схемами. Схема отношения определяется через атрибуты, определенные на доменах. Следовательно, все допустимые значения атрибутов должны удовлетворять условию принадлежности конкретному домену.

Подобные спецификации не исчерпывают свойства данных, которые целесообразно представлять в модели данных, в соответствии с предметной областью. Необходимы дополнительные средства выражения ограничений, накладываемых на значения данных и их соотношения, которые не могут быть выражены структурой

Определение. Логические ограничения, накладываемые на данные, называются *ограничениями целостности*.

Ограничение целостности — это свойство, которое для данного множества или отношения либо истинно, либо ложно. Это свойство можно определить следующим образом: если значения данных отвечают существующим знаниям об объекте, то соответствующие ограничения логические выражения принимают значение «истина». Это значение должно сохраняться для каждого возможного состояния, в котором может находиться объект.

Спецификация ограничений целостности обладает автономией по отношению к структурным спецификациям и реализуется средствами языка описания ограничений (ЯОО, или CDL – Constraints Definition Language).

Различают спецификации статических и динамических членов и соответственно два типа способов спецификации.

Статические спецификации выражают правила, которые определяют допустимые (достоверные) состояния базы данных. В этом случае обычно используют исчисление предикатов.

Динамические спецификации определяют допустимые переходы из одного состояния базы данных в другое. Эти спецификации зависят от используемых операций ЯМД.

Верификация ограничений выполняется с учетом следующих требований:

- непротиворечивость;
 - удовлетворяемость;
 - адекватность существующим знаниям о реальном мире.

В моделировании данных особую значимость имеют обобщенные ограничения, т.е. ограничения, относящиеся ко всем объектам и реализациям, а не к одной или нескольким конкретным реализациям.

Пример ограничения целостности: зарплата подчиненных не может быть больше зарплаты руководителя. Это утверждение, являющееся ограничением целостности, должно быть справедливо в любой момент времени и не должно зависеть от изменения зарплат подчиненных и руководителей.

Введем некоторые определения.

Рассмотрим некоторую схему базы данных — S . Пусть для этой схемы указано некоторое обобщенное ограничение целостности C .

и этой схеме соответствует некоторая реализация (состояние) базы данных — DBS_k (рис. 2.7).

МД

↓

 $S \rightarrow \{C_i\}$ $\dots DBS_1 \dots DBS_k \dots$

Рис. 2.7. Соотношения между схемой базы данных, реализацией и ограничениями целостности

Тогда:

1. Ограничение C_i будет:

- правильно построено, если оно соответствует синтаксическим правилам спецификации ограничений;
- удовлетворено состоянием DBS_k , если оно истинно для DBS_k ;
- удовлетворяется, если существует некоторое DBS_j , удовлетворяющее C_i ;
- недостоверно, если никакие состояния базы данных не удовлетворяют C_i .

2. Состояние базы данных:

- удовлетворяет схеме S , если удовлетворяет всем ее ограничениям;
- непротиворечиво, если удовлетворяет всем ограничениям.

3. Схема S является:

- удовлетворяемой, если существует некоторое состояние базы данных, удовлетворяющее ей;
- противоречивой, если никакое состояние базы данных ей не удовлетворяет.

Ограничения могут быть внутренними (неявными) и явными.

Внутренние ограничения предусматриваются самой моделью данных и тесно связаны со структурой базы данных. Операции манипулирования данными хорошо согласуются с ними, поэтому контроль соблюдения внутренних ограничений не вызывает трудностей и, как правило, не требует от разработчика каких-либо дополнительных усилий. Явные ограничения задаются разработчиками; обеспечение контроля явных ограничений — серьезная проблема реализации.

Вопросы

- Сформулируйте основные свойства множества в сильно типизированной модели данных.
 - Приведите свое понимание понятия модели данных.
 - Дайте определения основных структурных компонентов модели данных.
 - Дайте определение понятия отображения, покажите влияние отображения на структурные компоненты модели данных. Приведите примеры.
 - Опишите необходимость использования логических ограничений, накладываемых на данные. Приведите примеры.

ГЛАВА 3

МОДЕЛЬ ДАННЫХ «СУЩНОСТЬ-СВЯЗЬ»

WIBY-MAN-TOBACCO-HARVARD

В модели «сущность—связь» используется более естественное представление, в соответствии с которым реальный мир состоит из *сущностей и связей*. Эта модель основывается на некоторой важной семантической информации о реальном мире. Модель может обеспечить высокую степень независимости данных и основывается на теории множеств и реляционной теории.

Модель «сущность–связь» была предложена в 1976 г. Питером Ченом (Peter Chen) и с тех пор неоднократно усовершенствовалась и самим Ченом, и многими другими.

3.1. УРОВНИ ПРЕДСТАВЛЕНИЯ ИНФОРМАЦИИ

П. Чен, разрабатывая модель данных «сущность–связь», предложил следующие четыре уровня представления информации:

- жет следующие четыре уровня представления информации:

 1. Информация, относящаяся к объектам и связям, как она существует в нашем представлении.
 2. Структура информации, т.е. организация информации, в которой объекты и связи представлены данными.
 3. Структура данных, независимая от способа доступа (т.е. не связанная со схемами поиска, индексации и т.п.).
 4. Структура данных, зависимая от способа доступа.

Модель данных «сущность–связь» поддерживает первый и второй уровни абстракции. Третий уровень абстракции поддерживает реляционная модель данных, четвертый уровень — сетевая и иерархическая модели данных. Модель данных «сущность–связь» предназначена прежде всего для проектирования баз данных; нет ни одной СУБД, которая была бы основана на этой модели данных. В связи с этим в ней рассматриваются только два основных компонента из трех: определение структуры и ограничения целостности; набор операций для этой модели данных не рассматривается.

Прежде всего рассмотрим структурные компоненты модели данных.

3.2 УРОВЕНЬ 1. ИНФОРМАЦИЯ О СУЩНОСТИХ И СВЯЗЫХ

Информация, относящаяся к объектам и связям, как она существует в нашем представлении.

В соответствии с названием основными структурными компонентами модели данных являются сущности (entity) и связи (relationship).

тами модели данных являются сущности (entity) и связи (link).
П. Чен дал следующие определения сущности и связи [6].

Определение. Сущность — нечто, принадлежащее объективной реальности, облаченное в материальную форму или форму идеи; любой объект, который может быть идентифицирован некоторым способом, отличающим его от других объектов, и информацию о котором надо хранить в базе данных.

Примеры сущностей: ЛИЧНОСТЬ, КОМПАНИЯ, СДЕЛКА.

Определение. Связь — некоторая ассоциация, устанавливаемая между двумя или более сущностями.

Примеры связей:

ОТЕЦ—СЫН — связь между сущностями **ЛИЧНОСТЬ**, в связи вовлекаются разные экземпляры одной и той же сущности.

СОТРУДНИК ПРЕДПРИЯТИЯ — связь между сущностями ЛИЧНОСТЬ и ПРЕДПРИЯТИЕ.

База данных предприятия содержит информацию о сущностях и

связях, которые представляют интерес для этого предприятия.

3.2.1. Сущности: тип сущности и множество сущностей

Будем обозначать сущности, которые существуют в нашем воображении, через $\langle e \rangle$. В соответствии с определением сущности они обладают некоторыми свойствами, позволяющими их идентифицировать. Следовательно, каждая сущность относится к некоторому отличному от других множеству сущностей. Для каждого множества сущностей определяются некоторые свойства, общие для всех сущностей из множества. Если некоторая сущность относится к определенному множеству сущностей, то эта сущность обладает свойствами, общими для всех сущностей этого множества.

Применение к сущностям приемов построения абстракции позволяет получить *тип сущности* — некоторое обобщенное представление однородных сущностей. Например, имеем некоторое множество сущностей: {МОСКВА, ОРЕЛ, КУРСК, ...}. Обобщая сущности из этого множества, получаем тип сущности: ГОРОД. Экземпляр сущности — конкретный элемент из множества, например КУРСК. Таким образом, тип сущности — это интенсионал, а конкретное множество сущностей — экстенсионал типа сущности.

С каждым множеством сущностей связывается предикат, позволяющий проверить, принадлежит ли сущность данному множеству. Следовательно, предикат входит в число свойств, общих для множества сущностей. Например, для множества сущностей **СОТРУДНИК** можно определить следующий предикат: «человек, принятый на работу». Задавая предикат, можно включать в него дополнительные условия, например: «приказ о приеме на работу ...», или «возраст сотрудника в пределах ...», или «занимает определенную должность», или что-нибудь еще.

Ниже приведены примеры множеств сущностей:

- Множество СТУДЕНТЫ: {Саша, Катя, Мария, Илья, Егор, Олег}.
- Множество ПРОЕКТЫ: {Финансовый проект, Программный проект, Аналитический проект}.
- Множество КОМПОНЕНТЫ: {Монитор, Клавиатура, Мышь, Материнская плата}.
- Множество РАБОТНИКИ: {Сергей, Елена, Ольга, Николай, Мария}.
- Множество ТЕХНОЛОГИИ: {База данных, Система управления производством, Система управления качеством, Система управления рисками}.

Будем обозначать множество сущностей через $\langle E \rangle$.

Заметим, что множества сущностей не обязаны быть непересекающимися. Например, сущность, принадлежащая множеству сущностей СТУДЕНТ, принадлежит также и множеству сущностей ЛИЧНОСТЬ. В этом случае множество сущностей СТУДЕНТ является подмножеством множества сущностей ЛИЧНОСТЬ.

3.2.2. Связи, роли и множество связей

Связь — это ассоциация, устанавливаемая между сущностями. Для связи также определяются:

- множество связей;
- тип связи;
- экземпляр связи.

Множество связей — это математическое отношение между п сущностями, каждая из которых относится к некоторому множеству сущностей.

Обозначим множество связей через R :

$$R = \{\langle e_1, e_2, \dots, e_n \rangle \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}.$$

Здесь каждый кортеж отношения $\langle e_1, e_2, \dots, e_n \rangle$ является связью (relationship).

В этом определении множества сущностей не обязаны быть различными. Например, КОМПОНЕНТ — это связь между двумя сущностями из одного и того же множества ДЕТАЛЬ.

Тип связи определяется как ассоциация типов сущностей. Например, ассоциация типов сущностей ДЕТАЛЬ; ДЕТАЛЬ определяет тип связи КОМПОНЕНТ; ассоциация типов сущностей ПРОЕКТ и СОТРУДНИК определяет тип связи ИСПОЛНИТЕЛЬ ПРОЕКТА.

Роль сущности в связи — это функция, которую сущность выполняет в данной связи. Например, в связи КОМПОНЕНТ «основная» и «составляющая» — это роли. В связи ИСПОЛНИТЕЛЬ ПРОЕКТА ролями могут быть «проект» и «исполнитель». Обозначим роли сущностей в связи через r . Упорядочение сущностей в определении связи может отсутствовать, если в связи явно указаны роли сущностей:

$\langle e_1, e_2, \dots, e_n \rangle$ — в определении связи сущности упорядочены;
 $\langle r_1/e_1, r_2/e_2, \dots, r_n/e_n \rangle$ — в определении связи сущности не упорядочены; явно указаны роли сущностей.

Рассмотрим примеры. Для типа связи КОМПОНЕНТ: или $\langle e_1, e_2 \rangle$, где точно задано, что e_1 играет роль основной детали (или агрегата), а e_2 — роль составляющей части (или компонента), или же $\langle \text{основная}/e_1, \text{составляющая}/e_2 \rangle$, и тогда порядок перечисления типов сущностей в определении типа связи не важен. Для типа связи ИСПОЛНИТЕЛЬ ПРОЕКТА: или $\langle e_1, e_2 \rangle$, где точно задано, что

e_1 — это проект, над которым работает сотрудник, а e_2 — сотрудник, исполняющий проект, или же $\langle\text{проект}/e_1, \text{исполнитель}/e_2\rangle$, и тогда порядок перечисления типов сущностей в определении типа связи не важен.

3.2.3. Атрибут, значение и множество значений

В базе данных организации необходимо хранить информацию о сущностях и связях, интересующую данную организацию. Эту информацию получают путем наблюдения или измерения и выражают множеством пар «атрибут—значение».

Примеры значений: 3, «красный», «Иван», «Иванов». Значения классифицируются и объединяются в некоторые множества значений, например:

КОЛИЧЕСТВО = {3, 28, ...};

ЦВЕТ = {красный, зеленый, ...};

ФАМИЛИЯ = {Иванов, Петров, Сидоров, ...}

Будем обозначать значение через v , а множество значений — через V .

С каждым множеством значений связывается предикат, позволяющий проверить, принадлежит ли значение этому множеству.

Определение. Атрибут может быть определен как функция, отображающая:

- множество сущностей в множество значений – $f: E_i \rightarrow V_i$;
 - или множество связей в декартово произведение множеств значений – $f: R_i \rightarrow V_{i1} \times V_{i2} \times \dots \times V_{in}$.

Особенность функции заключается в том, что каждому экземпляру сущности соответствует точно одно значение из множества значений.

Например, рассмотрим множество сущностей **СОТРУДНИК**. Информация о сущностях определяется из множеств значений **НОМЕР СОТРУДНИКА**, **ФАМИЛИЯ**, **ИМЯ**, **ОТЧЕСТВО**, **КОЛИЧЕСТВО ЛЕТ**. Определены соответствующие атрибуты (рис. 3.1).

В приведенном примере можно выделить следующие ситуации:

- атрибут может отображать множество сущностей в одно множество значений, например атрибут *Номер сотрудника*;
 - атрибут может задавать отображение множества сущностей в несколько (декартово произведение) множеств значений, например атрибут *Полное имя*;
 - разные атрибуты могут задавать отображение одного и того же множества сущностей в одно и то же множество значений (или одну и ту же группу множеств значений), например атрибуты *Возраст* и *Стаж работы*.

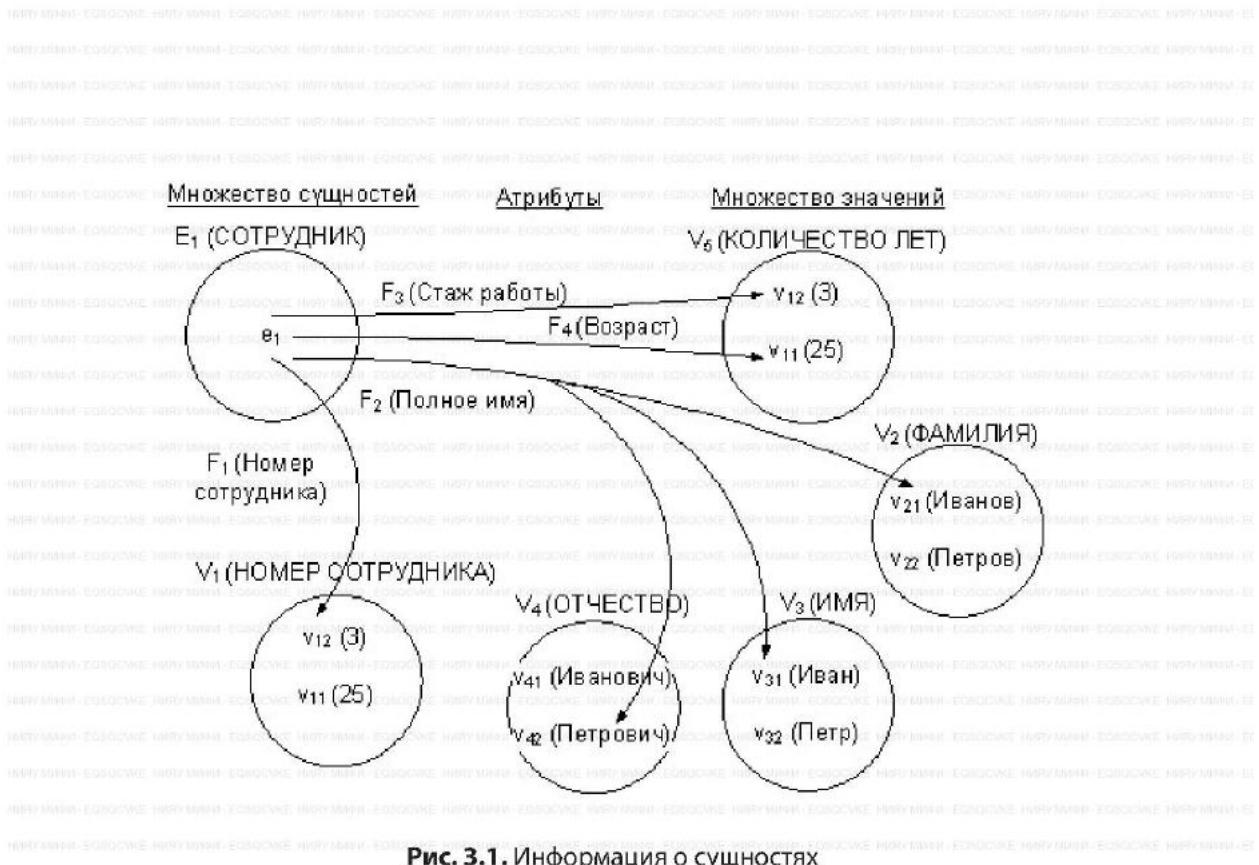


Рис. 3.1. Информация о сущностях

Таким образом:

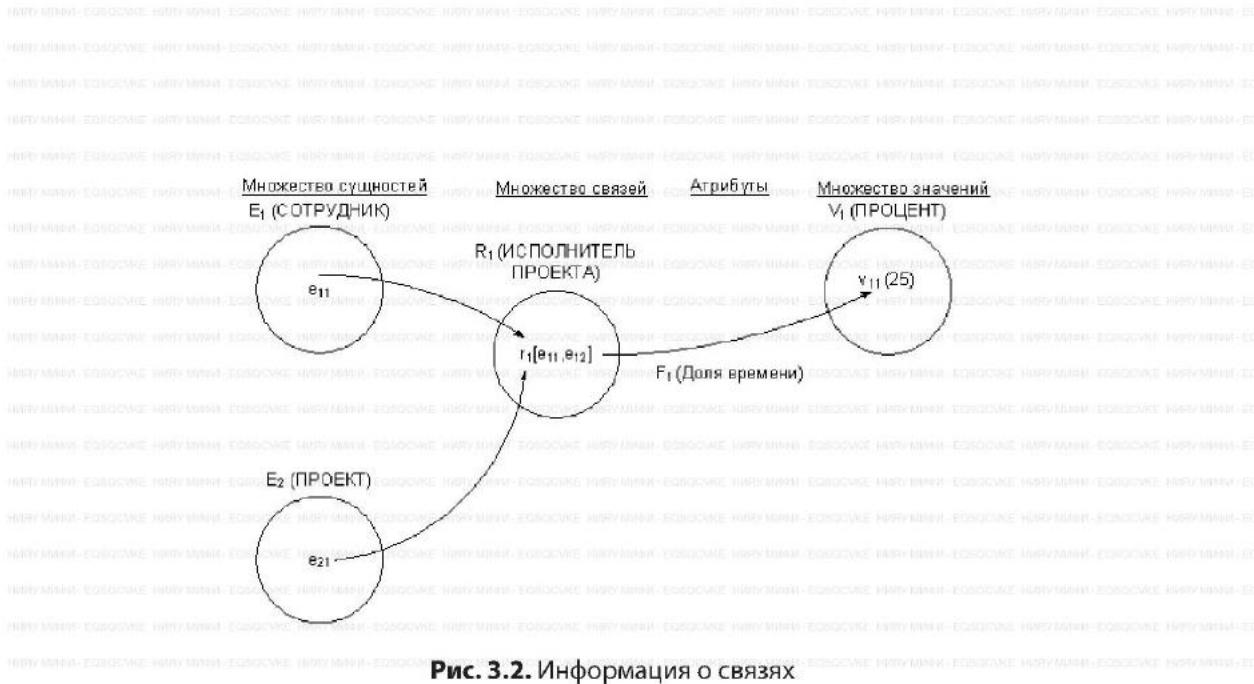
1. Атрибут и множество значений являются различными понятиями, хотя в некоторых случаях они могут иметь одно и то же имя (например, атрибут *Номер сотрудника* задает отображение множества объектов СОТРУДНИК в множество значений НОМЕР СОТРУДНИКА).

2. Атрибут определяется как функция. Следовательно, он отображает данный объект в одно значение (или одну группу значений в случае декартова произведения множеств значений).

Связи также могут иметь атрибуты.

Рассмотрим следующий пример. Определены множества сущностей СОТРУДНИК и ПРОЕКТ. Между ними определяется множество связей ИСПОЛНИТЕЛЬ ПРОЕКТА. На данном множестве связей можно определить атрибут Доля времени, представляющий долю времени, выделенную конкретному сотруднику на конкретный проект. Этот атрибут отображает связь на множество значений ПРОЦЕНТ. Данный атрибут не является ни атрибутом сущности СОТРУДНИК, ни атрибутом сущности ПРОЕКТ, так как его смысл зависит и от сотрудника, и от проекта (рис. 3.2).

Понятие атрибута связи важно для понимания семантики данных и определения функциональных зависимостей между данными.

**Рис. 3.2. Информация о связях**

3.3. УРОВЕНЬ 2. СТРУКТУРА ИНФОРМАЦИИ

Здесь рассматривается представление концептуальных объектов — сущностей, связей и значений, которые существовали в нашем воображении.

Представление значений не требует специального рассмотрения, так как существует непосредственное их представление, например: значения чисел — 12, 45, ...; значения строк — «красный», «зеленый», ...

Рассмотрим представление сущностей и связей, т.е. как организовать информацию о сущностях и связях. Информацию о сущностях и информацию о связях полезно разделять для идентификации функциональных зависимостей между данными.

3.3.1. Представление сущностей

П. Чен определил сущности как объекты, которые можно идентифицировать и отличать от других объектов; поэтому представление сущности означает ее идентификацию.

Для идентификации сущности в множестве сущностей выделяется ключ сущности — атрибут или группа атрибутов, такая, что отображение множества сущностей в одно или группу множеств значений является взаимно однозначным. Это означает, что ключ сущности однозначно определяет сущность. Например, в множестве сущностей СОТРУДНИК в качестве ключа сущности может быть выбран атрибут *Номер сотрудника*.

Ниже приведены примеры сущностей и их атрибутов:

- Сотрудник** – атрибуты: Номер сотрудника – PK, Полное имя, Возраст, Стаж работы, Количество лет.
- Фамилия** – атрибуты: Имя, Отчество.

Если не удается найти взаимно однозначное отображение на существующих данных или если желательна простота в идентификации объектов, можно искусственно определить атрибут и множество значений, обеспечивающих взаимно однозначное отображение.

Если существует несколько ключей, обычно определяют первичный ключ сущности (primary key — PK) и выбирают для него семантически значимый ключ.

Таким образом, представление сущности может быть организовано следующим образом (рис. 3.3).

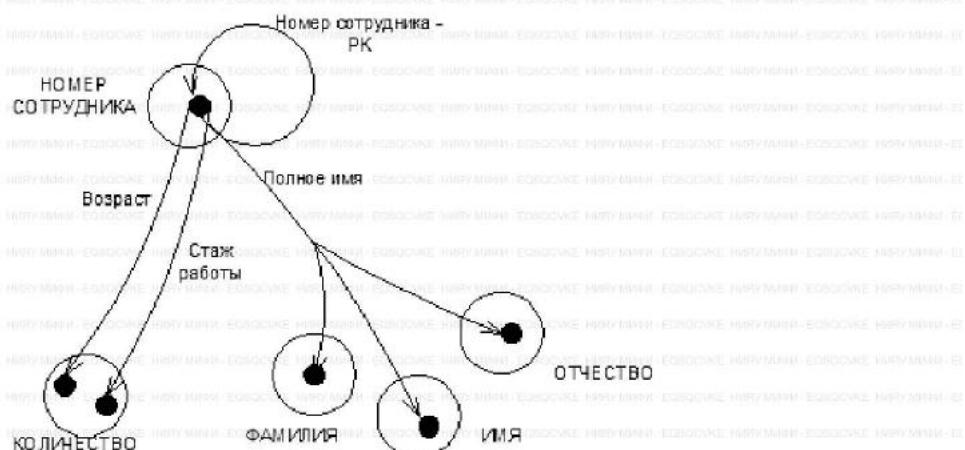


Рис. 3.3. Представление сущности

Информацию о сущностях удобно представлять в виде таблицы (табл. 3.1). Каждая строка таблицы относится к одной и той же сущности, а каждый столбец относится к множеству значений, которое, в свою очередь, относится к атрибуту. Порядок строк и столбцов не важен.

Таблица 3.1

Представление сущности

Сотрудник						
Атрибуты		Полное имя			Возраст	Стаж работы
Домены	Номер сотрудника	Фамилия	Имя	Отчество	Количество лет	Количество лет
Объект 1	128	Иванов	Иван	Иванович	25	5
Объект 2

3.3.2. Представление связи

Каждый экземпляр связи должен быть однозначно идентифицирован, и для этого также используется первичный ключ связи. Так как связь идентифицируется вовлеченными в нее сущностями, первичный ключ связи может быть представлен первичными ключами вовлеченных в связь сущностей.

Пример. Рассмотрим множество сущностей **СОТРУДНИК** с первичным ключом *Номер сотрудника* и множество сущностей **ПРОЕКТ** с первичным ключом *Номер проекта*. Оба первичных ключа могут быть определены на одном и том же домене **НОМЕР**. Рассмотрим связь **ИСПОЛНИТЕЛЬ ПРОЕКТА**. Вовлеченные в эту связь сущности представляются своими первичными ключами:

Номер сотрудника, Номер проекта.

Они и образуют первичный ключ связи ИСПОЛНИТЕЛЬ ПРОЕКТА. Важно отметить, что эти атрибуты — первичные ключи вовлеченных в связь сущностей, а не собственные атрибуты связи.

В табличном виде связи могут быть представлены следующим образом (табл. 3.2).

Таблица 3.2

Представление связи

	Первичный ключ связи	Атрибут связи
Вовлеченные в связь множества сущностей	СОТРУДНИК	ПРОЕКТ
Роль	ИСПОЛНИТЕЛЬ	ПРОЕКТ
Атрибуты сущностей	Номер сотрудника	Номер проекта
Множества значений	НОМЕР	НОМЕР
Экземпляр связи 1	128	1
Экземпляр связи 2

3.3.3. Некоторые особенности представления сущности и связи

В некоторых случаях сущности в множестве сущностей нельзя
уникально идентифицировать значениями только собственных ат-
рибутов; для уникальной идентификации необходимо использовать
и связь.

Рассмотрим пример. В некоторой организации для выполнения определенных сезонных работ формируются специальные временные бригады, которые могут быть в дальнейшем расформированы и созданы заново. Бригада состоит из руководителя и членов бригады, и все они являются сотрудниками данной организации. Один и тот же сотрудник может быть членом нескольких бригад. Бригада идентифицируется руководителем бригады.

Соответственно можно определить множества сущностей СОТРУДНИК и ЧЛЕН БРИГАДЫ и связь между этими сущностями НАЗНАЧЕНИЕ (сотрудник назначается членом бригады).

Для уникальной идентификации сотрудника в качестве первичного ключа можно использовать атрибут *Номер сотрудника*. Но этот атрибут не может идентифицировать данного сотрудника как члена некоторой конкретной бригады; необходимо еще указать, членом какой бригады является сотрудник (т.е. указать номер руководителя бригады). Следовательно, первичный ключ множества сущностей **ЧЛЕН БРИГАДЫ** включает номер сотрудника и номер руководителя бригады, т.е. учитывается связь **НАЗНАЧЕНИЕ**.

Таким образом, имеем два способа идентификации сущностей: только значениями собственных атрибутов или с учетом связей. В соответствии с этим определяются две формы отнесения сущностей:

- ответствий с этим определяются две формы отношения сущностей:

 - регулярное отношение сущности, если для идентификации сущностей используются только собственные атрибуты;
 - слабое отношение сущности, если для идентификации сущностей используются связи.

Соответственно, связи, в которые вовлечены слабые отношения сущностей, являются слабым отношением связи. Из этого следует, что любая связь между множеством сущностей ЧЛЕН БРИГАДЫ и любыми другими множествами сущностей (в том числе и множеством сущностей СОТРУДНИК) представляет собой слабое отношение связи.

Проведение различия между слабыми и регулярными отношениями сущности и связи полезно для поддержки ограничений целостности.

3.3.4 Язык отмечения геномных последовательностей «сущности» (см.)

Поскольку модель данных «сущность–связь» используется прежде всего для проектирования баз данных, в качестве языка описания данных был предложен графический язык диаграмм. П. Чен предложил следующие графические элементы для представления модели «сущность–связь» (рис. 3.4).



Рис. 3.4. Графические элементы для диаграммы

Линии, соединяющие сущности и связи, определяют, какие сущности вовлечены в ту или иную связь. Подписи линий определяют роли сущностей в связях. Атрибуты присоединяются к соответствующим сущностям и связям. Первичные ключи подчеркиваются.

3.3.5. Общая характеристика связей

1. Множество связей может быть определено более чем на двух множествах сущностей, например на трех — тернарная связь (рис. 3.5).

Здесь связь ПОСТАВКА определяет, какой поставщик поставляет определенную деталь для определенного проекта.



Рис. 3.5. Связь, определенная на трех множествах сущностей

2. Множество связей может быть определено на одном множестве сущностей (рис. 3.6).

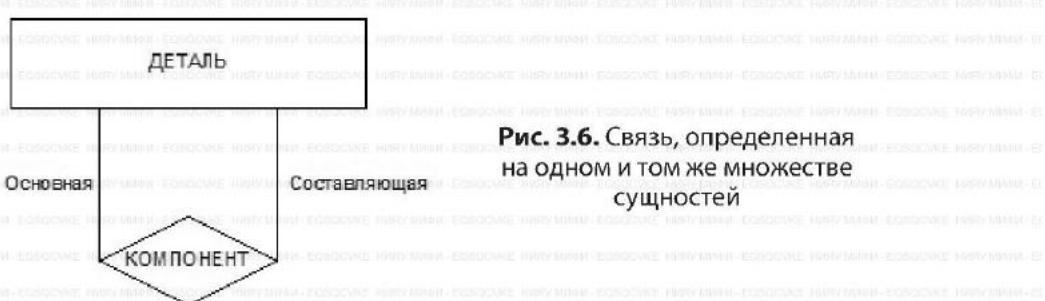


Рис. 3.6. Связь, определенная на одном и том же множестве сущностей

В этом случае необходимо указывать роль сущности в каждой связи. В данном примере деталь может быть основной для другой детали (например, клавиатура компьютера — основная деталь для клавиши) или может быть составляющей для другой детали (например, та же клавиатура для компьютера).

3. Может существовать несколько множеств связей, определенных на одних и тех же множествах сущностей (рис. 3.7).

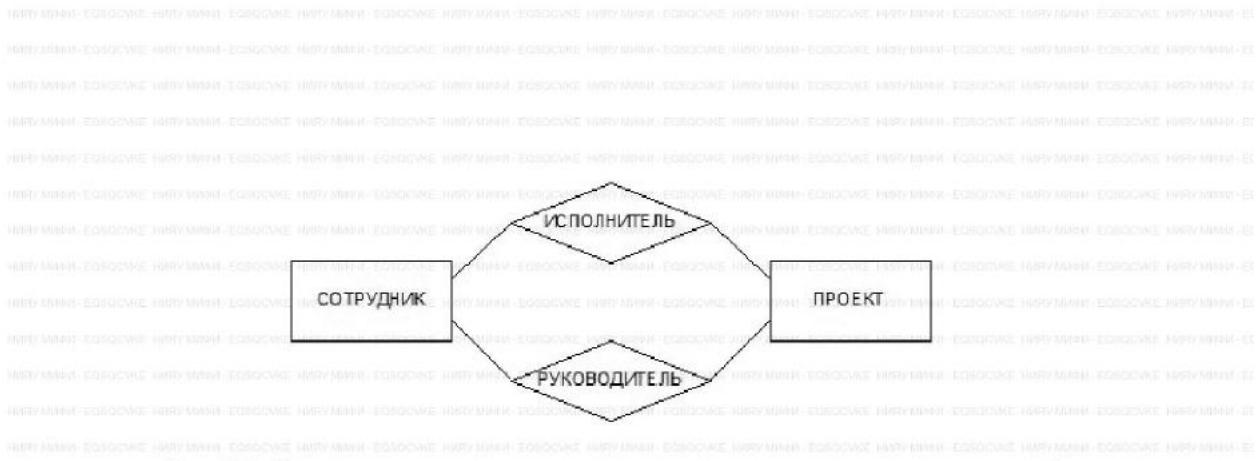


Рис. 3.7. Две связи, определенные на одних и тех же множествах сущностей

4. В диаграмме можно различать связи типа 1:п и п:н. Необходимый символ (1 или N) указывается у соответствующего множества сущностей, вовлеченного в связь (рис. 3.8).



Рис. 3.8. Указание типа связи

5. В диаграмме можно указать слабое отношение сущности (рис. 3.9).

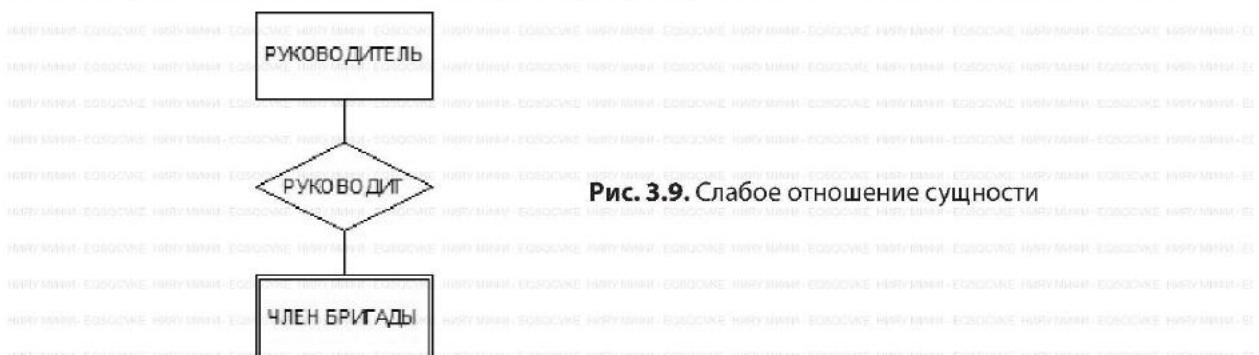


Рис. 3.9. Слабое отношение сущности

Пример диаграммы «сущность–связь» (рис. 3.10).

38

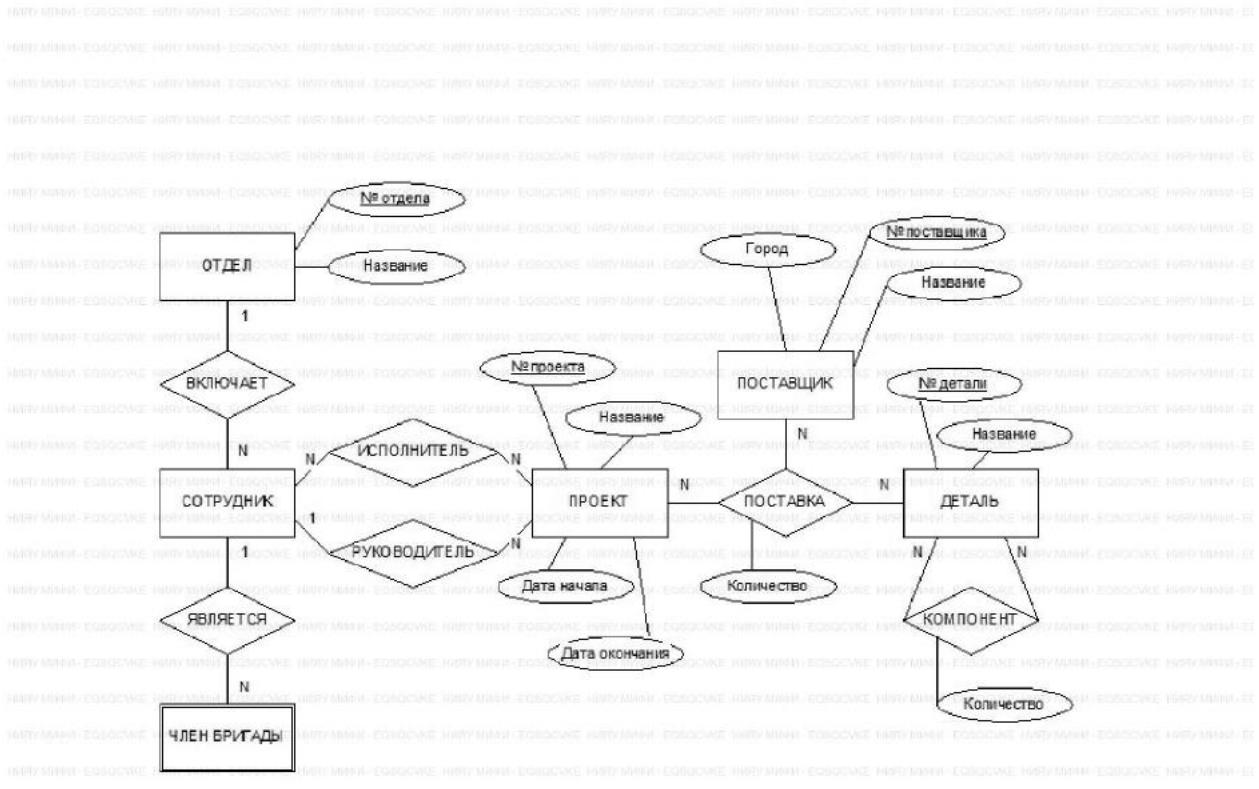


Рис. 3.10. Пример диаграммы «сущность–связь»

3.4. РАСШИРЕНАЯ МОДЕЛЬ ДАННЫХ «СУЩНОСТЬ–СВЯЗЬ»: НОТАЦИЯ IDEF1X

Как видно из рис. 3.10, иллюстрация концептуальной схемы базы данных с использованием графических элементов, предложенных П. Ченом, несколько громоздка. Поэтому были предложены и другие нотации для модели «сущность–связь». В данном издании используется нотация IDEF1x. Далее приводится краткое описание данной нотации. Подробную информацию можно найти в соответствующих изданиях и публикациях.

В нотации IDEF1x используются графические элементы для представления диаграммы «сущность–связь», несколько отличающиеся от элементов, введенных П. Ченом; кроме того, накладываются некоторые дополнительные ограничения. Но в главном сохраняются все положения модели, предложенной П. Ченом.

Основными структурными компонентами остаются **сущности** и **связи**.

Сущность также представляется прямоугольником. Регулярное отношение сущности модели П. Чена в IDEF1x называется **сущностью, независимой по идентификации**, и представляется **прямоугольником с прямыми углами**. Слабое отношение сущности в IDEF1x называется **сущностью, зависимой по идентификации**, и пред-

Допускается представление связей типа 1:n и n:n. Связи типа 1:n в IDEF1x называются определенными связями, типа n:n — неопределенными. Неопределенные связи могут быть использованы только на начальных этапах проектирования схемы базы данных и представляются линиями с жирными точками на двух концах связи (рис. 3.12). В окончательном варианте все неопределенные связи должны быть заменены определенными в соответствии с правилами, установленными в IDEF1x.

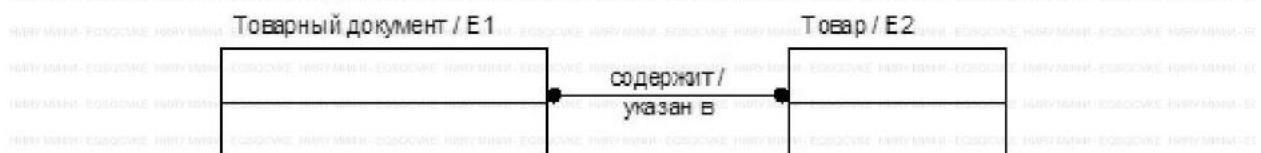


Рис. 3.12. Представление в IDEF1x неопределенной связи

В определенной связи сущность, находящаяся на конце связи «один», называется родительской, на конце «много» — дочерней. Определенная связь именуется только для одного отображения — родительской сущности в дочернюю. В дочерней сущности появляются специальные атрибуты, представляющие связь — внешний ключ, помеченный аббревиатурой FKm.n (Foreign Key), где m — номер внешнего ключа (одна сущность может участвовать в нескольких связях), n — номер атрибута в составе внешнего ключа (для составных ключей). Атрибуты внешнего ключа в дочерней сущности появляются в результате миграции (передачи) по связи атрибутов первичного ключа родительской сущности.

Определенные связи в IDEF1x соответствуют связям, в которых одно из отображений — функциональное. В IDEF1x различаются следующие разновидности определенной связи.

Определенная неидентифицирующая обязательная связь (одно из отображений — полное функциональное) представляется пунктирной линией с жирной точкой на конце дочерней сущности. Дочерняя сущность, вовлеченная в такую связь, идентифицируется только собственными атрибутами и представляет собой сущность, независимую по идентификации; поэтому атрибуты связи (внешнего ключа) размещаются в области прочих атрибутов. Каждый экземпляр дочерней сущности обязательно связан с одним экземпляром родительской сущности.

Пусть, например, определены сущности СОТРУДНИК (см. пример на рис. 3.11), ОТДЕЛ с атрибутами *Номер отдела* (первичный ключ) и *Название* (ключ) и ДОЛЖНОСТЬ с атрибутами *Код должности*, *Квалификация* (составной первичный ключ), *Название долж-*

ности, Оклад. Между сущностями ОТДЕЛ и СОТРУДНИК определена связь «состоит из/зачислен в», а между сущностями СОТРУДНИК и ДОЛЖНОСТЬ — связь «замещает/замещается»:

- должность замещается нулем или более сотрудников;
 - сотрудник замещает одну (конкретную) должность.

Сотрудник обязательно должен быть зачислен в отдел и должен замещать определенную должность. Такие связи представляются следующей диаграммой (рис. 3.13).



Рис. 3.13. Представление определенной неидентифицирующей обязательной связи

Определенная неидентифицирующая необязательная связь (одно из отображений — неполное функциональное) представляется пунктирной линией с жирной точкой на конце дочерней сущности и прозрачным ромбиком на стороне родительской сущности. Дочерняя сущность, вовлеченная в такую связь, также идентифицируется только собственными атрибутами и представляет сущность, независимую по идентификации. Но в данном случае могут существовать экземпляры дочерней сущности, не связанные ни с одним экземпляром родительской сущности.

Рассмотрим пример. Пусть определены сущности КЛАСС с атрибутами *Год обучения*, *Группа* (составной первичный ключ), *Дата формирования*, *Категория* и УЧИТЕЛЬ с атрибутами *Личный номер* (первичный ключ), *Фамилия И.О.*, *Ставка*. Между этими сущностями определена связь «назначен классным руководителем/имеет классного руководителя»:

- учитель назначен классным руководителем в нуль или более классов;
 - класс имеет одного классного руководителя или не имеет классного руководителя.

Такая связь может быть представлена диаграммой, приведенной ниже (рис. 3.14).

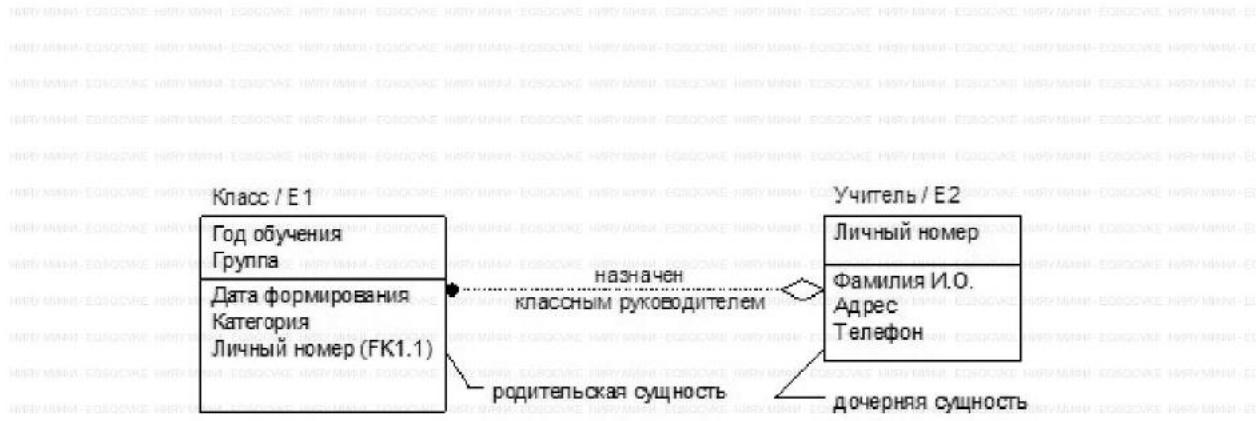


Рис. 3.14. Представление определенной неидентифицирующей необязательной связи

Определенная идентифицирующая обязательная связь (одно из отображений — полное функциональное) представляется сплошной линией с жирной точкой на конце дочерней сущности. Дочерняя сущность, вовлечеченная в такую связь, идентифицируется с учетом связи и представляет собой сущность, зависимую по идентификации. Атрибуты внешнего ключа в дочерней сущности, зависимой по идентификации, размещаются в области первичного ключа.

Пусть, например, определены сущность **СОТРУДНИК** в соответствии с примером, приведенным на рис. 3.11, и сущность **ЧЛЕН БРИГАДЫ** с атрибутами *Табельный номер сотрудника*, *Номер бригады*, *Дата зачисления в бригаду*, *Дата отчисления из бригады*. Между этими сущностями определена связь «является/является»:

- сотрудник **является** членом нуля или более бригад;
- член бригады **является** сотрудником.

Для идентификации экземпляра сущности **ЧЛЕН БРИГАДЫ** необходимо учитывать связь между этими сущностями (рис. 3.15).

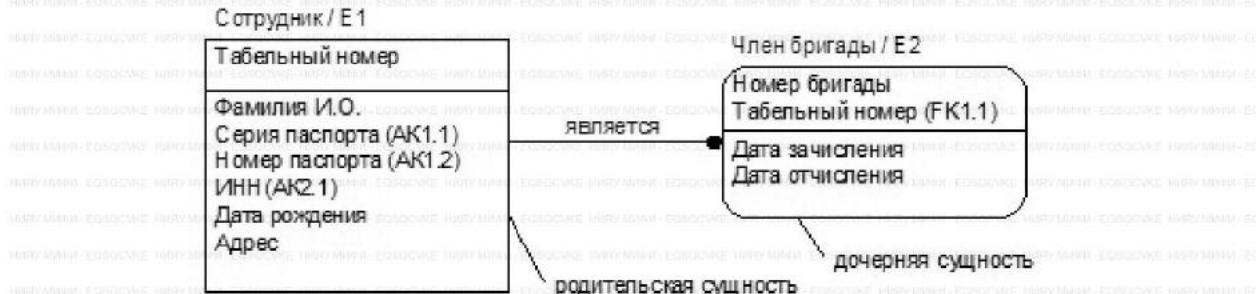


Рис. 3.15. Представление определенной идентифицирующей обязательной связи

Диаграмма «сущность—связь», соответствующая модели П. Чена (рис. 3.10), в нотации IDEF1x будет иметь следующий вид (рис. 3.16). Следует отметить, что связи типа n:n, допустимые в модели (и на

Ниже приведены примеры диаграмм «сущность–связь» в нотации IDEF1x.

диаграмма) П. Чена, в нотации IDEF1x представляются дополнительными сущностями (часто зависимыми по идентификации) с двумя связями типа 1:n.

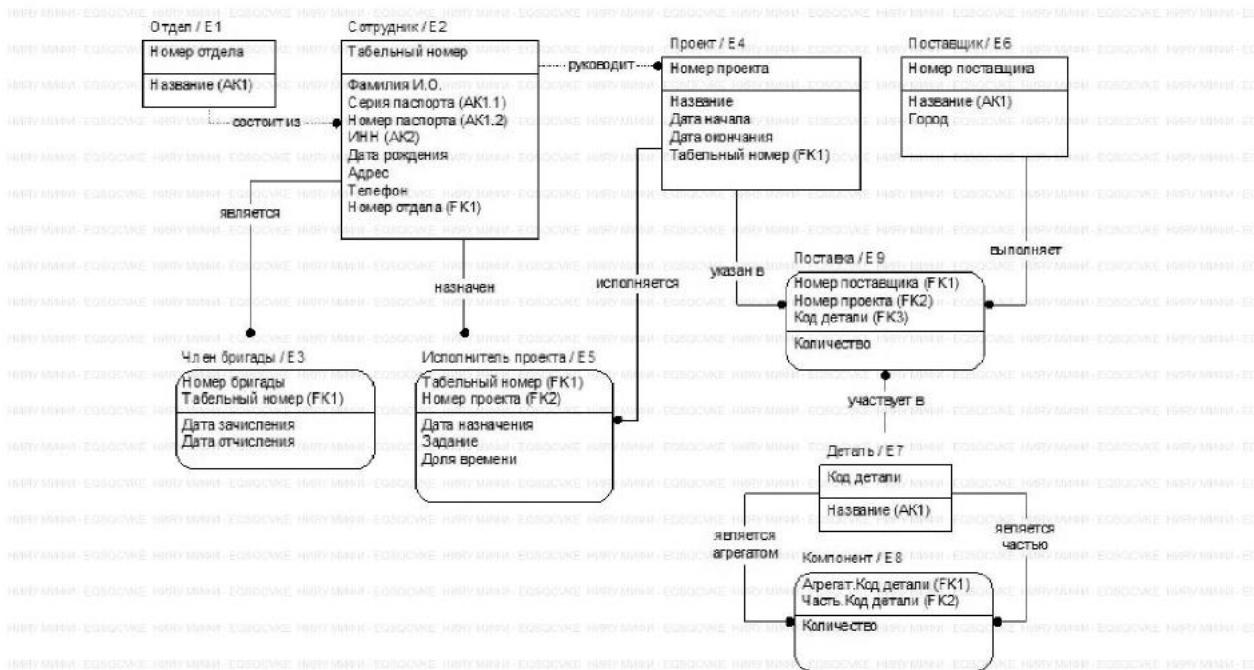


Рис. 3.16. Пример диаграммы «сущность–связь» в нотации IDEF1x

3.5. ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ В МОДЕЛИ «СУЩНОСТЬ–СВЯЗЬ»

Дополнительные ограничения целостности, накладываемые на данные, важны для описания используемых данных. Следует понимать, какие ограничения поддерживаются непосредственно используемой моделью данных, а какие требуют дополнительных усилий со стороны разработчика. Поэтому прежде всего следует определить типы ограничений целостности, а уже затем, учитывая, что модель «сущность–связь» используется только на этапе проектирования базы данных, — язык, с помощью которого можно наглядно представить сформулированные ограничения.

В общем случае в модели данных «сущность–связь» П. Чена рассматриваются следующие типы ограничений.

Ниже приведены примеры ограничений на атрибуты:

- Ограничения на допустимые значения в множестве значений**
- Ограничения на разрешенные значения некоторого атрибута**
- Ограничения на существующие значения в базе данных**

Ограничения на допустимые значения в множестве значений

Атрибут отображает сущность из множества сущностей на множество значений. Допустимые значения определяются значениями в соответствующем множестве значений. Например, определено множество сущностей СОТРУДНИК с атрибутами *Возраст* и *Стаж работы* (рис. 3.17). Оба атрибута определены на домене КОЛИЧЕСТВО ЛЕТ, для которого определено следующее условие принадлежности: целые числа не меньшие 0 и не большие 70. Соответственно, значения указанных атрибутов будут черпаться из диапазона чисел 0–70.

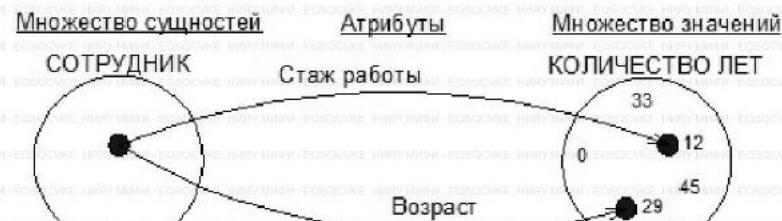


Рис. 3.17. Ограничения на допустимые значения

Ограничения на разрешенные значения некоторого атрибута

В отдельных случаях не все допустимые значения из множества значений являются разрешенными для некоторых атрибутов. Например, для того же множества сущностей разрешенное значение атрибута *Возраст* может быть определено условием: возраст сотрудника не менее 17 и не более 70 лет. Данное ограничение задается с помощью утверждения на естественном языке (возраст любого сотрудника не может быть менее 17 и более 70 лет) или предиката:

$\text{Возраст}(e) \geq 17 \text{ и } \text{Возраст}(e) \leq 70 \mid \text{для любого } e \in \text{СОТРУДНИК}$

Ограничения на существующие значения в базе данных

Здесь можно выделить две ситуации:

Ограничения на конкретные значения. Например, для того же множества сущностей СОТРУДНИК даны дополнительные атрибуты *Зарплата* и *Отчисления*, определенные на одном домене ДЕНЬГИ. Атрибут *Зарплата* определяет ежемесячный доход сотрудника, а атрибут *Отчисления* — также ежемесячные выплаты с дохода (например, налоги, плата за кредит и т.п.). Может быть определено следующее условие: отчисления не должны превышать зарплату сотрудника:

$\text{Отчисления}(e) < \text{Зарплата}(e) \mid \text{для любого } e \in \text{СОТРУДНИК}$

45

Другой пример. Некоторое предприятие состоит из нескольких отделов. Определены множества сущностей ПРЕДПРИЯТИЕ с атрибутом *Бюджет* и ОТДЕЛ также с атрибутом *Бюджет*; определен общий бюджет предприятия, в соответствии с которым определяются бюджеты отделов. Определяется условие: бюджет предприятия не должен быть меньше суммы бюджетов отделов:

Бюджет (e_1) $\geq \Sigma$ Бюджет ($e_{2,i}$) | для любого $e_1 \in$ ПРЕДПРИЯТИЕ
И всех $e_2 \in$ ОТДЕЛ предприятия e_1 .

Ограничения на множества значений. Этот тип ограничений позволяет определить условия принадлежности сущностей какому-то множеству в зависимости от существования каких-либо сущностей в другом множестве. Например, пусть определено множество сущностей СТУДЕНТ с атрибутом *Имя* и нужно определить множество сущностей РАБОТАЮЩИЙ СТУДЕНТ также с атрибутом *Имя*. Второе множество сущностей является подмножеством первого:

**Имя (е) | для любого $e \in$ РАБОТАЮЩИЙ СТУДЕНТ входит в Имя
(е) | $e \in$ СТУДЕНТ**

Вопросы

- Сформулируйте уровни представления информации и укажите модели данных, используемые на каждом уровне.
 - Приведите примеры для каждого уровня представления информации в модели данных «сущность–связь» П. Чена.
 - Опишите основные структурные компоненты, используемые в модели «сущность–связь», предложенной П. Ченом.
 - Сформулируйте понятие сущность в модели данных «сущность–связь» П. Чена. Приведите примеры пересекающихся и непересекающихся множеств сущностей.
 - Сформулируйте понятие связи в модели данных «сущность–связь» П. Чена. Приведите примеры отношений, ассоциаций и функций между сущностями.
 - Опишите все возможные ситуации представления атрибутов в модели данных «сущность–связь» П. Чена.
 - Перечислите типы связей в нотации IDEF1x. Дайте определения всем разновидностям определенных связей нотации IDEF1x.
 - Опишите все возможные ограничения, которые накладываются на данные в нотации IDEF1x.
 - Приведите основные отличия между моделью данных «сущность–связь», предложенной П. Ченом, и расширенной моделью данных «сущность–связь» в нотации IDEF1x.

ГЛАВА 4

ИЕРАРХИЧЕСКАЯ И СЕТЕВАЯ МОДЕЛИ ДАННЫХ

ГЛАВА 4

ИЕРАРХИЧЕСКАЯ И СЕТЕВАЯ МОДЕЛИ ДАННЫХ

4.1. ИЕРАРХИЧЕСКАЯ МОДЕЛЬ ДАННЫХ

Иерархическая модель данных (Hierarchical database model), как правило, представлена в виде несбалансированной древовидной структуры, имеющей однозначно определенные сегменты родитель–потомок. Такая структура предполагает достаточно высокий уровень избыточности данных, обычно в дочерних сегментах. Данные представлены набором записей со значениями. Каждый экземпляр записи определен соответствующим типом записи. Тип записи является эквивалентом таблицы реляционной модели данных. Экземпляр записи иерархической модели данных — строка реляционной таблицы. Связь между типами записей представляется иерархическим отношением родитель–потомок с кардинальностью 1:N.

Более формально, определения элементов иерархической модели можно сформулировать следующим образом:

- тип записи — определенная пользователем совокупность атрибутов;
 - экземпляр типа записи — конкретная реализация типа записи определенной длины, интерпретируемая соответствующим образом;
 - иерархия — набор типов записей, представленных в виде древовидной структуры;
 - иерархическая база данных — набор экземпляров типа записей, таких, что каждый экземпляр (за исключением экземпляра, являющегося корнем иерархии) имеет в точности одного родителя.

На рис. 4.1 определены четыре типа записи и соответствующие им элементы данных, образующие иерархическую структуру.



Рис. 4.1. Иерархическая структура. Предметная область — прокат автомобилей

Пример реализации представлен на рис. 4.2.

Пример реализации представлен на рис. 4.2.

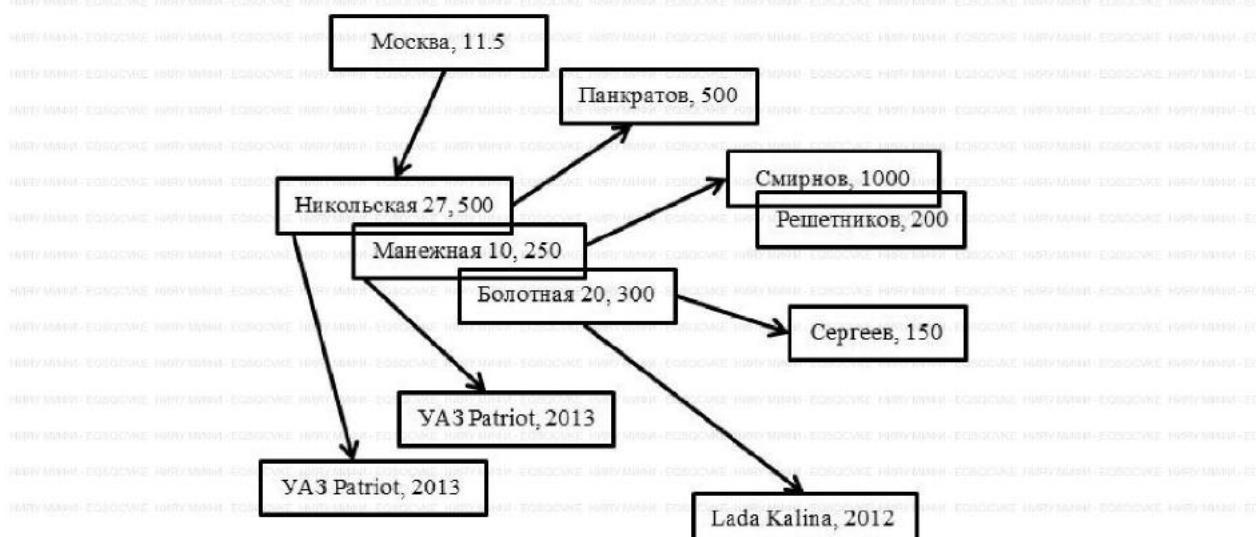


Рис. 4.2. Пример реализации иерархической структуры

4.2. СЕТЕВАЯ МОДЕЛЬ ДАННЫХ

Сетевая модель данных (Network Data Model), также известная как CODASYL Data Model, позволяет моделировать ситуации, при которых потомок может быть подчинен более чем одному родителю, обеспечивая тем самым реализацию отношений многие-ко-многим. В основе сетевой модели лежит концепция множеств (наборов), которая также представляется в терминах иерархической модели: множеством типов записей и множеством экземпляров типов записей. Один и тот же экземпляр типа записи может быть вовлечен в отношение с более чем одним набором типов записей, что и обеспечивает так называемый *multiparent* — концепцию, когда один и тот же подчиненный экземпляр типа записи может относиться одновременно к нескольким родительским. Связь между типами записей представляется отношением родитель–потомок, в котором один (родительский) экземпляр типа записи является владельцем и ноль, один или несколько экземпляров (дочерние) типа записи являются потомками. Причем один и тот же экземпляр типа записи может быть вовлечен в отношения с более чем одним родительским экземпляром разных типов записи, являясь тем самым потомком нескольких родителей.

Ниже приведены определения элементов сетевой модели:

Формально определения элементов сетевой модели можно сформулировать следующим образом:

- тип записи — определенная пользователем совокупность атрибутов;
- экземпляр типа записи — конкретная реализация типа записи определенной длины, интерпретируемая соответствующим образом;
- тип набора — бинарное отношение между одним родительским экземпляром (владельцем) типа записи и несколькими (ноль или более) дочерними экземплярами другого типа записи, так что для одного и того же экземпляра дочернего типа записи могут существовать отношения с экземплярами типов записей других владельцев;
- сетевая база данных — набор экземпляров типов записей, таких, что каждый экземпляр может иметь одного или нескольких родителей.

Иерархическая модель (см. рис. 4.1), преобразованная в сетьевую структуру, представлена на рис. 4.3. В данном примере избыточность сокращается за счет наличия у типа записи «Офис» двух родителей: «Город» и «Марка АМ».

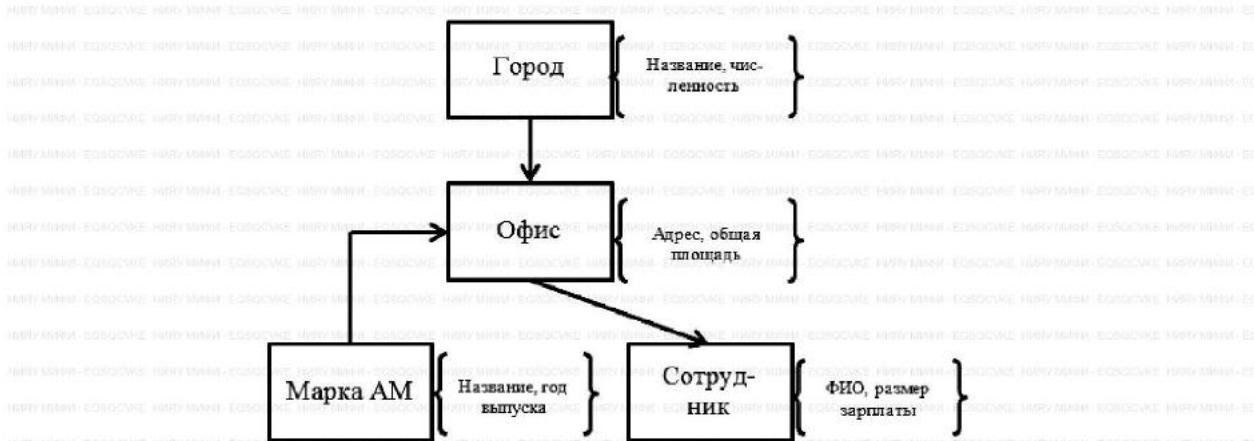


Рис. 4.3. Сетевая структура. Предметная область — прокат автомобилей

Пример реализации представлен на рис. 4.4.

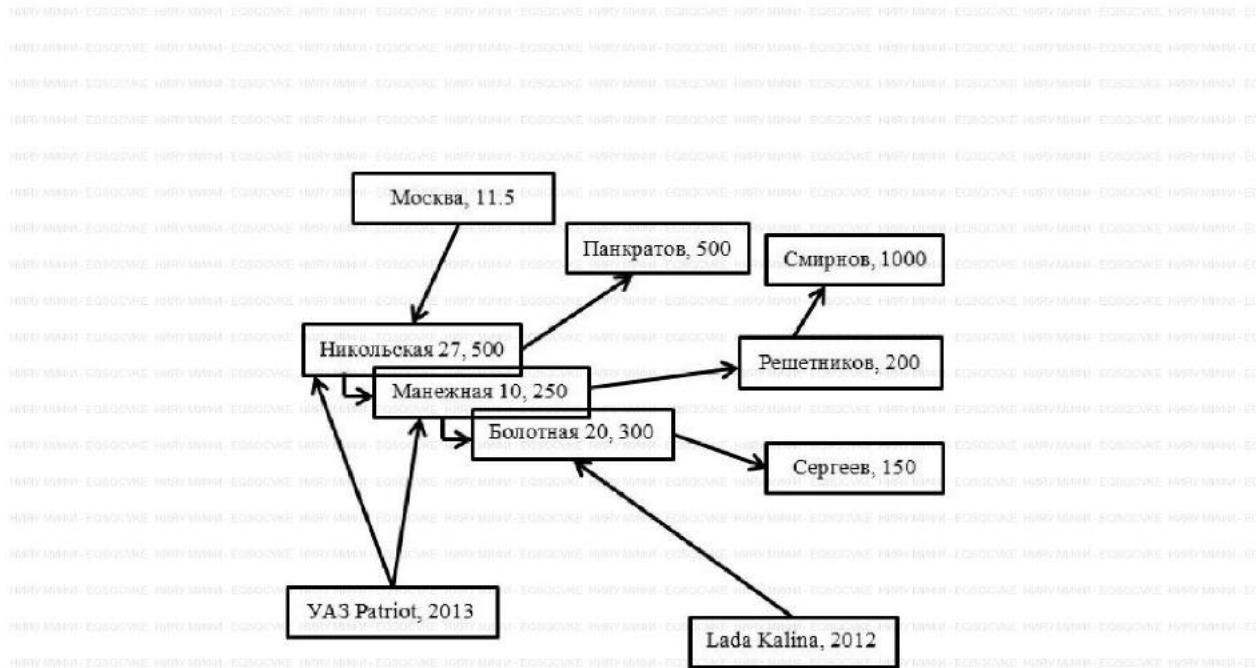


Рис. 4.4. Пример реализации сетевой структуры

Вопросы

- Сформулируйте основные отличия иерархической и сетевой моделей данных.
- Определите основные достоинства и недостатки иерархической и сетевой моделей данных.
- Опишите элементами иерархической и сетевой моделей данных фрагмент знаний вам предметной области.

5.1. БАЗОВЫЕ СТРУКТУРНЫЕ КОМПОНЕНТЫ РЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ

Базовыми структурными компонентами РМД являются:

- домены и атрибуты;
 - отношения;
 - связи.

5.1.1. Домены, атрибуты и отношения

Определение. Домен — множество элементов одного типа.

Э.Ф. Кодд определил простой домен, элементы которого имеют простые (атомарные) значения, и составной домен, элементы которого представляют собой отношения, построенные на простых доменах.

Примеры простых доменов:

ГОЛ = {1985, 2003, 2000};

Год (1985, 2005, 2009),
ДЕНЬГИ = {500, 1000, 850}

Пример составного домена, построенного на простых доменах

Год и деньги: ИСТОРИЯ ЗАРПЛАТЫ = {{<1985, 500>, <2000, 1000>}, {<2000, 850>, <1985, 850>, <2000, 500>, <2003, 1000>}}

В данном примере значением одного элемента составного домена является множество пар вида <ГОД, ДЕНЬГИ>

Отношение реляционной модели определяется в соответствии с его определением в теории множеств.

Определение. Пусть дана совокупность множеств D_1, D_2, \dots, D_n , не обязательно различных. Тогда отношение R , определенное на этих множествах, есть множество упорядоченных кортежей $\langle d_1, d_2, \dots, d_n \rangle$, где $d_i \in D_i$ для каждого $i = 1, 2, \dots, n$.

В реляционной модели данных множества D_i представляют собой

Пример: даны два домена $D_1 = \{a, b, c\}$ и $D_2 = \{1, 2\}$. Отношением, построенным на данных доменах, может быть $R_1 = \{\langle a, 2 \rangle, \langle c, 1 \rangle\}$. Другое отношение, построенное на этих же доменах: $R_2 = \{\langle a, 2 \rangle, \langle b, 2 \rangle, \langle a, 1 \rangle\}$.

Свойства отношения: *Симметрическое, рефлексивное, транзитивное.*

- кортежи отношения не упорядочены;
 - домены внутри кортежей упорядочены.

Определение. Атрибуты задают способ использования домена внутри отношения.

В связи с введением понятия атрибута в реляционной модели данных вводится понятие схемы отношения.

52. *Следует отметить, что в этом случае, если вспомогательный генетический материал не содержит генов, влияющих на синтез белка, то генетическая структура генома не изменяется.*

Другими словами, в нормализованном отношении могут быть использованы только простые домены. В соответствии с данным определением приведенный пример (см. рис. 5.1) представляет не-нормализованное отношение, которое не допускается в реляционной модели данных.

Ненормализованные отношения очень легко преобразовать в нормализованные. Для этого в схеме отношения составной домен заменяется составляющими его простыми доменами, а в реализации отношения значения атрибутов, определенных на других доменах, повторяются для каждого кортежа составного домена. Так, нормализация приведенного выше отношения даст следующее — нормализованное — отношение, представленное на рис. 5.2.

Номер сотрудника	Имя	Зарплата
1	Иванов	1985, 500
1	Иванов	2000, 1000
2	Петров	1985, 850
2	Петров	2000, 500
2	Петров	2003, 1000

Рис. 5.2. Нормализованное отношение СОТРУДНИК

Свойства отношения реляционной модели данных

1. Каждый атрибут отношения имеет уникальное в данном отношении имя.
 2. Каждый атрибут определен на каком-то одном домене.
 3. На одном и том же домене может быть определено несколько атрибутов.
 4. Имя атрибута может совпадать с именем домена.
 5. Порядок следования атрибутов не устанавливается (атрибуты в определении схемы отношения не упорядочены).
 6. В отношении нет совпадающих кортежей (каждый кортеж уникален).
 7. Порядок следования кортежей не устанавливается (кортежи в отношении не упорядочены).
 8. Отношение имеет имя, которое в схеме базы данных отличается от имен всех других отношений.

Примечание: часто в качестве доменов используются интуитивно понятные множества: например, в предыдущем примере интуитивно ясно, что *Номер отдела* — это число, а *Название* — это строка. В соответствии с этим в схеме отношения часто опускается указание имени домена:

БОЛСУЧЕ - НИВА МИНИ - БОЛСУЧЕ - НИВА АКРИЛ - БОЛСУЧЕ - НИВА МЮЛ-БОЛСУЧЕ

В реляционной модели данных отношение представляет собой единственный структурный компонент, используемый и для представления сущности, и для представления связи.

5.1.2. Представление сущности

Представление сущности означает возможность уникальной идентификации каждого отдельного кортежа отношения по значениям его атрибутов. Так как в отношении нет дубликатов кортежей, каждый кортеж уникален и может быть идентифицирован значениями всех своих атрибутов. Тем не менее, как правило, в кортежах можно выделить некоторое подмножество атрибутов, значения которых уникальны для всех реализаций отношения — настоящих, бывших в прошлом и будущих. Такие подмножества атрибутов представляют собой ключи.

Определение. Ключ — это совокупность атрибутов, которая однозначно идентифицирует каждый кортеж данного отношения.

Ключ может содержать дополнительные атрибуты, которые не обязательны для уникальной идентификации кортежа. Поэтому в РМД вводится понятие первичного ключа, состоящего только из тех атрибутов, которые действительно необходимы для уникальной идентификации кортежа.

Определение. *Первичный ключ (PK – Primary Key) – неизбыточный набор атрибутов, значения которых однозначно определяют кортеж отношения.*

Первичный ключ неизбыточен, если:

- а) состоит из одного атрибута;
б) состоит из нескольких атрибутов, но ни один из этих атрибутов не является лишним для однозначной идентификации каждого кортежа (если ключ состоит из нескольких атрибутов, он называется *составным*).

Таким образом, в соответствии с определением первичный ключ отношения обладает следующими двумя свойствами:

- уникальность — каждый кортеж в отношении единственным образом идентифицируется значением ключа;
 - неприводимость — никакое собственное подмножество ключа не обладает свойством уникальности.

Отношение может иметь только один первичный ключ. Если в отношении можно выделить несколько наборов атрибутов, каждый из которых уникально и неизбыточно идентифицирует каждый кортеж отношения, в таком случае один из них выбирается в качестве первичного ключа, а все остальные являются альтернативными ключами (AK — Alternate Key). Например, в отношении

КАФЕДРА (Номер кафедры, Название)

КАФЕДРА (Номер кафедры, Название)

каждый из атрибутов уникально идентифицирует каждый кортеж.

Если в качестве первичного ключа выбран атрибут *Номер кафедры*, тогда атрибут *Название* является альтернативным ключом.

В схеме отношения первичный ключ будем подчеркивать, а после альтернативных ключей добавлять аббревиатуру АК:

КАФЕДРА (*Номер кафедры*, *Название (АК)*)

КАФЕДРА (Номер кафедры, Название (АК)).

5.1.3. Связи

Связи между сущностями отражают взаимосвязи между конкретными экземплярами сущностей. Эти взаимосвязи представляются с помощью внешних ключей.

Определение. Внешний ключ (FK — Foreign Key) — это атрибут или некоторое множество атрибутов отношения R1, которые не являются собственными атрибутами отношения R1, но их значение совпадает со значениями первичного ключа некоторого отношения R2 (возможность идентичности R1 и R2 не исключается).

Атрибуты внешнего ключа представляют собой некоторые дополнительные атрибуты, которые не определяют саму сущность, но позволяют установить ассоциации с другой сущностью.

Основными типами связей между сущностями являются связи 1:n и n:n. Рассмотрим представление этих связей. Начнем со связи типа 1:n.

- СОТРУДНИК с атрибутами *Номер сотрудника* (первичный ключ), *Имя* и *Год рождения*;

- ОГДЕЛ с атрибутами *Номер отдела* (первичный ключ) и *Название*. Между этими отношениями определена связь типа 1:n — каждый сотрудник работает в одном определенном отделе, и в каждом отделе работают много сотрудников. На рис. 5.3 приведена соответствующая

Эта связь определяется атрибутом внешнего ключа в отношении **СОТРУДНИК**: в это отношение включается внешний ключ *Номер*

отдела, значения которого совпадают со значениями первичного ключа *Номер отдела* отношения ОТДЕЛ. В схеме отношения атрибуты внешнего ключа будем помечать аббревиатурой FK. Схемы отношений будут выглядеть следующим образом:

СОТРУДНИК (Номер сотрудника, Имя, Год рождения, Номер отдела (ЕК))

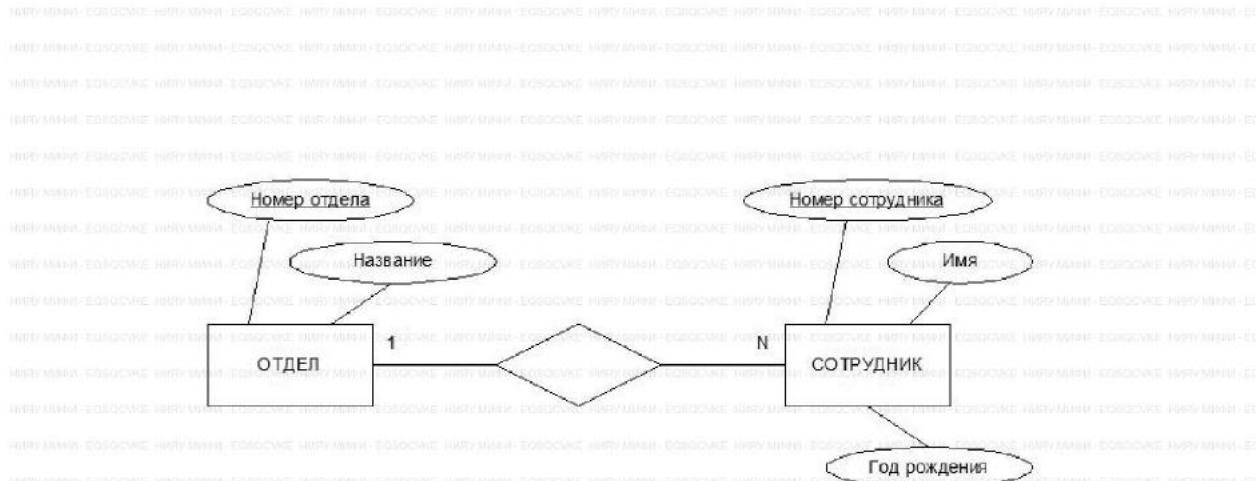


Рис. 5.3. Представление связи типа 1 : n в диаграмме П. Чена

Реализации, удовлетворяющие этим схемам отношений, приведены на рис. 5.4.

ОТДЕЛ	Номер отдела	Название
	Первичный ключ	Альтернативный ключ
	1	Бухгалтерия
	2	АХО
	3	Библиотека

СОТРУДНИК	Номер сотрудника	Имя	Год рождения	Номер отдела
	Первичный ключ			Внешний ключ
	1	Иванов	1953	1
	2	Петров	1970	3

Рис. 5.4. Пример представления связи типа 1 : n в РМД

В данном примере в отношении СОТРУДНИК может появиться кортеж $\langle 3, \text{Сидоров}, 1985, 3 \rangle$, но не может появиться кортеж $\langle 3, \text{Сидоров}, 1983, 5 \rangle$, так как в этом кортеже значению «5» внешнего ключа отношения СОТРУДНИК не соответствует никакое значение первичного ключа отношения ОТДЕЛ.

Таким образом, связи типа 1:n никак специально не представляются, только в отношении на стороне «много» появляются атрибуты внешнего ключа.

Следует отметить, что нотация IDEF1x полностью соответствует представлению связи в РМД (рис. 5.5).



Рис. 5.5. Представление связи типа 1 : n в нотации IDEF1x

ПОСТАВЩИК	Номер поставщика	Имя	Адрес
Первичный ключ	S1	Иванов	Москва
	S2	Сидоров	Санкт-Петербург
	S3	Петров	Тюмень

ПОСТАВЩИК	Номер поставщика	Имя	Адрес
	Первичный ключ		
	S1	Иванов	Москва
	S2	Сидоров	Санкт-Петербург
	S3	Петров	Тюмень

ДЕТАЛЬ	Номер детали	Название	Цена
1	Первичный ключ	Болт	18
2	P1	Винт	14
3	P2	Гайка	10
4	P3	Шуруп	
5	P4		

ПОСТАВКА	Первичный ключ отношения связи		Количество
	Номер поставщика	Номер детали	
ИМЯ ПОСТАВЩИКА	Внешний ключ отношения ПОСТАВЩИК	Внешний ключ отношения ДЕТАЛЬ	Собственный атрибут связи
	S1	P1	100
	S1	P2	200
	S2	P3	150

Рис. 5.7. Пример представления связи типа $p:p$ в РМЛ

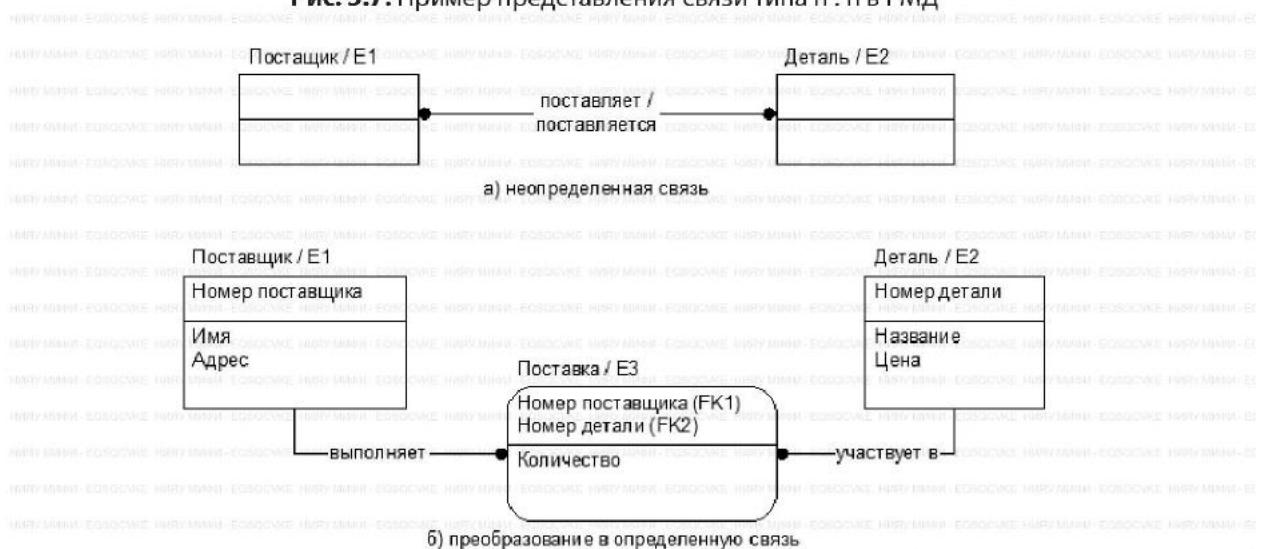


Рис. 5.8. Преобразование связи типа n:n в связь типа 1:n в нотации UMLx

5.2. ЦЕЛОСТНАЯ ЧАСТЬ РЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ

Реляционная модель данных определяет два базовых требования целостности, которые поддерживаются любой реляционной СУБД:

- требование целостности сущностей;
- требование целостности по ссылкам, или ссылочной целостности.

- требование целостности сущностей;
 - требование целостности по ссылкам, или **ссылочной целостности**.

5.2.1. Целостность сущностей

Объекту или сущности реального мира в реляционной базе данных соответствует кортеж отношения. Требование целостности сущностей состоит в следующем: любой кортеж любого отношения должен быть отличим от любого другого кортежа этого же отношения.

Это требование означает, что каждое отношение должно иметь первичный ключ. Данное требование выполняется автоматически, если в системе не нарушаются базовые свойства отношений.

Кроме этого, могут быть установлены и следующие ограничения:

- уникальность значения атрибутов; определяет альтернативные ключи отношения;
 - обязательность значения; при вставке новых или модификации существующих элементов отношения значения соответствующих атрибутов должны быть заданы;
 - допустимость значения атрибутов; вставляемые значения должны удовлетворять некоторому заданному условию.

5.2.2. Ссылочная целостность

Сылочные ограничения целостности в реляционной модели данных представляют собой условия, накладываемые на существование кортежей в связанных отношениях. Как уже говорилось, в реляционной базе данных связи между отношениями представляются с помощью внешних ключей. Вернемся к примеру, в котором рассматривались отношения **ОТДЕЛ** и **СОТРУДНИК** (см. рис. 5.5):

ОТДЕЛ (*Номер отдела, Название (АК)*);

**СОТРУДНИК (Номер сотрудника, Имя, Год рождения,
Номер отдела (FK)).**

Атрибут внешнего ключа *Номер отдела* в отношении СОТРУДНИКА позволяет получить полную информацию о том конкретном отделе, в котором работает конкретный сотрудник.

Обычно отношение, в котором определяется внешний ключ, называется *дочерним отношением*, а отношение, на которое ссылается внешний ключ, — *родительским отношением*. Так, в нашем примере отношение СОТРУДНИК — дочернее, а отношение ОТДЕЛ — родительское.

Требование ссылочной целостности состоит в следующем: значение атрибута внешнего ключа в любом кортеже дочернего отношения определяется значением внешнего ключа в кортеже родительской таблицы.

должно соответствовать значению атрибута первичного ключа в некотором кортеже родительского отношения.

ния должно соответствовать значению атрибута первичного ключа в некотором кортеже родительского отношения.

Другими словами, требование целостности по ссылкам означает, что должно существовать то, на что ссылаемся.

При выполнении операций, связанных с модификацией отношений (операции вставки и удаления кортежей, модификации значения соответствующего ключа), необходимо следить за требованиями целостности (рис. 5.9).

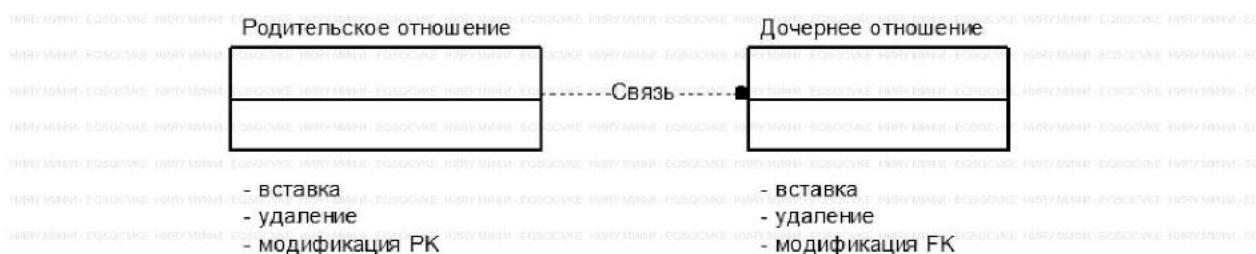


Рис. 5.9. Операции модификации родительского и дочернего отношений

Рассмотрим особенности выполнения этих операций.

1. Все операции с дочерним отношением должны удовлетворять требованиям ссылочной целостности;

- при вставке нового элемента этот элемент должен иметь допустимое значение атрибутов внешнего ключа;
 - удаление элемента выполняется без каких-либо ограничений;
 - при модификации внешнего ключа некоторого элемента этот элемент должен получить допустимое значение атрибутов внешнего ключа.

2. Операции с родительским отношением выполняются в соответствии со следующими правилами:

- вставка нового элемента выполняется без каких-либо ограничений;
 - удаление элемента не должно привести к нарушению ссылочной целостности. Если в дочернем отношении существует элемент, ссылающийся на удаляемый элемент родительского отношения, как поступить с ним? Здесь возможны три подхода:

- a) удаление элемента из родительского отношения не выполняется, если в дочернем отношении есть хотя бы один элемент, ссылающийся на удаляемый;
 - б) вместе с элементом родительского отношения удаляются все ссылающиеся на него элементы дочернего отношения;
 - в) атрибутам внешнего ключа дочернего отношения присваивается пустое значение (каждая СУБД использует собственный способ задания пустого значения); этот подход возможен,

если для атрибутов внешнего ключа дочернего отношения не установлено ограничение обязательности значения;

- модификация значения первичного ключа существующего элемента также не должна привести к нарушению ссылочной целостности. Здесь также возможны те же три подхода:
 - a) модификация первичного ключа элемента из родительского отношения не выполняется, если в дочернем отношении есть хотя бы один элемент, ссылающийся на модифицируемый;
 - b) вместе с элементом родительского отношения модифицируются значения атрибутов внешнего ключа всех ссылающихся на него элементов дочернего отношения;
 - c) атрибутам внешнего ключа дочернего отношения присваивается пустое значение; этот подход возможен, если для атрибутов внешнего ключа дочернего отношения не установлено ограничение обязательности значения.

5.3. МАНИПУЛЯЦИОННАЯ ЧАСТЬ РЕЛЯЦИОННОЙ МОДЕЛИ ДАННЫХ

5.3.1. Общая характеристика

В основе реляционной модели данных лежат некоторые разделы математической теории отношений и математической логики. Этот математический теоретический базис позволяет создавать эффективные языки запросов к реляционным базам данных.

Манипуляционную часть реляционной модели данных составляют спецификационные операторы, естественные для модели данных и базирующиеся на концепциях теории множеств.

В общем случае языки запросов в реляционной модели данных разбиваются на два класса:

- алгебраические языки (реляционная алгебра), позволяющие выражать запросы с помощью специальных операторов, применяемых к отношениям;
 - языки исчисления предикатов (реляционное исчисление), в которых запросы описывают требуемое множество кортежей путем спецификации предиката, которому должны удовлетворять эти кортежи.

Из приведенного определения следует:

- языки реляционной алгебры показывают, как получить результат;
 - языки реляционного исчисления показывают, что должно быть получено.

Языки реляционного исчисления, в свою очередь, делятся на два класса — в зависимости от того, что является примитивными объектами исчисления — кортежи или элементы доменов, являющиеся значениями некоторых атрибутов.

Таким образом, подводя итог сказанному выше, можно определить три вида языков запросов:

- языки реляционной алгебры;
 - языки реляционного исчисления с переменными — кортежами;
 - языки реляционного исчисления с переменными на доменах.

Языки реляционной алгебры позволяют выражать запросы средствами специализированных операций, применяемых к отношениям.

Языки реляционного исчисления позволяют выражать запросы путем спецификации предиката, которому должны удовлетворять кортежи (исчисление с переменными — кортежами) или элементы доменов (исчисление с переменными на доменах).

Такие теоретические языки запросов являются эталоном для оценки реально существующих языков запросов. Они были предложены Э.Ф. Коддом для представления минимальных возможностей любого реляционного языка запросов для реляционной модели. По своей выразительности (с точки зрения получения из реляционной базы данных необходимой информации) все три типа языков запросов эквивалентны между собой.

Реальные языки запросов обеспечивают функции соответствующего теоретического языка или их комбинации и, кроме того, реализуют некоторые дополнительные операции (арифметические и некоторые другие).

5.3.2. Реляционная алгебра. Общая характеристика

Реляционная алгебра предоставляет средства для записи и интерпретации выражений. Алгебраическое выражение имеет обычную структуру: совокупность операндов, разделенных знаками операций. Результат выражения определяется использованными в нем операндами и операциями. Реляционная алгебра представляет набор операций, использующих в качестве operandов отношения и возвращающих в качестве результата также отношение, т.е. представляет операции над отношениями:

Э.Ф. Кодд определил минимальный набор операций над отношениями; все операции, входящие в этот набор, можно разбить на две группы:

1. Теоретико-множественные операции – традиционные операции над множествами, определяемые теорией множеств; к ним относятся:

- объединение;
 - вычитание;
 - пересечение;
 - прямое (декартово) произведение.

2. Специальные реляционные операции; к ним относятся:

- **выбор (селекция);**
 - **проекция;**
 - **соединение;**
 - **деление.**

Особо следует выделить операцию переименования, относящуюся ко второй группе операций.

Из приведенных выше рассуждений можно сделать два важных вывода.

1. Операндами реляционных операций являются отношения; результатом также является отношение. Это свойство называют *свойством замкнутости*. Оно позволяет результат одной операции использовать в качестве исходных данных (операнда) для другой.

Следовательно, можно записывать *вложенные выражения*, т.е. выражения, в которых операнды представлены не простыми именами отношений, а другими выражениями:

R_1 on u_1 , R_2 on u_2 , R_3 ...

2. Используя термин *отношение*, следует помнить, что на самом деле мы имеем дело с двумя понятиями:

- схема отношения (интенсионал);
 - экземпляр (реализация) отношения (экстенсионал).

Так как реляционная операция дает в результате отношение, следовательно, оно также будет иметь, наряду с реализацией, и схему отношения

Так как схема отношения — это поименованная совокупность имен атрибутов, для реляционных операций определяются *правила наследования имен атрибутов*, на основании которых определяется схема отношения, являющегося результатом реляционной операции. Эти правила наследования позволяют предсказывать имена атрибутов на выходе операции.

Обязательным условием для определения схемы отношения является уникальность имен атрибутов в схеме. Для того чтобы результативное отношение имело правильную схему, содержащую правильные (уникальные в данной схеме) имена атрибутов, может быть востребована специальная *операция переименования*, позволяющая

Ниже приведены примеры операции переименования:

Пример 1. Рассмотрим отношение со схемой $S(A, B, C)$ и некоторой реализацией:

Схема: S

	A	B	C
a1	b1	c1	
a2	b1	c3	

Реализация: $S(A, B, SC)$

Схема: SC

	A	B	SC
a1	b1	c1	
a2	b1	c3	

Реализация: $S1(A, B, SC)$

Операция переименования в данном случае не изменяет схему отношения, а только меняет имя атрибута C , не влияя на его значение.

Пример 2. Рассмотрим пример. Пусть имеется отношение со схемой $S(A, B, C)$ и некоторой реализацией:

Можно использовать, например, следующую операцию переименования:

S: переименовать C в SC.

В результате получим отношение со следующей схемой $S1(A, B, SC)$ и той же реализацией:

	A	B	SC
a1	b1	c1	
a2	b1	c3	

Операции переименования в выражениях реляционной алгебры явно использоваться не будут, но там, где это необходимо, операция переименования будет упоминаться, с тем чтобы определить соответствующую схему отношения результата.

5.3.3. Теоретико-множественные операции

Как было сказано ранее, теоретико-множественные операции — это традиционные операции над множествами, определяемые теорией множеств; к ним относятся:

- объединение;
- вычитание;
- пересечение;
- прямое (декартово) произведение.

Но следует отметить, что реляционные операции имеют существенное отличие от классических, определенных в теории множеств.

Рассмотрим это отличие на примере операции объединения.

Операция объединения в теории множеств определяется следующим образом.

Определение. Даны два множества — $S1$ и $S2$. Объединением этих множеств является множество S , элементы которого принадлежат первому множеству $S1$ и (или) второму множеству $S2$:

$$S = S1 \cup S2 = \{s \mid s \in S1 \text{ и/или } s \in S2\}.$$

Ниже приведены примеры объединения множеств:

- Множество 1: {1, 3, 5, 7, 9}; Множество 2: {‘a’, ‘b’, ‘c’}.
- Объединение множеств: $D_1 \cup D_2 = \{1, 3, 5, 7, 9, ‘a’, ‘b’, ‘c’\}$.

Графически это можно представить так (рис. 5.10).



Рис. 5.10. Объединение множеств

В реляционной модели данных рассматривается объединение множеств — доменов и/или отношений.

Объединение доменов. В реляционной модели данных домен представляет собой множество элементов одного типа. Объединение доменов должно создать новый домен.

Пусть даны следующие домены:

$$D_1 = \{1, 3, 5, 7, 9\}; D_2 = \{‘a’, ‘b’, ‘c’\}; D_3 = \{2, 4, 6, 8\}.$$

Тогда в теории множеств определено новое множество:

$$S = D_1 \cup D_2 = \{1, 3, 5, 7, 9, ‘a’, ‘b’, ‘c’\}.$$

Но для реляционной модели данных полученное множество не является доменом, так как содержит элементы разных типов. Следовательно, данная операция в реляционной модели данных для указанных операндов не определена.

С другой стороны, новое множество

$$D = D_1 \cup D_3 = \{1, 3, 5, 7, 9, 2, 4, 6, 8\}$$

содержит элементы одного типа, т.е. является доменом. Следовательно, в данном случае, для данных операндов операция объединения определена.

Объединение отношений. В реляционной модели данных отношение представляет собой множество кортежей, удовлетворяющих одной схеме. Объединение отношений должно дать в результате также отношение.

Пусть определены следующие схемы отношений, определенные на приведенных выше доменах:

$$R_1(A_1:D_1, A_2:D_1); R_2(A_1:D_1, A_2:D_2); R_3(A_1:D_1, A_2:D_3).$$

Ниже приведены примеры объединения отношений:

- Одно отношение: $R_1(A_1:D_1, A_2:D_1)$.
- Две отношения: $R_1(A_1:D_1, A_2:D_1) \cup R_2(A_1:D_1, A_2:D_2)$.
- Три отношения: $R_1(A_1:D_1, A_2:D_1) \cup R_2(A_1:D_1, A_2:D_2) \cup R_3(A_1:D_1, A_2:D_3)$.

Ниже приведены примеры для иллюстрации этого понятия. Рассмотрим отношение R_1 , определенное на множестве $D_1 = \{1, 3\}$ и имеющее схему $\langle 1, 3 \rangle$. Оно определяется как $R_1 = \{(1, 3)\}$. Рассмотрим также отношение R_2 , определенное на множестве $D_2 = \{1, 2\}$ и имеющее схему $\langle 1, 2 \rangle$. Оно определяется как $R_2 = \{(1, 2), (2, 1)\}$. Рассмотрим также отношение R_3 , определенное на множестве $D_3 = \{1, 1\}$ и имеющее схему $\langle 1, 1 \rangle$. Оно определяется как $R_3 = \{(1, 1)\}$.

Реализации данных отношений могут иметь следующий вид:

$$r_1 = \{\langle 1, 3 \rangle, \langle 1, 1 \rangle\}; r_2 = \{\langle 1, 'a' \rangle, \langle 2, 'b' \rangle\}; r_3 = \{\langle 1, 2 \rangle, \langle 7, 2 \rangle\}.$$

Тогда в теории множеств определено новое множество:

$$r = r_1 \cup r_2 = \{\langle 1, 3 \rangle, \langle 1, 1 \rangle, \langle 1, 'a' \rangle, \langle 2, 'b' \rangle\}.$$

Но для реляционной модели данных полученное множество не является отношением, так как кортежи этого множества соответствуют разным схемам отношений. С другой стороны, множество

$$r = r_1 \cup r_3 = \{\langle 1, 3 \rangle, \langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 7, 2 \rangle\}$$

является отношением, хотя схемы исходных отношений R_1 и R_3 и разные.

В связи с этим в реляционной алгебре рассматривается свойство **совместимости по объединению**.

Определение. Простые домены считаются **совместимыми по объединению**, если они состоят из элементов одного типа.

Для приведенных выше примеров домены D_1 и D_2 несовместимы по объединению, а D_1 и D_3 — совместимы по объединению.

Определение. Два отношения считаются **совместимыми по объединению**, если:

- оба отношения имеют одно и то же множество атрибутов;
- одноименные атрибуты двух отношений определены на совместимых по объединению доменах.

Так, в приведенных выше примерах отношения R_1 и R_3 совместимы по объединению, так как их одноименные атрибуты определены на совместимых по объединению доменах.

Если нужно проверить совместимость по объединению отношений, использующих в своих схемах разные имена атрибутов, тогда в соответствии с семантикой атрибутов можно переименовать их, используя операцию переименования. После этого полученные отношения проверяются на совместимость по объединению. Например, пусть дано еще одно отношение $R_4(B_1:D_1, B_2:D_3)$ и надо проверить, совместимо ли по объединению данное отношение с отношением R_1 . Сначала, используя операцию переименования, получаем новое отношение $R_4(A_1:D_1, A_2:D_3)$:

$R_4 : \text{переименовать } B_1 \text{ в } A_1, B_2 \text{ в } A_2.$

После этого можно проверить отношения R_1 и R_4 на совместимость по объединению. С таким же успехом можно использовать операцию переименования:

$R_1: \text{переименовать } A_1 \text{ в } B_1, A_2 \text{ в } B_2.$

Ниже мы видим, что для любых трех элементов $a, b, c \in A$ из R_1 и R_2 мы имеем:

$$(a, b) \in R_1 \wedge (b, c) \in R_1 \Rightarrow (a, c) \in R_1$$

$$(a, b) \in R_2 \wedge (b, c) \in R_2 \Rightarrow (a, c) \in R_2$$

Рассмотрим теперь все операции реляционной алгебры. В определении операций и примерах строчными буквами обозначены реализации отношений, прописными — схемы отношений; запись $r(R)$ означает: реализация отношения r , удовлетворяющая схеме R .

Объединение отношений

Определение. Объединением двух отношений $r_1(R_1)$ и $r_2(R_2)$, совместных по объединению, называется отношение $s = r_1 \cup r_2$, для которого:

а) схема отношения совпадает с R_1 или R_2 ;

б) реализация отношения представляет множество кортежей, принадлежащих реализации r_1 и (или) r_2 .

Формальная запись:

даны $r_1(R_1), r_2(R_2), r_1 = \{t_{1i}\}, r_2 = \{t_{2i}\}, R_1 \equiv R, R_2 \equiv R$,

$s = r_1 \cup r_2 = s(R), s = \{t_i \mid t_i \in r_1 \text{ и/или } t_i \in r_2\}$.

Пример:

r_1	(A B C)	r_2	(A B C)	$s = r_1 \cup r_2$	(A B C)
	a1 b1 c1		a1 b2 c1		a1 b1 c1
	a1 b2 c1		a2 b2 c1		a1 b2 c1
	a2 b1 c2		a2 b2 c2		a2 b1 c2
					a2 b2 c1
					a2 b2 c2

Свойства операции:

- коммутативна — $r_1 \cup r_2 \equiv r_2 \cup r_1$;
- ассоциативна — $r_1 \cup (r_2 \cup r_3) \equiv (r_1 \cup r_2) \cup r_3 \equiv r_1 \cup r_2 \cup r_3$.

Вычитание отношений

Определение. Вычитанием двух отношений $r_1(R_1)$ и $r_2(R_2)$, совместных по объединению, называется отношение $s = r_1 - r_2$, для которого:

а) схема отношения совпадает с R_1 или R_2 ;

б) реализация отношения представляет множество кортежей, принадлежащих реализации r_1 , за исключением тех, которые имеются в r_2 .

Формальная запись:

даны $r_1(R_1), r_2(R_2), r_1 = \{t_{1i}\}, r_2 = \{t_{2i}\}, R_1 \equiv R, R_2 \equiv R$;

$s = r_1 - r_2 = s(R), s = \{t_i \mid t_i \in r_1 \text{ и } t_i \notin r_2\}$.

Ниже приведены примеры вычисления пересечения и разности отношений:

Пример:

r_1	(A B C)
	a1 b1 c1
	a1 b2 c1
	a2 b1 c2

r_2	(A B C)
	a1 b2 c1
	a2 b2 c1
	a2 b2 c2

$$s = r_1 \cap r_2 = \{(a1, b2, c1)\}$$

(A B C)
a1 b1 c1
a1 b2 c1
a2 b1 c2

Свойства операции:

- не коммутативна;
- не ассоциативна.

Пересечение отношений

Определение. Пересечением двух отношений $r_1(R_1)$ и $r_2(R_2)$, совместных по объединению, называется отношение $s = r_1 \cap r_2$, для которого:

- а) схема отношения совпадает с R_1 или R_2 ;
- б) реализация отношения представляет множество кортежей, содержащихся в r_1 , и в r_2 .

Формальная запись:

даны $r_1(R_1)$, $r_2(R_2)$, $r_1 = \{t_{1i}\}$, $r_2 = \{t_{2i}\}$, $R_1 \equiv R$, $R_2 \equiv R$;

$$s = r_1 \cap r_2 = s(R), s = \{t_i \mid t_i \in r_1 \text{ и } t_i \in r_2\}.$$

Пример:

r_1	(A B C)
	a1 b1 c1
	a1 b2 c1
	a2 b1 c2

r_2	(A B C)
	a1 b2 c1
	a2 b2 c1
	a2 b2 c2

$$s = r_1 \cap r_2 = \{(a1, b2, c1)\}$$

(A B C)
a1 b1 c1
a1 b2 c1

Свойства операции:

- коммутативна — $r_1 \cap r_2 \equiv r_2 \cap r_1$;
- ассоциативна — $r_1 \cap (r_2 \cap r_3) \equiv (r_1 \cap r_2) \cap r_3 \equiv r_1 \cap r_2 \cap r_3$.

Операцию пересечения можно выразить через операцию вычитания:

$$r_1 \cap r_2 = r_1 - (r_1 - r_2).$$

Декартово произведение отношений

Здесь отношения $r_1(R_1)$ и $r_2(R_2)$ могут иметь разные схемы, не обязательно совместимые по объединению. Перед выполнением операции декартова произведения необходимо переименовать атрибуты в схемах отношений R_1 или R_2 так, чтобы имена всех атрибутов были разными.

Ниже приведены примеры вычисления пересечения и разности отношений:

Пример:

r_1	(A B C)
	a1 b1 c1
	a1 b2 c1
	a2 b1 c2

r_2	(A B C)
	a1 b2 c1
	a2 b2 c1
	a2 b2 c2

$$s = r_1 \cap r_2 = \{(a1, b2, c1)\}$$

(A B C)
a1 b1 c1
a1 b2 c1

Определение. Декартовым произведением двух отношений $r_1(R_1)$ и

$r_2(R_2)$, схемы отношений которых не имеют одноименных атрибутов, т.е. $R_1 \cap R_2 = 0$, называется отношение $s = r_1 \times r_2$, для которого:

а) схема отношения определяется сцеплением (объединением) схем R_1 и R_2 ;

б) реализация отношения представляет множество кортежей, которое получается путем сцепления каждого кортежа из r_1 с каждым кортежем из r_2 .

[Единые государственные информационные системы Российской Федерации](#)

Формальная запись:
$$(\exists x)(B_x \wedge (\forall y)(A_y \rightarrow B_y)) \vdash (\exists x)(B_x \wedge (\forall y)(B_y \rightarrow B_x))$$

даны $r_1(R_1)$, $R_1(A_1, A_2, \dots, A_m)$, $r_2(R_2)$, $R_2(B_1, B_2, \dots, B_n)$,
 $r_1 = \{t_1\}$, $r_2 = \{t_2\}$, $R_1 \cap R_2 = \emptyset$:

$r_1 = (c_{11}), r_2 = (c_{21}), r_1 + r_2 = \phi$

$$s = r_1 \times r_2 = s(R), R(A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_n),$$

Свойство 3. Для любых элементов $s = \{u_i v_j | u_i \in r_1, v_j \in r_2\}$ множества r_1 и r_2 выполняется равенство $\text{сумма}(\text{сумма}(r_1) + \text{сумма}(r_2)) = \text{сумма}(s)$.

Пример: *Приемы языка и языковая культура в художественных произведениях*

$$(A \mid B) \quad r_2 \quad (X \mid Y \mid Z) \quad s = r_1 \times r_2 \quad (A \mid B \mid X \mid Y \mid Z)$$

al **b1** ~~WVYVWVWV - ECCCCVWV~~ **x1** **y1** **z2** ~~WVYVWVWV - ECCCCVWV~~ **al** **b1** **x1** **y1** **z2**

a2	b1	x2	y1	z1	a1	b1	x2	y1	z1
----	----	----	----	----	----	----	----	----	----

$x3$	$y1$	$z1$	$a1$	$b1$	$x3$	$y1$	$z1$
------	------	------	------	------	------	------	------

a2	b1	x1	y1	z2

TYRANN - ECOLOGIE, HIRY NAM N - ECOLOGIE, HIRY NAM N - ECOLOGIE, HIRY NAM N - ECOLOGIE, HIRY NAM N

a2 b1 x3 y1 z1

Свойства операции: *шина-входные низуники-входные низуники-входные низуники*

коммутативна — $r_1 \times r_2 \equiv r_2 \times r_1$; **ассоциативна** — $r \times (r \times r) \equiv (r \times r) \times r \equiv r \times r \times r$.

В теории множеств данная операция и не коммутативна, и не ас-

В теории множестве данная операция не коммутативна, и не ассоциативна, так как в множествах определен порядок перечисления

ементов в кортеже. Поскольку одно из свойств реляционной мо-

ли данных — отсутствие упорядоченности атрибутов, данная опе-
рация приобретает указанные свойства.

5.3.4. Специальные операции

5.4. Специальные операции

- К специальным операциям реляционной алгебры относятся:

 - проекция;
 - выбор (или селекция);
 - соединение;
 - деление.

Специальные операции определены только для нормализованных отношений. В этих операциях наряду с самими отношениями участвуют и их атрибуты. В отношениях реляционной модели данных к

атрибутам можно обращаться или по имени, или по их позиции в схеме отношений. Мы будем использовать обращение к атрибутам по имени.

атрибутам можно обращаться или по имени, или по их позиции в схеме отношений. Мы будем использовать обращение к атрибутам по имени.

Проекция

Данная операция является унарной операцией на отношениях, т.е. в этой операции участвует только одно отношение.

Определение. Проекцией отношения $r(R)$, $R = \{A_i\}$, на некоторый список имен атрибутов (подмножество атрибутов) L из R , $L \subseteq R$, называется отношение $s = \pi_L(r)$, для которого:

- схема отношения определяется списком L ;
 - реализация отношения есть множество кортежей, полученных из кортежей отношения g путем вычеркивания элементов, соответствующих атрибутам $R - L$, и исключением дубликатов.

Формальная запись:

даны $r(R)$, $R(A_1, A_2, \dots, A_n)$, $r = \{t_1 : A_1, t_2 : A_2, \dots, t_n : A_n\}$;

$s(L) = \pi_L(r), L(B_1, B_2, \dots, B_k), B_i \subseteq R, s = \{ \langle u_1 : B_1, u_2 : B_2, \dots, u_k : B_k \rangle |$
 таких, что $u_i = t_i$, если $B_i \equiv A_i\}$.

Пример:

r	(A)	B	C	D)
a1	b1	c2	d1	
a1	b1	c1	d2	
a2	b1	c3	d2	

$L = (A, B)$	$\pi_L(r)$	(A)	B)
		a1	b1
		a2	b1

Проекция дает вертикальное подмножество отношения

Проекция дает верти-

Свойство проекций:
если $Y \subseteq X \subseteq R$, то $\pi_Y(\pi_Y(r)) \equiv \pi_Y(r)$.

Выбор

Данную операцию называют еще *ограничением и селекцией*.
Также является унарной операцией над отношением.

Также является унарной операцией над отношением.

Определение. Выбором из отношения $r(R)$ по условию F называется отношение $s = \sigma_F(r)$, для которого:

- схема отношения совпадает со схемой R;
 - реализация отношения есть множество кортежей из Γ , удовлетворяющих условию F.

Формальная запись:

$s(R) = \sigma_F(r)$, $s = \{u_1 \mid u_i \in R \text{ и } F(u_i) - \text{истинно}\}$.

Условие (предикат) F записывается в соответствии со следующими правилами:

- в качестве операндов могут быть указаны атрибуты отношения и/или константы;
 - в качестве операций могут быть использованы операции отношения ($=$, \neq и т.д.) и логические операции (\wedge , \vee , \neg).

Для указания порядка вычисления предиката F в нем могут быть использованы круглые скобки.

Пример:

Выбор дает горизонтальное подмножество отношения.

Выбор дает горизонт
Свойства операции:

- коммутативна — $\sigma_{F_1}(\sigma_{F_2}(r)) = \sigma_{F_2}(\sigma_{F_1}(r)) = \sigma_{F_1 \wedge F_2}(r)$;
 - дистрибутивна относительно операций $\gamma = (\cap, \cup, -)$:
 $\sigma_F(r \gamma s) = \sigma_F(r) \gamma \sigma_F(s)$.

Операция выбора осуществляет ограничение кортежей исходного отношения до значений, удовлетворяющих условию.

6

В реляционной алгебре рассматриваются две модификации данной операции: естественное соединение и соединение по условию (или θ -соединение).

Естественное соединение

Определение. Естественным соединением отношений $r_1(R_1)$, $R_1 = XY$, и $r_2(R_2)$, $R_2 = YZ$, где Y — общее подмножество атрибутов из R_1 и R_2 , определенных на одних и тех же доменах, называется отношение $s(R) = r_1 \bowtie r_2$, для которого:

- схема отношения $R = R_1 \cup R_2 = XYZ$;
 - реализация отношения есть множество кортежей $\{t\}$, для которых $\pi_{XY}(t) \in r_1$ и $\pi_{YZ}(t) \in r_2$.

YOUNG-EGGOME HARRY YOUNG-EGGOME HARRY

Формальная запись:
даны $r_1(R_1)$, $R_1 = XY$, и $r_2(R_2)$, $R_2 = YZ$;

$$s(R) = r_1 \cup r_2, R = R_1 \cup R_2 = XYZ,$$

Приведенное естественное соединение называется *внутренним*, или *замкнутым*, так как результат операции содержит только те кортежи из двух отношений, которые имеют совпадающие значения одноименных атрибутов. Наряду с такой операцией, рассматривается еще операция *внешнего*, или *незамкнутого* естественного соединения, при этом существуют разновидности операции внешнего соединения: левое, правое и полное внешнее соединение.

Результат операции левого внешнего соединения $r_1 \bowtie r_2$ включает результат операции естественного соединения, дополненный и теми кортежами из r_1 , для которых нет соответствующих кортежей из r_2 ; для таких кортежей значения атрибутов, унаследованных от r_2 , устанавливаются как пустые значения.

Пример:

	r_1	(A B C)	r_2	(B C D)	$s = r_1 \bowtie r_2$	(A B C D)
HARRY MINNIE - ECONOMIC	a	b	c	b	c	d
HARRY MINNIE - ECONOMIC	d	b	c	b	c	e
HARRY MINNIE - ECONOMIC	b	b	f	a	d	b
HARRY MINNIE - ECONOMIC	c	a	d	a	b	f
HARRY MINNIE - ECONOMIC						—
HARRY MINNIE - ECONOMIC HARRY MINNIE - ECONOMIC HARRY MINNIE - ECONOMIC						c a d b

Результат операции правого внешнего соединения $r_1 \bowtie r_2$ включает результат операции естественного соединения, дополненный и теми кортежами из r_2 , для которых нет соответствующих кортежей из r_1 ; для таких кортежей значения атрибутов, унаследованных от r_1 , устанавливаются как пустые значения.

Пример:

Пример:					
r_1	(A B C)	r_2	(B C D)	$s = r_1 \bowtie r_2$	(A B C D)
	a b c		b c d	a b c d	
	d b c		b c e	a b c e	
	b b f		a d b	d b c d	
	c a d		a b f	d b c e	
				c a d b	
				- a b f	

Наконец, результат операции полного внешнего соединения $r_1 \times r_2$ включает результат операции естественного соединения, дополненный и кортежами из r_1 , для которых нет соответствующих кортежей из r_2 , и кортежами из r_2 , для которых нет соответствующих кортежей из r_1 , с пустыми значениями соответствующих унаследованных атрибутов.

Пример:

r_1	(A B C)	r_2	(B C D)	$s = r_1 \times r_2$	(A B C D)
	a b c		b c d		a b c d
	d b c		b c e		a b c e
	b b f		a d b		d b c d
	c a d		a b f		d b c e
					b b f —
					c a d b
					— a b f

Соединение по условию

Определение. Даны два отношения $r_1(R_1)$ и $r_2(R_2)$, для которых в R_1 и R_2 нет атрибутов с одинаковыми именами, т.е. $R_1 \cap R_2 = 0$. Пусть атрибут $A \in R_1$ сравним по условию θ с атрибутом $B \in R_2$ (условие θ определяется так же, как предикат F в операции выбора). Тогда *соединением* отношений $r_1(R_1)$ и $r_2(R_2)$ по условию θ называется отношение $s(R) = r_1 \bowtie_{A \theta B} r_2$, для которого:

- схема отношения $R = R_1 \cup R_2$:
 - реализация отношения есть множество кортежей, полученных сцеплением кортежей из r_1 и r_2 , удовлетворяющих условию АБВ.

Формальная запись:

даны $r_1(R_1)$ и $r_2(R_2)$, $R_1 \cap R_2 = 0$;

$s(R) = r_1 \bowtie_{\Theta \in B} r_2$, $R = R_1 \cup R_2$, $s = \{uv \mid \text{таких, что } u \in r_1, v \in r_2 \text{ и для } u \text{ и } v \text{ выполняется условие } \Theta\}$.

Атрибуты А и В могут быть составными, т.е. $A = \{A_1, A_2, \dots, A_n\}$, $B = \{B_1, B_2, \dots, B_n\}$. Тогда $A \theta B = [A_1 \theta_1 B_1, A_2 \theta_2 B_2, \dots, A_n \theta_n B_n]$. Операции θ_i могут быть разными. Например, пусть даны $r_1(A_1, A_2, A_3)$ и $r_2(B_1, B_2, B_3)$, и все атрибуты определены на числовых доменах. Тогда можно получить соединение по условию:

$$s = r_1 \wedge_{A_1 \leq B_1} A_2 = B_2, A_3 \geq B_3 \quad r_2$$

Ниже приведены примеры вычисления суммы и произведения отношений:

Пример:

r_1	(A B C)	r_2	(X Y)	$s = r_1 \times r_2$	(A B C X Y)
	1 2 3		3 1		1 2 3 3 1
	4 5 6		6 2		1 2 3 6 2
	7 8 9				4 5 6 6 2

Если в качестве операции θ используется операция $=$, такое соединение по условию называется экви соединением.

Деление

Определение. Даны два отношения $r_1(R_1)$ и $r_2(R_2)$, для которых $R_2 \subseteq R_1$ и R_2 не пусто. Частным от деления отношения r_1 на отношение r_2 называется отношение $s(R) = r_1 \div r_2$, для которого:

- схема отношения $R = R_1 - R_2$,
- реализация отношения есть множество кортежей t таких, что для всех $u_i \in r_2$ существует $v_j \in r_1$ такой, что $v_j(R_1 - R_2) = t$ и $v_j(R_2) = u_i$.

Формальная запись:

даны $r_1(R_1)$, $r_2(R_2)$, $R_2 \subseteq R_1$, $R_2 \neq 0$;
 $s(R) = r_1 \div r_2$, $R = R_1 - R_2$, $s = \{t_j \mid \forall u \in r_2 (tu \in r_1)\}$.
Другими словами, $s \times r_2 \in r_1$.

Пример:

r_1	(A B C D)	r_2	(C D)	$s = r_1 \div r_2$	(A B)
	a b c d		c d		b c
	a b c f		e f		e d
	b c e f				
	b c c d				
	e d e f				
	e d c d				
	e d c f				

Действительно, кортежи $\langle bcd \rangle$, $\langle bcef \rangle$, $\langle edcd \rangle$ и $\langle edef \rangle$ есть в отношении r_1 . Кортеж $\langle abcd \rangle$ есть в отношении r_1 , но кортежа $\langle abed \rangle$ нет, поэтому в s нет кортежа $\langle ab \rangle$.

Можно написать выражение реляционной алгебры, эквивалентное операции деления:

$$r_1 \div r_2 = \pi_{R_1-R_2}(r_1) - \pi_{R_1-R_2}((\pi_{R_1-R_2}(r_1) \times r_2) - r_1).$$

Рассмотрим некоторые примеры написания запросов к базе данных на языке реляционной алгебры.

Ниже приведены примеры запросов на языке SQL для выполнения различных операций с базой данных.

1. Пусть дана следующая схема базы данных:

$S(S\#, SN, SC)$ — ПОСТАВЩИК (Номер поставщика, Имя, Город);

$P(P\#, PN, PC)$ — ДЕТАЛЬ (Номер детали, Название, Цена);

$SP(S\#, P\#, QTY)$ — ПОСТАВКА (Номер поставщика, Номер детали, Количество).

1. Получить имена поставщиков, поставляющих деталь с номером P_1 :

$$\pi_{SN}(\sigma_{P\# = P_1}(S \bowtie SP)).$$

2. Получить имена поставщиков, не поставляющих деталь с номером P_1 :

$$\pi_{SN}(S) - \pi_{SN}(\sigma_{P\# \neq P_1}(S \bowtie SP)).$$

3. Получить имена поставщиков, поставляющих только деталь с номером P_1 :

$$\pi_{SN}(\sigma_{P\# = P_1}(S SP)) - \pi_{SN}(\sigma_{P\# \neq P_1}(S \bowtie SP)).$$

4. Получить имена поставщиков, поставляющих все детали:

$$\pi_{SN}((\pi_{S\#, P\#}(SP) \div \pi_{P\#}(P)) S).$$

5.3.5. Реляционное исчисление

Общие определения

Реляционное исчисление лежит в основе декларативного подхода к формулировке запроса к базе данных, суть которого заключается в следующем.

Запросу к базе данных соответствует формула некоторой формально-логической теории, которая описывает свойства желаемого результата.

Ответом на запрос служит множество объектов из области интерпретации (которой является база данных), на котором истинна формула, соответствующая запросу.

В качестве такой формально-логической теории используется **теория исчисления предикатов первого порядка**, в которой формула задается в виде **предиката**.

Понятие предиката

Даны некоторые попарно не пересекающиеся произвольные множества $D_1, D_2, \dots, D_n, D_i \cap D_j = \emptyset$ для любых $i \neq j$, и переменные x_1, x_2, \dots, x_n , принимающие значения из соответствующих множеств: $x_i \in D_i$ для любых i .

Тогда *предикатом* (или *предикатной функцией*) называется функция $P(x_1, x_2, \dots, x_n)$, принимающая одно из двух значений — 1 или 0 (истина или ложь).

Переменные x_1, x_2, \dots, x_n называются предикатными переменными. Множества D_1, D_2, \dots, D_n образуют область интерпретации предиката.

В записи предиката наряду с логическими операциями \wedge , \vee , \neg , смысл и использование которых традиционны, используются специальные операции — кванторы: всеобщности \forall и существования \exists . Смысл использования кванторов следующий:

- $\forall x (f(x))$ означает, что для всех значений « x » из области интерпретации формула $f(x)$ имеет значение «истина»;
 - $\exists x (f(x))$ означает, что существует по крайней мере одно значение « x » из области интерпретации, для которого формула $f(x)$ имеет значение «истина».

Примеры. Пусть переменная « x » определена на множестве «учебная группа»; $f(t)$ — утверждение « t получает стипендию»; тогда предикат $\exists x (f(x))$ имеет значение «истина», если хотя бы один член группы (неважно, кто именно) получает стипендию, и ложь, если никто в группе не получает стипендию.

Пусть переменная « x » определена на множестве «учебная группа»; $f(t)$ — утверждение « t не имеет задолженностей в сессию»; тогда предикат $\forall x (f(x))$ имеет значение «истина», если все члены группы не имеют задолженностей в сессию, и ложь, если хотя бы один член группы имеет задолженность.

Использование кванторов определяет понятие *свободного и связанных* вхождения переменных в предикатной формуле.

Вхождение переменной t в формулу ψ связано, если переменная t находится в формуле ψ в подформуле, начинающейся кванторами \forall или \exists , за которыми непосредственно следует переменная t ; тогда о кванторе говорят, что он связывает переменную t . В остальных случаях t входит в ψ свободно.

Рассмотрим следующий пример:

$$(\forall x(R(x,y) \vee \exists y(U(x,y,z) \wedge Q(x,y)))) \vee (\forall x(\exists r(U(x,y,r) \wedge (\exists x(F(x)))))$$

В примере пронумеровано использование переменных в связи с их появлением в кванторах и в формуле. В соответствии с этим в первой подформуле все вхождения « x » (помеченные цифрой 1) связаны квантором \forall ; первое вхождение « y » (помеченное цифрой 2) свободно; следующие вхождения « y » (помеченные цифры 3) связаны квантором \exists ; вхождение переменной « z » (помеченное цифрой 4) свободно. Во второй подформуле все вхождения переменной « x » (помеченные цифрами 5 и 8) связаны кванторами \forall и \exists соответственно;

**вхождение «у» (помечено цифровой 7) свободно; и наконец, вхож-
дение «и» (помечено цифровой 6) свободно.**

вхождение «у» (помеченное цифрой 7) свободно; и наконец, вхождение переменной g (помеченное цифрой 6) связано квантором Э.

Кванторы в реляционном исчислении определяют области значений и видимости переменных. Это означает следующее.

Все переменные, входящие в формулу, при построении которой не использовались кванторы, являются свободными. Это означает, что если для какого-то определенного набора свободных переменных при вычислении предикатной формулы получено значение «истина», значит, этот набор значений свободных переменных войдет в результат, определяемый предикатом.

Если же вхождение переменной в формулу связано квантором, тогда такая переменная не видна за пределами формулы, связавшей эту переменную. При вычислении значения такой формулы используется не одно значение переменной, что имеет место для свободных переменных, а все значения из области определения данной переменной.

Рассмотрим пример. Пусть дано множество десятичных цифр $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$; тогда подмножество, состоящее из четных цифр, можно определить следующим образом: множество всех значений y , таких, для которых выполняется условие:

$\exists x (x \in D \wedge x \div 2 = 0 \wedge y = x)$.

Приведенный предикат интерпретируется следующим образом: существует хотя бы одно значение $\langle x \rangle \in D$, которое четно и совпадает со значением некоторой свободной переменной «у». Следовательно, если свободная переменная у имеет конкретное значение, например, 6, приведенный предикат будет иметь значение «истина», и данное значение переменной у входит в результат. Если же переменная «у» будет иметь значение 7 или 12, тогда предикат будет иметь значение «ложь» и данное значение у не войдет в результат.

В реляционном исчислении областью интерпретации предиката является база данных, т.е. множество всех значений всех доменов, на которых определены все схемы отношений, определяющие схему базы данных. Соответствие между предикатом P и отношением $r(A_1, A_2, \dots, A_n)$ заключается в следующем.

Если $P(a_1, a_2, \dots, a_n) = 1$, то $\langle a_1, a_2, \dots, a_n \rangle$ есть выборка отношения $r(A_1, A_2, \dots, A_n)$, т.е.

$$\langle a_1, a_2, \dots, a_n \rangle \in L$$

Если $R(b_1 - b_2) = b_1 \lambda = 0$, то $b_1(b_1 - b_2) > 0$.

Если $P(b_1, b_2, \dots, b_n) = 0$, то $\langle b_1, b_2, \dots, b_n \rangle \notin \Gamma$.
 Таким образом, базисными понятиями исчисления являются по-

Как уже говорилось ранее, в зависимости от того, как определяется область значений используемых переменных, различают два вида реляционного исчисления:

- *реляционное исчисление с переменными-кортежами*, для которого областями определения переменных являются отношения базы данных, т.е. допустимым значением каждой переменной является кортеж некоторого отношения;
 - *реляционное исчисление с переменными на доменах*, для которого областями определения переменных являются домены, определяющие множество допустимых значений атрибутов.

Правильно построенные формулы служат для выражения условий, накладываемых на переменные (кортежи или на доменах).

Реляционное исчисление с переменными-кортежами

Как уже говорилось, в данном случае областью определения переменных являются отношения. Переменные-кортежи должны удовлетворять определенной схеме отношения R . Правильно построенная формула (wff – well formulated formula) определяет результаты отбора: выбираются те кортежи, для которых wff дает значение 1.

Обозначим правильно построенную формулу (предикат, значения которого 0 или 1) через $\psi(t)$.

Рассмотрим правила построения $\psi(t)$.

I. Основой формулы являются атомы, которые могут иметь значения 0 или 1.

Существует *три типа атомов* формулы $\psi(t)$.

1. Пусть $r(R)$ — некоторая реализация отношения, удовлетворяющая схеме R ; t — некоторая переменная-кортеж, удовлетворяющая схеме R . Тогда $r(t)$ — атом; означает, что t есть кортеж в отношении r (т.е. формула истинна, если $t \in r$).

2. Пусть $r(R)$ — некоторая реализация отношения, удовлетворяющая схеме R ; u и v — переменные-кортежи из отношения $r(R)$ (т.е. $u \in r, v \in r$); θ — арифметическая операция сравнения ($<$, $=$, $>$, \geq , \neq , \leq); A, B — атрибуты схемы отношения R , определенные на доменах, сравнимых по операции θ ; тогда $u[A] \theta v[B]$ — атом.

3. Пусть u — переменная-кортеж из отношения $r(R)$ (т.е. $u \in r$); θ — арифметическая операция сравнения ($<$, $=$, $>$, \geq , \neq , \leq); A, B — атрибуты схемы отношения R , определенные на доменах, сравнимые по операции θ ; c — константа из домена, на котором определен атрибут B ; тогда $u[A] \theta c$ (или $c \theta u[B]$) — атом.

В приведенных выше условиях запись $i[A]$ означает «значение переменной-кортежа i по атрибуту A ».

П. Формула реляционного исчисления $\psi(t)$, а также свободные и связанные вхождения переменных определяются по следующим рекурсивным правилам

1. Каждый атом есть формула. Все вхождения переменных-кортежей, упомянутых в атоме, являются свободными. Например, формула $r(t)$ утверждает, что переменная-кортеж t является кортежем отношения $r(R)$.

2. Если $x(R)$ — переменная-кортеж из отношения r со схемой R ; $\psi(x)$ — формула, в которой переменная « x » имеет свободное вхождение; тогда $\exists x(R)(\psi(x))$ — формула, в которой вхождение переменной « x » становится связанным квантором \exists . Данная формула утверждает, что существует хотя бы один кортеж « x » в отношении $r(R)$, такой, что при подстановке его в формулу $\psi(x)$ вместо всех свободных вхождений « x » получим значение «истина». Например, формула $\exists x(R)(r(x))$ утверждает, что отношение r не пусто (т.е. существует хотя бы один кортеж x , принадлежащий r).

3. Если $x(R)$ — переменная-кортеж из отношения r со схемой R ; $\psi(x)$ — формула, в которой переменная « x » имеет свободное вхождение; тогда $\forall x(R) (\psi(x))$ — формула, в которой вхождение переменной « x » становится связанным квантором \forall . Данная формула утверждает, что для всех кортежей « x » из отношения $r(R)$ при подстановке их в формулу $\psi(x)$ вместо всех свободных вхождений « x » получим значение «истина». Например, формула $\forall x(R) (x[A] \neq \emptyset)$ утверждает, что для всех кортежей значение атрибута A не пусто, т.е. атрибут A является обязательным.

4. Если $\psi(x)$ и $\phi(x)$ — формулы, тогда $\neg\psi(x)$, $\psi(x) \wedge \phi(x)$, $\psi(x) \vee \phi(x)$ — тоже формулы. Вхождения переменной « x » в эти формулы остаются свободными или связанными — такими, какими были в $\psi(x)$ или $\phi(x)$ соответственно. Например, если переменная « x » имела в $\phi(x)$ свободное вхождение, а в $\psi(x)$ — связанное, то в этих функциях сохраняется тип вхождения переменных.

5. Если $\psi(x)$ — формула, то $(\psi(x))$ — тоже формула; вхождение переменной « x » остается свободным или связанным — таким, каким оно было в $\psi(x)$.

6. Ничто иное не является формулой.

При вычислении формул используется порядок старшинства операций, определяемый правилами построения формулы:

а) атомы, в которых могут быть использованы арифметические операции сравнения;

б) кванторы:

Б) отрицание (-)

В) стрижка (1), Г) операция «И» (1)

15) операция «ИДИ» (х) АБОВСКОЕ, КИРГИЗИЯНСКИЙ, ЕДОСКОСКИЙ, КИЛУМЫНСКИЙ, ЕДОСКОСКИЙ, КИЛУМЫНСКИЙ, ЕДОСКОСКИЙ, КИЛУМЫНСКИЙ

Скобки используются для изменения порядка вычисления формулы.

Скобки используются для изменения порядка выполнения формул.

Определение. Выражение реляционного исчисления с переменными-кортежами имеет вид: $\{t(R) \mid \psi(t)\}$, где t — переменная-кортеж, удовлетворяющая схеме отношения R ; единственная переменная, имеющая свободное вхождение в формулу $\psi(t)$; $\psi(t)$ — правильно построенная формула.

Интерпретация выражения реляционного исчисления: множество кортежей t , удовлетворяющих схеме отношения R , таких, для которых правильно построенная формула $\psi(t)$ истинна.

Пример. Пусть есть отношение $R(\text{Имя}, \text{Стипендия})$; атрибут *Стипендия* определен на домене $D = \{\text{«да}, \text{«нет}\}$. Тогда выражение

позволит получить из отношения имена всех студентов, получающих стипендию.

Безопасные выражения

Выражение вида $\{t \mid \neg r(t)\}$ в общем случае определяет бесконечное отношение, что недопустимо. Поэтому в реляционном исчислении ограничиваются рассмотрением так называемых **безопасных** выражений вида $\{t \mid \psi(t)\}$, которые гарантированно дают ограниченное множество кортежей. В таких выражениях значения атрибутов кортежей t являются элементами некоторого ограниченного универсального множества — $\text{DOM}(\psi)$. Здесь $\text{DOM}(\psi)$ — унарное отношение, элементами которого являются символы, которые либо явно появляются в ψ , либо служат компонентами какого-либо кортежа в некотором отношении R , упоминаемом в ψ .

В книге Дж. Ульмана [3] приведено следующее определение: «Выражение $\{t \mid \psi(t)\}$ реляционного исчисления с переменными-кортежами назовем **безопасным**, если выполняются следующие условия:

1. Всякий раз, когда t удовлетворяет $\psi(t)$, каждый компонент t есть элемент $\text{DOM}(\psi)$.
 2. Для любого подвыражения ψ вида $(\exists u) (\omega(u))$ каждый компонент u принадлежит $\text{DOM}(\omega)$, если u удовлетворяет ω .
 3. Если для любого подвыражения ψ вида $(\forall u) (\omega(u))$ каждый компонент u не принадлежит $\text{DOM}(\omega)$, то u не удовлетворяет ω .

Условия 2 и 3 позволяют устанавливать истинность квалифицированной формулы $(\exists u)(\omega(u))$ или $(\forall u)(\omega(u))$, рассматривая только u , составленные из принадлежащих $\text{DOM}(\omega)$ символов. Например, любая формула $(\exists u)(R(u) \wedge \dots)$ удовлетворяет условию 2, а любая формула $(\forall u)(\neg R(u) \vee \dots)$ — условию 3.

Хотя условие 3 может показаться неинтуитивным, мы должны заметить, что формула $(\forall u)(\omega(u))$ логически эквивалентна формуле $\neg(\exists u)(\neg\omega(u))$. Последняя не является безопасной, если и только если существует некоторое u_0 , для которого истинно $\neg\omega(u_0)$ и u_0 не при-

надлежит домену формулы $\omega(u)$. Так как домены ω и $\neg\omega$ одни и те же, условие 3 устанавливает, что формула $(\forall u) (\omega(u))$ безопасна, когда безопасна формула $\neg(\exists u) (\neg\omega(u)) \dots$ ».

надлежит домену формулы $\neg\omega(u)$. Так как домены ω и $\neg\omega$ одни и те же, условие 3 устанавливает, что формула $(\forall u) (\omega(u))$ безопасна, когда безопасна формула $\neg(\exists u) (\neg\omega(u)) \dots$ ».

В той же книге доказана следующая теорема, которая здесь приводится без доказательства.

Если E – выражение реляционной алгебры, то существует эквивалентное ему безопасное выражение реляционного исчисления с переменными-кортежами.

В табл. 5.1 приводятся некоторые эквивалентности.

Таблица 5.1

Эквивалентности для реляционной алгебры и реляционного исчисления с переменными кортежами

Выражение реляционной алгебры	Выражение реляционного исчисления с переменными-кортежами
Объединение: $r_1 \cup r_2, r_1(R), r_2(R)$	$\{x(R) r_1(x) \vee r_2(x)\}$
Разность: $r_1 - r_2, r_1(R), r_2(R)$	$\{x(R) r_1(x) \wedge \neg r_2(x)\}$
Пересечение: $r_1 \cap r_2, r_1(R), r_2(R)$	$\{x(R) r_1(x) \wedge r_2(x)\}$
Декартово произведение: $r_1 \times r_2, r_1(R_1), r_2(R_2)$	$\{x(R_1 R_2) \exists u(R_1) \exists v(R_2) (r_1(u) \wedge r_2(v) \wedge x[R_1] = u[r_1] \wedge x[R_2] = v[R_2])\}$
Проекция: $\pi_L(r), r(R), L \subseteq R$	$\{t(L) \exists u(R) (r(u) \wedge u[L] = t[L])\}$
Выбор: $\sigma_F(r), r(R)$	$\{t(R) r(t) \wedge F'\}$
Естественное соединение: $r_1 \bowtie r_2, r_1(AB), r_2(BC)$	$\{t(ABC) \exists u(AB) \exists v(BC) (r_1(u) \wedge r_2(v) \wedge u[B] = v[B] \wedge t[AB] = u[AB] \wedge t[C] = v[C])\}$
Деление: $r_1 + r_2, r_1(AB), r_2(B)$	$\{t(A) \forall x(B) (r_2(x) \wedge \exists y(AB) (r_1(y) \wedge y[B] = x[B] \wedge y[A] = t[A])\}$

Рассмотрим те же примеры запросов, которые приводились для языка реляционной алгебры.

языка религиозной алгебры.

Дана та же схема базы данных:

S(S#, SN, SC) – ПОСТАВЩИК (Номер поставщика, Имя, Город);

P(P#, PN, PC) – ДЕТАЛЬ (Номер детали, Название, Цена);

SP(S#, P#, QTY) – ПОСТАВКА (Номер поставщика, Номер детали, Количество)

1. Нижу приведены имена поставщиков, поставляющих детали с номе-
ром Р1.

ИДИОМЫ-СЛОВОСОЧЕНИЯ ИХ ВЛИЯНИЕ НА ОБРАЗЫ МИРА

{t(SN)} | $\exists u(S) \exists v(SP)(S(u) \wedge SP(v) \wedge u[S\#] = v[S\#] \wedge v[P\#] =$

= ‘P1’ \wedge u[SN] = t[SN])\}.

Приложение 1 к Постановлению Правительства Российской Федерации от 12.02.2015 № 113

Пояснение: результат запроса — множество кортежей t , имеющих

только один атрибут SN, таких, что:

- Ниже приведены примеры запросов на языке SQL для решения задачи.
- существует, по крайней мере один кортеж из отношения **S** ($\exists u(S) \dots (S(u) \dots)$) и
 - существует, по крайней мере один кортеж из отношения **SP** ($\exists v(SP) \dots SP(v) \dots)$,
 - такие, что
 - значения этих кортежей по атрибуту **S#** совпадают ($u[S\#] = v[S\#]$),
 - значение кортежа **v** по атрибуту **P#** определяет интересующую нас деталь **P1** ($v[P\#] = 'P1'$) и
 - кортеж **v** определяет результат запроса ($u[SN] = t[SN]$).
2. Получить имена поставщиков, не поставляющих деталь с номером **P1**:
- $$\{t(SN) | \exists u(S)(S(u) \wedge \exists v(SP)(SP(v) \wedge v[P\#] = 'P1') \wedge u[S\#] = v[S\#] \wedge u[SN] = t[SN])\}.$$

То есть для кортежа из отношения **S**, определяющего результат запроса, не найдется кортеж из отношения **SP**, имеющий такое же значение по атрибуту **S#** и определяющий деталь **P1**.

3. Получить имена поставщиков, поставляющих только деталь с номером **P1**.

Этот запрос, по сути, объединяет два предыдущих: т.е. определяются кортежи, соответствующие поставляемой детали **P1**, и из этих кортежей удаляются те, которые соответствуют поставке других деталей:

$$\{t(SN) | \exists u(S) \exists v(SP)(S(u) \wedge SP(v) \wedge u[S\#] = v[S\#] \wedge v[P\#] = 'P1' \wedge u[SN] = t[SN] \wedge \neg \exists w(SP)(SP(w) \wedge w[S\#] = v[S\#] \wedge u[SN] = w[P\#] \neq 'P1'))\}.$$

4. Получить имена поставщиков, поставляющих все детали.

Поскольку этот запрос реализуется операцией деления реляционной алгебры, воспользуемся приведенной выше эквивалентностью. Рассмотрим сначала запрос, позволяющий получить номера поставщиков, поставляющих все детали:

$$\{t(S\#) | \forall w(P) \exists u(SP)(P(w) \wedge SP(v) \wedge w[P\#] = v[P\#] \wedge t[S\#] = v[S\#])\}.$$

Теперь добавим в этот запрос конструкции, необходимые для получения имени поставщика из отношения **S**:

$$\{t(SN) | \exists u(S) \forall w(P) \exists u(SP)(S(u) \wedge P(w) \wedge SP(v) \wedge w[P\#] = v[P\#] \wedge u[S\#] = v[S\#] \wedge t[S\#] = u[SN])\}.$$

Ниже приведены примеры выражений с переменными на доменах:

- $\exists x_1 \exists x_2 \ldots \exists x_n R(x_1, x_2, \ldots, x_n)$, где R — отношение, удовлетворяющее схеме $R(A_1, A_2, \ldots, A_n)$, и каждое x_i есть константа или переменная на домене;
- $\forall v \theta v$, где v и θ — константы или переменные, определенные на доменах, совместимых по операции θ , θ — арифметическая операция сравнения ($<, =, >, \geq, \neq, \leq$).

Реляционное исчисление с переменными на доменах

Выражение реляционного исчисления с переменными на доменах строится с использованием тех же средств, что и выражение реляционного исчисления с переменными-кортежами. Отличие состоит в том, что здесь областью определения являются домены.

Здесь также строится правильно определенная формула, основой которой являются атомы.

I. Атомы имеют следующий вид:

a) $r(x_1, x_2, \ldots, x_n)$, где r — отношение, удовлетворяющее схеме $R(A_1, A_2, \ldots, A_n)$,

б) θv , где v и θ — константы или переменные, определенные на доменах, совместимых по операции θ , θ — арифметическая операция сравнения ($<, =, >, \geq, \neq, \leq$).

II. Формула реляционного исчисления $\psi(t)$, а также свободные и связанные вхождения переменных определяются так же, как и для исчисления с переменными-кортежами.

Эквивалентность выражений реляционного исчисления

с переменными-кортежами и исчисления с переменными на доменах

Выражение исчисления с переменными на доменах, эквивалентное выражению исчисления с переменными-кортежами $\{t \mid \psi(t)\}$, конструируется в соответствии со следующими правилами.

Пусть есть переменная-кортеж $t(R)$, где $R = (A_1, A_2, \ldots, A_n)$ имеет арность n . Тогда вместо переменной-кортежа t вводятся n новых переменных на доменах t_1, t_2, \ldots, t_n , и заданное выражение исчисления с переменными-кортежами заменяется выражением $\{t_1, t_2, \ldots, t_n \mid \varphi'(t_1, t_2, \ldots, t_n)\}$. Здесь φ' представляет собой φ , в которой:

- любой атом $r(t)$ заменяется атомом $r(t_1, t_2, \ldots, t_n)$;
- каждое свободное вхождение $t[A_i]$ заменено переменной t_i ;
- для каждого квантора $\exists u$ или $\forall u$ вводится m новых переменных u_1, u_2, \ldots, u_m , где m — арность u . Кванторы $\exists u$ (или $\forall(u)$) заменяются кванторами $\exists u_1 \exists u_2 \ldots \exists u_m (\forall u_1 \forall u_2 \ldots \forall u_m$, соответственно), и в подчиненных кванторах выражениях $u[A_i]$ заменяются u_i , а $r(u) — r(u_1, u_2, \ldots, u_m)$.

Теорема. Для каждого безопасного выражения с переменными-кортежами существует эквивалентное безопасное выражение с переменными на доменах.

Пример. Даны отношения со схемами:

$S(S\#, SNAME, CITY)$ — поставщик;

$P(P\#, PNAME, PRICE)$ — деталь;

$SP(S\#, P\#, QTY)$ — поставка.

Написать выражение реляционного исчисления, позволяющее получить имена поставщиков, поставляющих деталь с номером $P1$:

а) выражение исчисления с переменными-кортежами:

а) выражение исчисления с переменными-кортежами:

$\{t(\text{SNAME}) \mid \exists u(S) \exists v(SP) (S(u) \wedge SP(v) \wedge u[\text{SNAME}] = t[\text{SNAME}] \wedge u[S\#] = v[S\#] \wedge v[P\#] = 'P1')\};$

б) выражение исчисления с переменными на доменах:

$\{t(\text{SNAME}) \mid \exists u_1(\text{S}\#) \exists u_2(\text{SNAME}) \exists u_3(\text{CITY}) \exists v_1(\text{S}\#) \exists v_2(\text{P}\#) \exists v_3(\text{OTY}) (S(u_1, u_2, u_3) \wedge SP(v_1, v_2, v_3) \wedge u_2 = t \wedge u_3 = v_3 \wedge v_2 = 'P2')\}$.

5.3.6. Общая характеристика языков манипулирования данными

Э.Ф. Кодд предложил использовать реляционное исчисление с переменными-кортежами как эталон для оценки языков манипулирования данными, основанных на реляционной модели. Дело обстоит так, что едва ли не во все современные языки запросов встраивается одна из трех рассмотренных выше нотаций, а в некоторые языки — их комбинация.

Язык, в котором можно (по крайней мере) моделировать исчисление с переменными-кортежами либо, что равносильно, реляционную алгебру или исчисление с переменными на доменах, называется **полным**.

Однако в реальных системах языки манипулирования данными обладают в общем случае возможностями, выходящими за рамки реляционного исчисления. Во всех языках манипулирования данными конечно же включаются команды вставки, удаления и модификации, которые не являются частью реляционной алгебры или реляционного исчисления. Кроме того, часто предоставляются возможности использовать в записи атомов арифметические операции и использовать агрегатные функции.

Далее приводится краткий обзор языков манипулирования данными, основанных на реляционной модели.

ISBL (Information System Base Language) — «чистый» язык реляционной алгебры. Язык запросов, разработанный в исследовательском центре фирмы IBM в Питерли (Англия) для использования в экспериментальной системе PRTV (Peterlee Relational Test Vehicle); весьма близок к реляционной алгебре. В данном языке нет агрегатных операций, а также средств для вставки, удаления и модификации кортежей.

SEQUEL (Structured English Query Language) — переход от алгебраических языков к языкам исчислений. Использует реляционную алгебру, но имеет синтаксис, напоминающий реляционное исчисление с переменными-кортежами. SEQUEL разработан в 1974 г. (первая публикация) в исследовательской лаборатории IBM в Сан-Хосе.

Центральная возможность языка — отображение: специальный вид селекции с последующей проекцией.

Центральная возможность языка — отображение: специальный вид селекции с последующей проекцией.

В 1976 г. на базе переработанной версии языка SEQUEL/2 корпорация IBM выпустила прототип СУБД, получивший название System R. Назначение этой пробной версии состояло в проверке осуществимости реляционной модели. Помимо прочего, важнейшим из результатов выполнения этого проекта можно считать разработку собственного языка SQL (Structured Query Language).

QUEL — язык реляционного исчисления с переменными-кортежами. Язык запросов реляционной СУБД INGRES, разработанной в Калифорнийском университете в Беркли в конце 1970-х гг. Включает широкий спектр операторов реляционного исчисления с переменными-кортежами, агрегатные функции. Этот язык является более структурированным, чем SQL, но его семантика менее близка к обычному английскому языку (позднее, когда язык SQL был принят как стандартный язык реляционных баз данных, СУБД INGRES была переведена на использование этого языка).

QBE (Query-By-Example) — язык исчисления с переменными на доменах. Язык разработан в Исследовательском центре IBM в Йорктаун-Хайтсе. Его специфика заключается в том, что он предназначен для работы с терминала. Клавиши терминала позволяют затребовать показ на экране одной или более формы таблиц. Необходимые запросы формулируются с использованием переменных на доменах и констант, как в реляционном исчислении с переменными на доменах. Включены агрегатные функции.

SQL (Structured Query Language) — язык, ориентированный на отображение: описывается отображение известного атрибута или множества атрибутов в искомый атрибут или множество атрибутов. В языке для указания роли имен отношений и атрибутов применяются ключевые слова. Первой из коммерческих реализаций СУБД, построенных на использовании языка SQL, была СУБД ORACLE V2, выпущенная в конце 1970-х гг. компанией Software Development Laboratories, которая ныне превратилась в корпорацию ORACLE.

В 1982 г. Американский национальный институт стандартов (ANSI) начал работу над стандартом языка реляционных баз данных. В 1983 г. к этой работе подключился Международный комитет по стандартизации (ISO). Совместными усилиями в 1987 г. был выпущен исходный вариант стандарта языка SQL, вызвавший волну критических замечаний. В 1992 г. была выпущена первая, существенно пересмотренная версия стандарта ISO, которую иногда называют SQL2 или SQL-92.

Язык SQL имеет два основных компонента:

- язык DDL (Data Definition Language), предназначенный для определения структуры базы данных;

- язык DML (Data Manipulation Language), предназначенный для выборки и обновления данных.

Язык SQL (точнее, его подмножество DML) является примером языка с трансформирующей ориентацией — языка, предназначенного для работы с таблицами с целью преобразования входных данных к требуемому виду. Это не процедурный язык, поэтому в нем необходимо указывать, какая информация должна быть получена, а не как ее можно получить. Иначе говоря, язык SQL не требует указания методов доступа к данным.

Вопросы

1. Сформулируйте достоинства и недостатки реляционной модели.
 2. Дайте определение базовых структурных компонентов реляционной модели данных.
 3. Приведите примеры использования целостной части реляционной модели данных.
 4. Приведите примеры использования манипуляционной части реляционной модели данных.
 5. Приведите особенности реализации языков на основе реляционной алгебры и реляционного исчисления.
 6. Перечислите все операции реляционной алгебры и особенности их вычисления. Приведите примеры записи выражений с использованием выражений реляционной алгебры.
 7. Даны следующие два отношения, атрибуты которых определены на домене целых чисел:

Укажите результаты вычисления следующих выражений реляционной алгебры:

- а) $r_1 \cup r_2$ б) $r_1 \cap r_2$ в) $r_1 - r_2$ г) $r_1 \times r_2$
 д) $\sigma_{A > 5}(r_1)$ е) $\sigma_{A > 5 \wedge X < 8}(r_2)$ ж) $\pi_{AC}(r_1)$ з) $\pi_{AX}(r_2)$
 и) $r_1 \text{ join } r_2$ к) $r_1 \text{ left join } r_2$ л) $r_1 \text{ right join } r_2$ м) $r_1 \text{ full join } r_2$

Здесь `join` определяет операцию естественного соединения.

8. Приведите правила формирования запросов при использовании реляционного исчисления с переменными-кортежами.

9. Приведите правила формирования запросов при использовании реляционного исчисления с переменными на доменах.

10. Приведите правила преобразования выражения реляционного исчисления с переменными-кортежами к выражению реляционного исчисления с переменными-именами.

ными на доменах.

11. Для всех выражений реляционной алгебры, указанных в п. 7, приведите эквивалентные выражения с использованием операторов \exists и \forall .

лентные выражения реляционного исчисления с переменными-кортежами и переменными на доменах.

Ниже приведены основные цели проектирования реляционных баз данных:

ГЛАВА 6

ТЕОРИЯ ПРОЕКТИРОВАНИЯ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

6.1. ЦЕЛИ ПРОЕКТИРОВАНИЯ

Основные цели:

- понизить избыточность данных;
- повысить надежность и достоверность данных (т.е. устраниТЬ некоторые аномалии).

Применительно к реляционным базам данных это сводится к решению следующих вопросов:

- какие отношения включать в состав базы данных;
- сколько отношений включить в состав базы данных.

Почему это важно?

Рассмотрим некоторый искусственный пример. Использовавшись ранее в примерах база данных ПОСТАВКА ТОВАРОВ включала информацию о поставщиках (*Номер поставщика — sId, Имя поставщика — sName, Место дислокации — city*), товарах (*Номер товара — pId, Название — pName, Стоимость — price*) и поставках (какой поставщик поставляет тот или иной товар и в каком Количество — qty).

Можно все данные разместить в одном отношении, в котором совокупность атрибутов sId и pId составляет первичный ключ. Схема данного отношения (в дальнейшем изложении оно будет использоваться постоянно):

ПОСТАВКА ИЗДЕЛИЙ (sId, sName, city, pId, pName, price, qty).

Но тогда возникают определенные проблемы:

- избыточность информации, так как, например, информация о поставщиках будет повторяться для каждого поставляемого им товара;
- возможны аномалии при выполнении операций:
 - обновления; если информация дублируется, то при модификации строк возможны ошибки: например, в одной строке изменили стоимость товара, а в другой, где также присутствует информация об этом же товаре, забыли;
 - включения; если нужно хранить информацию, например, о товаре, который еще никто не поставляет, то ее нельзя включить в данное отношение, так как первичный ключ отношения состоит из двух атрибутов — Sid и Pid и атрибуты первичного ключа не могут иметь пустое значение;

- удаления; если, например, удаляется информация о некотором товаре и некоторый поставщик поставляет только один этот товар, вместе с информацией о товаре удалится и информация о поставщике.

Если же базу данных представить тремя отношениями — ПОСТАВЩИК, ТОВАР, ПОСТАВКА, тогда проблемы частично снимаются: аномалии выполнения операций модификации, вставки и удаления снимаются, но некоторая управляемая избыточность информации сохраняется. Но в этом случае, для того чтобы выполнить запрос, включающий в себя полную информацию о поставках товаров, необходимо выполнить операцию соединения отношений, а данная операция является достаточно времязатратной.

Отсюда необходимо решить, какой вариант лучше. И если целесообразно использовать несколько отношений, то каким образом получить нужные отношения.

Эта задача и решается на этапе проектирования реляционной базы данных. Главное требование, которое предъявляется на этапе проектирования, заключается в гарантированном поддержании целостного состояния базы данных, т.е. анализируются и гарантированно поддерживаются ограничения целостности.

Как мы говорили ранее, все ограничения целостности могут быть разделены на два типа: внутренние ограничения, поддерживаемые моделью данных, и явные ограничения, за соблюдение которых отвечает разработчик базы данных.

Реляционная модель данных поддерживает следующие внутренние ограничения:

- категорийная целостность — так как ключ уникально идентифицирует и определяет каждый кортеж отношения, то никакой ключевой атрибут не может быть пустым;
 - ссылочная целостность — значение внешнего ключа дочернего отношения должно быть равно одному из текущих значений первичного ключа родительского отношения.

Явные ограничения, в свою очередь, можно разбить на две группы:

- семантические зависимости — зависимости между атрибутами отношения, определяемые предметной областью, например: сумма бюджетов всех отделов не должна превышать бюджет предприятия;
 - функциональные зависимости — дополнительные ограничения, накладываемые на реляционную схему; ограничения на значения одних атрибутов в зависимости от значений других атрибутов, например: во всех кортежах, где встречается один и тот же номер товара, название и цена товара также должны быть одинаковыми.

Любое априорное знание о различных ограничениях, накладываемых на совокупность данных, может принести большую пользу для достижения указанных целей. Один из способов формализации этих знаний — установление зависимостей между данными. Семантические зависимости могут быть удовлетворены только за счет использования специальных средств — триггеров и хранимых процедур, что требует от разработчика базы данных серьезных усилий на этапе разработки приложения. С другой стороны, знание функциональных зависимостей может привести к такому проектированию базы данных, что эти ограничения будут удовлетворяться без дополнительных усилий на этапе разработки приложения; для этих целей используется теория функциональных зависимостей.

6.2. ФУНКЦИОНАЛЬНЫЕ ЗАВИСИМОСТИ

6.2.1. Основные понятия

Имеем некоторую схему отношения — $R(A_1A_2...A_n)$. *Функциональная зависимость* представляет собой один из возможных типов зависимостей между атрибутами отношения. Она определяет, что:

- значение одного подмножества атрибутов $Y \subseteq R$ зависит от значения другого подмножества $X \subseteq R$. Например, в приведенном выше отношении ПОСТАВКА ИЗДЕЛИЙ атрибут city зависит от атрибута sId;
 - одному и тому же значению X соответствует одно и то же значение Y.

Возможные способы определения функциональных зависимостей:

- есть конкретная реализация отношения $r_1(R)$, и для нее на основании анализа конкретных значений атрибутов можно определить функциональные зависимости;
 - на основе анализа предметной области можно определить функциональные зависимости для всех возможных реализаций отношения $r_1(R), r_2(R), \dots$ в разные моменты времени.

Конечно, для проектирования базы данных необходимо использовать второй способ.

Определение функциональной зависимости. Пусть $R(A_1A_2...A_n)$ — схема отношения с атрибутами из некоторого универсального множества атрибутов $U = \{A_1, A_2, \dots, A_n\}$. Пусть также $X \subseteq U$ и $Y \subseteq U$ — некоторые подмножества множества атрибутов схемы R . Тогда говорят, что Y функционально зависит от X (или X функционально определяет Y) тогда и только тогда, когда для любой допустимой реализации отношения $r(R)$ каждое значение множества атрибутов X связано в точности с одним значением множества атрибутов Y .

Ниже приведены примеры функциональных зависимостей:

- Функциональная зависимость $f: X \rightarrow Y$.
Здесь X — детерминант, а Y — зависимость.

Другими словами, для любой допустимой реализации отношения $r(R)$ если какие-то два кортежа имеют одинаковые значения атрибутов из X , они обязательно имеют и одинаковые значения атрибутов из Y : $\pi_Y(\sigma_{X=x}(r))$ — всегда дает только один кортеж для любого значения x атрибутов X из r .

Примеры.

1. Очевидный пример: так как первичный ключ РК однозначно определяет каждый кортеж отношения, $PK \rightarrow A_1A_2...A_n$, а также и любое подмножество атрибутов из U .

2. Из рассматриваемого примера ПОСТАВКА ИЗДЕЛИЙ очевидно, что:

$sId \rightarrow sName$;

$pId \rightarrow pName$;

$(sId, pId) \rightarrow qty$.

Важно! Функциональные зависимости являются утверждением обо всех реализациях отношения, которые удовлетворяют схеме отношения R .

Нельзя, рассматривая конкретную реализацию отношения, на ее основе определить функциональные зависимости.

Рассмотрим пример. Пусть дана некоторая реализация отношения, удовлетворяющая схеме R :

R	sId	pId	qty
S1	P1	100	
S1	P2	100	
S2	P1	200	
S2	P3	200	
S3	P1	100	

Из приведенной реализации можно сделать вывод, что $sId \rightarrow qty$. Но в какой-то следующий момент времени в реализации отношения может появиться кортеж $\langle S1, P3, 200 \rangle$, который нарушает предполагаемую функциональную зависимость.

Как определять функциональные зависимости

Декларация функциональных зависимостей — решение, которое может быть принято только проектировщиком на основе анализа семантики атрибутов. Функциональные зависимости не могут быть доказаны, но они будут претворяться в жизнь средствами СУБД, если это предписано (это определяется установленными ограничениями целостности).

На что влияют функциональные зависимости

1. Функциональные зависимости гарантируют, что СУБД в дальнейшем будет поддерживать определенные ими ограничения целостности.

2. Возможно, обеспечат более эффективную реализацию отношений.

3. Но! Делят невозможным хранение некоторой информации.

3. Но! Делают невозможным хранение некоторой информации.

Рассмотрим пример, иллюстрирующий важность определения функциональных зависимостей. Пусть, например, определена следующая схема отношения:

ОТДЕЛ (Название, Номер помещения, Телефон).

Очевидно, что такая схема отношения определяет функциональную зависимость *Название* → *Телефон*. Возможная реализация отношения:

ОТДЕЛ	(Название)	Номер помещения	Телефон)
Бухгалтерия	128	123-4567	

Это означает, что никакой отель не может иметь несколько телевизоров.

Это означает, что никакой отдел не может иметь несколько телефонов.

Еще один пример. Уже говорилось, что в любом отношении есть функциональная зависимость $PK \rightarrow R$. Если для некоторой схемы R кроме указанной функциональной зависимости существуют еще и другие типа $A \rightarrow B$, то, вообще говоря, схема отношения R будет характеризоваться некоторой избыточностью. Действительно, рассмотрим следующую схему отношения:

ПОСТАВЩИК (Номер поставщика, Имя, Город, Код города).

В этой схеме определена функциональная зависимость
Город → Код города.

Следовательно, в каждом кортеже для каждого значения атрибута *Город* будет повторяться соответствующее значение атрибута *Код города*.

Чем это плохо?
Функциональные зависимости определяют некоторые ограничения на структуру базы данных.

Функциональные зависимости определяют некоторые ограничения целостности, которые должны проверяться при каждом обновлении состояния базы данных. Если таких ограничений много — слишком много времени будет тратиться на их проверку, что конечно же не очень хорошо. Например, если в некоторой реализации приведенного выше отношения ПОСТАВЩИК несколько десятков (или более) кортежей имеют одно и то же значение атрибута *Город* (например, *Москва*) и поменялся код этого города, необходимо внести изменения во все кортежи отношения.

Таким образом, функциональные зависимости рассматриваются

Таким образом, функциональные зависимости рассматриваются как средство задания ограничений целостности для схемы отношения R , и они будут проверяться.

Как определить все функциональные зависимости? Можно ли и как определить минимальное количество функциональных зависимостей?

6.2.2. Замыкание множества функциональных зависимостей

Для каждой схемы отношения существует вполне определенное конечное множество функциональных зависимостей. Некоторые функциональные зависимости определяются проектировщиком из анализа семантики атрибутов. Из них могут быть выведены другие функциональные зависимости. Например, если существует некоторая схема отношения $R(A,B,C)$ и для нее определены функциональные зависимости $A \rightarrow B$ и $B \rightarrow C$, то интуитивно понятно, что существует и функциональная зависимость $A \rightarrow C$.

Действительно, рассмотрим два кортежа $u \in r$ и $v \in r$, где r — некоторая реализация отношения, удовлетворяющая схеме R . Пусть $u[A] = v[A]$. Что можно сказать о $u[B]$ и $v[B]$? $u[C]$ и $v[C]$? Если эти кортежи не совпадают по атрибуту B , т.е. для этих кортежей $u[B] \neq v[B]$, значит, нарушена функциональная зависимость $A \rightarrow B$. Если же кортежи не совпадают по атрибуту C — $u[C] \neq v[C]$, тогда будет нарушена функциональная зависимость $B \rightarrow C$.

Рассмотрим некоторые определения.

Определение. Пусть F — множество функциональных зависимостей для схемы отношения R и имеется еще одна функциональная зависимость $f: X \rightarrow Y$. Говорят, что функциональная зависимость $X \rightarrow Y$ логически следует из F , если для любой реализации отношения $r(R)$, удовлетворяющей всем зависимостям из F , удовлетворяется также и зависимость $X \rightarrow Y$.

Пример. Пусть существует схема отношения $R(A,B,C)$ и для нее определено следующее множество функциональных зависимостей: $F = \{A \rightarrow B, B \rightarrow C\}$. Тогда функциональная зависимость $A \rightarrow C$ логически следует из F .

Определение. Логическим замыканием F (обозначается как F^+) называется полное множество функциональных зависимостей, которые логически следуют из F .

6.2.3 Правила вывода Армстронга

Важно иметь возможность из F получить F^+ . F^+ получается из F с помощью специальных правил вывода.

Правила вывода — это правила, устанавливающие, что если некоторая схема отношения R удовлетворяет определенной функциональной зависимости, то она должна удовлетворять и некоторым другим функциональным зависимостям.

Правила вывода позволяют по заданному множеству F функциональных зависимостей получить F^+ . Причем эти правила являются:

- полными в том смысле, что для заданного множества F функциональных зависимостей эти правила позволяют вывести все функциональные зависимости, принадлежащие F^+ ;
 - надежными (исчерпывающими) в том смысле, что, используя их, нельзя вывести из F некоторую функциональную зависимость, не принадлежащую F^+ .

Рассмотрим правила вывода. Введем следующие обозначения:

- $R(A_1, A_2, \dots, A_n)$ — схема отношения;
 - $U = \{A_1, A_2, \dots, A_n\}$ — универсальное множество атрибутов;
 - $X \subseteq U, Y \subseteq U, Z \subseteq U, W \subseteq U$ — некоторые подмножества атрибутов из U ;
 - XY — объединение атрибутов: $X \cup Y \equiv XY$.

1. Рефлексивность

Если имеется некоторая совокупность атрибутов $X = YZ$ (т.е. Y есть некоторое подмножество из X), тогда $X \rightarrow Y$

Это тривиальная функциональная зависимость, в которой зависимость (правая часть) содержится в детерминанте (левой части).

Из этой зависимости следует (если $Z \equiv \emptyset$) что $X \geq X'$

3. Пополнение или расширение левой части

Если существует функциональная зависимость $X \rightarrow Y$, то $XZ \rightarrow YZ$.

Важно, что функциональная зависимость $X \rightarrow Y$ или принадлежит F , или может быть логически выведена из F с помощью описываемых правил.

Пример. Пусть дана схема отношения $R(A, B, C, D)$, для которой определена функциональная зависимость $A \rightarrow B$. Рассмотрим некоторую реализацию отношения.

Дано				Следует								
R	(A	B	C	D)	A	→	B	A	C	→	B	C
	a1	b1	c1	d1	a1		b1	a1	c1		b1	c1
	a2	b2	c1	d1	a2		b2	a2	c1		b2	c1
	a1	b1	c1	d2	a1		b1	a1	c1		b1	c1
	a3	b2	c2	d3	a3		b2	a3	c2		b2	c2
	a1	b1	c2	d2	a1		b1	a1	c2		b1	c2

В соответствии с правилом 4, так как $X \rightarrow Y$ (по условию) и $YX \rightarrow YZ$ (выведено), имеем $XX \rightarrow YZ$ (или $X \rightarrow YZ$).

6. Декомпозиция (проективность)

Если $X \rightarrow YZ$, то $X \rightarrow Y$ и $X \rightarrow Z$.

Иллюстрирует данное правило пример, приведенный выше, в котором

тором просто нужно поменять местами пометки «Дано» и «Следует».

Доказательство. Дано: $X \rightarrow YZ$. Требуется получить $X \rightarrow Y$ и

X → Z. EUGENE HARRY MAYER, EGOSOCIOLOGY, HARRY MAYER, ZOROSOCIOLOGY, HARRY UHLEN, ZOROSOCIOLOGY, HARRY STEVENS, ZOROSOCIOLOGY, HARRY NIMMER, EGOSOCIOLOGY, HARRY MAYER, ZOROSOCIOLOGY, HARRY NIMMER, EGOSOCIOLOGY

В соответствии с правилом $1 \text{YZ} \rightarrow Y$ и $YZ \rightarrow Z$.

¹ В соответствии с правилом 3 (частный случай правила 4) из УМС МПАСА № 11-УМС-МПАСА-2010.

$X \rightarrow YZ$ и $YZ \rightarrow Y$ следует $X \rightarrow Y$; аналогично из $X \rightarrow YZ$ и $YZ \rightarrow Z$ следует $X \rightarrow Z$.

следует $X \rightarrow Z$.

7. Композиция

Если $X \rightarrow Y$ и $Z \rightarrow W$, то $XZ \rightarrow YW$.

Пример:

Дано Следует

дано				следует												
R	(A	B	C	D)	A	→	B	и	C	→	D	A	C	→	B	D
	a1	b1	c1	d1	a1		b1		c1		d1	a1	c1		b1	d1
	a2	b2	c1	d1	a2		b2		c1		d1	a2	c1		b2	d1
	a1	b1	c1	d1	a1		b1		c1		d1	a1	c1		b1	d1
	a3	b3	c2	d3	a3		b3		c2		d3	a3	c2		b3	d3

Доказательство. Дано: $X \rightarrow Y$ и $Z \rightarrow W$. Требуется получить $(X \wedge Z) \rightarrow (Y \wedge W)$.

В соответствии с правилом 2 из $X \rightarrow Y$ следует $XZ \rightarrow YZ$, а из $Z \rightarrow W$ следует $YZ \rightarrow YW$.

В соответствии с правилом 3 из $XZ \rightarrow YZ$ и $YZ \rightarrow YW$ следует $XZ \rightarrow YW$

6.2.4. Определение ключа

Ключ отношения определяется на основе функциональных зависимостей.

Рассмотрим некоторые определения.

Определение. Множество атрибутов Y функционально полно зависит от множества атрибутов X, если Y функционально зависит от X и не зависит функционально от любого собственного подмножества X.

Другими словами, если есть $X \rightarrow Y$, то для любого $Z \subset X$ нет зависимости $Z \rightarrow Y$ и, следовательно, $\neg(X \rightarrow Y)$.

Если детерминант представлен единственным атрибутом, тогда никакие проблемы при определении ключа не возникают. Если же ключ представлен совокупностью нескольких атрибутов, требуется проверить, действительно ли имеет место функционально полная зависимость.

Например, рассмотрим следующее отношение:

**ПОСТАВКА ИЗДЕЛИЙ (Sid, SNAME, CITY, Pid, PNAME, PRICE,
QTY).**

Для данного отношения определена функциональная зависимость $(\text{Sid}, \text{Pid}) \rightarrow \text{QTY}$, и она является функционально полной зависимостью.

С другой стороны, для этого же отношения определена и функциональная зависимость $(\text{Sid}, \text{Pid}) \rightarrow \text{CITY}$, но она не является функционально полной, так как существует функциональная зависимость $\text{Sid} \rightarrow \text{CITY}$.

Определение. Если R — схема отношения с атрибутами A_1, A_2, \dots, A_n и множеством функциональных зависимостей F , $X \subseteq U$ — некоторое подмножество атрибутов $\{A_1, A_2, \dots, A_n\}$, то X называется **ключом R** , если:

- а) функциональная зависимость $X \rightarrow A_1A_2...A_n$ принадлежит F^+ ,
 б) ни для какого собственного подмножества $Y \subset X$ функциональная зависимость $Y \rightarrow A_1A_2...A_n$ не принадлежит F^+ .

Другими словами, $X \rightarrow A_1A_2...A_n$ есть функционально полная зависимость.

Если X – первичный ключ отношения R , тогда из функциональной зависимости $X \rightarrow A_1A_2...A_n$ в соответствии с правилом 6 следуют функциональные зависимости $X \rightarrow A_i$ для $i = 1, 2, \dots, n$.

6.2.5. Декомпозиция с соединением без потерь

При проектировании базы данных может возникнуть ситуация, когда некоторое отношение должно быть разбито на несколько других отношений. Такой процесс разбиения называется *декомпозицией отношения*. При этом, поскольку отношение характеризуется схемой и имеет некоторую реализацию, декомпозиция отношения затрагивает и схему отношения, и его реализацию.

Определение. Декомпозицией схемы отношения $R(A_1, A_2, \dots, A_n)$ называется замена ее совокупностью подмножеств $\{R_i\}$ схемы R (подсхем) таких, что $R_1 \cup R_2 \cup \dots \cup R_k = R$. При этом необязательно, чтобы R_i были непересекающимися.

Декомпозиция реализации отношения определяется как совокупность проекций отношения на подсхемы, полученные при декомпозиции.

Ниже приведены примеры декомпозиции отношений:

- Пример 1.** Рассмотрим отношение R , определенное на множестве городов, имеющих различные функциональные зависимости:

В результате получили лишние строки, отсутствовавшие в исходном отношении.

Таким образом, возникает вопрос: как разбивать исходное отношение, чтобы не потерять информацию? Эта проблема получила название **декомпозиции с соединением без потери информации**. Вопрос о том, происходит ли потеря информации, тесно связан с функциональными зависимостями.

Определение. Пусть R — схема отношения, в результате декомпозиции которой получены схемы R_1, R_2, \dots, R_k , и F — множество функциональных зависимостей для R . Говорят, что эта декомпозиция обладает свойством соединения без потерь относительно F , если каждая реализация $r(R)$, удовлетворяющая F , может быть представлена в виде:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r).$$

Теорема. Пусть $R(A, B, C)$ — схема отношения с атрибутами (подмножествами атрибутов) A, B и C . Если данная схема отношения удовлетворяет функциональной зависимости $A \rightarrow B$, то осуществляется декомпозиция без потери на $\{R_1, R_2\}$, где $R_1 = \{A, B\}$ и $R_2 = \{A, C\}$.

Вернемся к приведенному выше примеру: $S(sId, status, city)$. Так как sId — первичный ключ отношения, то $sId \rightarrow city$. Отсюда декомпозиция без потери осуществляется на подсхемы $S1(sId, city)$ и $S2(sId, status)$. Функциональные зависимости $status \rightarrow sId$ или $status \rightarrow city$ отсутствуют, поэтому второй пример декомпозиции не сохраняет исходное отношение.

Но здесь может возникнуть еще одна проблема. Предположим, что в условиях данной задачи поставщик дислоцирован в конкретном городе, а каждый город имеет определенный статус. Следовательно, для приведенной схемы отношения имеют место следующие функциональные зависимости:

- $sId \rightarrow city$;
- $city \rightarrow status$.

В соответствии с теоремой можно предложить два способа декомпозиции без потери информации:

- a) относительно $sId \rightarrow city$: $S11(sId, city)$ и $S12(sId, status)$;
- b) относительно $city \rightarrow status$: $S21(city, sId)$ (или $S21(sId, city)$) — порядок задания имен атрибутов никакого значения не имеет) и $S22(city, status)$.

Какой из этих двух способов лучше?

Чтобы ответить на этот вопрос, введем еще одно определение.

Определение. Проекцией множества функциональных зависимостей F

на множество атрибутов Z ($\pi_Z(F)$) называется множество зависимостей $X \rightarrow Y$ в F^+ , таких, что $XY \subseteq Z$.

Рассмотрим все тот же пример.

Для исходного отношения определено следующее множество функциональных зависимостей:

$$F = \{s[d \rightarrow \text{city}, \text{city} \rightarrow \text{status}]\}$$

Из них по правилам вывода можно вывести следующие зависимости:

- $sId \rightarrow \text{status}$ — правило транзитивности;
 - $sId \rightarrow (\text{city}, \text{status})$ — правило объединения.

Отсюда,

$$F^+ = \{sId \rightarrow city, city \rightarrow status, sId \rightarrow status, sId \rightarrow (city, status)\}.$$

Для способа а) декомпозиции определены следующие функциональные зависимости:

- $sId \rightarrow status$ — из схемы отношения S11(sId, status);
 - $sId \rightarrow city$ — из схемы отношения S12(sId, city).

Для способа б) декомпозиции определены следующие функциональные зависимости:

- $sId \rightarrow city$ — из схемы отношения S21(sId , city);
 - $city \rightarrow status$ — из схемы отношения S22(city, status).

Определение. Декомпозиция сохраняет множество функциональных зависимостей F , если из объединения всех зависимостей, принадлежащих $\pi_{R_i}(F)$, для $i = 1, 2, \dots, k$, логически следуют все зависимости, принадлежащие F .

Из функциональных зависимостей, сохранившихся при способе декомпозиции а), нельзя логически вывести зависимость $\text{city} \rightarrow \text{status}$, т.е. данная декомпозиция не сохраняет функциональные зависимости.

Для способа декомпозиции б) все функциональные зависимости из Е сохранены.

Что дает сохранение функциональных зависимостей? Рассмотрим следующий пример.

Пусть дана следующая реализация отношения:

Выполним декомпозицию этого отношения способом а):

Выполним декомпозицию этого отношения способом а):

<u>S11</u>	<u>(sId, city)</u>	<u>S12</u>	<u>(sId, status)</u>
	S1 N1		S1 30
	S2 N2		S2 30
	S3 N3		S3 40

Предположим, что поставщик S2 переехал в город N3. Тогда в отношении S11 появится кортеж $\langle S2, N3 \rangle$. Но в отношении S12 сохраняется кортеж $\langle S2, 30 \rangle$, в соответствии с которым город N3 приобретает статус 30, что неправильно. Следовательно, для поддержания достоверного состояния реализаций потребуется изменить не только кортеж в S11, но и кортеж в S12, т.е. требуется зависимое обновление двух отношений.

Если же выполнить декомпозицию отношения способом б):

<u>S21 (sId, city)</u>			<u>S22 (city, status)</u>	
HARRYMANI-EGGSCWE	HARRYMANI	EGGSCWE	HARRYMANI	EGGSCWE
HARRYMANI-EGGSCWE	HARRYMANI	EGGSCWE	HARRYMANI	EGGSCWE
HARRYMANI-EGGSCWE	HARRYMANI	EGGSCWE	HARRYMANI	EGGSCWE

Тогда достаточно изменить кортеж в отношении S21: $\langle S2, N3 \rangle$; статус города N3 определен в отношении S22 независимо от того, какой поставщик в этом городе дислоцирован.

6.3. НОРМАЛИЗАЦИЯ ОТНОШЕНИЙ

Процесс проектирования реляционной базы данных предполагает выполнение операций нормализации отношений, использующих концепцию нормальных форм.

Определение. Отношение находится в некоторой нормальной форме, если оно удовлетворяет заданному набору условий.

Процесс нормализации был впервые предложен Коддом. Сначала были предложены только три нормальные формы: первая (1НФ), вторая (2НФ) и третья (3НФ). Затем Бойс и Кодд сформулировали более строгое определение 3НФ, которое получило название *нормальной формы Бойса–Кодда* (НФБК). Все эти нормальные формы основаны на функциональных зависимостях, существующих между атрибутами отношения.

Вслед за НФБК Fagin привел определения четвертой (4НФ) и пятой (5НФ) нормальных форм. Однако на практике эти нормальные формы более высоких порядков используются крайне редко.

Все эти нормальные формы можно представить следующим образом (рис. 6.1).

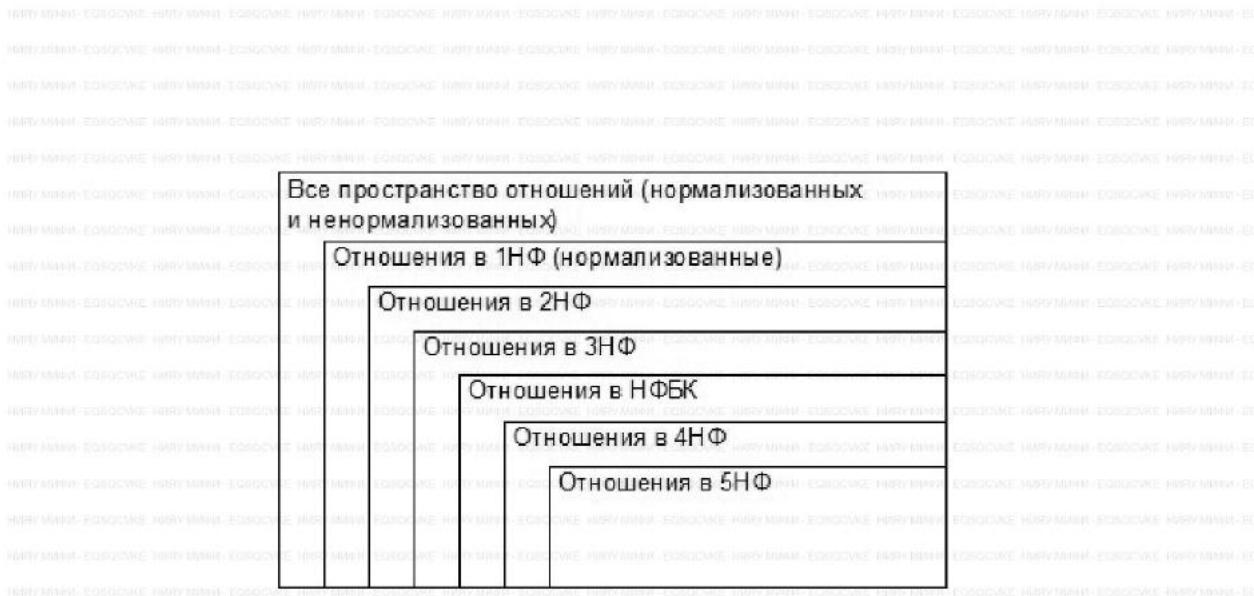


Рис. 6.1. Нормальные формы

Из этой диаграммы видно, что каждая более жесткая нормальная форма добавляет дополнительные условия к предыдущей: 2НФ — это 1НФ плюс дополнительные условия. Это означает, что отношение не может находиться в 2НФ и не находиться в 1НФ, но может находиться в 2НФ и не находиться в 3НФ.

6.3.1. Первая нормальная форма

Определение. Отношение находится в 1НФ тогда и только тогда, когда все используемые домены содержат только скалярные (атомарные, простые) значения.

Рассмотрим следующий пример. Пусть необходимо хранить информацию о поставщиках и поставляемых ими товарах. Информация о поставщиках включает в себя:

- номер поставщика sId, определенный на домене **НОМЕР**;
- имя поставщика sName, определенное на домене **ИМЯ**;
- город city, в котором размещается поставщик, определенный на домене **ГОРОД**;
- код города code, определенный на домене **КОД**.

Информация о поставляемых товарах включает в себя:

- код товара pId, определенный на домене **НОМЕР**;
- название товара pName, определенное на домене **НАЗВАНИЕ**;
- стоимость товара price, определенная на домене **ДЕНЬГИ**;
- количество поставляемых товаров qty, определенное на домене **КОЛИЧЕСТВО**.

Информация о поставляемых товарах ps может быть задана на составном домене **ПОСТАВКА: ПОСТАВКА (НОМЕР, НАЗВАНИЕ, ДЕНЬГИ, КОЛИЧЕСТВО)**.

Тогда можно определить следующее ненормализованное отношение ПОСТАВКА ТОВАРОВ:

ПОСТАВКА ТОВАРОВ (*sId : НОМЕР, sName : ИМЯ, city : ГОРОД, code : КОД, ps : ПОСТАВКА*).

Исходя из анализа семантики атрибутов, определим функциональные зависимости между ними:

$sId \rightarrow sName$, $sId \rightarrow city$, $city \rightarrow code$, $sId \rightarrow ps$.

Применяя правило транзитивности, получим функциональную зависимость $sId \rightarrow code$. Кроме того, из правила рефлексивности следует $sId \rightarrow sId$.

Отсюда, применив правило аддитивности, получим $sId \rightarrow (sId, sName, city, code, ps)$; следовательно, sId является первичным ключом данного отношения.

Возможная реализация такого отношения приведена в табл. 6.1.

Таблица 6.1

Ненормализованное отношение ПОСТАВКА ТОВАРОВ

Домен	НОМЕР	ИМЯ	ГОРОД	КОД	ПОСТАВКА (НОМЕР, НАЗВАНИЕ, ДЕНЬГИ, КОЛИЧЕСТВО)			
Атрибут	sId	sName	city	code	ps (pId, pName, price, qty)			
	S1	Smith	London	20	P1,	Nut,	12,	200
					P2,	Bolt,	17,	100
					P3,	Screw,	17,	100
	S2	Jones	Paris	10	P1,	Nut,	12,	150
					P2,	Bolt,	17,	200

Это ненормализованное отношение, так как значениями атрибута `ps` являются множества.

Данное отношение легко преобразовать в 1НФ. Для этого достаточно вместо составного домена ввести составляющие его простые домены, вместо одного атрибута *ps* ввести необходимые атрибуты, определенные на простых доменах, и повторить значения атрибутов *sId*, *sName*, *city*, *code* для каждой строки атрибута *ps* (табл. 6.2).

Определим функциональные зависимости данного отношения. Для поставщика сохраняются рассмотренные выше функциональные зависимости: $sId \rightarrow sName$, $sId \rightarrow city$, $city \rightarrow code$, и из правила транзитивности следует $sId \rightarrow code$. Из анализа семантики атрибутов, определяющих товар, находим следующие функциональные зависимости: $pId \rightarrow pName$, $pId \rightarrow price$. Значение атрибута qty не является собственным свойством ни товара, ни поставщика и зависит от совокупности атрибутов: $(sId, pId) \rightarrow qty$.

Таблица 6.2

Нормализованное отношение ПОСТАВКА ТОВАРОВ

Домен	Номер	Имя	Город	Код	Номер	Название	Деньги	Количество
Атрибут	sId	sName	city	code	pId	pName	price	qty
	S1	Smith	London	20	P1	Nut	12	200
	S1	Smith	London	20	P2	Bolt	17	100
	S1	Smith	London	20	P3	Screw	17	100
	S2	Jones	Paris	10	P1	Nut	12	150
	S2	Jones	Paris	10	P2	Bolt	17	200

Докажем, что если есть функциональная зависимость $X \rightarrow Y$, то существует и функциональная зависимость $XZ \rightarrow Y$. Так как есть $X \rightarrow Y$, то, применяя правило пополнения, получим $XZ \rightarrow YZ$. Из правила рефлексивности следует, что $YZ \rightarrow Y$. Отсюда, имея $XZ \rightarrow YZ$ и $YZ \rightarrow Y$ и применяя правило транзитивности, получим $XZ \rightarrow Y$, что требовалось доказать. Применяя выведенное соотношение к функциональным зависимостям, определенным для отношения ПОСТАВКА ТОВАРОВ, получим:

$(sId, pId) \rightarrow sName$ $(sId, pId) \rightarrow city$ $(sId, pId) \rightarrow code$

(sId, pId) \rightarrow sName, (sId, pId) \rightarrow city, (sId, pId) \rightarrow code,
 $(sId, pId) \rightarrow pName$ $(sId, pId) \rightarrow price$ $(sId, pId) \rightarrow qty$ (запахо):

$(sId, pId) \rightarrow pName, (sId, pId) \rightarrow pName, (sId, pId) \rightarrow \text{цту}$ (задание)

Отсюда, применив правило аддитивности, получим

т.е. атрибуты sId , pId образуют первичный ключ данного отношения.
Недостатки 1НФ: для нее существуют определенные аномалии

Главное, что свойственно ИНФ, — это избыточность информации, которая может привести к нарушению согласованности данных и даже к невозможности хранения некоторой информации.

Вернемся к нашему примеру и рассмотрим проблемы, которые могут возникнуть при манипулировании данными в таком отношении, находящемся в 1НФ.

Проблема обновления. Если поставщик S1 поставляет несколько товаров, в каждом кортеже отношения повторяются его имя, город и код города. Если, например, поставщик переезжает в другой город, для него изменяются значения атрибутов city и code. Следовательно, необходимо изменить информацию во всех кортежах, где упоминается данный поставщик. Если хотя бы в одном кортеже информация не будет обновлена, это приведет к нарушению достоверности данных.

Проблема вставки. Так как первичный ключ данного отношения является составным и атрибуты первичного ключа не могут иметь пустые значения, в таком отношении невозможно зарегистрировать нового поставщика, если он не поставляет ни одного товара, или новый товар, если он никем не поставляется.

Проблема удаления. Если некоторый товар изымается из поставок, как быть с поставщиками, которые в данный момент поставляют только этот товар? Удаляя весь кортеж, мы теряем информацию о поставщике.

Почему это так? Причиной таких аномалий является то, что в отношении, находящемся в 1НФ, все неключевые атрибуты *функционально* (а не функционально *полно*) зависят от атрибутов первичного ключа.

6.3.2. Вторая нормальная форма

Определение. Отношение находится в 2НФ тогда и только тогда, когда оно находится в 1НФ и каждый его неключевой атрибут функционально полно зависит от любого возможного ключа этого отношения.

Продолжим рассмотрение нашего примера. Так как первичным ключом отношения являются атрибуты (sId , pId), все остальные атрибуты должны функционально полно зависеть от ключа. Рассмотрим некоторые примеры. Так, зависимость $(sId, pId) \rightarrow qty$ является функционально полной, так как атрибут qty не зависит функционально ни от sId , ни от pId . Однако зависимости $(sId, pId) \rightarrow sName$ или $(sId, pId) \rightarrow price$ не являются функционально полными, так как существуют функциональные зависимости $sId \rightarrow sName$ и $pId \rightarrow price$. Следовательно, данное отношение не находится в 2НФ.

Так как $sId \rightarrow sName$, $sId \rightarrow city$, $sId \rightarrow code$, имеем $sId \rightarrow (sName, city, code)$. Аналогично из функциональных зависимостей $pId \rightarrow pName$ и $pId \rightarrow price$ можно вывести зависимость $pId \rightarrow (pName, price)$. Применяя правило декомпозиции на основе первой функциональной зависимости, можем разбить исходное отношение на два: ПОСТАВЩИК ($sId, sName, city, code$) с первичным ключом Sid , и ПОСТАВЛЯЕМЫЙ ТОВАР ($sId, pId, pName, price, qty$) с составным первичным ключом (sId, pId) . В отношении ПОСТАВЛЯЕМЫЙ ТОВАР атрибуты $pName$ и $price$ функционально зависят от атрибута pId , следовательно, данное отношение не находится в 2НФ. Вновь применяя правило декомпозиции на основе функциональной зависимости $pId \rightarrow (pName, price)$, разобьем отношение на два: ТОВАР ($pId, pName, price$) с первичным ключом pId и ПОСТАВКА (sId, pId, qty) с составным первичным ключом (sId, pId) . Таким образом, исходное отношение может быть разбито на три:

- ПОСТАВЩИК (sId, sName, city, code) с первичным ключом sId;
 - ТОВАР (pId, pName, price) с первичным ключом pId;
 - ПОСТАВКА (sId, pId, qty) с составным первичным ключом (sId, pId).

Легко убедиться, что все три отношения находятся в 2НФ.

Примечание: если при проектировании базы данных используется модель «сущность–связь» в нотации IDEF1x, тогда 2НФ достигается при соответствующем выборе независимых сущностей, а также при разрешении связей типа «многие-ко-многим». Так, в данном примере ПОСТАВКУ можно рассматривать как неопределенную связь между сущностями ПОСТАВЩИК и ТОВАР: один поставщик поставляет нуль или более товаров, один товар поставляется нулем или более поставщиков (рис. 6.2).

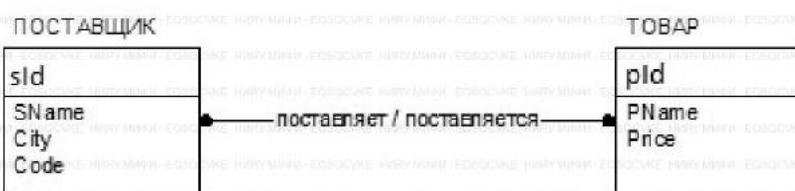


Рис. 6.2 Начальный этап проектирования с использованием IDEE1x

Разрежая неопределенную связь, получаем дополнительную сущность ПОСТАВКА (рис. 6.3).

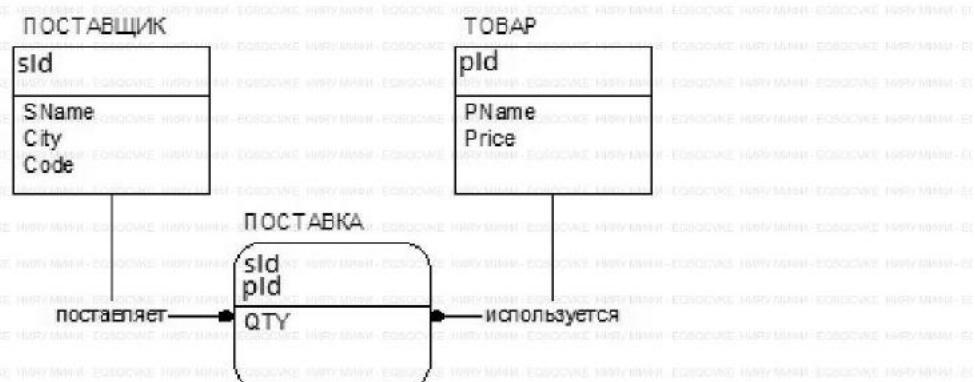


Рис. 6.3 Отношения удовлетворяющие $\exists H\phi$

Полученное в результате отношение ПОСТАВЩИК все еще не
свободно от определенных проблем.

Проблема вставки. Так как определена функциональная зависимость $\text{city} \rightarrow \text{code}$, может возникнуть необходимость включить значение кода для определенного города. Однако в отношении ПО

СТАВЩИК нельзя включить такую информацию, если в данном

СТАВЩИК нельзя включить такую информацию, если в данном городе нет ни одного поставщика.

Проблема обновления. Если меняется код некоторого города, необходимо изменить надлежащим образом информацию во всех кортежах отношения, в которых упоминается данный город.

Проблема удаления. Если в некотором городе размещен только один поставщик и информация об этом поставщике удаляется, то удаляется и информация о городе, которая в дальнейшем может представлять некоторый интерес.

Причиной таких аномалий является то, что в отношении ПОСТАВЩИК существует транзитивная зависимость: из $sId \rightarrow city$ и $city \rightarrow code$ следует $sId \rightarrow code$.

6.3.3. Третья нормальная форма

Определение. Отношение находится в ЗНФ тогда и только тогда, когда оно находится в 2НФ и каждый его не ключевой атрибут не транзитивно зависит от ключа

Так как в отношении ПОСТАВЩИК есть транзитивная зависимость $sId \rightarrow code$, используя рассмотренное выше правило декомпозиции с сохранением функциональных зависимостей, разобьем его на следующие два: ПОСТАВЩИК (sId , $sName$, $city$) и ГОРОД ($city$, $code$). В первом отношении первичный ключ sId сохраняется, а во втором отношении первичным ключом является атрибут $city$.

Примечание: если при проектировании базы данных используется IDEF1x, тогда на этапе построения полноатрибутной концептуальной схемы для каждой сущности определяются неключевые атрибуты. При этом:

- устанавливаются владельцы каждого атрибута (т.е. фактически определяются функциональные зависимости между атрибутами);
 - применяется правило отсутствия повторений.

Для сущности ПОСТАВЩИК владельцем атрибута city (место размещения поставщика) является поставщик, а владельцем атрибута code (код города) является атрибут city. В этом случае выделяется новая сущность — в данном примере ГОРОД (рис. 6.4).

Второй пример. Если значение некоторого «неудобного» атрибута встречается часто, вместо этого атрибута лучше выделить отдельную сущность, экземпляры которой представляют собой некоторые справочные данные. Так, если определяется сущность СОТРУДНИК с атрибутом *Должность*, значениями которого являются длинные строки, например «Инженер первого разряда» или «Инженер высшей категории», тогда имеет смысл выделить справочную сущность ЛОЖНОСТЬ (рис. 6.5).

ПОСТАВЩИК	
id	Атрибут Code - код города - уточняет атрибут City: City -> Code
SName	
City	
Code	



Рис. 6.4. Выделение транзитивных зависимостей

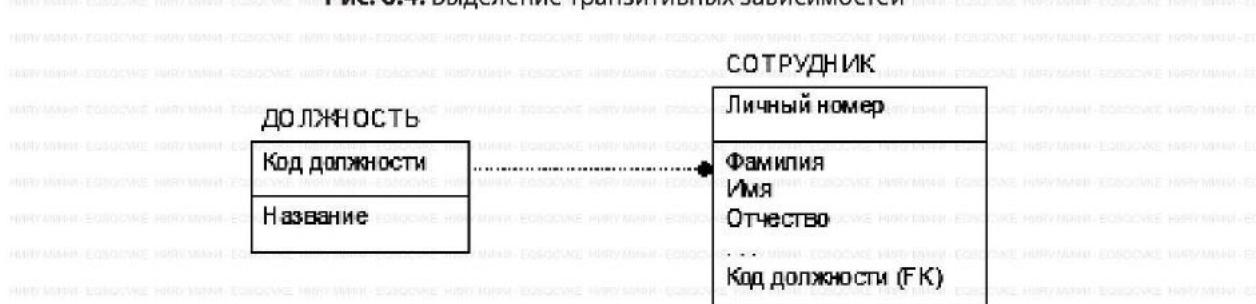


Рис. 6.5. Выделение справочных сущностей

6.3.4. Нормальная форма Бойса–Кодда

В рассмотренных выше примерах в отношениях был определен только один ключ. Однако в конкретной предметной области может иметь место следующая ситуация:

- отношение имеет несколько потенциальных ключей;
 - потенциальные ключи являются составными;
 - потенциальные ключи имеют общие атрибуты.

Такие ситуации на практике встречаются не очень часто, поэтому НФБК была определена позднее первых трех.

Рассмотрим следующий пример. Пусть определено отношение **ОБУЧЕНИЕ** с атрибутами S (студент), Р (предмет) и Т (преподаватель). Для этого отношения в предметной области определены следующие условия (функциональные зависимости).

1. Каждый студент изучает некоторый предмет под руководством только одного преподавателя;

Только одного преподавателя.

(S, P) → T

первой функциональной зависимости всегда можно узнать, какой именно предмет изучает студент под руководством конкретного преподавателя. Однако следует отметить, что такая декомпозиция не сохраняет функциональные зависимости, а значит, некоторые данные в этих двух отношениях не могут обновляться независимо. Например, если вместо какого-либо преподавателя, проводящего занятия по предмету, назначается другой преподаватель, необходимо изменить соответствующие кортежи и в отношении E_1 , и в отношении E_2 .

Примечание: при проектировании с использованием IDEF1x подобная ситуация выглядит следующим образом. В соответствии с заданными функциональными зависимостями определяются сущности ПРЕДМЕТ, ПРЕПОДАВАТЕЛЬ и ПРЕДМЕТ СТУДЕНТА (предмет, изучаемый студентом). Очевидно, что имеются связи типа 1 : n между сущностями ПРЕДМЕТ и ПРЕПОДАВАТЕЛЬ, ПРЕДМЕТ и ПРЕДМЕТ СТУДЕНТА (рис. 6.6).

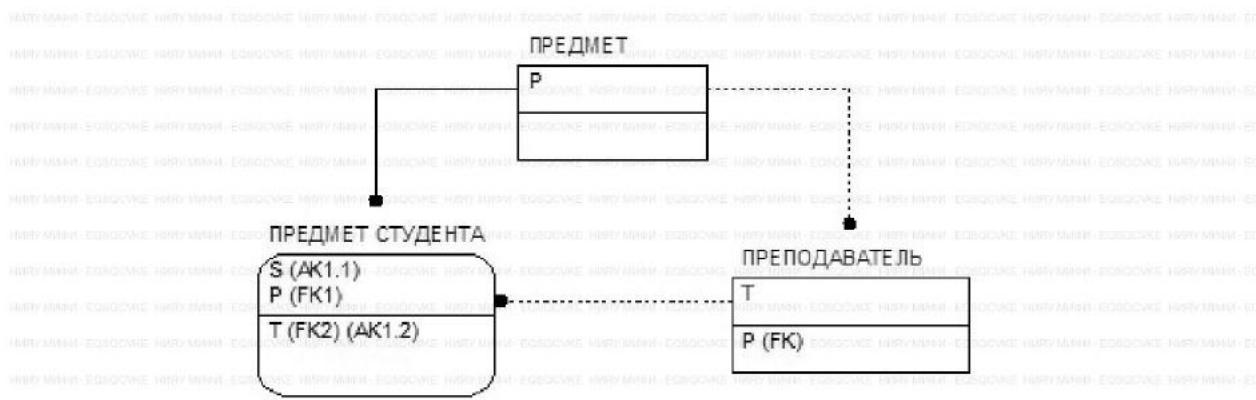


Рис. 6.6. Отображение требований предметной области: нарушение НФБК

Поскольку, как было показано выше, отношение ПРЕДМЕТ СТУДЕНТА имеет второй ключ, схема может быть представлена и по-другому (рис. 6.7).

Кроме того, так как студент изучает предмет под руководством определенного преподавателя, имеет место и связь между сущностями ПРЕДМЕТ СТУДЕНТА и ПРЕПОДАВАТЕЛЬ. В результате сущность ПРЕДМЕТ СТУДЕНТА не удовлетворяет НФБК.

Из концептуальной схемы, приведенной на рис. 6.7, видно, что связь между сущностями ПРЕДМЕТ и ПРЕДМЕТ СТУДЕНТА является лишней: так как каждый преподаватель ведет занятия только по одному предмету, можно выяснить, какой предмет изучает студент, зная, какой преподаватель проводит занятия с этим студентом.

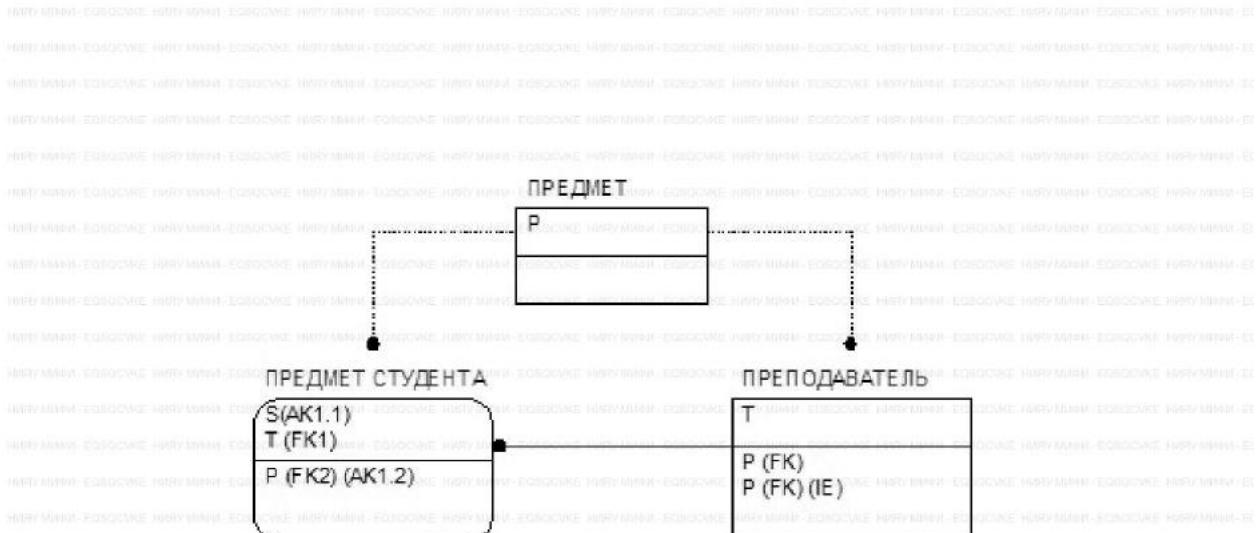


Рис. 6.7. Второй вариант отображения требований предметной области

Эту связь следует удалить, в результате чего сущность **ПРЕДМЕТ СТУДЕНТА** будет удовлетворять НФБК (рис. 6.8).

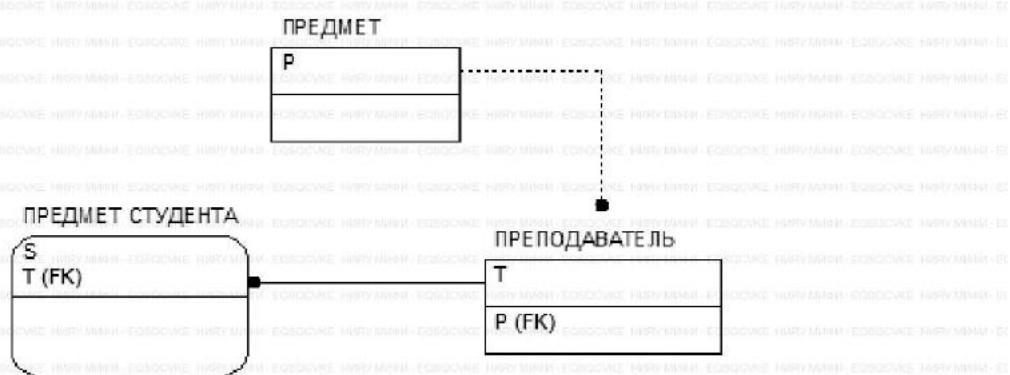


Рис. 6.8. Обеспечение НФБК

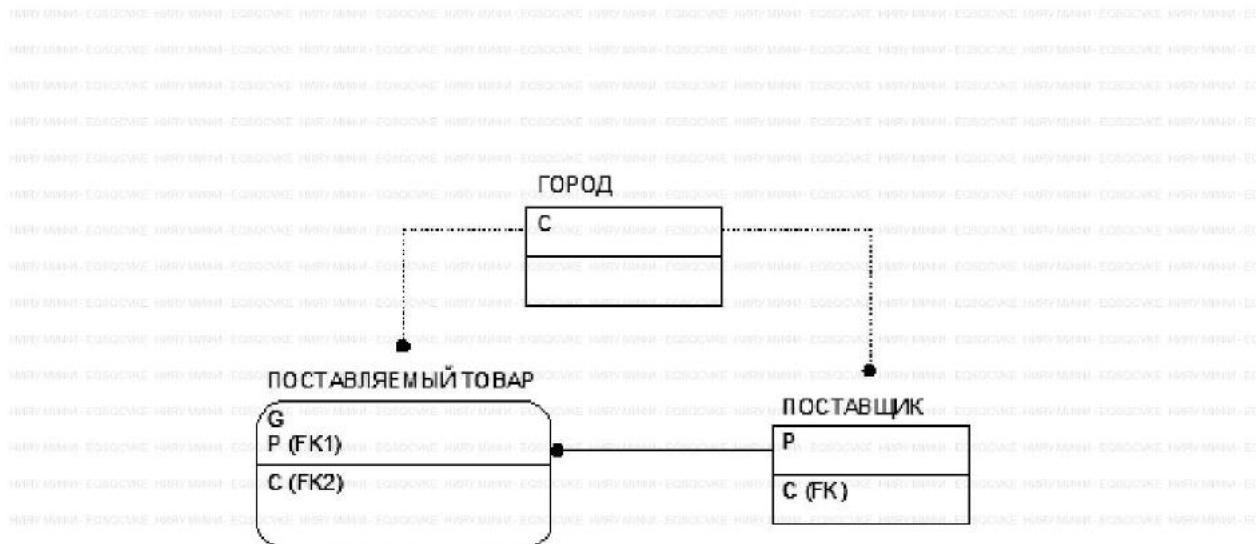
Следует иметь в виду, что подобные связи не всегда являются лишними.

Рассмотрим следующий пример. Имеются сущности **ПОСТАВЩИК** и **ТОВАР**, для которых определено следующее условие: каждый товар поставляется только одним поставщиком (но поставщик может поставлять много товаров). Кроме того, имеется сущность **ГОРОД**. Каждый поставщик дислоцируется в определенном городе, и каждый товар производится в определенном городе (рис. 6.9).

Можно увидеть, что данная схема внешне напоминает схему, приведенную на рис. 6.7.

Однако в данном примере связь между сущностями **ГОРОД** и **ПОСТАВЛЯЕМЫЙ ТОВАР** не является лишней, так как поставщик может быть дислоцирован в одном городе, а поставляемый им товар

112

**Рис. 6.9.** Схема, удовлетворяющая НФБК

производится в другом городе. Это означает, что, по сравнению с предыдущим примером здесь отсутствует функциональная зависимость $(G, C) \rightarrow P$ и сущности ПОСТАВЛЯЕМЫЙ ТОВАР нет альтернативного ключа (G, C) .

6.3.5. Многозначные зависимости

Рассмотрим некоторое ненормализованное отношение, задающее расписание занятий:

Учебный курс	День занятий	Студент
C1	Понедельник Среда	Иванов Петров Сидоров
C2	Пятница	Мягков Быков

Каждый кортеж такого отношения содержит индекс учебного курса, список дней недели, когда проводятся занятия, и список студентов, изучающих данный курс. Такое расписание означает, что занятия по каждому курсу проводятся во все указанные дни недели и все студенты посещают все занятия по курсу.

Предположения, которые могут быть сделаны.

1. Каждый курс может иметь произвольное количество дней занятий.
2. Каждый курс могут изучать произвольное количество студентов.
3. Дни занятий и студенты совершенно не зависят друг от друга, т.е. независимо от дня занятий состав группы студентов один и тот же.
4. День занятий может быть связан с любыми курсами.
5. Каждый студент может быть связан с любым курсом.

Ниже приведено описание отношения CDS:

Преобразуем данное отношение в нормализованное отношение

CDS:

CDS	Учебный курс	День занятий	Студент
	C1	Понедельник	Иванов
	C1	Понедельник	Петров
	C1	Понедельник	Сидоров
	C1	Среда	Иванов
	C1	Среда	Петров
	C1	Среда	Сидоров
	C2	Пятница	Мягков
	C2	Пятница	Быков

Важно отметить, что для рассматриваемых данных функциональные зависимости, кроме тривиальных, не заданы. Нормализованное отношение CDS означает, что кортеж

<Учебный курс:c, День занятий:d, Студент:s>

появляется в отношении тогда и только тогда, когда занятия по курсу с проводятся в день недели d и посещаются студентом s.

Тогда, принимая во внимание допустимость существования для заданного отношения всех возможных комбинаций дней занятий и студентов, можно утверждать, что для отношения CDS верно следующее ограничение: если в отношении существуют кортежи **<c, d1, s1>** и **<c, d2, s2>**, то также присутствуют и кортежи **<c, d2, s1>** и **<c, d1, s2>**.

Очевидно, что в данном отношении имеет место избыточность, которая может привести к проблемам.

Проблема вставки. Чтобы добавить в приведенное отношение информацию о том, что занятия по курсу C2 могут проводиться еще и в четверг, надо включить в отношение два кортежа: **<C2, четверг, Мягков>** и **<C2, четверг, Быков>**.

Проблема обновления. Чтобы перенести, например, день занятий по курсу C2 с пятницы на вторник, надо изменить данные в двух кортежах.

Проблема удаления. Чтобы отменить, например, занятия в понедельник по курсу C1, надо удалить из отношения три кортежа.

Тем не менее отношение CDS находится в НФБК, так как все атрибуты отношения образуют первичный ключ.

Интуитивно ясно, что эти проблемы вызваны тем, что студенты и дни занятий никак не связаны друг с другом. Можно исправить эту ситуацию, если разбить данное отношение на два:

CD(Учебный курс, День занятий) и CS (Учебный курс, Студент).

CD(*Учебный курс, День занятий*) и CS (*Учебный курс, Студент*).

CD	Учебный курс	День занятий	CS	Учебный курс	Студент
	C1	Понедельник		C1	Иванов
	C1	Среда		C1	Петров
	C2	Пятница		C1	Сидоров
				C2	Мягков
				C2	Быков

Оба отношения находятся в НФБК; более того, исходное отношение CDS может быть восстановлено с помощью операции естественного соединения отношений CD и CS, т.е. выполнена декомпозиция без потери информации, но эта декомпозиция выполнена не на основе функциональной зависимости.

Эти интуитивные идеи были теоретически сформулированы Фейгином (Fagin) в 1971 г. с помощью понятия многозначных зависимостей. Приведенная выше декомпозиция выполняется на основе многозначных зависимостей.

Определение. Пусть A, B, C — некоторое произвольное подмножество атрибутов схемы отношения $R(A, B, C)$. Тогда B многозначно зависит от A ($A \rightarrow\rightarrow B$) тогда и только тогда, когда множество значений B , соответствующее заданной паре $\langle A:a, C:c \rangle$ отношения R , зависит только от A , но не зависит от C .

Фейгин показал, что для данного отношения $R(A, B, C)$ многозначная зависимость $A \rightarrow\rightarrow B$ выполняется тогда и только тогда, когда также выполняется многозначная зависимость $A \rightarrow\rightarrow C$.

Таким образом, многозначные зависимости всегда образуют пары: $A \rightarrow B$ | C .

В нашем примере имеет место многозначная зависимость

Учебный курс ⇒⇒ День занятий | Студент

Всякая функциональная зависимость является многозначной, но не всякая многозначная зависимость является функциональной.

Фейгин сформулировал следующую теорему.

Теорема. Пусть A, B, C – некоторые множества атрибутов схемы отношения $R(A, B, C)$. Отношение R будет равно соединению его проекций $R_1(A, B)$ и $R_2(A, C)$ тогда и только тогда, когда для отношения R выполняется многозначная зависимость $A \rightarrow\rightarrow B$.

Ниже приведены примеры для иллюстрации различных форм нормализации:

- Нормализация до 1НФ**: если в сущности есть атрибуты, значение которых не является однозначным образом определяемым из оставшихся атрибутов.
- Нормализация до 2НФ**: если в сущности есть атрибуты, значение которых не является функционально зависимым от остальных атрибутов.
- Нормализация до 3НФ**: если в сущности есть атрибуты, значение которых не является функционально зависимым от других атрибутов, кроме того, что это следствие 2НФ.
- Нормализация до 4НФ**: если в сущности нет атрибутов, значение которых зависит от группы атрибутов, а не от отдельного атрибута.

6.3.6. Четвертая нормальная форма

Определение. Отношение R находится в 4НФ тогда и только тогда, когда в случае существования многозначной зависимости $A \rightarrow\rightarrow B$ все атрибуты отношения R функционально зависят от A.

Так как многозначные зависимости всегда образуют пары, а по определению все остальные атрибуты функционально (не много-значно) зависят от A, это означает, что зависимость атрибутов B от A также функциональная, т.е. отношение находится в 4НФ, если в нем отсутствуют многозначные зависимости.

В нашем примере отношение CDS не находится в 4НФ, так как в нем есть многозначная зависимость, которая не является функциональной.

Примечание: при проектировании с использованием IDEF1x такая ситуация может возникнуть в случае, например, когда определены три сущности — КУРС, СТУДЕНТ, ДЕНЬ ЗАНЯТИЙ и между сущностями КУРС и СТУДЕНТ, КУРС и ДЕНЬ ЗАНЯТИЙ существует связь типа п:п (рис. 6.10).

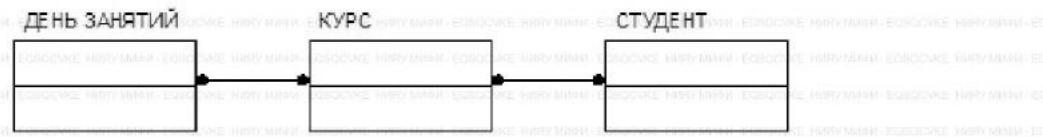


Рис. 6.10. Представление требований предметной области на первом этапе проектирования

Эти связи могут быть разрешены двумя способами.

1. Так как предполагается, что занятия по курсу обязательно посещаются студентами, можно ввести общую дополнительную сущность ЗАНЯТИЕ, для которой не будет выполняться 4НФ (рис. 6.11).

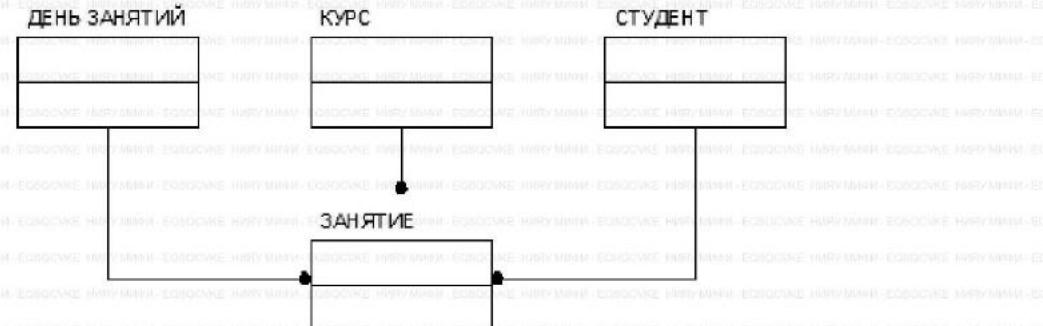


Рис. 6.11. Разрешение неопределенных связей: нарушение 4НФ

2. Если все-таки необходимо, чтобы отношения удовлетворяли 4НФ и допускается наличие данных о проведении занятий по курсу, когда еще не определен состав студентов, изучающих данный курс, тогда каждая неопределенная связь должна разрешаться самостоятельно. В этом случае согласование данных, заносимых в отношения КУРС СТУДЕНТА и ЗАНЯТИЕ ПО КУРСУ, должно осуществляться с помощью триггеров и хранимых процедур, чтобы не получилось так, что студент изучает некоторый курс, для которого не назначены дни занятий (рис. 6.12).



Рис. 6.10. Разрешение неопределенных связей: обеспечение требований 4НФ

6.3.7. Пятая нормальная форма

Все рассматриваемые ранее примеры использовали декомпозицию исходного отношения на две проекции (декомпозиция без потери). Это допущение успешно выполнялось вплоть до 4НФ. Однако существуют отношения, для которых нельзя выполнить декомпозицию без потери на две проекции, но которые можно подвергнуть декомпозиции без потери на три или более проекций.

Надо отметить, что в большинстве практических случаев такие задачи встречаются крайне редко. Поэтому здесь будет только дано определение 5НФ без каких-либо дополнительных пояснений. Подробное описание 5НФ можно найти, например, в [2].

Определение. Отношение $R(A_1A_2...A_n)$, где A_1, A_2, \dots, A_n — произвольные подмножества множества атрибутов R , удовлетворяет зависимости соединения $*(A_1, A_2, \dots, A_n)$ тогда и только тогда, когда оно равносильно соединению своих проекций с подмножествами атрибутов A_1, A_2, \dots, A_n (т.е. восстанавливается без потерь путем соединения своих проекций).

Например, если определено отношение со схемой $R(S, P, J)$, то это отношение удовлетворяет зависимости соединения $*(SP, PJ, SJ)$,

если оно равносильно соединению своих проекций на подмножества
стремится к (S, P) , (P, P) и (S, P) .

если оно равносильно соединению своих проекций на подмножества атрибутов $\{S, P\}$, $\{P, J\}$ и $\{S, J\}$.

Определение. Отношение R находится в 5НФ тогда и только тогда, когда любая зависимость соединения в отношении R следует из существования некоторого потенциального ключа отношения R (или каждая зависимость соединения в отношении R подразумевается потенциальными ключами отношения R).

5НФ называют еще нормальной формой проекции – соединения.

Вопросы

- Сформулируйте основные цели проектирования. Приведите примеры проблем, возникающих, когда пренебрегают основными целями проектирования.
 - Дайте исчерпывающее описание понятия функциональной зависимости. Приведите несколько жизненных примеров.
 - Приведите пример декомпозиции с потерей информации. Докажите, что декомпозиция с потерей информации не сохраняет множество функциональных зависимостей.
 - Приведите примеры декомпозиции без потери информации, при которой сохраняются и не сохраняются функциональные зависимости. Какие проблемы возникают при использовании декомпозиции без потери информации, не сохраняющей функциональные зависимости.
 - Сформулируйте основную цель нормализации отношений. Дайте определения всех известных вам нормальных форм. Приведите примеры.

ГЛАВА 7

ЯЗЫК SQL

Как упоминалось ранее, появление языка SQL (Structured Query Language) связано с разработкой фирмой IBM прототипа реляционной СУБД System R (1976 г.). В дальнейшем этот язык занял главенствующее положение в среде реляционных СУБД.

Работа над стандартом языка SQL была начата в начале 1980-х гг. Американским национальным институтом стандартов (American National Standards Institute – ANSI) и Международной организацией по стандартизации (International Organization for Standardization – ISO). В результате в 1986 г. ANSI представил первый вариант стандарта, известный как SQL86, который в 1987 г. был утвержден ISO (SQL87). Неофициально этот стандарт получил название SQL1. Дальнейшая работа над стандартом привела к расширению возможностей языка, в частности, в области обеспечения целостности данных; в результате в 1989 г. первый стандарт языка был расширен и получил название SQL89. Практически одновременно была начата работа над новой версией стандарта SQL2, которая должна была заменить стандарт SQL89. Новая версия стандарта была принята в 1992 г. и получила название SQL-92. Следующим стандартом стал SQL:1999 (SQL3); в него были добавлены базовые процедурные расширения и некоторые объектно-ориентированные возможности. В настоящее время действует стандарт, принятый в 2003 г. (SQL:2003) с небольшими модификациями, внесенными позже.

Ни одна реальная СУБД не поддерживает стандарт SQL в полном объеме; производители СУБД часто предлагают собственные расширения SQL за счет включения в состав языка дополнительных функций и возможностей, особенно в той его части, которая касается реализации возможностей управления процессом обработки данных — процедурной логики. Эти дополнения принято называть *расширениями* языка SQL. Каждая конкретная реализация языка, включающая те или иные расширения, представляет собой *диалект* языка SQL. Каждая реляционная СУБД использует свой диалект SQL, например PL SQL в СУБД Oracle, Transact SQL в СУБД MS SQL Server и др.

Полное описание возможностей языка SQL-92, включенных в стандарт, даже без рассмотрения различных расширений представляет собой отдельное весьма объемное издание, поэтому в данной книге будут рассмотрены лишь основные аспекты стандарта SQL-92, а также некоторые диалекты языка, используемые для разработки процедурной логики.

В данной главе рассматриваются следующие вопросы: средства языка SQL, используемые для представления структурных компонентов и ограничений целостности (подмножество DDL: типы данных, правила записи констант и выражений; правила создания таблиц); возможности языка SQL, относящиеся к средствам обработки данных (подмножество DML: операции вставки, удаления, обновления данных; формирование различных запросов); возможности диалектов языка SQL для реализации процедурной логики (расширенный SQL: реализация триггеров и хранимых процедур), а также некоторые вопросы, относящиеся к оптимизации (создание индексов, построение и оценка плана выполнения запроса).

Все примеры, иллюстрирующие соответствующие конструкции языка SQL, соответствуют диалекту языка, определяемого стандартом SQL-92. Особенности диалектов SQL для СУБД MS SQL Server и Oracle указываются в примечаниях по ходу описания соответствующих конструкций. При изучении языка полезно выполнять соответствующие примеры в конкретной СУБД; в этом случае тексты примеров должны быть скорректированы с учетом диалекта SQL для конкретной СУБД.

В описании использованы следующие обозначения.

- В описании использованы следующие обозначения:

 1. Прописными буквами полужирным шрифтом записываются те фрагменты, которые при кодировании предложения SQL должны копироваться именно так, как указано в описании, например **CREATE TABLE**.
 2. Строчными буквами и курсивом выделены те понятия, которые при кодировании предложения SQL должны заменяться собственными значениями; например, запись *имя_таблицы* означает, что вместо нее должно быть указано конкретное имя таблицы.
 3. Специальные символы, встречающиеся в описании предложения и выделенные полужирным шрифтом, должны кодироваться так, как они записаны.
 4. Если в записи какого-то фрагмента допускается несколько разных значений и необходимо выбрать какое-то одно, допустимые значения перечисляются через символ |, например: **INT | FLOAT | REAL**. Если для данного фрагмента допускается значение по умолчанию, оно должно указываться первым в списке.
 5. Необязательная часть предложения заключается в квадратные скобки; при кодировании предложения она может быть опущена.
 6. Обязательная часть предложения заключается в фигурные скобки.
 7. Многоточие означает, что предшествующая ему часть предложения может быть повторена произвольное число раз.

Ниже приведены примеры использования комментариев в языке SQL:

```
-- Это комментарий
-- это тоже
/* это тоже
   комментарий */
```

В записи исходных текстов часто возникает необходимость добавлять комментарии, поясняющие особенности использования соответствующих конструкций языка. Комментарии в языке SQL начинаются двумя знаками «минус» и заканчиваются концом строки, например:

-- это комментарий

Допускается использовать многострочный комментарий в виде

```
/ * это тоже
   комментарий */
```

В записи текста комментария могут быть использованы любые символы.

7.1. ОБЩАЯ ХАРАКТЕРИСТИКА ЯЗЫКА SQL

Язык SQL включает в себя два подмножества:

- подмножество языка для описания данных (SQL–DDL);
- подмножество языка для манипулирования данными (SQL–DML).

Подмножество языка описания данных предоставляет средства для создания объектов базы данных, к которым прежде всего относится таблица. При определении таблицы необходимо указать используемые для колонок таблицы типы данных, а также ограничения, накладываемые на данные. Эти ограничения проверяются тогда, когда выполняются соответствующие операции с данными, определяемые подмножеством языка манипулирования данными. В связи с этим упомянутые подмножества языка SQL–DDL и DML очень тесно связаны между собой и используют некоторые общие возможности языка SQL, с представления которых и начинается данное описание.

Помимо указанных выше, в современных версиях SQL выделяют еще одно подмножество языка — для управления доступом к данным (SQL–DCL — Data Control Language).

Возможности языка SQL будут проиллюстрированы на некотором конкретном примере.

7.2. ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ «ПРОКАТ АВТОМОБИЛЕЙ»

Прежде всего следует отметить, что рассматриваемая здесь база данных проката автотранспортных средств является исключительно учебной базой данных. Ее основное назначение заключается в том, чтобы предоставить некоторый набор данных, на основе которых можно изучить синтаксис и семантику основных предложений SQL,

а также получить представление о принципах создания запросов к базе данных. База данных содержит информацию о несуществующей на самом деле организации, которая предоставляет услуги в области проката транспортных средств.

Проектирование любой базы данных начинается с анализа бизнес-правил и принципов функционирования соответствующей организации. В соответствии с этим предположим, что наша организация имеет большой парк автотранспортных средств и представительства (офисы) в различных городах и странах, в которых оформляются договоры с клиентами на прокат (аренду) автотранспортных средств. Клиентами организации могут быть частные лица, имеющие водительское удостоверение или другой документ, удостоверяющий личность. Если клиент не имеет водительского удостоверения, он может арендовать транспортное средство только с водителем (дополнительное условие аренды).

В договоре указываются все условия аренды (начало аренды, срок, контактное лицо организации для связи при возникновении каких-либо нештатных ситуаций и др.), условия получения и возврата автотранспортного средства (забирать автомобиль можно разными вариантами: самостоятельно на специализированных стоянках компании либо доставка за дополнительную плату по конкретному адресу, и так же можно вернуть), могут быть указаны различные дополнительные условия (например, страховка, франшиза — часть, которую оплачивает клиент в случае повреждений, количество водителей, количество бесплатных километров, входящих в стоимость, и т.п.). Договор считается вступившим в силу только после его оплаты.

Договор может быть аннулирован до наступления срока аренды; в этом случае, если договор уже был оплачен, клиенту возвращается стоимость договора. Все факты перечисления денег должны фиксироваться.

Оплата осуществляется либо банковской картой, если автомобиль резервируется через Интернет заранее, либо наличными или картой в случае без предварительного бронирования. Также при возврате автомобиля возможны дополнительные оплаты, связанные со штрафами при использовании автомобиля. Оплата договора может быть выполнена в любой валюте с учетом текущего курса на день оплаты.

Клиент, получивший автотранспортное средство, фиксирует его возврат в любом офисе организации. При возврате транспортного средства возможен перерасчет услуг (если, например, нарушены сроки договора).

Полноатрибутная схема спроектированной базы данных в соответствии с методологией IDEF1x приведена на рис. 7.1.

Ниже приведена схема базы данных, описанная в Приложении 1. Схема показывает структуру и взаимосвязи между различными объектами. Ключевые элементы схемы включают:

- Страна**: Имеет атрибут **Код страны** и **Название**.
- Клиент**: Имеет атрибуты **Фамилия**, **Имя**, **Отчество**, **Город**, **Адрес** и **Код клиента**.
- Платежный документ**: Имеет атрибуты **Номер договора**, **Тип документа**, **Дата оплаты**, **Сумма** и **Код записи (PK)**.
- Договор аренды**: Имеет атрибуты **Номер договора**, **Код клиента** (FK), **Код офиса** (FK), **Дата оформления**, **Дата оплаты**, **Срок аренды**, **Дата начала аренды**, **Фактическая дата окончания аренды**, **Менеджер**, **Телефон** и **Договор выкуплен**.
- Уточнения договора**: Имеет атрибуты **Номер договора (PK)** и **Номер записи (PK)**.
- Дополнительные условия аренды**: Имеет атрибуты **Номер записи**, **Условие**, **Код записи (PK)** и **Стойкость**.
- Офис проекта**: Имеет атрибуты **Код офиса**, **Город** и **Адрес**.
- Категория**: Имеет атрибуты **Код категории** и **Название**.
- Транспортное средство**: Имеет атрибуты **Код ТС**, **Название**, **Стандарт проекта**, **Марка**, **Модель**, **Цвет**, **Глонасс (AK1.1)** и **Код записи (PK)**.
- Документ единицы**: Имеет атрибуты **Код записи** и **Название (AK1.1)**.
- Курс денежной единицы**: Имеет атрибуты **Номер записи (PK)** для **Код записи (PK)** и **Дата**.
- Значение курса**: Имеет атрибут **Значение курса**.

В Приложении 1 приведено описание схемы базы данных в соответствии с реляционной моделью данных.

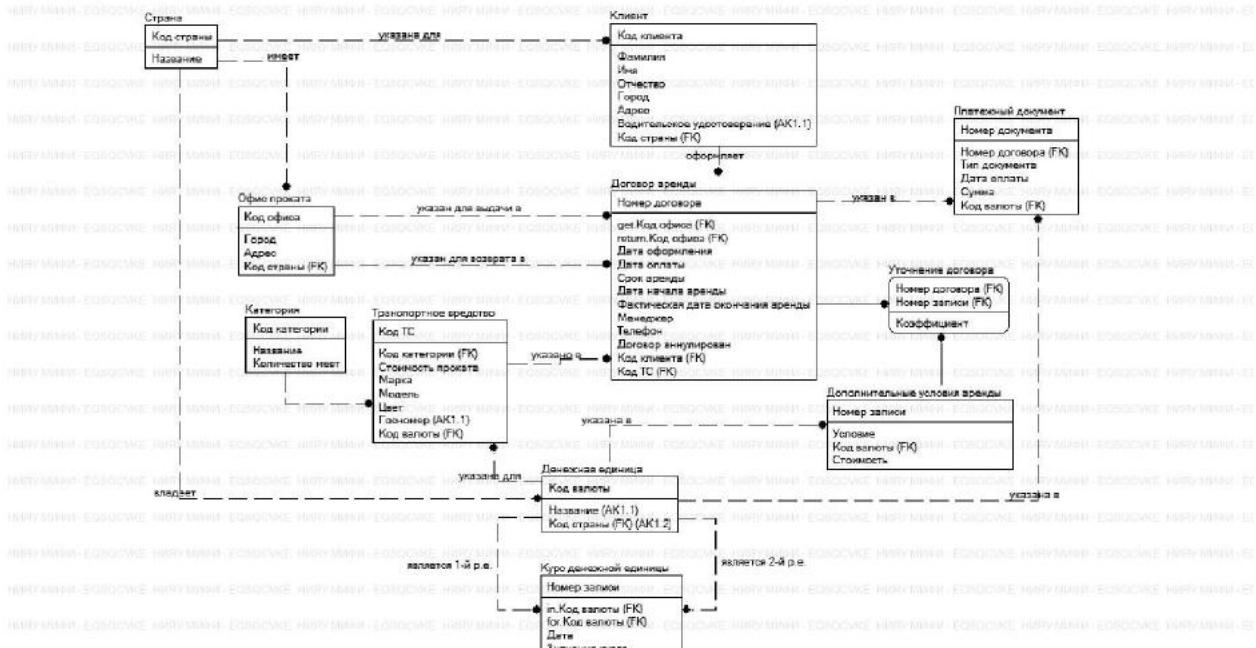


Рис. 7.1. Полноатрибутивная схема базы данных проката автотранспортных средств

7.3. ПРЕДСТАВЛЕНИЕ ДАННЫХ

7.3.1. Способы именования объектов

В общем случае полное имя любого объекта базы данных состоит из нескольких частей, разделяемых символом «точка». Конкретное представление имени объекта определяется конкретной СУБД. Мы ограничимся следующим представлением имени объекта, которое используется в именовании объектов в любой СУБД:

имя_схемы . собственное_имя_объекта.

Имя_схемы в именовании объекта является необязательным; если оно не указывается, в качестве его значения подставляется значение по умолчанию, которое определяется конкретной СУБД.

В дальнейшем мы будем использовать в имени объекта только **собственные имена**.

Для именования объектов базы данных используются *идентификаторы*. Идентификатор объекта создается, когда создается сам объект. В дальнейшем он используется для ссылок на соответствующий объект.

Идентификаторы языка делятся на обычные идентификаторы и идентификаторы с ограничителями. *Обычный идентификатор* — это последовательность букв, цифр и символов подчеркивания (_), начинающаяся с буквы. Обычный идентификатор не должен совпадать с ключевыми словами SQL. Обычные идентификаторы никак не выделяются в предложениях SQL.

Примеры:

A12 Tab Name DeptID

Язык SQL считается языком, не чувствительным к регистру, т.е. в записи обычных идентификаторов и ключевых слов SQL можно использовать и прописные, и строчные буквы английского алфавита, например идентификаторы `DeptID` и `DEPTID` определяют одно и то же имя; слова `SELECT` и `select` определяют одно и то же ключевое слово. Однако принято ключевые слова SQL, определяющие название предложений, записывать прописными буквами: `CREATE TABLE`, а не, например, `Create Table`. В дальнейшем в примерах будут использованы следующие соглашения: ключевые слова SQL записываются прописными буквами; собственные идентификаторы, используемые для именования объектов базы данных, записываются строчными буквами.

Идентификаторы с ограничителями — любой текст, заключенный в парные двойные кавычки (стандарт ISO) или в квадратные скобки (MS SQL Server), например:

«wkly-sal» [region 5].

Правила использования идентификаторов с ограничителями зависят от конкретной СУБД. В предложениях SQL такие идентификаторы также должны быть записаны с ограничителями.

Идентификаторы с ограничителями целесообразно использовать, в частности, в тех случаях, когда по каким-либо причинам необходимо использовать имя, совпадающее с зарезервированным словом, например [**UNION**] или «**UNION**», или многословное имя, отражающее семантику объекта, например «**department name**» или [**department name**]. В последнем случае можно обойтись и без идентификатора с ограничителем, если не разделять слова пробелами, но каждое слово в имени начинать с прописной буквы, например **departmentName**.

Максимальная длина идентификатора определяется конкретной СУБД. Так, в MS SQL Server длина идентификатора не может превышать 128 символов.

7.3.2. Основные типы данных

Современные СУБД поддерживают определенный набор встроенных типов данных, используемых, в частности, для определения колонок таблицы. Для всех типов данных допускается особое NULL-значение, которое по своей сути означает отсутствие значения. Для каждого типа данных допускается свое NULL-значение, которое отличается от любого не NULL-значения этого же типа. В языке нет возможности представить NULL-значение, но в некоторых случаях для его обозначения используется ключевое слово NULL.

В общем случае базовые встроенные типы данных можно разделить на три группы: *числовые типы данных*, *строковые типы данных* и *типы данных, используемые для представления даты и времени*.

Числовые типы данных

Числовые типы данных используются для представления точных и приближенных чисел. При задании числовых типов данных рассматриваются такие их характеристики, как **точность** и **масштаб** представления. Точность представления данных числовых типов определяется как общее количество цифр в представлении числа (двоичных или десятичных, в зависимости от представления). Масштаб определяет количество десятичных цифр в дробной части представления числа. Для точных чисел масштаб = 0.

Конкретное представление чисел в памяти машины определяется реализацией.

Для представления точных чисел используются следующие типы данных:

SMALLINT — короткое целое со знаком; точность представления — обычно 15 двоичных цифр, масштаб — 0. Это означает, что область хранения данных такого типа — 2 байта, диапазон представления — от -32 768 до 32 767 (от -2^{15} до $2^{15} - 1$).

INT (или **INTEGER**) — целое со знаком; точность представления — обычно 31 двоичная цифра, масштаб — 0. Это означает, что область хранения данных такого типа — 4 байта, диапазон представления — от $-2\ 147\ 483\ 648$ до $2\ 147\ 483\ 647$ (от -2^{31} до $2^{31} - 1$).

DECIMAL[(p[,s])] (или **DEC[(p[,s])]**) – десятичное число; представляется в виде десятичного числа с неявно заданным положением десятичной запятой, определяемым значениями точности p и масштаба s числа. Точность p задает общее количество цифр в записи числа, $p \leq p_{max}$. Значение p_{max} зависит от реализации (так, для MS SQL Server $p_{max} = 38$). Масштаб s задает количество десятичных цифр в дробной части числа и не может быть отрицательным или превышать точность числа: $0 \leq s \leq p$. Если значение s опущено, оно принимается равным 0. Если опущено значение p, оно принимается равным некоторому значению по умолчанию, которое зависит от

реализации. Так, для MS SQL Server это значение по умолчанию — 18. Таким образом, для MS SQL Server указание **DECIMAL** эквивалентно заданию **DECIMAL (18, 0)**.

реализации. Так, для MS SQL Server это значение по умолчанию — 18. Таким образом, для MS SQL Server указание **DECIMAL** эквивалентно заданию **DECIMAL (18,0)**.

Область хранения данных числовых типов также зависит от реализации. Диапазон представления чисел зависит от заданных значений точности и масштаба.

NUMERIC[(p[,s])] (или **NUM[(p[,s])]**) — полностью эквивалентно **DECIMAL**.

Примечание. В СУБД MS SQL Server для представления точных чисел используются еще и следующие типы данных:

ВТ — целый тип данных; допустимые значения 0 и 1. Область хранения — байт.

нения — целое число байтов (например, если в таблице определены 8 колонок с типом данных BIT, SQL Server может хранить значения всех этих колонок в одном байте).

TINYINT — целое без знака; область хранения — 1 байт; диапазон представления — от 0 до 255.

BIGINT — длинное целое со знаком; область хранения — 8 байтов; диапазон представления — от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 (от -263 до 263 - 1).

SMALLMONEY — целое число со знаком для представления денежных величин; в записи числа младшие 4 цифры определяют дробную часть денежной единицы (центы, копейки, ... — в зависимости от используемой денежной единицы); область хранения — 4 байта; диапазон представления — от -214 748.3648 до 214 748.3647.

MONEY — целое число со знаком для представления денежных величин, в записи которого также младшие 4 цифры определяют дробную часть денежной единицы; область хранения — 8 байтов; диапазон представления — от -922 337 203 685 477.5808 до 922 337 203 685 477.5807.

В СУБД Oracle для представления числовых типов используется еще один мини-символический тип — **NUMBER.**

Так, числовым типам данных стандарта SQL-92 **SMALLINT**, **INTEGER**, **INT** соответствует тип данных **NUMBER**.

Типам данных **NUMERIC[(p,s)]** и **DECIMAL[(p,s)]** соответствует **NUMBER[(p,s)]**.

Тип данных **MONEY** в СУБД Oracle отсутствует, однако его можно представить, например, как **NUMBER(19, 4)**, соответственно **SMALLMONEY** — как **NUMBER(10, 4)**.

Для представления приближенных чисел используются следующие типы данных

FLOAT(n) — вещественное число обычной или двойной точности в зависимости от того, как задано значение n; это значение определяет количество двоичных цифр в записи мантиссы числа. В общем случае $1 \leq n \leq 53$. Значения $1 \leq n \leq 24$ соответствуют вещественному

числу обычной точности (независимо от значения n для представления числа используется 31 бит, или 4 байта), $25 \leq n \leq 53$ — двойной точности (независимо от значения n для представления числа используется 63 бита, или 8 байтов). Если значение n не указано, по умолчанию принимается значение 53 (вещественное число двойной точности). Диапазон представления числа зависит от значения n . Для $n = 53$ абсолютная величина числа может быть представлена как 0 или в диапазоне от $2,23 \cdot 10^{-308}$ до $1,79 \cdot 10^{+308}$.

REAL — вещественные числа обычной точности; эквивалентно заданию FLOAT(24). Абсолютная величина числа может быть представлена как 0 или в диапазоне от $1,18 \cdot 10^{-38}$ до $3,40 \cdot 10^{+38}$.

DOUBLE PRECISION — вещественные числа двойной точности; эквивалентно заданию FLOAT(53).

Примечание. В СУБД Oracle для представления вещественных чисел используются следующие типы данных:

FLOAT(n) — точность представлена в диапазоне $1 \leq n \leq 126$ двоичных цифр;

BINARY_FLOAT — 32-битный тип одинарной точности;

BINARY_DOUBLE — 64-битный тип двойной точности.

Числовые константы

Числовые константы, в свою очередь, делятся на целые, вещественные и десятичные. Все константы обладают свойством NOT NULL. Значение числовой константы (-0) эквивалентно значению 0.

Целая константа определяет целое значение со знаком или без знака. Целая константа не содержит десятичной точки. Тип целой константы определяется ее значением. Так, целая константа имеет тип INTEGER, если ее значение лежит в диапазоне 4-байтного значения, или тип decimal, если ее значение выходит за диапазон значений целочисленного типа.

Примеры:

64 -15 +100 32767 720176 12345678901

Десятичная константа представляется числом со знаком или без знака, которое либо содержит десятичную точку, либо значение константы выходит за диапазон представления целых чисел. Точность константы определяется общим количеством цифр в ее записи, включая лидирующие и завершающие нули. Масштаб числа определяется количеством цифр, записанных справа от десятичной точки, включая завершающие нули.

Примеры:

25.5 (точность константы 3, масштаб 1) +37589.333333333
1000. -15.

Если в записи двух последних констант точка не будет указана, тогда эти константы будут иметь тип INT.

Вещественная константа (или *константа с плавающей запятой*) определяет вещественное число, в записи которого используются два числа, разделенных буквой Е. В записи первого числа могут быть указаны знак и десятичная точка. Второе число представляется в виде целого числа, которому может предшествовать знак. Тип данных вещественной константы — вещественное число двойной точности. Значением константы является произведение первого числа на 10 в степени, равной второму числу в записи константы.

Примеры:

15E1 (значение константы $15 \cdot 10^1 = 150$) 2.E5 2.2E-1 +5E+2

Строковые типы данных

Строковые типы данных используются для представления символьных или битовых строк.

Символьная строка — это последовательность символов, представляющих текстовые (символьные) данные. Длина строки представляет собой количество символов в последовательности. Если длина строки равна нулю, значение такой строки определяется как *пустая строка*. Пустая строка не эквивалентна строке, имеющей NULL-значение.

В зависимости от способа кодирования символов рассматривают обычные символы и Unicode-символы (или символы национальных алфавитов).

Для представления обычных символьных строк, в которых каждый символ строки кодируется одним байтом, используются следующие типы данных.

CHAR[(len)] — символьные строки фиксированной длины, заданной параметром len — количеством символов в строке. Значение len лежит в диапазоне $1 \leq \text{len} \leq \text{maxLength}$. Значение maxLength определяется конкретной СУБД. Так, для MS SQL Server maxLength = 8000. Для представления строки выделяется в точности len байтов. Если значение len не указано, оно принимается равным 1. Все значения данного типа имеют одинаковую длину len. Если в записи символьной строки данного типа использовано меньшее, чем len, количество символов, строка дополняется пробелами справа до len символов. Если в записи строки использовано большее, чем len, количество символов, строка усекается справа до len символов, что в определенной ситуации может вызвать сообщение об ошибке (например, при вставке значения в таблицу).

VARCHAR[(len)] — символьные строки переменной длины. Значение len задает максимально возможную длину строки (количество символов в строке) и лежит в диапазоне $1 \leq \text{len} \leq \text{maxLen}$. Если в за-

**Писи строки использовано количество символов $rlen \leq len$,
длина такой строки будет равна $rlen$. Строки переменной длины хра-
нятся в памяти вместе с информацией о реальной длине соответству-
ющей строки.**

писи строки использовано количество символов $rlen \leq len$, длина такой строки будет равна $rlen$. Строки переменной длины хранятся в памяти вместе с информацией о реальной длине соответствующей строки.

Примечание. В СУБД MS SQL Server для хранения строки выделяется $rlen + 2$ байта (дополнительные 2 байта используются для хранения действительной длины строки). Если в записи строки использовано большее, чем len , количество символов, строка усекается справа до len символов, и длина такой строки будет равна len . Опять же в некоторых ситуациях это может привести к ошибке. Если значение len не указано, оно принимается равным 1.

В СУБД **Oracle** существует расширенная версия типа данных **VARCHAR**, которая называется **VARCHAR2[(len)]**. У типа данных **VARCHAR2** значение **len** имеет максимально возможную длину строки в диапазоне $1 \leq \text{len} \leq 4000$ байт.

Для представления символьных Unicode-строк, в которых каждый символ кодируется двумя байтами, используются типы данных **NCHAR**[(len)] и **NVARCHAR**[(len)]. Они эквивалентны типам **CHAR** и **VARCHAR**, соответственно.

Битовая строка представляет собой последовательность байтов, содержащих двоичный код. Для представления битовых строк используются следующие типы данных.

BIT[(len)] — битовые строки фиксированной длины, заданной параметром len. Длина битовой строки определяется количеством битов в записи строки. Значение len лежит в диапазоне $1 \leq \text{len} \leq \text{maxLength}$. Значение maxLength зависит от реализации. Если значение len не указано, оно принимается равным 1.

BIT VARYING[(len)] — битовые строки переменной длины. Значение len задает максимально возможную длину строки и лежит в диапазоне $1 \leq \text{len} \leq \text{maxLength}$. Если значение len не указано, оно принимается равным 1.

Примечание. В СУБД MS SQL Server для представления битовых строк используются следующие типы:

BINARY[(len)] — битовые строки фиксированной длины, заданной параметром len. Значение len лежит в диапазоне $1 \leq \text{len} \leq 8000$. Для представления строки выделяется len байтов. Если значение len не указано, оно принимается равным 1.

VARBINARY[(len)] — битовые строки переменной длины. Значение len задает максимально возможную длину строки и лежит в диапазоне $1 \leq \text{len} \leq 8000$. Если в записи строки использовано количество байтов rlen $\leq \text{len}$, длина такой строки будет равна rlen. Для представления строки выделяется rlen + 2 байта. Если значение len не указано, оно принимается равным 1.

В СУБД **Oracle** для представления битовых строк используется тип данных **ROW[(len)]**; значение len определяет длину строки в байтах и лежит в диапазоне $1 \leq \text{len} \leq 2000$.

Строковые константы

Строковые константы, как и строковые типы данных, делятся на две группы — символьные и двоичные.

Символьная строковая константа определяет строку символов переменной длины и представляется последовательностью алфавитно-цифровых символов, заключенную в парные одиночные кавычки (апострофы), например 'string for example'. Значением строковой константы является сама строка, расположенная между апострофами.

Длина символьной строковой константы определяется количеством символов в записи константы. Так, в приведенном примере длина константы равна 18. Длина символьной строковой константы не должна превышать значения `maxLen`, зависящего от реализации.

Если в записи константы необходимо использовать сам символ апострофа, он должен быть удвоен. Так, запись 'DON''T CHANGE' определяет строку длиной 12 символов, значением которой является строка DON'T CHANGE.

Строковые символьные константы чувствительны к регистру, используемому при записи букв. Так, константы 'string for example' и 'String for Example' имеют разное значение.

Если нужно представить Unicode-символьную строковую константу, в которой каждый символ отображается в двух байтах, записи константы должен предшествовать символ N (в верхнем регистре). Так, константа 'string' представляет собой обычную константу длиной 6 символов и отображается в области памяти размером 6 байтов, тогда как константа N 'string' — это Unicode-константа длиной также 6 символов, но отображается она в области памяти длиной 12 байтов.

Представление двоичной строковой константы зависит от реализации. В соответствии со стандартом SQL-92 двоичные строковые константы могут быть заданы в виде строк, состоящих из двоичных или из шестнадцатеричных цифр, которым предшествует символ B или X соответственно, например: B'101100' X'12af0'. В MS SQL Server двоичная строковая константа представляет собой строку шестнадцатеричных цифр, начинающуюся префиксом 0x, например 0xAE, 0x12Ef, 0x69048AEFDD010E. Запись 0x представляет пустую строку.

Типы данных дата-время

Типы данных дата–время используются для представления даты и времени. Хотя значения даты и времени можно использовать с

некоторыми арифметическими и строковыми операциями, они не являются ни числами, ни строками.

Стандарт SQL-92 определяет следующие типы данных для представления даты — времени: **DATE**, **TIME**, **TIMESTAMP**, **INTERVAL**. Однако в текущих реализациях синтаксис и семантика этих типов могут различаться, и каждая СУБД предоставляет свои средства для представления данных типа даты—время и обработки этих данных.

Представление даты и времени:

DATE — представление даты; значение, состоящие из трех частей: год (YEAR), месяц (MONTH) и день (DAY);

TIME [(n)] — представление времени; значение, представленное также в виде трех частей: часы (HOUR), минуты (MINUTE) и секунды (SECOND). При задании секунд может быть указана дробная часть; n определяет количество цифр в дробной части; если не указано — предполагается значение 0;

TIMESTAMP [(n)] — представление даты и времени; содержит поля год (YEAR), месяц (MONTH), день (DAY), часы (HOUR), минуты (MINUTE) и секунды (SECOND). При задании секунд также может быть указана дробная часть: n определяет количество цифр в дробной части. Если n не указано, по умолчанию предполагается значение 6.

Расположение соответствующих частей в представлении даты и времени считается упорядоченным в соответствии со следующим перечислением: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND.

Интервалы. Интервальный тип задается следующим образом:

INTERVAL квалификатор интервала.

Квалификатор_интервала представляется следующим образом:

название_поля | начало_интервала TO конец_интервала;

название_поля, начало_интервала, конец_интервала — это одно из слов из упорядоченной последовательности YEAR, MONTH, DAY, HOUR, MINUTE, SECOND; при этом значение **начало_интервала** должно обязательно предшествовать значению **конец_интервала**.

Стандарт SQL поддерживает два типа интервалов: интервалы *год–месяц* (*year-month*) и интервалы *день–время* (*day-time*). Интервал год–месяц содержит только указание года (YEAR) и месяца внутри года (MONTH; допустимые значения 0–11). Интервал день–время содержит указание дня (DAY), часа внутри дня (HOUR; допустимые значения 0–23), минуты внутри часа (MINUTE; допустимые значения 0–59) и секунды внутри минуты (SECOND); допускается задание дробной части секунд.

Примеры:

INTERVAL YEAR TO MONTH

INTERVAL TIME

INTERVAL DAY TO MINUTE

Ниже приведены примеры использования функций для работы с датами и временем в языке SQL.

Примечание. В СУБД MS SQL Server интервальный тип не поддерживается.

Для представления даты и времени используются типы данных **DATE**, **TIME** и следующие дополнительные типы данных.

DATETIME2 [(n)] — соответствует типу данных **TIMESTAMP** стандарта SQL-92.

DATETIME — совместное представление даты и времени, время представляется в 24-часовом формате; область хранения — 8 байтов. Диапазон представления даты — от 1 января 1753 года до 31 декабря 9999 года; диапазон представления времени — от 00:00:00 до 23:59:59.997; точность представления времени — дробная часть секунды представляется в виде 0.000, 0.003 или 0.007 сек.

SMALLDATETIME — совместное представление даты и времени, время представляется в 24-часовом формате; область хранения — 4 байта.

Диапазон представления даты — от 1 января 1900 года до 6 июня 2079 года; диапазон представления времени — от 00:00:00 до 23:59:59; точность представления времени — 1 мин. Данный тип несовместим со стандартом ANSI или ISO 8601.

В СУБД **Oracle** поддерживаются интервальные типы данных и не поддерживается тип данных **TIME**. Представление даты и времени задается одним типом данных **DATE**, диапазон представления которого находится в пределах от 1 января 4712 года до н.э. до 31 декабря 4712 года нашей эры. Точность представления времени — 1 секунда.

Константы для представления даты и времени

Стандарт SQL определяет следующие правила задания констант типа даты–времени:

DATE 'год-месяц-день';

TIME 'часы : минуты : секунды .дробная_часть';

TIMESTAMP 'год-месяц-день пробел часы : минуты : секунды .дробная_часть'.

Дробная часть секунд при задании времени не обязательна и может отсутствовать.

Примеры:

DATE '2014-6-20' TIME '12:30:45' TIMESTAMP '2014-6-20

12:30:45'

Для задания текущих значений даты и времени используются встроенные функции:

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP

Константы для задания интервала имеют следующий вид:

a) задание константы для интервала типа год–месяц;

INTERVAL [знак] 'год-месяц' квалификатор интервала год_месяц

где год–месяц задается в виде: *год* [*год*–*месяц*];

где год–месяц задается в виде: *год | [год-]месяц;*

б) задание константы для интервала типа день–время:

INTERVAL [знак] 'день–время' квалификатор интервала **день_месяц**
где **день–время** задается следующим образом:

Примеры:

INTERVAL '2014-6' YEAR TO MONTH

INTERVAL '25' DAY

INTERVAL - 130:38 ! HOUR TO MINUTE

Примечание. В СУБД MS SQL Server нет специальных констант, представляющих дату и/или время; для этих целей используются обычные символьные строки, содержащие значение даты и/или времени в соответствии с одним из указанных выше форматов.

Примеры задания даты и/или времени:

'December 5, 2013', '5 December, 2013', '20131205', '12/5/13'

12/5/13 14:30:34 14:30:35 50

'December 5, 2013 14:30:25.50'
Для задания текущих значений даты–времени используются специальные встроенные функции языка Transact-SQL:

SYSDATETIME() — возвращает значение типа DATETIME2(7), со-

GETDATE() — возвращает значение типа **DATETIME** (7), содержащее текущие дату и время компьютера, на котором установлен SQL Server.

CURRENT_TIMESTAMP — возвращает значение типа DATETIME, содержащее текущие дату и время компьютера, на котором установлен SQL Server.

GETDATE() — возвращает значение типа DATETIME, содержащее текущие дату и время компьютера, на котором установлен SQL Server.

В СУБД **Oracle** для задания текущих значений даты-времени используются специальная встроенная функция **SYSDATE**. Она возвращает

зуется специальная встроенной функция `SYSDATE`. Она возвращает значение типа `DATE`, содержащее текущие дату и время операционной системы, на которой развернута СУБД.

7.3.3. Выражения

В общем случае выражение представляет собой совокупность **операндов**, разделенных знаками **операций**.

При вычислении выражения на основании значений operandов определяется его результат.

В зависимости от типа результата выражения можно разделить на

следующие категории:

- *скалярное выражение*; в результате вычисления скалярного выражения получается одно-единственное значение любого типа или NULL;
 - *предикат*; некоторая разновидность скалярного выражения; в результате вычисления предиката получается одно из трех значений — истина, ложь или не определено;
 - *таблично-строковое выражение*; в результате вычисления выражения данного типа получается одна строка таблицы, содержащая одно или более значений, или NULL;
 - *табличное выражение*; в результате вычисления табличного выражения получается таблица, состоящая из одной или более

Для выражения каждой категории определяются соответствующие типы operandов и операции, правила их записи, вычисления и использования в конструкциях языка **SOL**.

Скалярные выражения

Операнды, используемые в записи одного скалярного выражения, должны иметь сопоставимые типы. Поэтому в общем случае скалярные выражения делятся на три типа — в соответствии с представлением встроенных типов данных: *числовые выражения*, *строковые выражения* и *выражения с данными типа дата — время*.

В записи скалярных выражений соответствующего типа, наряду со знаками операций, определяемыми стандартом SQL, в конкретном расширении SQL могут быть использованы и свои собственные знаки операций.

В качестве операндов скалярного выражения могут быть использованы следующие конструкции:

- константа;
 - имя колонки;
 - вызов функции;
 - скалярный подзапрос;
 - case-выражение;
 - cast-спецификация;
 - скалярное выражение, заключенное в круглые скобки.

Операнду может предшествовать знак плюс (+) или минус (-). Скалярное выражение определяет одно-единственное значение. Если хотя бы один из операндов в выражении имеет значение NULL, результатом вычисления выражения является NULL; в противном случае результат вычисления выражения определяется типом и значениями операндов.

Рассмотрим правила записи скалярных выражений разных типов

Таким образом, мы можем записать скалярные выражения разных типов.

Числовые выражения

В записи числовых выражений используются операнды, относящиеся к числовым типам, и традиционный набор арифметических операций:

- префиксная операция **+** (унарный плюс) не изменяет значение своего операнда;
 - префиксная операция **-** (унарный минус) изменяет знак ненулевого операнда;
 - бинарные (инфиксные) операции **(+, -, *, /)** определяют сложение, вычитание, умножение и деление соответственно. Значение второго операнда в операции деления не должно быть равно нулю.

В общем случае все числовые типы считаются сопоставимыми, поэтому разрешается использование в одном выражении операндов, имеющих разные числовые типы. При этом в каждой реализации определяются правила преобразования типов.

Если оба операнда, участвующие в бинарной операции, относятся к целому типу, тип операнда определяется как `INT`. Операция деления выполняется как целочисленная операция (дробная часть результата отбрасывается).

Если хотя бы один из операндов относится к вещественному типу, оба операнда преобразуются к типу двойной точности и результат имеет тип вещественного числа двойной точности.

Если один из операндов относится к десятичному типу, а другой — к целому, целочисленный operand преобразуется к десятичному типу в соответствии с правилами, принятыми в конкретной СУБД.

Строковые выражения

В записи строковых выражений используются операнды строковых типов и определена только одна операция конкатенации, которая может иметь разные обозначения. Так, стандарт SQL определяет для задания операции конкатенации символы `||`, а в языке Transact SQL MS SQL Server операция конкатенации обозначается символом `+`. Операция выполняет сцепление двух строковых operandов.

Операндами операции конкатенации должны быть совместимые строки. Не допускается использование в одном выражении символьных строк и битовых строк; символьных строк обычных и Unicode-символьных строк.

Результат определяется как строка, состоящая из значения первого операнда, за которым непосредственно следует значение второго операнда.

Выражения с данными типа дата–время

Стандарт SQL-92 допускает следующие виды выражений, использующих данные типа дата–время и интервальные. Обозначим через dt данные типа дата — время, it — данные интервальных типов и n — данные целочисленного типа. Тогда определены следующие виды выражений:

- $dt - dt \rightarrow$ дает в результате значение интервального типа;
 - $dt + it, dt - it, it + dt \rightarrow$ дают в результате значения типа дата — время;
 - $it + it, it - it, it \cdot n, it / n, n \cdot it \rightarrow$ дают в результате значение интервального типа.

Следует учесть, что данные типа `дата`–`время` имеют специфическое представление в различных СУБД, поэтому и запись выражений такого типа зависит от используемой СУБД.

Примечание. В СУБД Oracle и MS SQL Server все операции, использующие данные типа дата–время, реализованы с помощью специальных встроенных функций.

Функции

Функции

Функция представляет собой некоторую операцию, на которую можно ссылаться по имени. За именем функции должны следовать парные круглые скобки, содержащие список аргументов (список аргументов может отсутствовать).

Поддерживаются два типа функций:

- *встроенные функции*, включенные в состав конкретной реализации языка и поддерживаемые средствами самой СУБД;
 - *функции, определенные пользователем*; они создаются с помощью предложения CREATE FUNCTION.

В данном разделе будут рассмотрены только встроенные функции; функции, определенные пользователем, будут рассмотрены позже (см. п. 7.7.5).

Встроенные функции можно разделить на два типа: агрегатные функции и скалярные.

Аргументом *агрегатной функции* является совокупность значений одного типа (обычно множество значений некоторой колонки таблицы). Агрегатная функция возвращает в качестве результата скалярное значение (возможно, NULL) и может быть указана в предложениях SQL **везде**, где может быть использовано скалярное выражение.

Аргументами скалярной функции являются отдельные скалярные значения, которые могут относиться к разным типам и иметь разные значения; например, здесь могут быть использованы отдельные значения какой-либо колонки таблицы, константы. Скалярная функция возвращает в качестве результата скалярное значение (возможно,

NULL) и может быть указана в предложениях SQL везде, где может быть использовано выражение.

Надо отметить, что стандарт SQL-92 определяет относительно небольшое количество встроенных функций, к которым могут быть

небольшое количество встроенных функций, к которым могут быть отнесены агрегатные функции и некоторые функции обработки строковых данных. В реальных СУБД состав встроенных функций существенно расширен. Прежде всего это относится к функциям обработки данных типа дата–время, поскольку представление таких данных зависит от реализации. В связи с этим далее будут рассмотрены агрегатные функции. Стандартные функции обработки строк и некоторые их модификации, принятые в конкретных СУБД, а также конкретные функции обработки данных типа дата–время для разных СУБД приведены в Приложении 3. При необходимости, следует обращаться к технической документации по каждой СУБД.

Агрегатные функции

Агрегатные функции, включенные в стандарт SQL-92, приведены в табл. 7.1.

Таблица 7.1

Агрегатные функции

Функция	Описание
AVG	Возвращает среднее арифметическое для совокупности чисел
COUNT	Возвращает количество строк или значений в совокупности строк или значений
MAX	Возвращает максимальное значение из множества значений
MIN	Возвращает минимальное значение из множества значений
SUM	Возвращает сумму чисел из множества чисел

Общий синтаксис этих функций:

имя функции([**ALL** | **DISTINCT**] выражение)

Квалификатор ALL указывает, что в вычислениях участвуют все значения, квалификатор DISTINCT указывает, что в вычислениях участвуют только уникальные значения; если имеются дубликаты, избыточные дублирующие значения удаляются. Если квалификатор не указан, предполагается ALL.

Функция COUNT имеет еще одну форму записи:

COUNT(*)

Все функции, кроме COUNT (*), игнорируют пустые (NULL) значения в колонке. Если аргументом функции является пустое множество значений, все функции, кроме COUNT (), возвращают неопределенное значение (NULL); функция COUNT (*выражение*) возвращает значение 0; функция COUNT (*) возвращает количество неопределенных значений.

Аргументами функций AVG () и SUM () могут быть только числовые значения; функции MAX () и MIN () допускают указание в качестве аргумента значений, для которых определены операции сравнения, например данные типа дата–время.

Функция COUNT() возвращает целое значение; тип результата остальных функций определяется типом аргумента.

Рассмотрим некоторые примеры использования агрегатных функций.

Пусть в некоторой таблице колонка coll целочисленного типа имеет следующее множество значений: {12, 5, NULL, 12, 8, 35, 5, NULL, 12}. Результат вычисления агрегатных функций для этой колонки приведен в табл. 7.2.

Таблица 7.2

Результаты вычисления агрегатных функций

count (*)	count (col1)	count (distinct col1)	sum (col1)	sum (distinct col1)	min (col1)	min (distinct col1)	max (col1)	max (distinct col1)	avg (col1)	avg (distinct col1)
9	7	4	89	60	5	5	35	35	12	15

Если колонка `coll` имеет следующее множество значений: {`NULL`, `NULL`}, тогда результат вычисления агрегатных функций будет следующим (табл. 7.3).

Таблица 7.3

Результаты обработки пустого множества значений

count(*)	count(coll)	count(distinct coll)	Все остальные функции
2	0	0	NULL

Скалярный подзапрос

Скалярный подзапрос, который можно записывать в скалярных выражениях, представляется в виде предложения `SELECT`, заключенного в круглые скобки (правила записи предложения `SELECT` будут рассмотрены позднее — см. п. 7.5.4). Он должен возвращать результат, содержащий максимум одно значение, т.е. только одну строку, имеющую только одну колонку. Если скалярный подзапрос не возвращает строку результата, тогда результатом выражения является значение `NULL`.

Само выражение

Case-выражение

Case-выражение позволяет определить результат на основе проверки одного или нескольких условий. Такая проверка может быть осуществлена двумя способами.

Первый способ имеет следующий вид:

Ниже приведены примеры использования оператора CASE:

```

CASE WHEN условие_поиска1 THEN выражение1 [WHEN условие_поискаК THEN выражениеK] ... ELSE выражениеL END

```

Вычисление выражения CASE, представленного таким образом, начинается с вычисления *условия_поиска1*. Если в результате будет получено значение false (ложь), вычисляется *условие_поискаК*, и т.д. Как только для какого-либо *условия_поискаК* будет получено значение true (истина), вычисляется связанное с ним *выражениеК*, определяющее результат вычисления выражения CASE, и на этом вычисления прекращаются. Если все условия поиска дают в результате значение false, результатом выражения CASE будет результат вычисления *выражениеЛ*, если присутствует ELSE, или же NULL, если ELSE не указано.

Следует отметить, что если при вычислении какого-либо *условия поиска* получается неопределенное значение (из-за наличия NULL-значений), такое значение не является истинным и интерпретируется как false.

Второй способ записи выражения CASE имеет следующий вид:

```

CASE выражение WHEN выражение1 THEN результирующее_выражение1 [ WHEN выражение2 THEN результирующее_выражение2 ...] [ ELSE результирующее_выражениеЛ END

```

При вычислении выражения CASE, заданного таким способом, результат вычисления *выражения* последовательно сравнивается с результатами вычисления *выражения1*, *выражения2* и т.д. (в соответствии с этим, тип результата вычисления *выражения* должен быть сравнимым с типом результатов вычисления выражений, записанных после WHEN). Как только очередное сравнение окажется успешным (*выражение = выражениеК*), вычисляется соответствующее *результатирующее_выражениеК*, определяющее результат вычисления выражения CASE, и вычисления заканчиваются. Если же значение *выражения* не совпадает ни с одним из результатов выражений, указанных после WHEN, тогда результатом вычисления выражения CASE является результат вычисления *результатирующего_выражениеЛ*, если указано ELSE, или NULL, если ELSE отсутствует.

Для обеих форм записи выражения CASE должны выполняться следующие условия:

- после ключевых слов THEN и ELSE вместо выражения может быть указано ключевое слово NULL;
- по крайней мере после хотя бы одного THEN не должно быть указано NULL;
- результаты всех выражений, записанных после THEN и ELSE, должны иметь совместимый тип данных.

Для обеих форм записи выражения CASE проверки, указанные в выражении, выполняются в порядке их записи.

Cast-спецификация

CAST-спецификация осуществляет преобразование типа и записывается следующим образом:

`CAST (выражение AS тип_данных)`

Результатом вычисления `CAST` является результат вычисления выражения, преобразованный к указанному типу данных. Если в качестве выражения записано `NULL`, `CAST` возвращает значение `NULL`, имеющее указанный тип данных. *Тип_данных* — название существующего типа данных. Если тип данных имеет дополнительные атрибуты, например длина, или точность и масштаб, эти атрибуты должны быть указаны вместе с типом данных.

Ограничения на использование типов данных:

- если в качестве операнда указано выражение, тогда допустимый тип данных определяется типом результата выражения (исходного типа данных);
 - если в качестве операнда указано ключевое слово NULL, тогда можно использовать любой существующий тип данных.

Если числовые данные преобразуются к символьному типу, в результате будет получена строка фиксированной длины. Если символьные данные преобразуются к числовому типу, результат преобразования зависит от заданного числового типа.

7.3.4. Предикаты

В записи предложений языка SQL широко используются *предикаты*. Предикат представляет собой некоторое выражение, которое определяет в результате вычисления одно значение — истина (true), ложь (false) или неопределенное (unknown). Стандарт SQL включает достаточно широкий набор предикатов. Особенности реализации предикатов зависят от конкретной СУБД.

Предикаты сравнения

ПРЕДКАТЫ СРАВНЕНИЯ

Предикат сравнения имеет следующий

Каждый из операндов — *операнд1* и *операнд2* — представляет собой скалярное выражение, возвращающее результат сопоставимого типа.

Используются следующие операции сравнения: равно ($=$), больше ($>$), меньше ($<$), больше или равно (\geq), меньше или равно (\leq), не равно (\neq).

Примечание. В СУБД MS SQL Server допускается использовать еще и следующие знаки операций: не равно ($!=$), не больше ($<$), не меньше ($>$).

В СУБД Oracle допускается использовать еще и не равно ($!=$), знаки операций не больше ($>$) и не меньше ($<$) не допускаются.

В СУБД **Oracle** допускается использовать еще и не равно (`!=`), знаки операций не больше (`>`) и не меньше (`<`) не допускаются.

Если хотя бы один из операндов имеет значение NULL, результат вычисления предиката — unknown. В противном случае результат вычисления предиката — true, если условие, заданное предикатом, выполняется в точности, или false, если условие не выполняется.

Предикаты сравнения с квантором

Предикат сравнения с квантором имеет следующий вид:

операнд операнда, спасибо за сайт, забыл выставить подпись

операцию операций_сравнения кодоватора таблицы-инициализации

Операнд задается в виде скалярного выражения.

Операция Сравнения – любая из перечисленных выше.
Каждая запись включает в себя следующие строки: **Имя | Фамилия | Год**

Квантор задается одним из следующих:
Кванторы **АЛН** и **СОМП** активизируются

Табличный_подзапрос должен возвращать таблицу, состоящую из

Условие, определяемое операцией сравнения, проверяется для

Если используется квантор ALL, тогда результат вычисления пре-
каждой строки из подзапроса.

- `true`, если подзапрос возвращает пустое множество строк или условие, определяемое операцией сравнения, выполняется для всех строк подзапроса;
 - `false`, если условие, определяемое операцией сравнения, не выполняется хотя бы для одной строки подзапроса;
 - `unknown`, если результат вычисления предиката не `true` и не `false`.

Если используются кванторы ANY или SOME, тогда результат вычисления предиката:

- **true**, если условие, определяемое операцией сравнения, выполняется хотя бы для одной строки подзапроса;
 - **false**, если подзапрос возвращает пустое множество строк или условие, определяемое операцией сравнения, не выполняется ни для одной строки подзапроса;
 - **unknown**, если результат вычисления предиката не **true** и не **false**.

Предикат BETWEEN

Предикат BETWEEN имеет следующий вид:

such as *and1* [not] BETWEEN *and2* AND *and3*

Onepano1 [NOT] BETWEEN Onepano2 AND Onepano3

Каждый из операндов представляет собой скалярное выражение. Значения всех операндов, используемых в предикате, должны быть сопоставимыми и допускать сравнение; кроме того, должно выполняться соотношение *операнд2* \leq *операнд3*.

Предикат *операнд1* BETWEEN *операнд2* AND *операнд3* эквивалентен

условию поиска

operand1 >= operand2 AND operand1 <= operand3

Предикат *операнд1 NOT BETWEEN операнд2 AND операнд3* эквивалентен условию поиска

`NOT (операнд1 >= операнд2 AND операнд1 <= операнд3)`

NOT (операнд1) = операнд2 AND операнд1 <= операнд3
или **операнд1 < операнд2 OR операнд1 > операнд3.**

Результатом вычисления предиката BETWEEN является значение true, если данное условие выполняется, false, если не выполняется, и unknown, если один из operandов имеет значение NULL. Результатом вычисления предиката NOT BETWEEN является значение true, если данное условие не выполняется, false, если выполняется, и unknown, если один из operandов имеет значение NULL.

Предикат IN

Предикат IN имеет следующий вид:

операнд [NOT] IN (список значений) | таблица | подзапрос

Оператор [NOT] IN (список_значений) | табличный_позвозапрос

Список_значений представляет собой перечисление отдельных значений через запятую в произвольном порядке. Значение *операнда* должно быть сопоставимым со значениями выражений из списка.

Табличный_подзапрос должен возвращать таблицу, состоящую из совокупности строк, содержащих только одну колонку.

Результатом вычисления предиката TN является значение

- true, если значение операнда совпадает хотя бы с одним значением из списка;
 - false, если значение операнда не совпадает ни с одним значением из списка и в списке нет неопределенных значений;
 - unknown, если значение выражения не совпадает ни с одним значением из списка и в списке есть неопределенные значения.

Результатом вычисления предиката NOT IN является значение:

- true, если значение выражения не совпадает ни с одним значением из списка и в списке нет неопределенных значений;
 - false, если значение выражения совпадает хотя бы с одним значением из списка;
 - unknown, если значение выражения не совпадает ни с одним значением из списка и в списке есть неопределенные значения.

Предикат LIKE

Предикат `LIKE` имеет следующий вид:

строковое_выражение [NOT] LIKE шаблон [ESCAPE escape-символ]

напоминание о том, что в языке SQL есть специальные символы, называемые метасимволами, которые позволяют выполнять различные операции с данными.

Предикат LIKE проверяет, соответствует ли строка, являющаяся результатом вычисления *строкового выражения*, заданному шаблону.

Результатом вычисления предиката LIKE является значение true, если значение выражения удовлетворяет шаблону, false, если не удовлетворяет, и unknown, если хотя бы один из аргументов имеет значение NULL. Результатом вычисления предиката NOT LIKE является значение true, если значение выражения не удовлетворяет шаблону, false, если удовлетворяет, и unknown, если хотя бы один из аргументов имеет значение NULL.

Шаблон представляется в виде строки, в которой могут быть использованы следующие специальные метасимволы:

— (символ подчеркивания) — соответствует одному (любому) символу;

% — соответствует любой (возможно, пустой) цепочке символов.

Любой другой символ соответствует самому себе.

Если в записи шаблона необходимо использовать обычные символы _ и/или % без их специального смысла, таким символам должен предшествовать *escape-символ*. Если он не задан, по умолчанию используется символ \. Так, шаблон 'A%' соответствует любой строке символов, начинающейся буквой A; шаблон 'A\%' соответствует строке, состоящей в точности из двух символов — A и %. Если нужно отменить специальный смысл самого символа \, ему также должен предшествовать *escape-символ* \. Так, шаблон 'A\\%' соответствует любой строке символов, начинающейся последовательностью из двух символов — A и \; шаблон 'A\\\\%' соответствует строке символов, состоящей в точности из трех символов — A, \ и %.

Примечание. В СУБД MS SQL Server в записи шаблона допускается использовать еще следующие специальные метасимволы:

[множество_символов] — соответствует одному какому-либо символу из указанного множества символов;

[^множество_символов] — соответствует одному какому-либо символу, кроме тех, которые включены в указанное множество символов. Множество символов задается либо перечислением нужных символов в любом порядке, например fxd12, либо заданием диапазона символов в виде *первый — последний*, например A-F. При этом первый и последний символы должны принадлежать одному и тому же множеству символов, обладающих внутренней упорядоченностью, — букв (прописных или строчных) латинского алфавита или цифр. Обязательно первый символ должен предшествовать последнему символу в упорядоченной последовательности символов.

Допускается при задании множества символов использовать оба способа.

Примеры:

[a-c] — соответствует одному из символов a, b, c;

[^0-9] — соответствует любому символу, кроме цифры;

[x0-9yz] — соответствует одному из символов x, y, z или любой цифре.

В СУБД **Oracle** поддерживается использование регулярных выражений, в том числе и для описания шаблона в условии поиска. Набор метасимволов соответствует стандарту POSIX. Для этого используют встроенные функции: REGEXP_LIKE, REGEXP_REPLACE, REGEXP_INSTR, REGEXP_SUBSTR, REGEXP_COUNT.

Предикат NULL

Предикат `NHL` имеет следующий вид:

enlarged to [not] null

Операнд может быть задан в виде скалярного выражения

Результат вычисления предиката `IS NULL` — true, если значение операнда `NULL`, и false, если значение операнда не `NULL`. Результат вычисления предиката `IS NOT NULL` — true, если значение операнда не `NULL`, и false, если значение операнда `NULL`.

Предикат EXISTS

Предикат EXIST

Результат вычисления предиката — true, если подзапрос возвращает хотя бы одну непустую строку, и false, если подзапрос не содержит ни одной строки.

7.3.5 Услуги поиска

Условие поиска используется в предложениях SQL и представляет собой выражение, которое, как и предикат, определяет в результате вычисления одно значение — истина (true), ложь (false) или неопределенное (unknown).

В записи условия поиска используются **логические операции** — унарная операция НЕ (**NOT**) и бинарные операции И (**AND**) и ИЛИ (**OR**). Для логических операций сохраняются традиционные приоритеты: NOT (наивысший приоритет), AND, OR (самый низкий приоритет). Круглые скобки позволяют изменить порядок вычисления условия поиска.

В общем случае условие поиска имеет следующий вид:

[**ноги**] склоняй [башмаки], склоняй [ноги] склоняй!

[NOT] операндо [бинарная_операци

- предикат (любой из рассмотренных выше);
 - условие поиска, заключенное в круглые скобки.

Ниже приведены основные логические операции:

- NOT** (логическое отрицание): возвращает значение **false**, если значение предыдущего выражения было **true**, и наоборот.
- P AND Q** (логическое умножение): возвращает значение **true**, если оба выражения истинны, и **false** в противном случае.
- P OR Q** (логическое сложение): возвращает значение **true**, если хотя бы одно из выражений истинно, и **false** в противном случае.

Значением операнда может быть одно из трех значений — true, false и unknown. В соответствии с этим логические операции реализуют троичную логику. Правила вычисления логических операций приведены в табл. 7.4.

Таблица 7.4

Правила вычисления логических операций

P	Q	NOT P	P AND Q	P OR Q
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false
false	unknown	true	false	unknown
unknown	true	unknown	unknown	true
unknown	false	unknown	false	unknown
unknown	unknown	unknown	unknown	unknown

Вопросы

- Охарактеризуйте основные направления использования языка SQL.
- Приведите основные типы данных, используемые в языке SQL (возможно, для того диалекта языка, который вы изучаете).
- Каковы особенности представления данных типа дата–время?
- Предположим, что в некоторой таблице есть колонка `col` типа INT, которая определяет тип сотрудника в соответствии со следующим соглашением: значение 1 соответствует штатному сотруднику, значение 2 — внештатному; другие значения недопустимы. Используя конструкцию CASE, запишите на языке SQL выражение, результатом которого является текстовая строка, соответствующая указанному в колонке типу сотрудника.
- Напишите предикат, результат вычисления которого — истина, если значение некоторой колонки `col` типа INT таблицы попадает в промежуток $[-100, 100]$, и ложь в противном случае. Рассмотрите разные варианты написания предиката.
- Напишите предикат, результат вычисления которого — истина, если значение некоторой колонки `col` типа VARCHAR(20) таблицы начинается двумя любыми цифрами, после которых следуют любые символы, или ложь в противном случае. Рассмотрите разные варианты написания предиката.
- Напишите предикат, результат которого — истина, если значение некоторой колонки `col` типа INT совпадает с каким-либо одним значением из списка $\{10, 20, 30, 40\}$, и ложь в противном случае. Рассмотрите разные варианты написания предиката.

7.4. СРЕДСТВА SQL ДЛЯ ОПИСАНИЯ ОБЪЕКТОВ БАЗЫ ДАННЫХ

Язык определения данных (ЯОД) для реляционной модели данных включает следующие возможности:

- создание домена;
 - создание отношения;
 - определение ограничений целостности.

Соответствие между структурными компонентами реляционной модели данных и элементами реляционной базы данных, определяемыми с помощью SQL, приведено в табл. 7.5.

Таблица 7.5

Соответствие между компонентами РМД и элементами базы данных

Структурный компонент реляционной модели данных	Элемент реляционной базы данных
Отношение	Таблица
Кортеж отношения	Строка таблицы
Атрибут	Колонка таблицы
Домен	Тип данных

Описание ограничений целостности, поддерживаемых моделью данных, осуществляется также средствами языка описания данных. Типы ограничений целостности, определяемые средствами SQL, приведены в табл. 7.6.

Таблица 7.6

Определение ограничений целостности в SQL

Тип ограничения целостности	Представление средствами SQL
Уникальность значений	PRIMARY KEY или UNIQUE
Обязательность значений	NULL или NOT NULL
Допустимость значений	CHECK
Ссылочное ограничение	FOREIGN KEY

Если база данных была спроектирована с использованием модели сущность–связь и представлена в виде ER-диаграммы, ее преобразование в реляционную базу данных осуществляется достаточно просто, руководствуясь следующими рекомендациями.

- Каждой сущности ставится в соответствие таблица базы данных.
 - Атрибуты сущности — это колонки соответствующих таблиц, для которых выбираются подходящие типы данных в соответствии с описанием предметной области.
 - Если ER-диаграмма разработана в соответствии с рекомендациями IDEF1x, как в рассматриваемом примере, тогда все ключевые атрибуты выделены на диаграмме и для них используются необходимые ограничения целостности.

4. Определяются дополнительные ограничения целостности, определяемые предметной областью и отсутствующие на ER-диаграмме.

В Приложении 1 приведено описание схемы спроектированной базы данных проката автотранспортных средств в соответствии с приведенными рекомендациями.

В реально существующих реляционных СУБД (РСУБД), работающих на основе реляционной модели данных, в качестве ЯОД используется некоторое подмножество языка SQL, предоставляющее соответствующие средства. Так, в стандарте SQL-92 определено предложение CREATE DOMAIN для создания доменов, однако далеко не все СУБД такую возможность поддерживают, или же они используют собственные средства для определения множества допустимых значений. Но во всех РСУБД есть предложение CREATE TABLE, позволяющее создать таблицу (отношение реляционной модели данных) и определить для нее необходимые ограничения целостности.

Подмножество языка SQL для описания данных включает в свой состав три основных предложения:

- **CREATE** *тип_объекта* — создать соответствующий объект базы данных;
 - **ALTER** *тип_объекта* — изменить соответствующий объект базы данных;

- *DROP тип_объекта* — удалить указанный объект базы данных.

В данном разделе в качестве объекта базы данных рассматриваются два объекта: таблица и последовательность. В соответствии с этим, в качестве *типа_объекта* используются ключевые слова TABLE и SEQUENCE.

7.4.1. Создание таблицы

Рассмотрим основной синтаксис предложения CREATE TABLE для создания базовой таблицы базы данных в пределах тех возможностей, которые определяются реляционной моделью данных (т.е. отображение схемы отношения и указание всех необходимых ограничений целостности).

Предложение CREATE TABLE для создания базовой таблицы имеет следующий синтаксис:

```
CREATE TABLE имя_таблицы (  
    элемент_таблицы  
    [, ...]  
)
```

Здесь *имя_таблицы* — идентификатор, уникальный в пределах схемы, задает имя таблицы; элемент *таблицы* — представляет опре-

напоминаниях. В Oracle ограничение целостности на таблицу определяется с помощью предложения `CONSTRAINT`, в MS SQL Server — с помощью предложения `ALTER TABLE`. Для определения колонки или ограничение целостности на таблицу, при этом в предложении `CREATE TABLE` должно присутствовать по крайней мере одно определение колонки.

Определение колонки имеет следующий вид:

`имя_колонки тип_данных [ограничение_обязательности] [ограничение_целостности_на_колонку] [спецификация_генерируемого_значения]`

где **имя_колонки** — идентификатор, уникальный в пределах данной таблицы; задает имя соответствующей колонки; **тип_данных** — тип данных для колонки; **ограничение_обязательности** — задает ограничение обязательности значения данной колонки; **ограничение_целостности_на_колонку** — ограничения целостности (одно или несколько), накладываемые на данную колонку; **спецификация_генерируемого_значения** — указывает, какое значение должно быть вставлено в данную колонку в строке таблицы, если в операции вставки значение данной колонки не указано.

Ограничение обязательности, ограничения целостности на колонку и спецификация генерируемого значения могут быть записаны в произвольном порядке.

Примечание. В СУБД MS SQL Server в определении колонки допускается еще использование атрибута автоинкрементного значения.

В СУБД Oracle в определении колонки явно не допускается использование атрибута автоинкрементного значения. Данный механизм реализуется посредством использования специальных объектов базы данных — триггеров и последовательностей.

Ограничение целостности на таблицу — это ограничение целостности, накладываемое на одну или несколько колонок создаваемой таблицы.

Если некоторое ограничение целостности накладывается на одну колонку таблицы, оно может быть задано и как ограничение целостности на колонку, и как ограничение целостности на таблицу. Если же некоторое ограничение целостности затрагивает несколько колонок таблицы, например, составной первичный ключ, или какое-то соотношение между значениями нескольких колонок таблицы, такое ограничение может быть задано только как ограничение целостности на таблицу. При этом ограничение целостности на таблицу должно быть указано после того, как будут определены все колонки, используемые в этом ограничении.

Рассмотрим правила записи отдельных компонентов в определении элемента таблицы.

Ограничение обязательности

При задании ограничения обязательности допускается одно из двух значений — **NOT NULL** или **NULL**.

NOT NULL — признак обязательности значения данной колонки, означающий, что значение данной колонки не может быть опущено.

NULL — признак необязательности значения данной колонки; указывает, что значение данной колонки может отсутствовать.

Если ограничение обязательности не указано, принимается значение по умолчанию, определяемое конкретной СУБД; обычно это **NULL**.

Ограничения целостности

Ограничения целостности Ограничение целостности на колонку и на таблицу задается следующим образом:

[CONSTRAINT имя_ограничения] ограничение [атрибуты_ограничения]

Здесь *имя_ограничения* — задает имя ограничения. Должно быть уникальным в пределах схемы. Рекомендуется при задании имени

уникальным в пределах схемы. Рекомендуется при задании имени ограничения использовать в нем имя таблицы, тип ограничения и некоторую дополнительную информацию, гарантирующую уникальность имени. Например, если в таблице с именем `tab1` определяется ограничение уникальности, его имя может быть определено как `tab1_UQ_01`, учитывая, что ограничений уникальности в одной таблице может быть несколько. Если имя ограничения не указывается, система сама генерирует уникальный идентификатор;

- *ограничение* — задает конкретное ограничение целостности. Конкретный способ записи ограничения зависит от того, какое ограничение — на колонку или на таблицу — используется;
 - *атрибуты_ограничения* — дополнительные параметры, зависящие от типа ограничения.

Рассмотрим правила задания ограничений

Ограничение уникальности

Ограничение уникальности на колонку имеет вид: **UNIQUE** или **PRIMARY KEY**. Ограничение уникальности на таблицу имеет вид: **UNIQUE(имя_колонки, ...)** или **PRIMARY KEY(имя_колонки, ...)**. В этом случае после указания типа ограничения (**UNIQUE** или **PRIMARY KEY**) перечисляются имена колонок таблицы, которые используются в данном ограничении.

- UNIQUE** — определяет альтернативный ключ;
 - PRIMARY KEY** — определяет первичный ключ.

Колонки, для которых определено ограничение уникальности (UNIQUE или PRIMARY KEY), должны быть объявлены как NOT NULL.

Ниже приведены примеры определения ограничений на колонки:

```
CREATE TABLE tabl (
    colA INT NOT NULL CONSTRAINT tabl_pk PRIMARY KEY,
    colB INT NOT NULL,
    colC INT NOT NULL
);
```

Примечание. В СУБД Oracle и MS SQL Server допускается определять ограничение уникальности **UNIQUE** и для колонок, которые объявлены как **NULL**; в этом случае в таблице может появиться только одна строка, в которой значение данной колонки — **NULL**.

Одна и та же последовательность колонок может быть указана только в одном ограничении уникальности (**UNIQUE** или **PRIMARY KEY**). В определении таблицы может быть указано несколько ограничений **UNIQUE** и только одно ограничение **PRIMARY KEY**.

В приведенных ниже примерах для задания типов данных используется диалект языка SQL, соответствующий стандарту SQL-92.

1. Ограничение на колонку

```
CREATE TABLE tabl (
    colA INT NOT NULL CONSTRAINT tabl_pk PRIMARY KEY,
```

Эквивалентно следующему ограничению на таблицу:

```
CREATE TABLE tabl (
    colA INT NOT NULL,
    CONSTRAINT tabl_pk PRIMARY KEY(colA),
```

Неправильное определение составного первичного ключа:

```
CREATE TABLE tabl (
    colA INT NOT NULL,
    colB INT NOT NULL,
    PRIMARY KEY(colA, colB),
```

Неправильное определение составного первичного ключа:

```
CREATE TABLE tabl (
    colA INT NOT NULL PRIMARY KEY,
    colB INT NOT NULL PRIMARY KEY,
```

Примечание. В СУБД Oracle и MS SQL Server при задании табличного ограничения уникальности порядок перечисления имен колонок должен соответствовать порядку определения данных колонок в предложении CREATE TABLE. Нарушение этого правила может проявиться в дальнейшем при определении и использовании дочерних таблиц, ссылающихся на данную таблицу.

Ограничение допустимости значения

Ограничение, определяющее допустимость знания, имеет вид

CHECK (установка)

Здесь *условие* записывается в соответствии с правилами записи *условия поиска*, рассмотренными ранее (см. п. 7.3.5), с учетом следующих особенностей:

- в записи операндов условия должны упоминаться имена колонок только из данной таблицы;
 - в качестве операндов условия могут использоваться константные выражения (т.е. выражения, содержащие только константные значения);
 - в качестве операндов условия могут использоваться функции, возвращающие текущие дату–время;
 - в записи операндов условия не допускаются подзапросы.

Условие, записанное в ограничении на колонку, может содержать имя только данной колонки; условие, записанное в ограничении на таблицу, может содержать любые (одно или более) имена колонок из данной таблицы.

Примеры

Примеры:

1. Ограничение на количественные

эквивалентно следующему ограничению на таблицу:

2. Следующее ограничение может быть представлено только как ограничение на таблицу:

CREATE TABLE *tbl_1* (

```

CREATE TABLE tab1(
    ...
    colA INT NOT NULL,
    colB INT NOT NULL,
    CHECK(colB > colA),
    ...
)

```

Неправильное определение ограничения:

Неправильное определение ограничения:

```
CREATE TABLE tab1(
    colA INT NOT NULL,
    colB INT NOT NULL CHECK(colB > colA),
)

```

Сылочное ограничение

Ссылочное ограничение на колонку задается следующим образом:

[**FOREIGN KEY**] REFERENCES *имя_родительской_таблицы*
[(*имя_колонки_AK*, ...)] [ON DELETE *правило_удаления*] [ON UP-
DATE *правило обновления*]

Ссыльное ограничение на таблицу задается следующим образом:

FOREIGN KEY (имя_колонки_FK, ...) REFERENCES имя_родительской_таблицы [(имя_колонки_AK, ...)] [ON DELETE правило удаления] [ON UPDATE правило обновления]

Здесь имя *родительской таблицы* — имя родительской таблицы, записанное в первом столбце

которую ссылается создаваемая дочерняя таблица; *имя_колонки_AK* — имена колонок ключа в родительской таблице.

список имен колонок ключа должен соответствовать множеству имен колонок первичного ключа или уникального ключа, определенных в родительской таблице. Если список имен колонок не указан, предполагается, что задан первичный ключ родительской таблицы;

имя_колонки_FK – имена колонок внешнего ключа в создаваемой (и в дочерней) таблице. Каждое имя колонки в списке должно соответствовать имени колонки, определенной в дочерней таблице. Порядок перечисления колонок в ограничении должен соответствовать порядку определения колонок в таблице. В списке не должно одно и то же имя колонки указываться дважды. Колонки, перечисленные в ограничении, должны по порядку записи, количеству и типам данных совпадать с колонками первичного или альтернативного ключа родительской таблицы. Это означает, что в определении внешнего ключа должно быть указано столько же колонок, сколько колонок указано в первичном или альтернативном ключе родительской таблицы, и тип данных n-й колонки в списке колонок внешнего ключа должен совпадать с типом данных n-й колонки в списке колонок первичного или альтернативного ключа родительской таблицы;

правило_удаления — определяет, какие действия с дочерней таблицей должны быть выполнены при удалении записи из родительской таблицы. Допускаются следующие значения:

NO ACTION — удаление записи из родительской таблицы не происходит, если в дочерней таблице есть хотя бы одна запись, ссылающаяся на удаляемую;

CASCADE — вместе с удаляемой записью из родительской таблицы удаляются все ссылающиеся на нее записи из дочерней таблицы;

SET NULL — при удалении записи из родительской таблицы в колонки внешнего ключа ссылающихся на нее записей дочерней таблицы записывается значение NULL (колонки внешнего ключа должны допускать NULL-значение);

SET DEFAULT — при удалении записи из родительской таблицы в колонки внешнего ключа ссылающихся на нее записей дочерней таблицы записывается значение по умолчанию, определенное для колонок внешнего ключа; для колонок внешнего ключа в этом случае должна быть указана спецификация генерируемого значения;

правило_обновления — определяет, какие действия с дочерней таблицей должны быть выполнены при модификации ключа в родительской таблице. Допускаются те же значения, что и в правиле удаления.

Если правило удаления (и/или обновления) не указано, по умолчанию принимается NO ACTION.

Примечание. В СУБД Oracle не поддерживаются правила SET DEFAULT и NO ACTION. Правило NO ACTION определяется как правило по умолчанию и не должно явно указываться в определении ограничения. Явное использование этих правил при задании ссылочного ограничения приведет к сообщению об ошибке.

Спецификация генерируемого значения

При определении колонки в таблице можно для колонки указать спецификацию генерируемого значения. Данная спецификация позволяет указать, какое значение должно быть вставлено в строку таблицы, если в операции вставки значение данной колонки не указано.

Спецификация генерируемого значения задается следующим образом:

www.ecocycle.org • 800-338-1112

DEFAULT значение В качестве значения может быть задана некоторая константа (или константное выражение), тип которой должен соответствовать типу данных колонки; функция, возвращающая текущие дату–время (для колонки соответствующего типа) или NULL. Указанное значение, за исключением функции даты–времени, вычисляется один раз, во время создания таблицы, и используется в дальнейшем в операциях вставки. Если же в качестве значения указана функция даты–времени, соответствующее значение вычисляется в тот момент, когда выполняется операция вставки в таблицу новой строки.

Ниже приведены примеры определения атрибутов для колонок:

```

CREATE TABLE tab1 (
    colA INT NOT NULL DEFAULT 0,
    colB INT IDENTITY(1,1)
);

```

Пример:

```
CREATE TABLE tab1 (
```

```
    ...  
    colA INT NOT NULL DEFAULT 0,
```

```
)
```

Примечание. В СУБД MS SQL Server спецификация генерируемого значения рассматривается как некоторое дополнительное ограничение целостности, накладываемое на колонку, и может быть задано в виде:

```
[CONSTRAINT имя_ограничения] DEFAULT значение
```

Задание автоинкрементного значения. В СУБД MS SQL Server в определении колонки может быть указан дополнительный атрибут, задающий автоинкрементное значение. Этот атрибут имеет следующий вид:

```
IDENTITY [ (начальное_значение[, приращение]) ]
```

Начальное значение и приращение представляют собой целые константы, большие нуля. Если они не указаны, по умолчанию предполагается значение 1, т.е. задание IDENTITY эквивалентно заданию IDENTITY(1,1).

В таблице может быть определена только одна колонка с атрибутом IDENTITY, и эта колонка должна иметь целочисленный тип. Обычно атрибут IDENTITY используется для определения суррогатного первичного ключа таблицы.

Пример:

```
CREATE TABLE tab1 (
```

```
    tabId INT NOT NULL PRIMARY KEY IDENTITY,
```

```
)
```

Данный атрибут определяет последовательность целых чисел, которая используется при вставке в таблицу новых строк. Для каждой таблицы, использующей атрибут IDENTITY, определяется своя последовательность. Каждая новая строка, вставляемая в таблицу, использует следующее значение из соответствующей последовательности.

В СУБД Oracle определение автоинкрементного ключа задается с использованием двух объектов: триггера уровня строки типа BEFORE и последовательности (SEQUENCE). Объект последовательность (SEQUENCE) используется для генерации последовательности уникальных целых чисел. Для создания последовательности используется предложение CREATE SEQUENCE.

Ниже приведены примеры создания таблиц в SQL-диалекте.

Пример:

```
CREATE SEQUENCE country_pk_seq
    START WITH 1000
    INCREMENT BY 1;
```

В представленном примере создается последовательность с именем `country_pk_seq`, первое число последовательности равно 1000, шаг между числами последовательности равен 1. Созданная последовательность используется в триггере, который срабатывает для каждой добавляемой строки таблицы и присваивает очередное значение последовательности атрибуту, который определяется как **автоматический/инкрементный**.

Пример:

```
CREATE TRIGGER country_pk_tr
    BEFORE INSERT ON country
    FOR EACH ROW
    BEGIN
```

```
    :new.countryid := country_pk_seq.nextval;
END country_pk_tr;
```

Подробное описание правил определения и использования последовательностей и триггеров будет представлено далее (см. пп. 7.4.4, 7.7.7, дополнительную литературу и документацию).

7.4.2. Примеры создания таблиц

Рассмотрим примеры создания таблиц.

Пусть имеется следующий фрагмент полноатрибутной схемы базы данных (рис. 7.2).

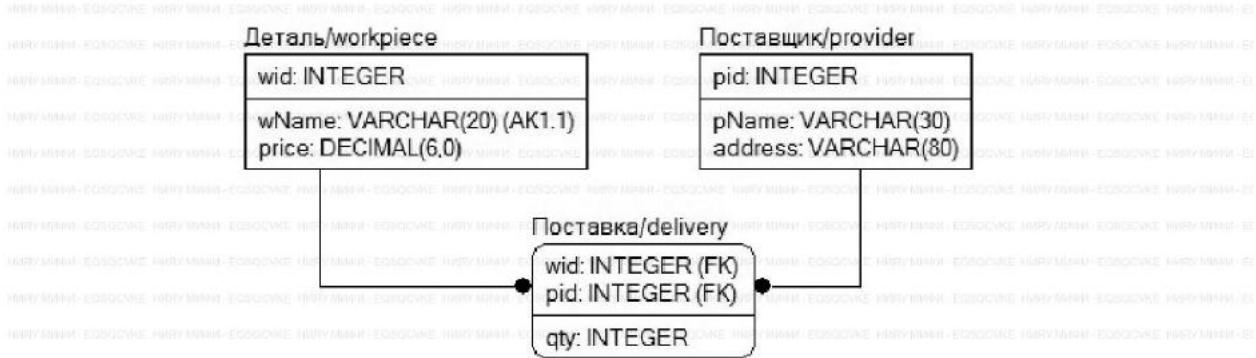


Рис. 7.2. Фрагмент полноатрибутной схемы базы данных «Поставка»

В приведенных ниже примерах создания таблиц использован диалект SQL, соответствующий стандарту SQL-92. Для выполнения указанных примеров в конкретной СУБД следует скорректировать приведенные примеры с учетом диалекта SQL конкретной СУБД.

Ниже приведены примеры определения ограничений для таблицы workpiece.

Пример 1. Отношение ДЕТАЛЬ (таблица workpiece) может быть определено следующим образом:

```
CREATE TABLE workpiece(
    wid INT NOT NULL CONSTRAINT workpiece_PK PRIMARY KEY,
    wName VARCHAR(20) NOT NULL CONSTRAINT workpiece_UQ_01
        UNIQUE, --1
    price DECIMAL(6,0) NOT NULL CONSTRAINT workpiece_CH_01
        CHECK(price > 0) --3
)
```

Пояснения:

- 1 — определены ограничения первичного ключа (ограничение PRIMARY KEY, имя ограничения — workpiece_PK) и обязательности значения;
- 2 — определены ограничения уникальности (имя ограничения — workpiece_UQ_01) и обязательности значения;
- 3 — определено условие (имя ограничения — workpiece_CH_01): значение атрибута должно быть строго положительным.

При определении ограничений полезно задавать их имена, которые обычно строятся по следующему правилу: в имя ограничения включается имя таблицы, затем условное название типа ограничения и номер ограничения данного типа, если их в данной таблице может быть несколько, или смысловое название ограничения.

Пример 2. Отношение связи ПОСТАВКА может быть определено следующим образом. Так как отношение ПОСТАВКА является отношением связи (дочерним отношением) между отношениями ПОСТАВЩИК и ДЕТАЛЬ (родительскими отношениями), прежде чем определять дочернее отношение, необходимо определить родительские отношения.

Определение отношения ДЕТАЛЬ приведено выше.

Отношение ПОСТАВЩИК (таблица provider) может быть определено следующим образом:

```
CREATE TABLE provider(
    pid INT NOT NULL,
    pName VARCHAR(30) NOT NULL,
    address VARCHAR(80),
    CONSTRAINT provider_PK PRIMARY KEY(pid) -- 4
)
```

Пояснения:

- 4 — как говорилось выше, ограничение, накладываемое на одну колонку таблицы, может быть представлено и как ограничение на колонку, и как ограничение на таблицу. В определении таблицы workpiece первый ключ был определен при определении колонки

156

Например, ограничение на колонку, а в определении данной таблицы первичный ключ определен как ограничение на таблицу.

Теперь определим отношение связи ПОСТАВКА (таблица delivery):

```
CREATE TABLE delivery (
    pid INT NOT NULL CONSTRAINT delivery_FK_01 REFERENCES provider,
    wid INT NOT NULL,
    qty INT NOT NULL CONSTRAINT delivery_CH_01 CHECK(qty > 0),
    CONSTRAINT delivery_PK PRIMARY KEY(pid, wid),
    CONSTRAINT delivery_FK_02 FOREIGN KEY(wid) REFERENCES workpiece --7 )
```

Пояснения:

5 — определяется внешний ключ; в ограничении внешнего ключа (ключевое слово REFERENCES) указывается родительская таблица и ее первичный ключ; по умолчанию правило удаления определяется как NO ACTION;

6 – определяется ограничение первичного ключа; так как в данном примере первичный ключ составной, определение первичного ключа может быть задано только как ограничение на таблицу (ограничение внешнего ключа в данном примере, устанавливаемое только для одного атрибута, является ограничением на колонку).

7 — также определяется внешний ключ, только в виде ограничения на таблицу. Имя колонки данной таблицы, являющейся внешним ключом, указывается после ключевых слов FOREIGN KEY; REFERENCES указывает родительскую таблицу; правило удаления также определяется как NO ACTION.

Пример 3. Ранее была спроектирована схема базы данных для организации проката автотранспортных средств. SQL-скрипт создания базы данных для СУБД **MS SQL Server** и **Oracle** приведен в Приложении 2.

7.4.3. Удаление и модификация таблиц

С помощью предложений `DROP TABLE` и `ALTER TABLE` можно удалять существующие таблицы или изменять их структуру. В соответствующих предложениях должно быть указано имя существующей таблицы.

Удаление таблицы

Удаление таблицы
Удаление таблицы осуществляется с помощью предложения **DROP TABLE**, имеющего следующий вид:

напоминаниях. Для удаления таблицы необходимо выполнить предложение **DROP TABLE**, в котором указывается имя таблицы:

DROP TABLE имя_таблицы

Операция удаления не будет выполнена, если удаляется родительская таблица, на которую ссылается хотя бы одна дочерняя таблица.

Прежде чем удалять родительскую таблицу, необходимо удалить все дочерние таблицы, ссылающиеся на нее.

При удалении таблицы удаляются все индексы и ограничения, определенные для данной таблицы.

Пример. Пусть даны следующие определения таблиц:

```
CREATE TABLE t1 (
    t1Id INT NOT NULL PRIMARY KEY,
    t1Name VARCHAR(50)
);
CREATE TABLE t2 (
    t2Id INT NOT NULL PRIMARY KEY,
    t1Id INT NOT NULL REFERENCES t1
);
```

Тогда предложение **DROP TABLE t1** не выполнится из-за наличия ссылки на таблицу **t1** из таблицы **t2**.

Если, тем не менее, требуется удалить таблицу **t1**, надо сначала удалить таблицу **t2** или ограничение внешнего ключа из таблицы **t2**, и только после этого можно удалить таблицу **t1**.

Примечание. В СУБД **Oracle** допускается еще один способ задания предложения:

DROP TABLE имя_таблицы CASCADE CONSTRAINTS

В этом случае удаляется родительская таблица и все ограничения внешних ключей из дочерних таблиц, ссылающиеся на данную таблицу. Так, для приведенных выше примеров определения таблиц предложение **DROP TABLE t1 CASCADE CONSTRAINTS** приведет к удалению таблицы **t1** и ограничению внешнего ключа из таблицы **t2**.

Модификация таблицы

Модификация таблицы осуществляется с помощью предложения **ALTER TABLE**. Модификация таблицы предполагает изменение структуры таблицы; при этом можно добавлять новые и удалять существующие колонки, изменять некоторые характеристики существующих колонок, а также добавлять новые и удалять существующие ограничения целостности. В одном предложении **ALTER TABLE** могут быть указаны несколько операций, изменяющих структуру таблицы.

В общем случае предложение **ALTER TABLE** имеет следующий вид:

158

ALTER TABLE *имя_таблицы* *операция* [*операция* ...]

Рассмотрим некоторые возможности, предоставляемые предложением **ALTER TABLE**.

Добавление новых колонок

Добавление новой колонки в таблицу выполняется с помощью следующей операции:

APP определение коленки

Определение колонки имеет такой же вид, как и описание колонки, используемое в предложении CREATE TABLE:

имя новой колонки тип данных [ограничение обязательности]

[ограничение_на_колонку] **[значение_по_умолчанию]**

При добавлении новой колонки в таблицу необходимо обеспечить

ЧИТАТЬ ВЫПОЛНЕНИЕ СЛЕДУЮЩИХ УСЛОВИЙ:

- имя новой колонки не должно совпадать с именами колонок, уже существующих в таблице;
 - если для новой колонки указано ограничение обязательности NOT NULL, должно быть обязательно задано значение по умолчанию с помощью DEFAULT.

Пример. Пусть имеется таблица `equipment`, в которой находится информация об оборудовании, выделенном для некоторых отделов предприятия. В эту таблицу необходимо добавить колонку с именем `equip_qty` для указания количества оборудования, предоставленного отделу. Значение данной колонки является обязательным; по умолчанию должно подставляться значение 1:

ALTER TABLE equipment

ADD equip_qty SMALLINT NOT NULL DEFAULT 1

Изменение характеристик существующей колонки

Надо отметить, что стандарт SQL-92 с помощью предложения ALTER TABLE для существующей колонки допускает только установку или удаление значения по умолчанию. В соответствии с этим операция может быть задана одним из двух способов:

ALTER COLUMN имя_колонки SET DEFAULT значение

ИПН

ALTER COLUMN имя_колонки DROP DEFAULT

Конкретные СУБД могут существенно изме-

предложения ALTER TABLE в этой области.

Примечание. В СУБД MS SQL Server операция не выполняется, если удаляемая колонка упоминается в любом ограничении целостности.

Так, предложение

ALTER TABLE t2 DROP COLUMN t1id

```
ALTER TABLE cz DROP COLUMN ctyp
```

также вызовет сообщение об ошибке.

В СУБД Oracle допускается еще один способ удаления колонки:

DROP COLUMN имя_колонки CASCADE CONSTRAINTS

В этом случае колонка удаляется вместе со всеми ограничениями, в которых упоминается данная колонка. Так, если в таблице `t1` колонка `t1id` определена как первичный ключ, а в таблице `t2` определен внешний ключ, ссылающийся на колонку `t1id`, тогда предложение

ALTER TABLE t1 DROP COLUMN tlid CASCADE CONSTRAINTS приведет к удалению колонки tlid и ограничению внешнего ключа из таблицы t2.

Добавление ограничений целостности

Добавление новых ограничений
с помощью следующей операции:

ABB [CONSTRAINT] ица, ограничение] ограничение

ADD [CONSTRAINT *имя_ограничения*] ограничение

Данная операция позволяет добавить в таблицу ограничения уникальности, допустимости значения или ссылочное ограничение. Ограничение задается точно так же, как и табличное ограничение в предложении CREATE TABLE.

Пример 1. Предположим, что существует таблица с именем equipment, в которой колонка equip_owner определяет отдел, для которого выделено оборудование. Необходимо добавить в таблицу ссылочное ограничение целостности, в соответствии с которым колонка equip_owner должна представлять некоторый отдел из тех, которые определены в таблице dept. Первичным ключом таблицы dept является колонка с именем deptNo. Если из таблицы dept удаляется некоторый отдел, которому выделено оборудование, значение колонки equip_owner для такого оборудования должно быть установлено как неопределенное (NULL). Новому ограничению необходимо присвоить имя deptEquip:

```
ALTER TABLE equipment
ADD CONSTRAINT deptQuip
FOREIGN KEY(equip_owner)
REFERENCES dept(deptNo)
ON DELETE SET NULL;
```

Пример 2. В таблицу `employee` следует добавить ограничение с именем `unique_name`, пропорциональное, что каждый сотрудник должен

иметь суммарное значение зарплаты (колонка salary) и комиссии (наличие, если), превышающее значение 20 000;

иметь суммарное значение зарплаты (колонка salary) и комиссионных (колонка comm), превышающее значение 30 000:

```
ALTER TABLE employee ADD CONSTRAINT revenue CHECK  
(salary + comm > 30_000)
```

Примечание. В СУБД MS SQL Server данная операция в предложении ALTER TABLE может быть использована и для добавления существующей колонке значения по умолчанию. В приведенном ниже примере колонке col в таблице tt добавляется значение по умолчанию 0:

ALTER TABLE tt ADD CONSTRAINT ttDefault DEFAULT 0 FOR col
или

```
ALTER TABLE tt ADD DEFAULT 0 FOR col
```

Если для некоторой колонки надо переустановить значение по умолчанию, эта операция выполняется в два этапа: сначала удаляется су-

ществующее значение по умолчанию, а затем устанавливается новое. В СУБД **Oracle** для добавления существующей колонке значения по

умолчанию используется операция `MODIFY`. В приведенном ниже примере колонке `col` в таблице `tt` добавляется значение по умолча-

ALTER TABLE tt MODIFY col DEFAULT 0

Удаление ограничений целостности

Существующие в таблице ограничения целостности могут быть удалены с помощью следующей операции:

записи с помощью следующей операции:

И здесь, если на удаляемое ограничение целостности есть ссылки из каких-либо других ограничений целостности, ограничение не уда-

Пример 1. Удалить из таблицы employee ограничение revenue, добавленное в предыдущем примере.

обавленное в предыдущем примере.

Пример 2. Пусть в таблице dept на колонке deptNo определено ограничение первичного ключа с именем dept_PK, а в таблице employee определено ограничение внешнего ключа deptQuip, ссылающееся на первичный ключ таблицы dept. Тогда следующее предложение:

```
ALTER TABLE employee DROP CONSTRAINT deptQuip
```

Примечание. В СУБД Oracle допускается еще одна форма задания данной

DROP CONSTRAINT *имя_ограничения* **CASCADE**

В этом случае удаляется указанное ограничение вместе с другими ограничениями, ссылающимися на данное. Например, предложение ALTER TABLE dept DROP CONSTRAINT dept_pk CASCADE приведет к удалению ограничения первичного ключа dept_pk и ограничению внешнего ключа из таблицы employee.

В СУБД MS SQL Server данная операция в предложении ALTER TABLE может быть использована и для удаления у существующей колонки значения по умолчанию, например:

```
ALTER TABLE tt DROP CONSTRAINT ttDefault
```

```
ALTER TABLE tt DROP CONSTRAINT ttDefault
```

7.4.4. Создание и использование последовательности

В стандарте SQL-92 объект последовательность не определен; поэтому создание и использование последовательности зависит от реализации.

В общем случае, последовательность создается с помощью следующего предложения:

CREATE SEQUENCE *имя_последовательности* [**AS** *целый_тип_данных*] [INCREMENT_BY *шаг_счетчика*] [MINVALUE *минимальное_значение*] [MAXVALUE *максимальное_значение*] [CYCLE] [NO CYCLE]

[START WITH целое] [INCREMENT BY целое]
[минимальное_значение] [максимальное_значение] [цикличность]
[кэш]

Как видно, при определении последовательности обязательным является только первый компонент — `CREATE SEQUENCE имя_последовательности`. Остальные компоненты не обязательны и могут быть опущены; в этом случае для них устанавливаются значения по умолчанию, определяемые в соответствии с некоторыми соглашениями.

AS определяет тип данных для элементов последовательности.

AS — определяет тип данных для элементов последовательности.
Должен задавать обязательно целочисленный тип.

Примечание. В СУБД Oracle конструкция AS не указывается.
В СУБД MS SQL Server:

В СУБД MS SQL Server, если не указано AS, принимается тип данных BIGINT.

START WITH — определяет начальное значение для элементов последовательности; если не указано, в качестве начального значения берется минимальное значение, установленное для данной последовательности.

INCREMENT BY — задает шаг изменения значений элементов последовательности. Так, если обозначить начальное значение через *s*, а шаг изменения — через *h*, тогда при первом обращении к данной последовательности будет получено значение *s*, при вто-

ром — $s + h$, при третьем — $s + 2h$ и т.д. Значение шага изменения может быть положительным — тогда задается возрастающая последовательность, и отрицательным — тогда задается убывающая последовательность.

Минимальное значение — определяет минимальное значение для данной последовательности. Может быть задано в виде **MINVALUE** [*целая_константа*] или **NO MINVALUE**. Если минимальное значение явно не указано, тогда в качестве минимального значения принимается: для возрастающей последовательности — минимально возможное значение, соответствующее типу данных элементов последовательности; для убывающей последовательности — максимально возможное значение, соответствующее типу данных элементов последовательности.

Максимальное значение — определяет максимальное значение для данной последовательности. Может быть задано в виде **MAXVALUE** [*целая_константа*] или **NO MAXVALUE**. Если максимальное значение явно не указано, тогда в качестве максимального значения принимается: для возрастающей последовательности — максимально возможное значение, соответствующее типу данных элементов последовательности; для убывающей последовательности — минимально возможное значение, соответствующее типу данных элементов последовательности.

Цикличность — задается в виде **CYCLE** или **NO CYCLE**; определяет, что происходит при достижении максимального значения. Если задано **CYCLE**, тогда при достижении максимального значения следующее обращение к последовательности приводит к циклическому переходу к минимальному значению. Если задано **NO CYCLE** или цикличность не указана, тогда при достижении максимального значения следующее обращение к последовательности приводит к генерации исключения (сообщения об ошибке).

Кэш — задается в виде **CACHE** [*целое*] или **NO CACHE**; определяет размер кэша, используемый для генерации значений элементов последовательности, что влияет на производительность. Если указано **NO CACHE**, тогда каждое обращение к последовательности предполагает обращение к внешней памяти. Если используется **CACHE целое**, тогда указанное количество значений элементов последовательности сохраняется в кэше и для их получения не требуется обращения к внешней памяти. Если, например, задано **CACHE 5**, то только при каждом пятом обращении к последовательности будет выполняться обращение к внешней памяти.

Работа с последовательностью в разных СУБД реализуется разными средствами.

Ниже приведены примеры использования оператора `CREATE SEQUENCE`:

```
CREATE SEQUENCE seq1
    START WITH 1
    INCREMENT BY 1
    NOCACHE;
```

Так, в СУБД **MS SQL Server** для того, чтобы получить очередное значение последовательности, используется специальное предложение:

`NEXT VALUE FOR имя_последовательности`

В СУБД **Oracle** для этих целей используются следующие предложения SQL:

`имя_последовательности.CURVAL` — получить текущее значение последовательности

`имя_последовательности.NEXTVAL` — получить очередное значение последовательности

Для удаления последовательности используется предложение

`DROP SEQUENCE имя_последовательности`

7.4.5. Язык управления доступом DCL (Data Control Language)

Для управления доступом к объектам СУБД используются команды языка управления доступом:

`GRANT` — предоставляет пользователю полномочия доступа к объекту базы данных или его содержимому;

`REVOKE` — отменяет предоставленные ранее полномочия.

Привилегии доступа могут быть предоставлены/отменены как на управление самим объектом базы данных, так и на управление содержимым объекта базы данных.

В общем случае полномочия доступа предоставляются с помощью следующего предложения:

`GRANT имя_привилегии ON имя_объекта TO имя_пользователя`
и отменяются с помощью следующего предложения:

`REVOKE имя_привилегии ON имя_объекта FROM имя_пользователя`

Здесь **`имя_привилегии`** — имена предложений SQL, которые могут или не могут выполняться пользователем; к ним относятся `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `EXECUTE`;

`имя_объекта` — объект базы данных, на который предоставляется или аннулируется доступ: таблица, представление, процедура, функция;

`имя_пользователя` — имя пользователя (имя схемы) базы данных, для которого предоставляются или аннулируются соответствующие полномочия доступа.

Рассмотрим некоторые примеры.

Пример 1. Предположим, необходимо предоставить доступ пользователю базы данных с именем `hr` на чтение и изменение таблицы `employee`:

Ниже приведены примеры использования оператора GRANT для предоставления доступа к таблице employee:

```
GRANT SELECT, UPDATE ON employee TO hr;
```

Пример 2. Предположим, необходимо отменить доступ пользователю базы данных с именем hr на изменение таблицы employee:

```
REVOKE UPDATE ON employee TO hr;
```

Вопросы

- Создайте две таблицы; в первой (родительской) таблице первичный ключ состоит из двух атрибутов; во второй (дочерней) таблице первичный ключ состоит из трех атрибутов: один — собственный атрибут таблицы, а два других — составной первичный ключ родительской таблицы.
- Создайте все таблицы предметной области «Прокат автотранспортных средств».
- Создайте таблицы с использованием ограничений ссылочной целостности на уровне колонки и таблицы с использованием ограничений ссылочной целостности на уровне таблицы.
- Придумайте пример, в котором ограничение допустимости значения может быть задано только как ограничение уровня таблицы, а не ограничение уровня колонки.
- В существующей таблице добавьте/удалите колонку, а также измените тип данных и ограничение обязательности значений.
- При наличии двух таблиц, одна из которых родительская, вторая — дочерняя, удалите родительскую таблицу. С какими проблемами можно столкнуться при удалении таблицы? Приведите обоснование этих проблем.
- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= 0)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= col2)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= col2)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= col2)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= col2)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

- Дано следующее предложение SQL:

```
CREATE TABLE tab(
    col1 INT NOT NULL PRIMARY KEY,
    col2 INT DEFAULT 0,
    col3 INT CHECK(col3 >= col2)
);
```

Укажите, содержит ли данное предложение ошибки. Если да, исправьте их.

10. Даны следующие предложения SQL:

```

CREATE TABLE tab1(
    tab1Id INT NOT NULL PRIMARY KEY,
    col2 INT NOT NULL UNIQUE,
    col3 INT CHECK(col3 >= 0)
);

CREATE TABLE tab2(
    tab2Id INT NOT NULL PRIMARY KEY,
    tab1Id INT NOT NULL REFERENCES tab1,
    col3 VARCHAR(20) NOT NULL
);

```

Укажите, содержит ли данные предложения ошибки. Если да, исправьте их.

11. Даны следующие предложения S1:

```
CREATE TABLE tab1(
    tab1Id INT NOT NULL PRIMARY KEY,
    col2 INT NOT NULL UNIQUE,
    col3 INT CHECK(col3 >= 0)
);

CREATE TABLE tab2(
    tab2Id INT NOT NULL PRIMARY KEY,
    tab1Id INT NOT NULL REFERENCES tab1 (col2),
    col3 VARCHAR(20) NOT NULL
);
```

Укажите, содержит ли данные предложения ошибки. Если да, исправьте их.

13. Данные следующие предложения SOI:

```

CREATE TABLE tab1(
    tab1Id INT NOT NULL PRIMARY KEY,
    col2 INT NOT NULL UNIQUE,
    col3 INT NOT NULL CHECK(col3 >= 0)
);
CREATE TABLE tab2(
    tab2Id INT NOT NULL PRIMARY KEY,
    tab1Id INT NOT NULL REFERENCES tab1(col3),
    col13 VARCHAR(20) NOT NULL
);

```

МІСТОВИЙ ПОДІЛІЛЬНИК ВІДНОВЛЕННЯ ПОДІЛІЛЬНОГО МІСТОВИХ ПОСІДІЛЬНИКІВ

7.5. ПОДМНОЖЕСТВО SQL ДЛЯ МАНИПУЛИРОВАНИЯ ДАННЫМИ

Подмножество SQL для манипулирования данными — DML — включает в свой состав предложения `INSERT`, `DELETE`, `UPDATE` и `SELECT`. Первые три предложения реализуют операции модификации данных; они позволяют добавлять новые строки в таблицы базы данных, удалять и модифицировать существующие строки. Последнее предложение (`SELECT`) — самое мощное предложение SQL, которое позволяет формировать различные запросы к базе данных; с одной стороны, оно реализует все возможности, определяемые реляционным исчислением с переменными-кортежами (а следовательно, и реляционной алгеброй), а с другой — предоставляет дополнительные возможности, такие как реализация арифметических вычислений и специальных агрегатных функций, применяемых к множеству значений. Надо отметить, что предложение `SELECT` используется не только как автономное предложение языка SQL, но и как часть других конструкций языка. В этих ситуациях на предложение `SELECT` мы будем ссылаться как на запрос или подзапрос в зависимости от того, в каких конструкциях оно используется.

В соответствии с этим сначала рассмотрим предложения модификации данных (в которых по необходимости будем использовать понятия и простейшие примеры запроса и (или) подзапроса), далее — общий синтаксис предложения SELECT и завершим данную тему рассмотрением разнообразных примеров, реализующих различные запросы к базе данных.

Здесь будут рассмотрены наиболее часто используемые возможности предложения **SELECT**. Полное описание предложения **SELECT** со всеми его возможностями и особенностями использования в конкретной СУБД можно найти в соответствующей дополнительной литературе и документации для каждой СУБД.

7.5.1. Предложение INSERT

Предложение INSERT позволяет вставлять в таблицу одну или несколько новых строк.

Существует несколько форм использования предложения **INSERT**.

Вставка в таблицу новых строк

Предложение `INSERT` имеет следующий вид:

Предложение INSERT имеет следующий вид:

INSERT INTO имя_таблицы [(имя_колонки, ...)] VALUES (значение, ...), (...), ...

168

напоминание, включая ограничения целостности, идентификаторы и т. д.).
Все ограничения целостности, определенные в таблице, должны быть определены в предложении CREATE TABLE.

имя_таблицы — должно указывать существующую таблицу базы данных, в которую вставляются новые значения (строки);

(имя_колонки, ...) — указывает имена колонок, для которых в данном предложении задаются значения. Каждое имя называет имя колонки в данной таблице. Имена колонок могут быть перечислены в любом порядке, и одно и то же имя колонки не должно указываться дважды. Если список имен колонок опущен, предполагается, что перечислены все имена колонок данной таблицы в том порядке, в каком эти колонки располагаются в таблице (порядок расположения колонок в таблице определяется порядком их перечисления в предложении CREATE TABLE). Если в списке имен колонок указаны не все колонки таблицы, для тех колонок, которые не указаны, при создании таблицы должны быть определены либо атрибут NULL, либо атрибут генерируемого значения (DEFAULT).

(значение, ...) — определяет список значений, вставляемых в одну строку. Если необходимо вставить несколько строк, соответствующие списки значений перечисляются через запятую.

Количество значений, указанных в каждом списке в одном предложении INSERT, должно соответствовать явлому или неявному списку имен колонок. Первое значение вставляется в первую колонку списка имен, второе — во вторую и т.д. Тип вставляемых значений должен соответствовать типу соответствующих им колонок.

Каждое значение, указанное в списке значений, должно быть представлено некоторым выражением или ключевыми словами NULL или DEFAULT.

В качестве выражения может быть использовано любое скалярное выражение, описанное ранее (см. п. 7.3.3).

NULL — определяет пустое значение; слово может быть указано только для колонки, для которой разрешены неопределенные значения.

DEFAULT — указывает, что должно быть использовано значение по умолчанию, заданное в предложении CREATE TABLE конструкцией DEFAULT.

Если при создании таблицы для нее были определены какие-либо ограничения целостности, при вставке в таблицу новых строк для каждой строки проверяются все ограничения. Если хотя бы одно ограничение будет нарушено, ни одна новая строка в таблицу не вставляется и операция завершается с ошибкой.

Примеры. Пусть таблица tab1 была создана следующим образом:

```
CREATE TABLE tab1 (
    id_t INT NOT NULL PRIMARY KEY,
    id_num1 INT DEFAULT 0,
    id_num2 INT
```

Ниже приведены примеры вставки в таблицу с использованием оператора `INSERT INTO`:

```
INSERT INTO tabl VALUES (12, 10, 3), (10, DEFAULT, NULL);
INSERT INTO tabl(id_t, id_num1) VALUES (5, 128), (15, 5);
```

Пример 1. Если в данную таблицу необходимо вставить несколько новых строк, предложение `INSERT` может быть записано следующими способами:

```
INSERT INTO tabl VALUES (12, 10, 3), (10, DEFAULT, NULL);
INSERT INTO tabl(id_t, id_num1) VALUES (5, 128), (15, 5);
```

Пример 2. При попытке выполнить следующее предложение:

```
INSERT INTO tabl VALUES (3, 10, 3), (1, DEFAULT, NULL),
(3, 3, 8);
```

будет получено сообщение о нарушении ограничения первичного ключа, и ни одна из указанных в предложении трех строк вставлена не будет.

Примечание. При вставке новых значений в таблицу может использоваться объект последовательность.

Пусть, например, были определены следующие объекты базы данных:

```
CREATE SEQUENCE sq1 START WITH 1 INCREMENT BY 1;
CREATE TABLE tabl (
    id_t INT NOT NULL PRIMARY KEY,
    id_num1 INT DEFAULT 0,
    id_num2 INT
```

В СУБД MS SQL Server для вставки нового значения из некоторой ранее созданной последовательности используется функция `NEXT VALUE FOR имя_последовательности`.

Тогда при выполнении следующего предложения:

```
INSERT INTO tabl VALUES (NEXT VALUE FOR sq1, 10, 3),
(NEXT VALUE FOR sq1, DEFAULT, NULL)
```

в таблицу будут вставлены две строки, в которых в колонке `id_t` будет вставлено значение, например, 1, во второй строке — значение 2.

Если же будет выполнено следующее предложение:

```
INSERT INTO tabl VALUES (NEXT VALUE FOR sq1, NEXT VALUE FOR sq1, 128)
```

тогда в таблицу `tabl` будет вставлена одна строка, в которой в колонки `id_t` и `id_num1` будет вставлено одно и то же значение — например 3.

В СУБД Oracle для вставки нового значения из некоторой ранее созданной последовательности используется конструкция вида: `имя_последовательности.NEXTVAL`. Добавление данных в таблицу `tabl` будет выглядеть следующим образом:

```
INSERT INTO tab1 VALUES (sql.NEXTVAL, 10, 3)
INSERT INTO tab1 VALUES (sql.NEXTVAL, DEFAULT, NULL)
INSERT INTO tab1 VALUES (sql.NEXTVAL, sql.NEXTVAL,
128)
```

Результаты вставки будут такими же, как и для СУБД MS SQL Server.

Примечание. В СУБД MS SQL Server, если какая-то колонка таблицы определена со свойством IDENTITY, эта колонка не должна указываться в списке имен колонок и для нее не должно задаваться значение в списке значений. Например, если таблица tab1 была создана так:

TAK.

GRE

```
CREATE TABLE tab1 (
    id_t INT NOT NULL,
    id_num1 INT DEFAULT 0,
    id_num2 INT
)
```

тогда вставка строк в таблицу выполняется следующим образом:

```
INSERT INTO tab1 VALUES (10, 3), (DEFAULT, NULL)
```

```
INSERT INTO tab1(id, num1) VALUES (128), (5)
```

В СУБД Oracle, если какая-то к

В СУБД Oracle, если какая-то колонка таблицы заполняется автоматически с использованием объектов триггера и последовательности (аналог свойства **IDENTITY** в **MS SQL Server**), эта колонка также не должна указываться в списке имен колонок и для нее не должно задаваться значение в списке значений.

В этом случае для ранее созданной таблицы `tab1` сначала создается триггер уровня строки (правила использования триггеров будут рассмотрены позднее — см. п. 7.7.7):

```
CREATE TRIGGER tab1_pk_tr  
BEFORE INSERT ON tab1  
FOR EACH ROW
```

```
BEGIN :new.id := sql.NEXTVAL;
```

END tab_pk_tr;

Автоинкрементная колонка заполняется значением последовательности с помощью созданного триггера. Вставка строк в таблицу выполняется следующим образом:

```
INSERT INTO tabl(id_num1, id_num2) VALUES (10, 3), (DE-FAULT, NULL)
```

В этом случае предложение `INSERT` имеет следующий вид:

INSERT INTO имя_таблицы [(имя_колонки, ...)] DEFAULT VALUES

Данная форма позволяет вставить в таблицу только одну строку эквивалента следующему предложению:

и эквивалентна следующему предложению:

Ниже приведены примеры использования оператора `INSERT`:

```
INSERT INTO имя_таблицы [(имя_колонки, ...)] VALUES (значение1, значение2, ...)
```

где `значение1, значение2, ...` — это или значения по умолчанию (`DEFAULT`), указанные в предложении `CREATE TABLE` для данных колонок, или значения `NULL`, если для соответствующих колонок не указаны ограничения обязательности и конструкция `DEFAULT`. Если для какой-либо колонки в таблице указано ограничение обязательности `NOT NULL` и отсутствует конструкция `DEFAULT`, предложение `INSERT` завершится с ошибкой.

Пример. Пусть таблица `tab2` была создана следующим образом:

```
CREATE TABLE tab2 (
    id_num1 INT DEFAULT 0,
    id_num2 INT
```

Тогда предложение

```
INSERT INTO tab2 DEFAULT VALUES
```

эквивалентно предложению

```
INSERT INTO tab2 VALUES (0, NULL)
```

или, что то же самое,

```
INSERT INTO tab2 VALUES (DEFAULT, NULL)
```

Примечание. В СУБД MS SQL Server, если какая-то колонка таблицы определена со свойством `IDENTITY`, и для всех остальных колонок либо определены значения по умолчанию, либо отсутствует ограничение обязательности, новая строка может быть вставлена следующим образом:

```
INSERT INTO tab1 DEFAULT VALUES
```

В СУБД Oracle данная конструкция не поддерживается.

Вставка в таблицу результата выполнения запроса

В этом случае предложение `INSERT` имеет следующий вид:

```
INSERT INTO имя_таблицы [(имя_колонки, ...)] запрос
```

В данной форме совокупность новых строк, которые должны быть вставлены, определяется в виде результата выполнения *запроса*. Надо отметить, что понятия запроса и подзапроса активно используются в SQL. Мы уже сталкивались со скалярным подзапросом в п. 7.3.3; здесь используется понятие запроса. Подробно правила написания запросов и подзапросов рассматриваются начиная с п. 7.5.4.

В данном варианте использования предложения `INSERT` в таблицу может быть вставлена одна строка, много строк или ни одной.

строки. Количество колонок в результате выполнения запроса должно совпадать с количеством колонок, указанным (явно или неявно) в списке имен. Значение первой колонки результата вставляется в первую колонку списка имен, второй колонки результата — во вторую колонку списка, и т.д.

строки. Количество колонок в результате выполнения запроса должно совпадать с количеством колонок, указанным (явно или неявно) в списке имен. Значение первой колонки результата вставляется в первую колонку списка имен, второй колонки результата — во вторую колонку списка, и т.д.

Пример 1. Пусть таблица `tab3` имеет такую же структуру, как и таблица `tab1`, определенная выше. Допустим, что для колонки `id_num2` таблицы `tab3` установлено ограничение `NOT NULL`. Необходимо вставить в таблицу `tab3` все строки из `tab1`, в которых в колонке `id_num2` есть определенное (не `NULL`) значение:

INSERT INTO tab3

```
SELECT * FROM tab1 WHERE id num2 IS NOT NULL -- 3anpoc
```

Если в запросе используется та же таблица, в которую вставляются новые строки, запрос выполняется перед вставкой в таблицу хотя бы одной новой строки и далее полученный результат добавляется в таблицу. Так, в следующем примере в таблице `tab3` будут про-дублированы все строки:

INSERT INTO tab3

```
SELECT * FROM tab3 -- 3appoc
```

Если при создании таблицы для нее были определены какие-либо ограничения целостности, при вставке в таблицу новых строк из запроса для каждой строки проверяются все ограничения. Если хотя бы одно ограничение будет нарушено, ни одна новая строка из запроса в таблицу не вставляется, и операция завершается с ошибкой.

Пример 2. Если в таблице `tab1` есть хотя бы одна строка, в которой значение колонки `id_num2` не установлено (т.е. `NULL`), предложение

```
INSERT INTO tab3 SELECT * FROM tab1
```

завершится с ошибкой и в таблицу tab3 не будет вставлена ни одна строка.

7.5.3 Предложение DELETE

Предложение **DELETE** позволяет удалять из таблицы все или некоторые строки.

В общем случае предложение DELETE имеет следующий вид:

DELETE FROM имя_таблицы [WHERE условие_поиска]

имя_таблицы — должно указывать существующую таблицу базы данных:

условие_поиска — записывается в соответствии с правилами, рассмотренными в п. 7.3.5, и определяет условие отбора удаляемых строк. Если конструкция WHERE опущена, удаляются все строки таблицы.

В условии поиска в конструкции WHERE можно ссылаться только на колонки таблицы, указанной в предложении DELETE (если условие поиска содержит подзапрос, в подзапросе могут использоваться другие таблицы и, соответственно, их колонки). Условие поиска применяется к каждой строке таблицы, и удаляются те строки, для которых результатом условия поиска является значение true. Если условие поиска содержит подзапрос, можно считать, что подзапрос будет выполняться каждый раз, когда условие поиска применяется к строке, и результаты подзапроса используются в условии поиска. Если подзапрос ссылается на таблицу, указанную в этом же предложении DELETE, или на зависимые таблицы по правилу удаления CASCADE или SET NULL, подзапрос выполняется полностью, прежде чем будет удалена хотя бы одна строка.

При удалении строк из родительской таблицы проверяются ограничения ссылочной целостности и правила удаления, определенные в дочерних таблицах. Если для какой-либо из дочерних таблиц определено (явно или по умолчанию) правило удаления NO ACTION, строки из родительской таблицы, на которые есть ссылки из дочерних таблиц, не удаляются, и предложение DELETE завершается с ошибкой. Если операция удаления не защищена правилом удаления NO ACTION, выбранные строки удаляются. Кроме того, оказывается воздействие на строки дочерних таблиц, ссылающиеся на удаляемые строки:

- колонкам внешнего ключа любых строк, зависящих по связям с правилом удаления SET NULL, присваивается значение NULL;
 - колонкам внешнего ключа любых строк, зависящих по связям с правилом удаления SET DEFAULT, присваивается значение по умолчанию (DEFAULT), определенное для данной колонки при создании таблицы;
 - любые строки, зависящие по связям с правилом удаления CASCADE, также удаляются, и правила удаления применяются, в свою очередь, и к этим строкам.

Если при выполнении предложения `DELETE`, удаляющего несколько строк, возникла ошибка, никакие изменения в данной таблице не происходят.

Пример 1. Удалить из упомянутой выше таблицы `tab1` строки, в которых значение колонки `id_num2` не установлено:

Пример 2. Удалить из таблицы tab2 все строки (т.е. опустошить таблицу):

DELETE FROM tab2

тельной литературе и документации по диалекту SQL соответствующей СУБД.

тельной литературе и документации по диалекту SQL соответствующей СУБД.

Пример 1. В приведенной выше таблице `tab1` для всех строк, в которых значение колонки `id_num2` не установлено, заменить это значение на 0:

```
UPDATE tab1  
SET id_num2 = 0  
WHERE id_num2 IS NULL;
```

Пример 2. Удвоить значение колонки `id_num1` в таблице `tab1` для всех строк, в которых значение этой колонки больше 0:

```
UPDATE tab1 SET id_num1 = id_num1 + id_num1 -- или id_num1 = 2 * id_num1 WHERE id_num1 > 0
```

Пример 3. Для всех строк таблицы `tab1`, в которых в двух колонках — `id_num1` и `id_num2` — указаны значения по умолчанию, переустановить эти значения на 15 и 10 соответственно:

```
UPDATE tab1 SET id_num1 = 15, id_num2 = 10 WHERE id_num1 = 0 AND id_num2 IS NULL;
```

7.5.4. Формирование запросов к базе данных

Ранее неоднократно использовалось понятие *запроса*. Введем теперь более точное определение этого понятия.

Запрос представляет собой некоторую форму обращения к базе данных, позволяющую получить из нее необходимые данные, удовлетворяющие некоторому критерию отбора. Для этих целей в языке SQL предусмотрено особое предложение — **SELECT**, которое включает в себя достаточно широкий спектр возможностей, позволяющий удовлетворить все потребности пользователей, работающих с базой данных.

Синтаксические правила написания запросов с помощью предложения `SELECT` зависят в том числе и от того, каким образом — самостоятельно или в составе других предложений SQL — используется предложение `SELECT`. Обычно понятие *запрос* определяет такую форму использования предложения SQL, которая допускается везде — и для выборки данных при формировании необходимых отчетов, и в составе других предложений SQL (в частности, в предложении `INSERT`, которое было рассмотрено выше). Если предложение `SELECT` используется в другом предложении `SELECT`, тогда говорят об использовании *подзапроса*, т.е. запроса, вложенного в другой запрос.

Ниже приведены примеры использования конструкции `WITH` в запросах к базе данных.

Если же предложение SELECT используется непосредственно для выборки данных, оно обладает дополнительными возможностями, использование которых недопустимо в других ситуациях. В этом случае мы будем говорить о расширенном запросе.

Таким образом, можно определить следующие понятия:

- **расширенный запрос** — извлечение из базы данных необходимой информации, удовлетворяющей некоторым заданным критериям;
- **запрос** — частный случай расширенного запроса, который может быть использован в составе других предложений SQL;
- **подзапрос** — частный случай запроса, когда запрос используется в составе другого запроса (подзапрос) или в записи выражений (скалярный подзапрос).

В общем случае расширенный запрос имеет следующий вид:

WITH общее_табличное_выражение

запрос

ORDER BY список_для_упорядочивания

В записи расширенного запроса конструкции `WITH` и `ORDER BY` не обязательны и любая из них (или они обе) может быть опущена.

Запрос имеет следующий вид:

предложение_SELECT

теоретико-множественная операция

предложение_SELECT

Использование теоретико-множественных операций не обязательно, и в простейшем случае запрос представляется единственным предложением `SELECT`.

Далее мы рассмотрим правила записи и использования основного запроса — предложение `SELECT` и теоретико-множественные операции; правила формирования расширенного запроса — конструкции `WITH` и `ORDER BY`. Для иллюстрации этих правил будет использован фрагмент базы данных поставок, созданный в п. 7.4.2. Текущее состояние базы данных приведено на рис. 7.3.

Завершим рассмотрение формирования запросов соответствующими примерами, использующими учебную базу данных проката автотранспортных средств.

Все приводимые далее примеры соответствуют диалекту стандарта SQL-92, который в основном поддерживается всеми реляционными СУБД. Однако, если при выполнении приводимых примеров будут обнаружены какие-либо ошибки или несоответствия с

<i>Поставщик - provider</i>			<i>Деталь - workpiece</i>		
<i>pid</i>	<i>pName</i>	<i>address</i>	<i>wid</i>	<i>wName</i>	<i>price</i>
<i>PK</i>	<i>имя</i>	<i>адрес</i>	<i>PK</i>	<i>название</i>	<i>стоимость</i>
P1	Иванов	Москва	W1	винт	18
P2	Петров	Орел	W2	болт	25
P3	Сидоров	Псков	W3	гайка	23
P4	Сергеев	Орел			
P5	Ильин	Ижевск			

<i>Поставка - delivery</i>		
<i>pid</i>	<i>wid</i>	<i>qty</i>
<i>FK1</i>	<i>FK2</i>	<i>количество</i>
P1	W1	150
P1	W2	100
P2	W1	200
P3	W1	150
P3	W2	200
P3	W3	180
P5	W2	140

Рис. 7.3. Текущее состояние базы данных поставок

представленными примерами, следует обратиться к документации по соответствующей СУБД.

7.5.5. Предложение SELECT

В общем случае предложение SELECT имеет следующий вид:

SELECT список_вывода

FROM источники

WHERE условие_отбора_строк

GROUP BY список_для_группирования

HAVING условие_отбора_групп

В результате выполнения запроса, представленного предложением SELECT, формируется результирующая таблица, вид и состав которой определяются следующим образом:

- конструкция SELECT определяет совокупность колонок результатирующей таблицы и их имен;
- конструкция FROM определяет, из каких таблиц базы данных извлекаются данные в результирующую таблицу;
- конструкция WHERE определяет дополнительные условия, накладываемые на строки данных, которые должны войти в результирующую таблицу;
- конструкция GROUP BY позволяет сгруппировать строки по определенному признаку для последующей обработки; определяет некоторую совокупность групп, каждая из которых состоит из нескольких строк данных;
- конструкция HAVING позволяет из всех групп выбрать некоторые группы, удовлетворяющие дополнительным условиям.

Примечание. В общем случае запрос к базе данных может возвращать большое количество строк. В зависимости от конкретных условий может возникнуть необходимость ограничить количество строк, выводимых в результате выполнения запроса. Конкретный способ задания ограничения на количество получаемых строк зависит от реализации.

Все конструкции, кроме `SELECT` и `FROM`, являются необязательными и могут быть опущены; если же все они или некоторые из них указываются, они должны быть записаны именно в таком порядке, в каком представлены в описании предложения `SELECT`.

Не вдаваясь в особенности работы СУБД, можно считать, что конструкции запроса обрабатываются последовательно, в некотором порядке; на каждом этапе формируется некоторая промежуточная результирующая таблица, которая используется в качестве исходной таблицы на следующем этапе. Порядок обработки запроса следующий.

1. **FROM** — выделяются таблицы и представления, из которых выбираются нужные данные.
 2. **WHERE** — из выбранных таблиц выбираются строки, удовлетворяющие указанному условию отбора строк.
 3. **GROUP BY** — выбранные строки (или все, если не указывается конструкция WHERE) объединяются в группы (группируются) по указанным признакам.
 4. **HAVING** — из созданных групп выделяются группы, удовлетворяющие условию отбора групп.
 5. **SELECT** — из выбранных групп в соответствии со списком вывода выделяются и обрабатываются необходимые данные, создавая строки и столбцы результирующей таблицы.

Рассмотрим особенности задания каждой конструкции предложения `SELECT`.

Конструкция зерката

Конструкция `SELECT` имеет следующий вид:

СВИДЕТЕЛИ

SELECT список_запроса определяет, какую информацию нужно получить из базы данных и имеет следующий вид:

[РЕДАКТОР] [1-1] **[РЕДАКТОР]** [1-2] **[РЕДАКТОР]** [1-3]

[DISTINCT] [ALL] *expression* [[AS] *alias*]

Список вывода определяет структуру результатирующей таблицы.

DISTINCT или **ALL**. Слово **DISTINCT** пред-

зультирующей таблице встречаются повторяющиеся строки, выво-

дить только одну (любую) строку из группы повторяющихся строк,

тогда как слово **ALL** предписывает выводить все строки. Если ни одно из этих слов не указано, по умолчанию предполагается, что задано **ALL**.

В качестве элемента списка вывода может быть указан символ ***** или использовано любое скалярное выражение, которое строится в соответствии с правилами записи выражений, рассмотренными ранее (см. п. 7.3.3), в том числе, и скалярный подзапрос, возвращающий единственное значение. Символ ***** означает, что нужно получить данные из всех колонок всех таблиц, указанных в запросе (в конструкции **FROM**). Для выборки необходимых данных из таблицы в записи выражения используются имена колонок таблиц. Каждое имя колонки должно однозначно идентифицировать колонку таблицы. Если в запросе используются несколько таблиц, имеющих одинаковые имена колонок, и в списке вывода необходимо указать имя такой колонки, оно должно быть уточнено именем таблицы: **имя_таблицы.имя_колонки** — получается так называемое **уточненное имя**.

Если в запросе используются несколько таблиц и из некоторой таблицы нужно получить все колонки, используется конструкция *имя_таблицы.**.

При формировании результирующей таблицы каждая колонка этой таблицы получает некоторое собственное имя. Конструкция AS позволяет явно указать желаемое имя колонки результирующей таблицы. Если эта конструкция не используется, имя колонки формируется системой.

Имена результирующих колонок формируются в соответствии со следующими правилами:

- если указана конструкция AS, именем результирующей колонки является имя, указанное в этой конструкции;
 - если конструкция AS не указана и результирующая колонка получается из колонки таблицы, тогда именем результирующей колонки является неуточненное имя колонки таблицы;
 - в остальных случаях результирующая колонка будет безымянной. Таким колонкам назначаются свои имена, зависящие от реализации.

Примечание. В СУБД MS SQL Server такие колонки именуются как (по column name).

В СУБД Oracle заголовок таких колонок содержит тот же текст выражения, который был указан в конструкции `SELECT`.

Во избежание таких зависимостей от реализации целесообразно в конструкции SELECT для вычисляемых колонок, в которых указаны выражения, использовать конструкцию AS.

Рассмотрим некоторые примеры решения задания конструкции `SELECT`.

Пример 1. Предложению

напоминание о том, что в конструкции SELECT можно использовать конструкцию DISTINCT для удаления из результата повторяющихся строк. Важно отметить, что DISTINCT не удаляет строки, а просто не включает в результат те строки, которые уже были в нем.

Строение конструкции SELECT

Все конструкции языка SQL начинаются с ключевого слова SELECT, за которым следует список полей, из которых будет формироваться результат.

Следующий вид:

SELECT [DISTINCT | ALL] ТОП (выражение) список_вывода

где выражение задает максимальное количество строк, которое должно быть выведено в качестве результата.

Например:

SELECT TOP(10) * FROM delivery

Кроме того, в конструкции SELECT может быть использована последовательность

SELECT NEXT VALUE FOR имя_последовательности...

В этом случае из указанной последовательности выбирается очередное значение, которое подставляется в список вывода. При этом, если в конструкции SELECT используется несколько обращений к одной и той же последовательности, например

SELECT NEXT VALUE FOR sq1, NEXT VALUE FOR sq1, NEXT VALUE FOR sq1, ...

все эти обращения вернут одно и то же значение.

Конструкция FROM

Конструкция FROM в общем случае может быть представлена следующим образом:

FROM элемент1, элемент2, ...

Каждый элемент может быть представлен в виде ссылки на некоторый объект данных, соединения таблиц или вложенного табличного выражения.

Ссылка на объект данных имеет следующий вид:

имя_объекта [[AS] корреляционное_имя]

В качестве имени объекта может быть использовано имя существующей таблицы базы данных, имя общего табличного выражения (см. п. 7.5.7) или имя представления (см. п. 7.5.8).

Корреляционное имя, если оно используется, заменяет (в пределах данного предложения SELECT) непосредственно предшествующее ему имя объекта. Если, например, корреляционное имя указано для таблицы, именно оно должно использоваться при записи уточненного имени колонки такой таблицы. Если одна и та же таблица указывается в конструкции FROM дважды, по крайней мере, после одного указания должно следовать корреляционное имя.

Рассмотрим некоторые примеры.

Пример 1. Предложение

SELECT * FROM provider

позволяет получить полное содержимое таблицы provider:

182

Ниже приведены примеры соединения таблиц:

pid	pName	address
P1	Иванов	Москва
P2	Петров	Орел
P3	Сидоров	Псков
P4	Сергеев	Орел
P5	Ильин	Ижевск

Пример 2. Предложение

SELECT p.* , w.* FROM provider p, workpiece w

реализует операцию декартова произведения двух таблиц — provider и workpiece; в предложении используются корреляционные имена для обеих таблиц:

pid	pName	address	wid	wName	price
P1	Иванов	Москва	W1	винт	18
P1	Иванов	Москва	W2	болт	25
P1	Иванов	Москва	W3	гайка	23
P2	Петров	Орел	W1	винт	18
P2	Петров	Орел	W2	болт	25
P2	Петров	Орел	W3	гайка	23
P3	Сидоров	Псков	W1	винт	18
P3	Сидоров	Псков	W2	болт	25
P3	Сидоров	Псков	W3	гайка	23
P4	Сергеев	Орел	W1	винт	18
P4	Сергеев	Орел	W2	болт	25
P4	Сергеев	Орел	W3	гайка	23
P5	Ильин	Ижевск	W1	винт	18
P5	Ильин	Ижевск	W2	болт	25
P5	Ильин	Ижевск	W3	гайка	23

Соединение таблиц

записывается следующим образом:

элемент1 [тип_соединения] JOIN элемент2 ON условие_соединения

Здесь каждый элемент также может быть представлен в виде ссылки на объект данных, соединения таблиц или вложенного табличного выражения.

В качестве типа соединения может быть указано внутреннее (INNER) или внешнее (OUTER) соединение. Если тип соединения не указан, по умолчанию принимается внутреннее соединение.

Для внешнего соединения должен быть обязательно указан его вид: левое (LEFT), правое (RIGHT) или полное (FULL) внешнее со-

Ниже приведены результаты всех операций соединения.

Пример 1. Внутреннее соединение

Условие соединения записывается аналогично условию поиска, за единение. При этом ключевое слово **OUTER** может быть опущено; так, указание **LEFT JOIN** эквивалентно **LEFT OUTER JOIN**.

Условие соединения записывается аналогично условию поиска, за следующими ограничениями:

- любая колонка, на которую ссылается выражение в условии соединения, должна быть колонкой одной из таблиц, указанных в соответствующей операции соединения; обычно используются колонки первичного и внешнего ключей из соединяемых таблиц;

- условие соединения не может содержать никаких подзапросов.

Соединение таблиц реализует соответствующую операцию соединения реляционной алгебры.

Примеры. Пусть даны две таблицы — t_1 и t_2 , имеющие следующее содержимое:

t_1		a	b	t_2		x	y
		A	11			A	21
		B	12			C	22
		C	13			D	23

Ниже приведены результаты всех операций соединения.

Пример 1. Внутреннее соединение —

```
SELECT * FROM t1 INNER JOIN t2 ON t1.a = t2.x
```

a	b	x	y
A	11	A	21
C	13	C	22

Пример 2. Левое внешнее соединение —

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.a = t2.x
```

a	b	x	y
A	11	A	21
B	12	-	-
C	13	C	22

Для наглядности здесь и далее отсутствующие значения для колонок таблицы отмечены символом «-».

Примечание. Представление отсутствующего (неустановленного) значения колонки зависит от реализации. Так, в СУБД **MS SQL Server** в таких случаях выводится слово **NUL**, в СУБД **Oracle** позиция остается пустой.

Пример 3. Правое внешнее соединение —

```
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.a = t2.x
```

184

Ниже приведены примеры полного внешнего соединения:

a	b	x	y
A	11	A	21
C	13	C	22
-	-	D	23

Пример 4. Полное внешнее соединение —

```
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.a = t2.x
```

Ниже приведены примеры полного внешнего соединения для таблицы t1 и t2:

a	b	x	y
A	11	A	21
B	12	-	-
C	13	C	22
-	-	D	23

Пример 5. Вернемся к базе данных поставок. Предложение

```
SELECT p.pn, w.wn, d.qty
FROM provider p JOIN delivery d ON p.pid = d.pid
JOIN workpiece w on d.wid = w.wid
```

вернет следующий результат:

rName	wName	qty
Иванов	винт	150
Иванов	болт	100
Петров	винт	200
Сидоров	винт	150
Сидоров	болт	200
Сидоров	гайка	180
Ильин	болт	140

В соответствии с определением элементов, используемых в соединении таблиц, соединение может быть вложено в другое соединение.

Несколько соединений обычно выполняются в порядке слева направо, основываясь на положении требуемого условия соединения. Для улучшения читабельности при использовании вложенных соединений рекомендуется использовать скобки. Например, конструкция

```
tb1 LEFT JOIN tb2 ON tb1.c1=tb2.c1
RIGHT JOIN tb3 LEFT JOIN tb4 ON tb3.c2=tb4.c2
ON tb1.c3=tb3.c3
```

эквивалентна следующей:

```
(tb1 LEFT JOIN tb2 ON tb1.c1=tb2.c1)
RIGHT JOIN
(tb3 LEFT JOIN tb4 ON tb3.c2=tb4.c2)
```


Ниже приведены примеры запросов к базе данных для получения информации о поставщиках и заказах.

Здесь имя результирующей таблицы — **t**, имя колонки в этой таблице — **part**; для такой формы подзапроса имя колонки должно быть указано обязательно.

```
(SELECT qty / 100 AS part FROM delivery) AS t
```

Здесь имя результирующей таблицы — **t**, имя колонки в этой таблице — **part**.

```
(SELECT qty / 100 AS part FROM delivery) AS t(cnt)
```

Здесь имя результирующей таблицы — **t**, имя колонки в этой таблице — **cnt**.

```
(SELECT qty / 100 FROM delivery) AS t
```

Для данного табличного выражения будет выведено сообщение об ошибке — колонка результирующей таблицы не имеет имени.

На колонки вложенного табличного выражения можно ссылаться из любых мест запроса с помощью корреляционного имени. В общем случае вложенное табличное выражение может быть указано в любой конструкции **FROM**.

Пример 1. Предложение

```
SELECT t.tn as "имя"
FROM (SELECT p.pName FROM provider p) AS t(tn)
```

вернет следующий результат:

имя

Иванов

Петров

Сидоров

Сергеев

Ильин

Пример 2. Предложение

```
SELECT p.pid, pName, qty
FROM provider p JOIN
(SELECT d.pid, d.qty FROM delivery d) AS t
ON p.pid = t.pid
```

вернет следующий результат:

pid	pName	qty
P1	Иванов	150
P1	Иванов	100
P2	Петров	200
P3	Сидоров	150
P3	Сидоров	200
P3	Сидоров	180
P5	Ильин	140

Вложенным табличным выражением не могут быть использованы имена, ссылающиеся на таблицы, использованные в основном запросе.

Конструкция WHERE

Конструкция WHERE задает условие отбора строк и имеет следующий вид:

WHERE условие_поиска

В результате обработки данной конструкции будут получены строки, для которых указанное условие_поиска истинно.

Условие поиска записывается в соответствии с правилами, рассмотренными выше (см. п. 7.3.5), с учетом следующих ограничений:

- в условии поиска каждое используемое имя колонки или уточненное имя должно уникально идентифицировать колонку исходной таблицы или вложенного табличного выражения, указанных в конструкции FROM;
- в условии поиска конструкции WHERE не могут быть использованы агрегатные функции.

С помощью конструкции WHERE можно реализовать внутреннее соединение. Так, запрос

```
SELECT * FROM provider p INNER JOIN delivery d
```

```
ON p.pid = d.pid
```

эквивалентен запросу

```
SELECT * FROM provider p, delivery d
```

```
WHERE p.pid = d.pid
```

Однако внешние соединения с помощью только подобной конструкции WHERE реализовать нельзя.

Пример 1. Предложение

```
SELECT * FROM delivery WHERE qty > 150
```

вернет следующий результат:

```
188
```

Ниже приведены примеры использования подзапросов в конструкции WHERE:

pid	wid	qty
P2	W1	200
P3	W2	200
P3	W3	180

Пример 2. Предложение

```
SELECT * FROM provider
WHERE pid IN (SELECT DISTINCT pid FROM delivery)
```

вернет следующий результат:

pid	pName	address
P1	Иванов	Москва
P2	Петров	Орел
P3	Сидоров	Псков
P5	Ильин	Ижевск

В данном предложении в условии поиска в конструкции WHERE использованы предикат IN и подзапрос.

Конструкция GROUP BY

Конструкция GROUP BY имеет следующий вид:

GROUP BY элемент [, элемент ...]

Конструкция GROUP BY включается в предложение SELECT в тех случаях, когда в список вывода конструкции SELECT включаются в том числе и выражения, содержащие агрегатные функции.

Элемент конструкции GROUP BY представляется в виде некоторого выражения группирования, содержащего имена колонок и используемого для определения группы строк. Каждое имя колонки, включенное в выражение группирования, должно однозначно идентифицировать колонку исходной таблицы.

Каждому элементу списка вывода, содержащему имена колонок вне агрегатных функций, должно соответствовать выражение группирования. В простейшем случае выражение группирования представляется именем одной колонки, причем это имя колонки может отсутствовать в списке вывода. Если в списке вывода в конструкции SELECT указан скалярный подзапрос, он может быть использован вместе с агрегатными функциями и не должен указываться в списке группирования (см. приведенные далее примеры написания запросов).

Пример. Требуется для каждого поставщика получить суммарное количество поставляемых им деталей. Тогда предложение:

```
SELECT pid, sum(qty) AS Total
FROM delivery
GROUP BY pid
```

вернет следующий результат:

вернет следующий результат:

	pid	Total
HARRY MANN - EDGECOME	P1	250
HARRY MANN - EDGECOME	P2	200
HARRY MANN - EDGECOME	P3	530
HARRY MANN - EDGECOME	P5	140

Если в списке вывода используется конструкция AS для задания имени колонки в результирующей таблице, в выражении группирования должно использоваться не это имя колонки, а само выражение, записанное в списке вывода, или имена колонок, участвующих в записи выражения. Например, если конструкция SELECT содержит следующий список вывода:

```
SELECT a * b AS total, count(id) AS count
```

тогда при использовании группирования следующие предложения SELECT завершатся с ошибкой:

```
SELECT a * b AS total, count(id) FROM t ...  
GROUP BY total
```

или

```
SELECT a * b AS total, count(id) FROM t ...  
GROUP BY 1
```

Чтобы получить требуемый результат, данное предложение должно быть дополнено тем,

SELECT a * b AS total, count(id) FROM t ... GROUP BY a * b

```
SELECT a * b AS total, count(id) FROM t ... GROUP BY a, b
```

Выражение группирования не может включать скалярный подзапрос. Чтобы выполнять группирование с использованием таких выражений, используются вложенные табличные выражения.

Результатом выполнения операции GROUP BY является множество групп строк. Каждая группа состоит из множества строк, имеющих одно и то же значение выражения группирования. При выполнении группирования все значения NULL из выражения группировки

Kognitivne ištaknute

Конструции НАУЧНС имеет следующий вид:

Конструкция HAVING и ORDER BY

HAVING условие_поиска

Конструкция HAVING определяет промежуточную таблицу результатов, которая состоит из групп, полученных на предыдущих этапах вычисления SELECT, для которых условие поиска истинно.

Ниже приведены примеры использования конструкции **HAVING**:

```

SELECT pid, sum(qty) AS Total
FROM delivery
GROUP BY pid
HAVING count(wid) > 1

```

В условии поиска конструкции HAVING допускается использование имен колонок, однозначно идентифицирующих каждую группу исходной таблицы, и агрегатных функций, применяемых к полученным группам.

Пример. Требуется получить суммарное количество поставляемых деталей только для тех поставщиков, которые поставляют несколько деталей. В этом случае предложение

```

SELECT pid, sum(qty) AS Total
FROM delivery
GROUP BY pid
HAVING count(wid) > 1

```

вернет следующий результат:

pid	Total
P1	250
P3	530

Допускается использование конструкции **HAVING** без конструкции **GROUP BY**; в этом случае считается, что в операции **HAVING** участвует только одна группа, и в списке вывода в конструкции **SELECT** могут быть указаны только агрегатные функции и константы.

7.5.6. Использование теоретико-множественных операций

Как упоминалось ранее, запрос в общем случае может быть представлен следующим образом:

предложение_SELECT

теоретико-множественная_операция

предложение_SELECT

**...
в качестве операций используются:
UNION или UNION ALL — для операции объединения;
EXCEPT — для операции вычитания;
INTERSECT — для операции пересечения.**

Указанные операции реализуют соответствующие теоретико-множественные операции реляционной алгебры. Ключевое слово **ALL** для операции **UNION** указывает, что в результат должны быть включены все строки независимо от того, встречаются дубликаты среди строк или нет. Если ключевое слово **ALL** не указано, из результата удаляются все дубликаты. Две строки считаются дубликатами, если каждое значение первой строки равно соответствующему значению

второй. При анализе дубликатов два значения NULL считаются равными.

второй. При анализе дубликатов два значения NULL считаются равными.

Примечание. В СУБД Oracle операция вычитания определяется ключевым словом **MINUS**.

Обозначим через $r1$ результирующую таблицу, полученную при выполнении первого предложения `SELECT`, и через $r2$ — результирующую таблицу, полученную при выполнении второго предложения `SELECT`. Количество колонок в таблицах $r1$ и $r2$ и типы данных для совпадающих по порядку следования колонок должны быть одинаковыми. Имена колонок в результирующей таблице определяются именами колонок из результирующей таблицы $r1$.

Так, например, если таблица `t1` содержит колонки с именами `a` и `b`, а таблица `t2` — колонки с именами `x` и `y`, то в результате выполнения любого из следующих запросов:

```
SELECT a, b FROM t1 UNION SELECT x, y FROM t2
SELECT a, b FROM t1 UNION SELECT * FROM t2
SELECT * FROM t1 UNION SELECT x, y FROM t2
SELECT * FROM t1 UNION SELECT * FROM t2
```

будет получена результирующая таблица, колонки которой имеют имена *a* и *b*. Такая же таблица будет получена, если вместо операции *UNION* используется любая другая теоретико-множественная операция.

Если же первое предложение SELECT имеет вид

```
SELECT a AS "First Name", b AS "Second Name" FROM t1
```

то независимо от того, как будет записано второе предложение SELECT, колонки результирующей таблицы будут иметь имена First Name и Second Name.

В табл. 7.7 приведены результаты вычисления выражений $r1$ операции $r2$ для всех теоретико-множественных операций.

Если в записи полного запроса используется несколько теоретико-множественных операций, они выполняются в порядке слева направо, причем операция `INTERSECT` имеет более высокий приоритет, чем операции `UNION` и `EXCEPT` (эти операции имеют одинаковый приоритет). Скобки могут изменить порядок вычисления операций.

Например, пусть имеются три таблицы — t1, t2 и t3

HARRY MINNIE-EGGCAKE	t1	a	b	t2	x	y	t3	c1	c2
	A	1		A	4		A	2	
	A	2		B	2		B	3	
	C	1		B	3		C	1	

Таблица 7.7

Результаты вычисления теоретико-множественных операций

Результаты базовых операций множественных операций					
		UNION ALL	UNION	EXCEPT / MINUS	INTERSECT
HARRY MAMA - EGBODUO	1	1	1	2	1
HARRY MINIM - EGBODUO	1	1	2	5	3
HARRY MINIM - EGBODUO	1	3	1		4
HARRY MINIM - EGBODUO	2	3	1	4	
HARRY MINIM - EGBODUO	2	3	1	5	
HARRY MINIM - EGBODUO	2	3	2		
HARRY MINIM - EGBODUO	3	4	2		
HARRY MINIM - EGBODUO	4		2		
HARRY MINIM - EGBODUO	4		3		
HARRY MINIM - EGBODUO	5		3		
HARRY MINIM - EGBODUO			3		
HARRY MINIM - EGBODUO			3		
HARRY MINIM - EGBODUO			3		
HARRY MINIM - EGBODUO			4		
HARRY MINIM - EGBODUO			4		
HARRY MINIM - EGBODUO			4		
HARRY MINIM - EGBODUO			5		

Тогда при выполнении запроса

```
SELECT * FROM t1 UNION SELECT * FROM t2 INTERSECT  
SELECT * FROM t3
```

будет получен следующий результат:

Задача №1		Задача №2		Задача №3	
Вариант	Решение	Вариант	Решение	Вариант	Решение
A	1	A	2	A	3
B	3	B	1	B	2
C	1	C	3	C	2

т.е. сначала выполняется операция `INTERSECT`, а затем — `UNION`.

При выполнении следующего запроса, в котором скобки явно определяют порядок вычислений:

(SELECT * FROM t1 UNION SELECT * FROM t2) INTERSECT

```
(SELECT * FROM t1 UNION SELECT * FROM t2) INTERSECT  
SELECT * FROM t3
```

SELECT FROM CS будет получен следующий результат:

будет получен следующий результат: Файл → Выход из приложения → Выход из приложения

a	b
A	2
B	3
C	1

Ниже приведены примеры использования конструкции ORDER BY в различных сценариях.

Примечание. Как уже упоминалось ранее, в СУБД MS SQL Server для ограничения количества выводимых строк в предложении SELECT используется конструкция TOP (*выражение*). Следует отметить, что эта конструкция влияет на результат выполнения только того предложения SELECT, в которое она включена, а не на общий результат запроса. Так, в результате выполнения следующего запроса:

```
SELECT TOP(5) * FROM t1 UNION ALL SELECT * FROM t2
```

ограничение вывода (5 строк) влияет только на первое предложение SELECT, и результат будет содержать 5 строк из таблицы t1 и все строки из таблицы t2.

7.5.7. Формирование расширенных запросов

Как упоминалось выше, расширенный запрос имеет следующий вид:

WITH общее_табличное_выражение

запрос

ORDER BY список_для_упорядочивания

Конструкции WITH и ORDER BY не обязательны и могут быть опущены. Рассмотрим эти конструкции подробнее.

Конструкция ORDER BY

Конструкция ORDER BY имеет следующий синтаксис:

ORDER BY ключ_сортировки [направление_сортировки], ...

В качестве *ключа_сортировки* могут быть указаны *простое имя колонки* или *простое целое*. *Простое имя колонки* определяет именованную колонку из списка вывода конструкции SELECT. Если элемент списка вывода задается с использованием конструкции AS *новое_имя*, тогда в качестве ключа сортировки может быть использовано это новое имя. *Простое целое* задает порядковый номер колонки результирующей таблицы начиная с 1.

Если, например, в предложении SELECT используется следующий список вывода:

```
SELECT col1 AS "New Name", ...
```

тогда для него конструкция ORDER BY может быть задана любым из следующих способов:

ORDER BY col1

ORDER BY "New name"

ORDER BY 1

В качестве *направления сортировки* может быть указано ключевое слово ASC или DESC. Значение ASC предписывает выполнять сорти-

Ниже приведены примеры сортировки:

```
SELECT * FROM employees ORDER BY hiredate DESC;
```

Сортировка в порядке убывания даты найма.

```
SELECT * FROM employees ORDER BY hiredate ASC;
```

Сортировка в порядке возрастания даты найма.

Сортировка в возрастющем порядке, DESC — в убывающем порядке. Если направление сортировки не указано, по умолчанию принимается ASC.

Конструкция `ORDER BY` определяет упорядочение строк результирующей таблицы. Если указана только одна спецификация сортировки (ключ сортировки с соответствующим направлением), строки упорядочиваются по значениям указанной спецификации. Если указано несколько спецификаций сортировки, строки упорядочиваются сначала по значениям первой спецификации, затем по значениям второй и т.д.

Упорядочение выполняется в соответствии с правилами сравнения. Значение `NULL` считается большим, чем любое другое значение. Если конструкция `ORDER BY` не может полностью упорядочить строки, строки с одинаковыми значениями всех указанных в ключах сортировки колонок выводятся в произвольном порядке.

Примечание. Как уже упоминалось ранее, в СУБД **MS SQL Server** для ограничения количества выводимых строк в предложении `SELECT` используется конструкция `TOP (выражение)`. Следует отметить, что эта конструкция влияет на результат выполнения предложения `SELECT` после того, как будет выполнена сортировка, заданная конструкцией `ORDER BY`. Так, запрос

```
SELECT TOP(1) qty FROM delivery ORDER BY 1 DESC
```

позволяет получить максимальное значение колонки `qty` у таблице `delivery`.

Конструкция `WITH`

Как упоминалось ранее, конструкция `WITH` используется при задании расширенных запросов и определяет *общее табличное выражение* — вычисляемую таблицу, которая может использоваться в основном запросе наряду с обычными таблицами базы данных. Конструкция `WITH` не включена в стандарт SQL-92, но практически все СУБД включают эту конструкцию в свое расширение SQL. Далее рассматриваются общие возможности использования данной конструкции; для уточнения особенностей ее использования в конкретном расширении языка SQL необходимо обратиться к соответствующей дополнительной литературе и документации.

В расширенном запросе ключевое слово `WITH` может появиться только один раз, но оно может определять несколько табличных выражений, перечисляемых через запятую:

`WITH общее_табличное_выражение [, ...]`

Общее табличное выражение задается следующим образом:

`имя_вычисляемой_таблицы [(имя_колонки [, ...])]`

AS

Основной запрос, определяемый в конструкции WITH, формируется в соответствии с правилами записи запросов, рассмотренными выше; в частности, в запросе могут, в том числе, использоваться и теоретико-множественные операции.

Следует обратить внимание на следующие особенности.

1. В общем табличном выражении допускается использование только основного запроса, т.е. вложенные конструкции WITH недопустимы.
 2. Основной запрос обязательно заключается в круглые скобки.
 3. Определенная в конструкции WITH вычисляемая таблица доступна только в следующем за конструкцией WITH запросе; за пределами запроса указанная вычисляемая таблица не определена.

Формат вычисляемой таблицы определяется основным запросом, указанным после слова AS: вычисляемая таблица будет иметь столько колонок, сколько элементов указано в соответствующей конструкции SELECT. Имена колонок определяются либо явно после имени вычисляемой таблицы, либо, если имена колонок вычисляемой таблицы не указаны, списком вывода конструкции SELECT основного запроса. При этом неименованные колонки в основном запросе не допускаются.

Следующие два примера использования конструкции WITH являются корректными и определяют вычисляемую таблицу с колонками `wid` и `total`, которая затем используется в предложении `SELECT`:

- ```
a) WITH derivedTable AS (SELECT wid, sum(qty) AS total FROM delivery GROUP BY wid)
SELECT * FROM derivedTable
b) WITH derivedTable(wid, total) AS (SELECT wid, sum(qty) FROM delivery GROUP BY wid)
SELECT * FROM derivedTable
```

При выполнении любого из двух приведенных расширенных запросов будет получен следующий результат:

| wid | total |
|-----|-------|
| W1  | 500   |
| W2  | 440   |
| W3  | 180   |

**Следующее использование конструкции WITH:**

Следующее использование конструкции WITH:

WITH derivedTable

AS

```
(SELECT wid, sum(qty) FROM delivery GROUP BY wid)
SELECT * FROM derivedTable
```

предлагают сообщение об ошибке

приведет к сообщению об ошибке — не определено имя для второй колонки.

Как упоминалось выше, конструкция **WITH** позволяет определять несколько табличных выражений. Рассмотрим следующий пример.

Пусть имеется таблица, в которой хранятся результаты проведения соревнований, например, результаты матчей по футболу. Каждая команда участвует, по крайней мере, в двух матчах — в роли хозяина и в роли гостя. Таблица (назовем ее *game*) имеет следующий вид:

| Команда – хозяин встречи | Команда – гость | Дата проведения матча | Забито хозяином | Пропущено хозяином |
|--------------------------|-----------------|-----------------------|-----------------|--------------------|
| cowner                   | cguest          | matchDate             | scored          | conceded           |
| cmd1                     | cmd2            | 12.07.2014            | 2               | 1                  |
| cmd2                     | cmd1            | 25.07.2014            | 3               | 2                  |

В первом матче от 12 июля команда cmd1 победила команду cmd2 со счетом 2:1, тогда как в ответном матче от 25 июля команда cmd1 проиграла команде cmd2 со счетом 2:3. Соответственно, каждая из двух команд провела по две встречи и имеет по одной победе. Пусть обязательно выполняется условие, что в таблице нет записей, когда какая-либо команда не участвует в матче в роли хозяина или в роли гостя.

Пусть нужно сформировать общий итог всех встреч в виде

Команда | Общее количество матчей | Количество побед

Требуемый результат может быть получен с использованием двух табличных выражений. Первое (`tab1`) вычисляет для каждой команды количество матчей и побед в роли хозяина встречи, второе (`tab2`) — количество матчей и побед в роли гостя; с учетом указанных выше ограничений оба табличных выражения будут иметь один и тот же состав команд:

WITTH

*= — первое табличное выражение*

tab1(cmd match win) AS

(SELECT count(cowner) AS

```
 sum(CASE WHEN scored > conceded THEN 1 ELSE 0 END)
```

sum(CASE WHEN scored > conceded THEN 1 ELSE 0 END) AS Conceded\_Wins FROM game GROUP BY souper )

FROM game GROUP BY counter );

**— второе табличное выражение**

```
tab2(cmd, match, win) AS
 (SELECT cguest, count(cguest),
 sum(CASE WHEN conceded > scored THEN 1 ELSE 0 END)
 FROM game GROUP BY cguest)
```

**— основной запрос**

```
SELECT tab1.cmd as command,
 tab1.match + tab2.match as games,
 tab1.win + tab2.win as win
FROM tab1 JOIN tab2 ON tab1.cmd = tab2.cmd
```

Здесь следует обратить внимание на то, что в табличных выражениях в качестве аргумента агрегатной функции (в данном примере это функция sum) используется выражение CASE, которое для каждой строки таблицы (т.е. каждой игры) возвращает одно из двух значений: 1, если команда победила в данной игре, и 0, если команда проиграла или свела встречу к ничьей.

Если в конструкции WITH определено несколько общих табличных выражений, тогда в табличном выражении могут использоваться определенные ранее в этой же конструкции WITH другие табличные выражения.

Вернемся к базе данных поставок. Если требуется найти поставщика, поставившего максимальное суммарное количество деталей, требуемый результат может быть получен разными способами, например так:

```

a) WITH
tab1(pid, qty) AS
 (SELECT pid, sum(qty) FROM delivery GROUP BY pid)
SELECT pid, qty AS maxqty FROM tab1
WHERE qty = (SELECT max(qty) FROM tab1)

```

Здесь использовано одно общее табличное выражение, в котором для каждого поставщика вычисляется суммарное количество поставляемых им товаров, а требуемый результат формируется в запросе.

```

6) WITH
tab1(pid, qty) AS
 (SELECT pid, sum(qty) FROM delivery GROUP BY pid),
tab2(pid, maxqty) AS
 (SELECT pid, qty FROM tab1 WHERE qty = (SELECT
max(qty) FROM tab1))
SELECT * FROM tab2

```

В этом варианте используются два табличных выражения; в первом (с именем `tab1`) для каждого поставщика вычисляется суммарное количество поставляемых им товаров, а во втором (с именем

**tab2**) используется первое (с именем **tab1**) и формируется необходимый результат; сам запрос всего лишь выводит содержимое сформированной вычисляемой таблицы **tab2**.

таб2), используется первое (с именем таб1), и формируется необходимый результат; сам запрос всего лишь выводит содержимое сформированной вычисляемой таблицы таб2.

Оба запроса возвращают один и тот же результат:

| pid | maxqty |
|-----|--------|
| P3  | 530    |

Допускается рекурсивное табличное выражение, в котором используется это же самое табличное выражение. Основной запрос в рекурсивном табличном выражении состоит из двух частей — основной и рекурсивной, объединенных операцией UNION ALL:

## и рекурсивной основная часть

## Основы чи- HINTON M.L.

#### **UNION ALL**

Основная часть выполняет начальную инициализацию вычисляемой таблицы и не должна содержать ссылку на определяемое табличное выражение.

Рекурсивная часть формируется в соответствии со следующими правилами:

- в конструкции FROM предложения SELECT рекурсивной части допускается только одна ссылка на определяемое табличное выражение;
  - в предложении SELECT рекурсивной части не допускаются конструкции DISTINCT, GROUP BY, HAVING, агрегатные функции;
  - в предложении SELECT рекурсивной части не допускается использование внешнего соединения и подзапросов.

Схема формирования вычисляемой таблицы в рекурсивном табличном выражении следующая.

Сначала выполняется основная часть рекурсивного табличного выражения, которая формирует начальное состояние вычисляемой таблицы. Далее начинает выполняться рекурсивная часть, которая организует цикл. Выбирается первая строка вычисляемой таблицы, подставляется в запрос рекурсивной части, и все выбранные строки добавляются в вычисляемую таблицу. Указанные действия выполняются для второй строки вычисляемой таблицы, третьей и т.д. до тех пор, пока не будут обработаны все строки вычисляемой таблицы, включая и те, которые добавляются при выполнении запроса рекурсивной части.

Рассмотрим пример

Пусть имеется следующая таблица `tab`:

|                        | first | second |
|------------------------|-------|--------|
| HARRY MAMA- EDGAR WINE | 1     | 2      |
| HARRY MAMA- EDGAR WINE | 1     | 3      |

Ниже приведен запрос, использующий рекурсивное табличное выражение:

```

WITH
v1(a, b) AS
(
 SELECT first, second FROM tab -- основная часть
 UNION ALL
 SELECT first, b FROM tab, v1 WHERE v1.a < tab.first
 -- рекурсивная часть
 SELECT * FROM v1 ORDER BY a, b
)

```

Выполняется следующий расширенный запрос, использующий рекурсивное табличное выражение:

```

WITH
v1(a, b) AS
(
 SELECT first, second FROM tab -- основная часть
 UNION ALL
 SELECT first, b FROM tab, v1 WHERE v1.a < tab.first
 -- рекурсивная часть
 SELECT * FROM v1 ORDER BY a, b
)

```

При выполнении основной части будет сформировано начальное состояние вычисляемой таблицы v1:

| v1 | a | b | tab | first | second |
|----|---|---|-----|-------|--------|
|    | 1 | 2 |     | 1     | 2      |
|    | 1 | 3 |     | 1     | 3      |
|    | 2 | 2 |     | 2     | 2      |
|    | 3 | 1 |     | 3     | 1      |

Далее начинает работать рекурсивная часть общего табличного выражения. Выбирается первая строка таблицы v1 (строка  $<1, 2>$ ) и для нее из таблицы tab выбираются те строки, в которых в первой колонке (first) находится значение, превышающее значение первой колонки (a) выбранной строки. Данному условию удовлетворяют две последние строки таблицы tab (строки  $<2, 2>$  и  $<3, 1>$ ), и в вычисляемую таблицу будут добавлены две новые строки со значениями first (из tab) и b (из v1) — строки  $<2, 2>$  и  $<3, 1>$ :

| v1 | a | b | tab | first | second |
|----|---|---|-----|-------|--------|
|    | 1 | 2 |     | 2     | 2      |
|    | 1 | 3 |     | 3     | 1      |
|    | 2 | 2 |     |       |        |
|    | 3 | 1 |     |       |        |

результат обработки первой строки вычисляемой таблицы в рекурсивной части рекурсивного табличного выражения

Далее те же действия выполняются для второй строки таблицы v1 (строка  $<1, 3>$ ), в результате будут добавлены еще две строки — строки  $<2, 3>$  и  $<3, 3>$ :

200

| $v1$ | $a$ | $b$ | $tab$ | $first$ | $second$ |
|------|-----|-----|-------|---------|----------|
|      | 1   | 2   |       | 2       | 2        |
|      | 1   | 3   |       | 3       | 1        |
|      | 2   | 2   |       |         |          |
|      | 3   | 1   |       |         |          |
|      | 2   | 2   |       |         |          |
|      | 3   | 2   |       |         |          |
|      | 2   | 3   |       |         |          |
|      | 3   | 3   |       |         |          |

*результат обработки второй строки вычисляемой таблицы в рекурсивной части*

Для третьей строки вычисляемой таблицы условию отбора удовлетворяет только третья строка таблицы `tab`, и в таблицу `v1` будет добавлена только одна строка:

Для четвертой строки таблицы `v1` в таблице `tab` подходящих данных нет, и в таблицу `v1` ничего не будет добавлено. Для пятой строки добавится еще одна строчка:

| v1 | a | b | tab | first | second |
|----|---|---|-----|-------|--------|
|    | 1 | 2 |     | 3     | 1      |
|    | 1 | 3 |     |       |        |
|    | 2 | 2 |     |       |        |
|    | 3 | 1 |     |       |        |
|    | 2 | 2 |     |       |        |
|    | 3 | 2 |     |       |        |
|    | 2 | 3 |     |       |        |
|    | 3 | 3 |     |       |        |
|    | 3 | 2 |     |       |        |
|    | 3 | 2 |     |       |        |

*результат обработки пятой строки вычисляемой таблицы*

**Шестая строка опять не приведет к изменению вычисляемой таб-**

Шестая строка опять не приведет к изменению вычисляемой таблицы, а седьмая добавит в нее новую строку:

| v1                                 | a | b | tab | first | second |
|------------------------------------|---|---|-----|-------|--------|
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 1 | 2 |     | 3     | 1      |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 1 | 3 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 2 | 2 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 3 | 1 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 2 | 2 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 3 | 2 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 2 | 3 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 3 | 3 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 3 | 2 |     |       |        |
| HARRY MINNIE-EDOGOWIE-HARRY MINNIE | 3 | 3 |     |       |        |

Все остальные строки не приведут к добавлению в вычисляемую таблицу новых данных, и на этом выполнение рекурсивного табличного выражения закончится. В итоге расширенный запрос приведет к формированию следующего результата:

При написании рекурсивной части рекурсивных табличных выражений нужно внимательно следить за тем, чтобы рекурсия не была бесконечной. Так, в приведенном выше примере будет получена бесконечная рекурсия, если в конструкции WHERE рекурсивной части использовать условие

vl.a = tab first

#### 7.5.8. Создание представлений — CREATE VIEW

**Представление — это объект базы данных, созданный на основе запросов к одной или нескольким таблицам и (или) другим представлениям.**

Представления используются в системах баз данных достаточно активно. Это объясняется следующими причинами.

- Ограничение доступа к данным** — представления позволяют ограничивать конкретным пользователям или группе пользователей доступ к данным как на уровне строк, так и на уровне колонок таблиц, участвующих в запросе, на основе которого сформировано представление.
  - Обеспечение независимости от данных** — представления позволяют сохранять неизменными свою структуру, даже если изменяется структура таблиц и связи между таблицами (в той их части, которая не участвует в создании представления), на которых построено представление. Например, если в таблицы, на которых построено представление, добавляются новые колонки или из этих таблиц удаляются существующие колонки, не участвующие в создании представления, такие изменения таблиц не повлияют на использование уже созданных представлений.
  - Упрощенное представление сложных запросов** — представления позволяют скрыть в себе запросы со сложной логической структурой и большим объемом сложно читаемого кода.
  - Различная интерпретация одних и тех же данных** — представления позволяют по-разному интерпретировать одни и те же данные, размещенные в одном и том же источнике.

Для создания представлений используется предложение **CREATE VIEW**, которое имеет следующий вид:

`CREATE VIEW имя_представления [(имя_колонки...)] AS запрос`

Здесь *имя\_представления* — именует представление. Имя не должно совпадать с именами существующих в базе данных таблиц и других представлений;

*имя\_колонки* — именует колонку представления. Если указывается список имен колонок, он должен содержать столько имен, сколько колонок определено в результирующей таблице запроса. Каждое *имя\_колонки* должно быть уникальным и неугочненным. Если список имен колонок не указан, колонки представления наследуют имена колонок результирующей таблицы запроса. Список имен колонок должен быть указан, если результирующая таблица запроса имеет несколько колонок с одинаковыми именами или неименованные колонки:

As — указывает начало определения представления;

**запрос** — определяет представление. Запрос выполняется тогда, когда данное представление используется в других запросах. В любой момент времени представление содержит строки, получающиеся в результате выполнения предложения `SELECT`.

Представление и общее табличное выражение имеют похожий синтаксис, но и существенные отличия, которые заключаются в следующем.

1. Общее табличное выражение существует и может использоваться только во время выполнения расширенного запроса, за пределами этого запроса оно не определено. Представление существует как самостоятельный объект базы данных, который можно использовать в любых запросах. Если представление более не требуется, его можно удалить с помощью предложения

**DROP VIEW** имя представления

2. В запросе, определяющем представление, может быть использована конструкция `WITH`.

Рассмотрим примеры для таблицы game, которая использовалась для иллюстрации встроенных табличных выражений.

Пример 1. Для формирования общего итога встреч можно использовать два представления: для команды — хозяина встречи и для команды — гостя (действуют те же ограничения на правила проведения матчей):

```

CREATE VIEW v1(cmd, match, win) AS
SELECT cowner, count(cowner),
 sum(CASE WHEN scored > conceded THEN 1 ELSE 0 END)
FROM game GROUP BY cowner
CREATE VIEW v2(cmd, match, win) AS
SELECT cguest, count(cguest),
 sum(CASE WHEN conceded > scored THEN 1 ELSE 0 END)

```

FROM game GROUP BY cguest

```

 SELECT v1.cmd as command,
 v1.match + v2.match as games,
 v1.win + v2.win as win
 FROM v1 JOIN v2 ON v1.cmd = v2.cmd

```

Именно в этот момент времени, когда выполняется основной запрос, выполняются и запросы, указанные в определении представлений.

Если эти два представления не нужны, их следует удалить:

**DROP VIEW v1**

Пример 2. Можно создать представление, в котором используется конструкция WITH:

```
CREATE VIEW v1(command, games, win) AS
WITH
tab1(cmd, match, win) AS
(SELECT cowner, count(cowner),
sum(CASE WHEN scored > conceded THEN 1 ELSE 0 END)
FROM game GROUP BY cowner),
tab2(cmd, match, win) AS
(SELECT cguest, count(cguest),
sum(CASE WHEN conceded > scored THEN 1 ELSE 0 END)
FROM game GROUP BY cguest)
SELECT tab1.cmd, tab1.match + tab2.match, tab1.win +
tab2.win
FROM tab1 JOIN tab2 ON tab1.cmd = tab2.cmd
```

#### Использование представления:

**Неслужебное представление:**  
**SELECT \* FROM v1**

Наконец, удаление представления:

## Вопросы

1. Заполните все таблицы предметной области «Прокат автотранспортных средств».
  2. Удалите из таблицы «Транспортное средство» все строки, в которых значение атрибута *Марка* равно «BMW». С какими проблемами можно столкнуться при удалении строк? Приведите обоснование этих проблем.
  3. Измените текущее значение колонки «Код клиента» в таблице «Договор аренды» на значение, равное 987654321. С какими проблемами можно столкнуться при изменении данных? Приведите обоснование этих проблем.
  4. Используя данные из базы данных проката автотранспортных средств, приведите результаты выполнения следующих предложений SQL:

результаты выполнения следующих предложений SQL:

a) `SELECT * FROM officeRental WHERE officeId NOT IN (SELECT retOffice FROM rentalContract);`

б) `SELECT cl.* FROM client cl JOIN rentalContract rc ON cl.clientId = rc.clientId WHERE undo IS NULL;`

в) `SELECT v.brand, v.model, count(v.vehicleId) FROM vehicle v LEFT JOIN rentalContracr rc ON v.vehicleId = rc.vehicleId GROUP BY v.brand, v.model;`

5. Напишите следующие запросы:

- а) получить информацию об офисах проката, в которых не был сдан в аренду ни один автомобиль марки «BMW»;**

- б) получить информацию об автомобилях, которые сдавались в аренду и возвращались только в разных офисах;
  - в) получить информацию о клиентах, которые возвращали автомобиль с нарушением сроков аренды;
  - г) получить информацию об офисах проката, в которые возвращались более одного автомобиля, при условии, что возвращаемый автомобиль был взят в аренду в этом же офисе;
  - д) для всех автотранспортных средств получить их марку и модель, а также доход от их аренды в рублях;
  - е) для каждого клиента получить количество оформленных им договоров аренды автотранспортных средств и общую продолжительность аренды в днях.



## 7.6. ПРИМЕРЫ НАПИСАНИЯ ЗАПРОСОВ



## 7.7. РЕАЛИЗАЦИЯ ПРОЦЕДУРНОЙ ЛОГИКИ

## Вопросы

1. Каково назначение хранимых процедур?
  2. Каким образом организуется передача исходных данных в хранимые процедуры?  
Укажите типы параметров процедуры.
  3. Каким образом осуществляется вызов хранимых процедур?
  4. Как реализуется обработка ошибок в хранимых процедурах?
  5. Укажите назначение курсоров, основные этапы создания и использования курсора в хранимой процедуре.
  6. Укажите назначение функций, правила использования функций.
  7. Что такое триггер? С какой целью используются триггеры?
  8. Укажите основные отличия между триггерами и хранимыми процедурами.

8. Укажите основные отличия между триггерами и хранимыми процедурами.

ГЛАВА 8

## УПРАВЛЕНИЕ ПАРАЛЛЕЛИЗМОМ В СУБД

## 8.1. ПОНЯТИЕ ТРАНЗАКЦИИ

**Архитектура современных СУБД предполагает, что в системе одновременно могут работать много клиентов. Каждый клиент должен иметь возможность работать независимо от того, кто еще работает в системе, т.е. при многопользовательском режиме работы клиентов должна быть обеспечена индивидуальность их работы.**

Есть проблемы, присущие коллективному доступу; они должны быть разрешены на уровне СУБД, но и разработчики приложения должны иметь представление об этих проблемах. Каждая конкретная СУБД в общем случае использует собственные механизмы управления коллективным доступом; тем не менее есть общие понятия, присущие разным СУБД, о которых следует иметь определенное представление.

**Определение.** Транзакция — это логическая единица работы, которая лежит в основе проблемы параллелизма.

**Главная задача управления параллелизмом заключается в следующем:**

- или все изменения, сделанные транзакцией, фиксируются и де-

- или все изменения, сделанные транзакцией, фиксируются и делаются постоянными (успешное завершение транзакции);
  - или транзакция никак не влияет на состояние базы данных (ошибки при выполнении транзакции, откат транзакции).

Транзакция имеет начало; оно может определяться неявно, например когда программа подсоединяется к БД, или явно.

Транзакция имеет явный конец, который отмечается предложениями SQL **COMMIT** или **ROLLBACK**. Если транзакция завершается успешно, она сообщает об этом серверу базы данных с помощью **COMMIT** — зафиксировать изменения. После этого все изменения в базе данных, сделанные данной транзакцией, становятся постоянными. Если в процессе выполнения транзакции обнаруживается какая-либо ошибка, транзакция завершается неуспешно и сообщает об этом с помощью **ROLLBACK**. В этом случае все изменения базы данных, сделанные данной транзакцией, аннулируются — выполняется откат транзакции (рис. 8.1).

После выполнения предложения COMMIT или ROLLBACK может начаться новая транзакция.

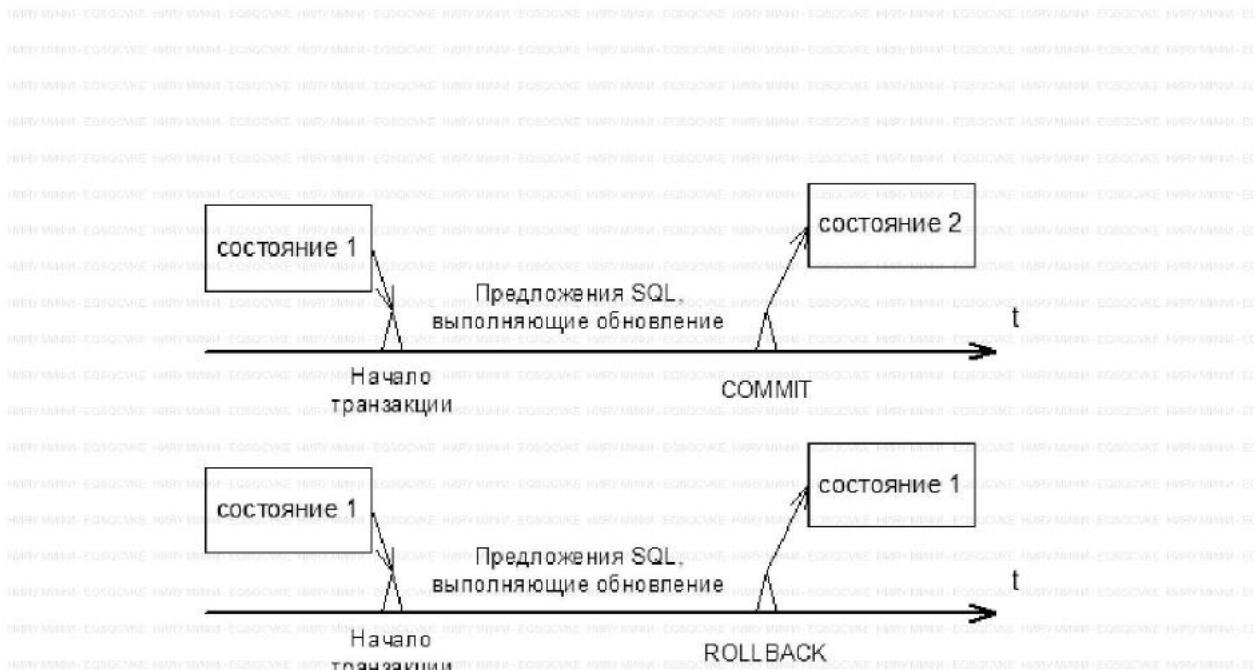


Рис. 8.1. Завершение транзакций

**Примечание.** В СУБД MS SQL Server существует несколько способов определения транзакций.

В обычных условиях СУБД работает в так называемом режиме автocomмита (auto commit); в этом режиме каждое предложение SQL представляет собой транзакцию, т.е. когда выполняется любое предложение SQL — INSERT, DELETE или UPDATE, транзакция начинается и сразу заканчивается, когда заканчивается выполнение данного предложения; при этом, если предложение SQL выполнилось успешно, транзакция заканчивается по COMMIT, если же были обнаружены какие-либо ошибки — по ROLLBACK. В этом режиме, если требуется определить транзакцию, содержащую несколько предложений SQL, нужно явно указать начало транзакции — с помощью предложения BEGIN TRANSACTION и конец транзакции — с помощью соответствующих предложений COMMIT или ROLLBACK.

Кроме этого, СУБД может работать в режиме неявного определения транзакции. Этот режим должен быть установлен с помощью специального предложения

```
SET IMPLICIT_TRANSACTIONS ON;
```

В этом режиме новая транзакция начинается (неявно), когда выполняется первое предложение SQL после установки данного режима или после завершившейся в данном режиме транзакции (т.е. предложение BEGIN TRANSACTION не требуется), а заканчивается при выполнении соответствующих предложений COMMIT или ROLLBACK.

В СУБД Oracle все выполняемые SQL-выражения предполагают начало транзакции по умолчанию.

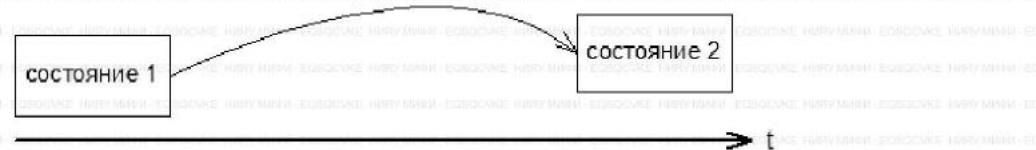
Очевидно, что транзакции не должны влиять друг на друга. Предполагается, что каждая транзакция автономно управляет разделяемой базой данных.

Введем определение транзакции.

Транзакция — это логическая единица работы, удовлетворяющая следующим четырем свойствам, получившим название АСИД:

- атомарность (A) — в транзакции выполняется все или ничего; транзакция не может выполниться частично;
  - согласованность (C) — транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние;
  - изолированность (I) — каждая транзакция выполняется независимо от других транзакций;
  - долговременность (D) — если изменения состояния базы данных, сделанные транзакцией, зафиксированы, они будут сохранены, даже если потом произойдет сбой.

Таким образом, транзакция переводит базу данных из одного целостного состояния в другое, причем в промежуточных точках база данных может находиться в несогласованном состоянии (рис. 8.2).



**Рис. 8.2.** Изменение состояния БД

СУБД гарантирует, что отдельные команды SQL сами по себе атомарны (т.е. выполняются полностью или не выполняются вовсе).

Транзакция в результате своей работы воздействует на состояние базы данных двумя способами:

- явно — выполняя команды подмножества DDL INSERT, DELETE, UPDATE;
  - неявно — возможно влияние на работу других транзакций, когда эти транзакции считывают данные, измененные текущей транзакцией.

При явном воздействии результаты работы транзакции могут быть аннулированы с помощью ROLLBACK.

При неявном воздействии может возникнуть ситуация, при которой некоторая транзакция, например 1, считывает данные, изменившиеся какой-то другой транзакцией, например 2, после чего транзакция 2 аннулирует сделанные ею изменения. В результате транзакция 1 имеет данные, не соответствующие целостному состоянию

иные минимумы. Единственное различие между первичной и вторичной транзакциями в том, что первичная транзакция имеет базу данных, на которую она влияет, а вторичная транзакция не имеет. Устранение такого вторичного влияния реализуется на основе механизма блокировок.

## 8.2. ПРОБЛЕМЫ ПАРАЛЛЕЛИЗМА

Если транзакции не удовлетворяют свойствам АСИД, при параллельной работе таких транзакций возможно возникновение следующих проблем:

- потеря результатов обновления;
- незафиксированные зависимости;
- несовместимый анализ.

Рассмотрим эти проблемы подробнее.

### 8.2.1. Потеря результатов обновления

Пусть выполняются параллельно две транзакции, каждая из которых выполняет извлечение некоторой записи ( $P$ ), ее анализ и последующее обновление; между извлечением и обновлением могут выполняться какие-либо другие вычисления.

Порядок выполнения транзакций при их параллельной работе может быть таким, какой представлен на рис. 8.3.

#### Транзакция 1

##### Извлечение Р

##### Обновление Р

#### Транзакция 2

##### Извлечение Р

##### Обновление Р

Рис. 8.3. Потеря результатов обновления

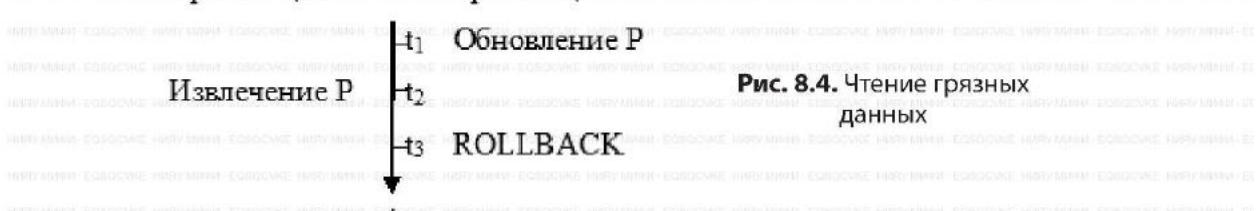
Если теперь после момента времени  $t_4$  транзакции завершатся успешно, каким будет состояние записи  $P$ ? В зависимости от того, какая из двух транзакций завершится позднее, будут утеряны изменения, сделанные ранее завершенной транзакцией.

### 8.2.2. Незафиксированные зависимости

Здесь можно, в свою очередь, выделить две ситуации, каждая из которых определяется тем, что одна из двух параллельно выполняющихся транзакций завершается откатом (ROLLBACK).

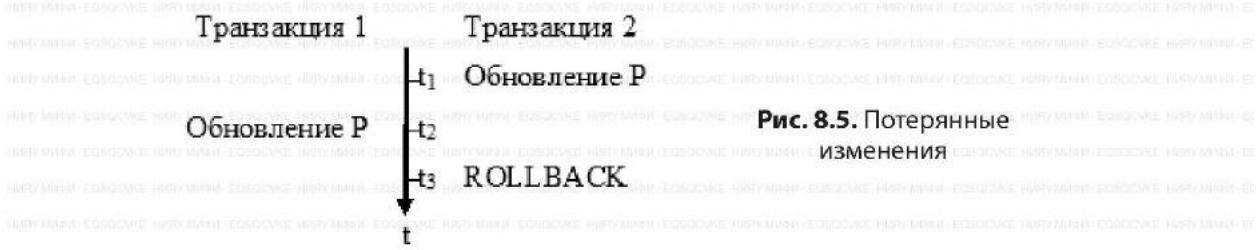
**Чтение грязных данных. Порядок выполнения транзакций при их**

*Чтение грязных данных.* Порядок выполнения транзакций при их параллельной работе представлен на рис. 8.4.



Транзакция 1 прочитала данные, измененные транзакцией 2, но в последующем транзакция 2 отменила свои изменения. Какие данные обрабатывает транзакция 1?

**Отсутствие потерянных изменений.** Порядок выполнения транзакций при их параллельной работе представлен на рис. 8.5.



Транзакция 1 изменяет данные после того, как эти данные были изменены транзакцией 2. Если теперь транзакция 2 аннулирует свои изменения, состояние БД будет восстановлено таким, каким оно было перед началом выполнения транзакции 2. А значит, изменения, сделанные транзакцией 1, также будут аннулированы.

### 8.2.3 Несовместимый анализ

Здесь можно познакомиться с ее спектром

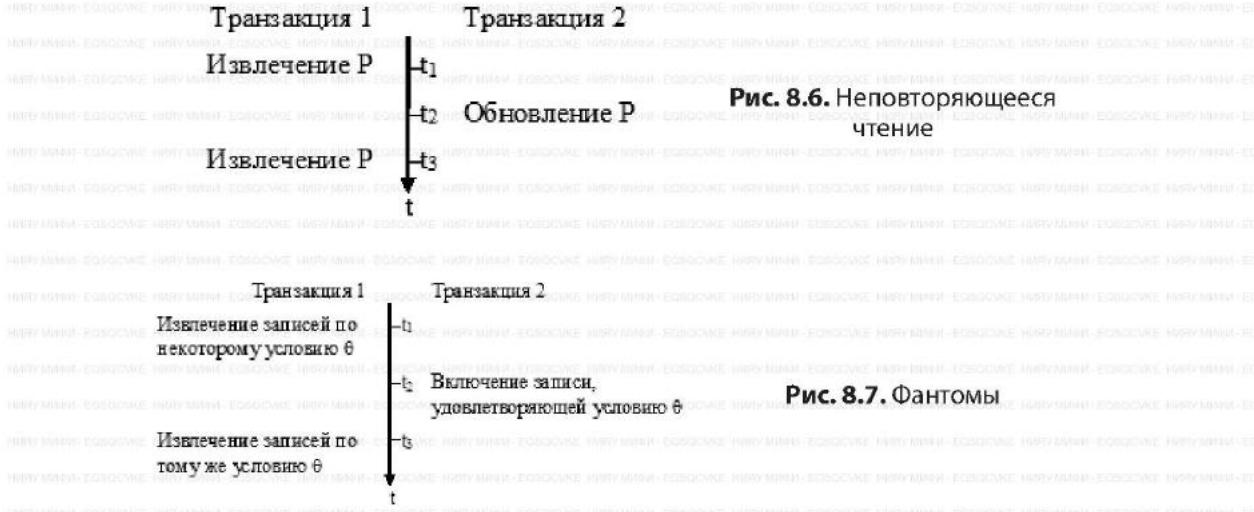
*Отсутствие неповторяющихся чтений. Порядок выполнения*

транзакций при их параллельной работе представлен на рис. 8.6.

Транзакция 2 изменяет данные, прочитанные транзакцией 1. Если после этого транзакция 1 повторно читает те же данные, она получит уже другие результаты.

**Фантомы.** Порядок выполнения транзакций при их параллельной работе представлен на рис. 8.7.

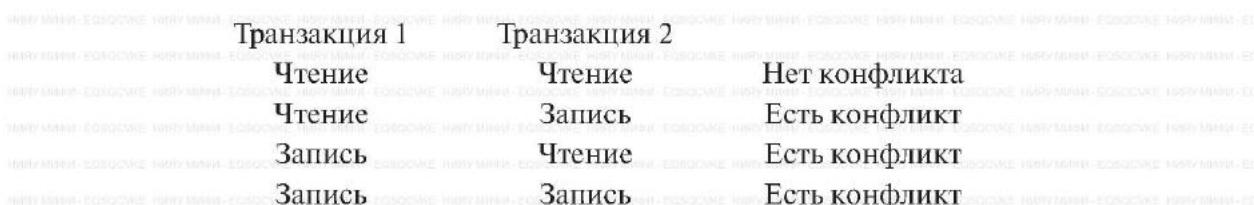
работе представлен на рис. 8.7.



## **Рис. 8.6. Неповторяющееся чтение**

Транзакция 1 считывает несколько записей, удовлетворяющих некоторому условию, после чего транзакция 2 добавляет новую запись, удовлетворяющую этому условию. Если после этого транзакция 1 повторно прочитает записи, удовлетворяющие тому же условию, она получит другой набор записей — в нем появились дополнительные записи-фантомы.

Если проанализировать приведенные выше проблемы, можно увидеть, что конфликты возникают, когда транзакции параллельно пытаются выполнить операции чтения или записи данных. Возможны следующие ситуации:



Для устранения таких конфликтов может быть использован механизм блокировок.

Блокировки

**Основная идея использования механизма блокировок заключается в следующем:** на время выполнения какой-либо операции с некоторым объектом транзакция должна заблокировать доступ к этому объекту со стороны других транзакций.

Традиционно используются два типа блокировок:  
использование  $S_{block}$  (Shared-lock) — блокировка

- разделяемая **S-блокировка** (**Shared lock**), или блокировка чтением;

- монопольная X-блокировка (eXclusive lock), или блокировка записи.

Разделяемая S-блокировка устанавливается, когда транзакция читает данные. Несколько транзакций одновременно могут читать одни и те же записи, поэтому S-блокировка на один и тот же объект может быть установлена несколькими транзакциями. Если же какая-либо транзакция читает данные, никакая другая транзакция не может эти данные модифицировать.

Чтение данных осуществляется с помощью команды **SELECT**, следовательно, при выполнении этой команды на извлекаемые записи устанавливается **S-блокировка**.

Монопольная X-блокировка устанавливается, когда транзакция изменяет некоторые данные. Только одна транзакция может установить на запись X-блокировку. Никакая другая транзакция не может изменять или читать эти данные.

Изменение данных осуществляется с помощью команд **INSERT**, **DELETE**, **UPDATE**, следовательно, при выполнении этих команд на модифицируемые записи устанавливается X-блокировка.

Для того чтобы при параллельной работе транзакций одни транзакции не мешали другим, транзакции должны устанавливать и снимать блокировки на записи. Существование блокировок приведено на рис. 8.8.

|    |    |   |   |
|----|----|---|---|
|    | T2 | S | X |
| T1 |    | + | - |
| S  | -  |   | - |

**Рис. 8.8.** Существование блокировок

**Блокировки устанавливаются и снимаются в соответствии с определенным протоколом доступа к данным.**

## Протокол доступа к данным

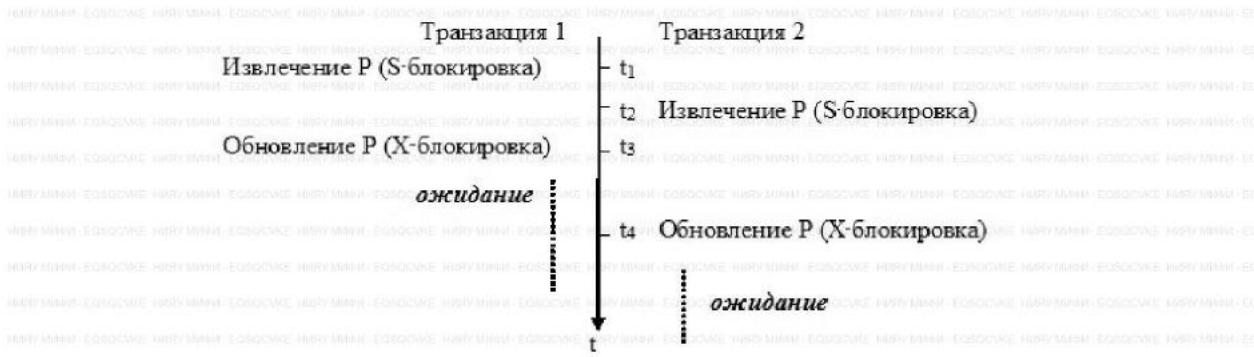
1. Транзакция, желающая извлечь некоторую запись, должна установить на эту запись S-блокировку.
  2. Транзакция, желающая изменить некоторую запись, должна установить на эту запись X-блокировку.
  3. Блокировки устанавливаются неявно, автоматически, при выполнении соответствующих команд SQL.
  4. Транзакция, установившая на некоторую запись S-блокировку, может заменить ее X-блокировкой.
  5. Транзакция, установившая на некоторую запись X-блокировку, не может заменить ее S-блокировкой.

- Блокировки устанавливаются в соответствии с правилами существования блокировок (см. рис. 8.8). Если блокировка, запрошеннная транзакцией А, отвергается из-за конфликта с транзакцией В, транзакция А переводится в состояние ожидания.
  - Блокировки сохраняются до конца транзакции и снимаются, когда транзакция завершается.

Механизм блокировок снимает проблемы, связанные с влиянием одних транзакций на другие, но, к сожалению, приводит к появлению другой проблемы — проблемы тупиков.

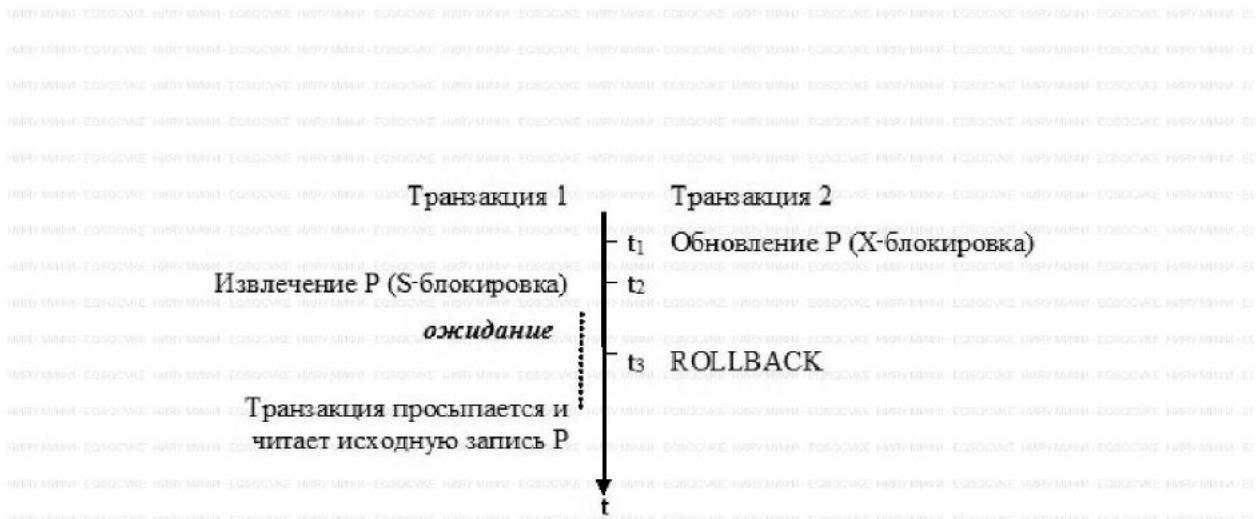
Вернемся вновь к рассмотренным выше проблемам параллелизма и посмотрим, какое влияние оказывает на них механизм блокировок.

*Потеря результатов обновления.* Транзакция 1, читающая запись P, устанавливает на эту запись S-блокировку. Транзакция 2, читающая эту же запись, также устанавливает на нее S-блокировку. Далее транзакция 1, желая изменить запись P, пытается установить на нее X-блокировку и не может из-за конфликта с транзакцией 2; в результате она переводится в состояние ожидания. То же самое происходит и с транзакцией 2. В результате возникает тупиковая ситуация — каждая из транзакций ожидает завершение другой транзакции и не может продолжить свою работу (рис. 8.9).



**Рис. 8.9.** Устранение потери результатов обновления — туник

**Чтение грязных данных.** Транзакция 2, модифицирующая запись P, устанавливает на нее X-блокировку. В результате транзакция 1, желающая прочитать эту запись, пытается установить на нее S-блокировку и переводится в состояние ожидания. Когда транзакция 2 завершается по **ROLLBACK**, восстанавливается исходное состояние записи P и X-блокировка снимается, транзакция 1 просыпается, устанавливает на запись P S-блокировку и успешно читает правильные данные (рис. 8.10).

**Рис. 8.10.** Устранение чтения грязных данных

**Потерянные изменения.** Транзакция 2, модифицирующая запись Р, устанавливает на нее X-блокировку. В результате транзакция 1, также желающая модифицировать эту запись, пытается установить на нее X-блокировку и переводится в состояние ожидания. Когда транзакция 2 завершается по ROLLBACK, восстанавливается исходное состояние записи Р и X-блокировка снимается. Транзакция 1 просыпается, устанавливает на запись Р свою X-блокировку и успешно модифицирует правильные данные (рис. 8.11).

**Рис. 8.11.** Устранение потерянных изменений

**Неповторяющееся чтение.** Транзакция 1, читающая запись Р, устанавливает на нее S-блокировку. Транзакция 2, желающая модифицировать эту запись, пытается установить на нее X-блокировку и переводится в состояние ожидания. В результате транзакция 1 в следующий момент времени успешно повторно читает ту же запись Р (рис. 8.12).



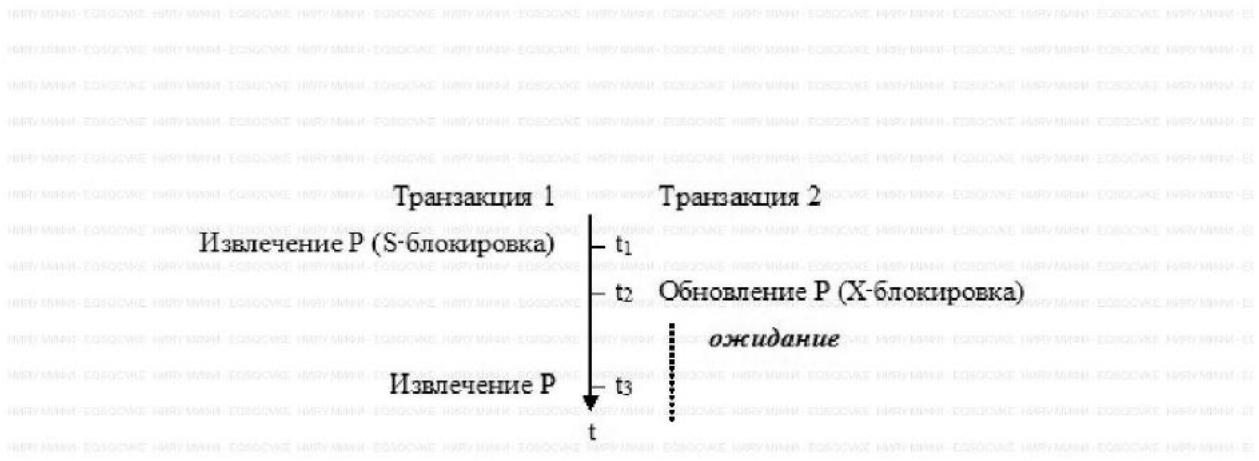


Рис. 8.12. Неповторяющееся чтение

**Фантомы.** Блокировки могут устанавливаться на разные объекты базы данных — строки (записи) таблицы, всю таблицу целиком и др. Использование блокировок, устанавливаемых только на извлекаемые и модифицируемые записи, не устраняет проблему фантомов. Если устанавливается блокировка на всю таблицу, проблема фантомов решается, но сильно ухудшается эффективность выполнения транзакций, так как операция модификации какой-либо одной записи в таблице препятствует работе с другими записями этой таблицы.

Надо отметить, что механизм блокировок — не единственный способ устранения конфликтов, возникающих при одновременном доступе к данным со стороны разных транзакций. Каждая СУБД в общем случае использует свои собственные механизмы, и при необходимости следует обращаться к соответствующим дополнительным материалам и технической документации для конкретной СУБД. Тем не менее в каждой СУБД транзакция, желающая модифицировать некоторые данные, устанавливает на эти данные монопольную X-блокировку.

### Вопросы

- Сформулируйте необходимость использования механизма транзакций в современных системах баз данных.
- Объясните, почему ситуация, близкая к идеальной с точки зрения строгого соблюдения всех свойств АСИД, снижает степень параллелизма при коллективном доступе к одни и тем же данным, и наоборот.
- Проведите сравнительный анализ типов блокировок двух-трех современных СУБД.

ГЛАВА 9

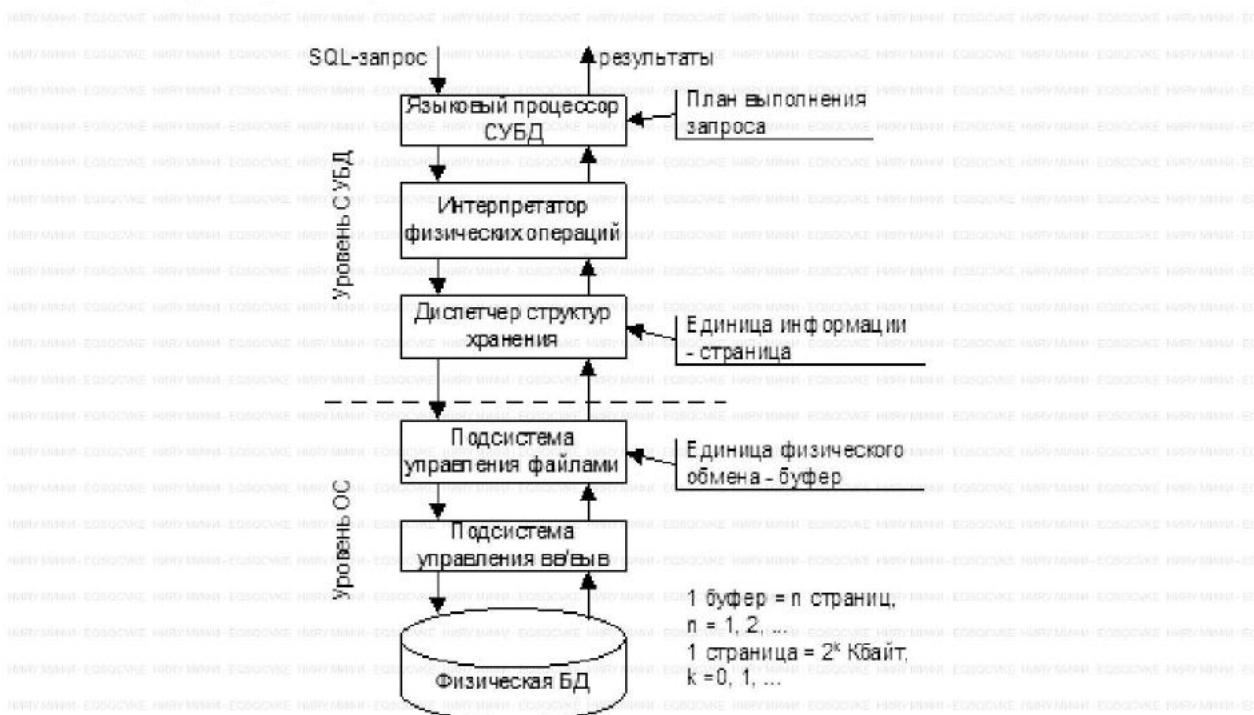
## **ВНУТРЕННИЕ СТРУКТУРЫ ХРАНЕНИЯ**

## **9.1. СТРУКТУРНАЯ СХЕМА ОБРАБОТКИ ЗАПРОСА**

**Важными особенностями реляционной СУБД, влияющими на организацию хранения данных во внешней памяти, являются следующие:**

- наличие двух уровней системы для организации доступа к данным;
  - поддержка отношений-каталогов;
  - регулярность структуры данных.

*Наличие двух уровней.* Рассмотрим структурную схему обработки запроса (рис. 9.1).



**Рис. 9.1.** Структурная схема обработки запроса

При такой организации подсистема нижнего уровня должна поддерживать во внешней памяти набор базовых структур, конкретная интерпретация которых входит в число функций подсистемы верхнего уровня.

**Поддержка отношений-каталогов.** Каталоги базы данных содержат информацию, связанную с объектами базы данных (имена, свойства и т.п.). Эта информация поддерживается системой языкового уровня. С точки зрения структур внешней памяти отношение-каталог ничем не отличается от обычного отношения.

*Регулярность структуры данных.* Так как основным объектом базы данных является таблица, главный набор объектов внешней памяти может иметь очень простую регулярную структуру. При этом необходимо обеспечить возможность эффективного выполнения операторов языкового уровня как над одним отношением, так и над несколькими отношениями.

Для этого во внешней памяти должны поддерживаться дополнительные управляющие структуры — индексы.

Отсюда возникают следующие разновидности объектов базы данных, размещаемых во внешней памяти:

- строка отношения (строка таблицы) — основная часть базы данных, большей частью непосредственно видимая пользователем;
  - индексы — объекты, создаваемые по инициативе пользователя или верхнего уровня системы из соображений повышения эффективности выполнения запросов;
  - управляющая (служебная) информация, поддерживаемая для удовлетворения внутренних потребностей нижнего уровня системы (например, информация о свободной памяти).

Это и определяет структуру файлов базы данных.

Конкретная организация файлов определяется конкретной СУБД, однако перечисленные выше компоненты в том или ином виде присутствуют в любых файлах баз данных.

Тем не менее имеются общие принципы представления во внешней памяти основных объектов базы данных — таблиц и индексов.

Таблицы обычно хранятся во внешней памяти по строкам. Однаковые значения атрибутов в разных строках дублируются. Такой способ хранения обеспечивает удобный доступ ко всей строке, но могут потребоваться лишние обращения к памяти, если нужно получить часть строки.

Организация доступа к данным в общем случае использует два способа получения данных:

- последовательная выборка данных, позволяющая получить совокупность всех строк таблицы, возможно упорядоченных по каким-либо атрибутам; такую выборку характеризует следующее предложение:  
**SELECT \* FROM таблица**
  - произвольная выборка данных, позволяющая получить конкретные строки таблицы, удовлетворяющие некоторому критерию

**рию отбора; часто в качестве такого критерия указывается условие совпадения с конкретным значением некоторых (уни-кальных) атрибутов:**

рию отбора; часто в качестве такого критерия указывается условие совпадения с конкретным значением некоторых (уни-кальных) атрибутов:

**SELECT \* FROM *таблица* WHERE *первичный\_ключ* = *значение***

Для обеспечения эффективного доступа к данным, размещаемым во внешней памяти, должны использоваться соответствующие методы доступа и дополнительные структуры, позволяющие эффективно выполнять любые запросы.

Обычно рассматриваются два метода доступа к данным:

- последовательный метод доступа, при котором некоторая целевая запись размещается непосредственно после записи, предшествующей целевой; чтобы получить целевую запись, надо последовательно обратиться ко всем предшествующим цели записям;
  - произвольный метод доступа, при котором положение целевой записи в общем случае не зависит от расположения предшествующей; целевая запись может быть получена при непосредственном обращении к ней.

Последовательный метод доступа позволяет легко организовать последовательную выборку данных, но не дает удобных средств организации произвольной выборки: произвольная выборка также сводится к обычному сканированию таблицы. Произвольный метод доступа при соответствующей организации данных позволяет эффективно реализовать и последовательную, и произвольную выборки данных. Поэтому в современных СУБД используются произвольные методы доступа. На их основе создаются дополнительные структуры — индексы, обеспечивающие быстрый доступ к требуемым данным. В настоящее время широко распространены методы доступа, основанные на использовании древовидных структур. Эти методы доступа предполагают использование специальных объектов базы данных — индексов, организованных в виде некоторой древовидной структуры. Уникальные индексы создаются на основе ключей таблицы, и в первую очередь на основе первичного ключа.

Каждый индекс ассоциируется с записью данных (строкой таблицы). Чтобы найти конкретную запись, сначала нужно по заданному значению ключа найти ассоциированный с записью индекс.

Существуют различные древовидные структуры; прежде всего это — бинарные деревья поиска, но могут быть определены и другие древовидные структуры. Рассмотрим их подробнее.

## **9.2. БИНАРНЫЕ ДЕРЕВЬЯ**

## 9.2. БИНАРНЫЕ ДЕРЕВЬЯ

Прежде всего рассмотрим двоичные (или бинарные) деревья поиска. Они могут обеспечить разумный компромисс и для произвольной, и для последовательной выборки данных.

Бинарное дерево поиска представляет собой упорядоченную тройку  $(T_L, R, T_R)$ , где  $R$  – корневая вершина дерева,  $T_L, T_R$  – левое и правое поддеревья вершины  $R$  соответственно.

Каждое из поддеревьев может быть пустым или представляет собой такое же бинарное дерево.  $T_L$  и  $T_R$  содержат соответственно  $N_L$  и  $N_R$  вершин,  $N_L \geq 0$ ,  $N_R \geq 0$ . Общее количество записей в бинарном дереве:  $NR = N_L + N_R + 1$ .

Бинарное дерево поиска на множестве NR ключей есть бинарное дерево  $T_{NR}$ , в котором каждая вершина помечена отдельным ключом и расположена в соответствии с определенным порядком ключей. Для любой вершины  $i$ ключи вершин в его левом поддереве «меньше» ключа вершины  $i$ , который, в свою очередь, «меньше» ключей вершин в его правом поддереве.

В каждой вершине бинарного дерева должны храниться:

- значение вершины, т.е. значение полного первичного ключа и некоторая запись RowId, указывающая на размещение соответствующей строки таблицы;
  - левый и правый указатели на поддеревья.

**Механизм поиска очевиден.**

При анализе алгоритма поиска определяют длину пути до целевой записи как количество просмотренных вершин дерева от корня до целевой вершины.

Вставка в бинарное дерево проста, но при этом можно получить разные формы дерева — от линейного списка (рис. 9.2, *a*) до сбалансированного дерева (рис. 9.2, *б*). Наиболее часто встречается нечто среднее между этими двумя крайними случаями (рис. 9.2, *в*).

Поскольку доступ к вершинам в деревьях поиска осуществляется на основании значений ключей, в иллюстрациях представлены только значения ключей; но нужно помнить, что с каждым ключом связана еще и запись RowId, определяющая размещение информации.

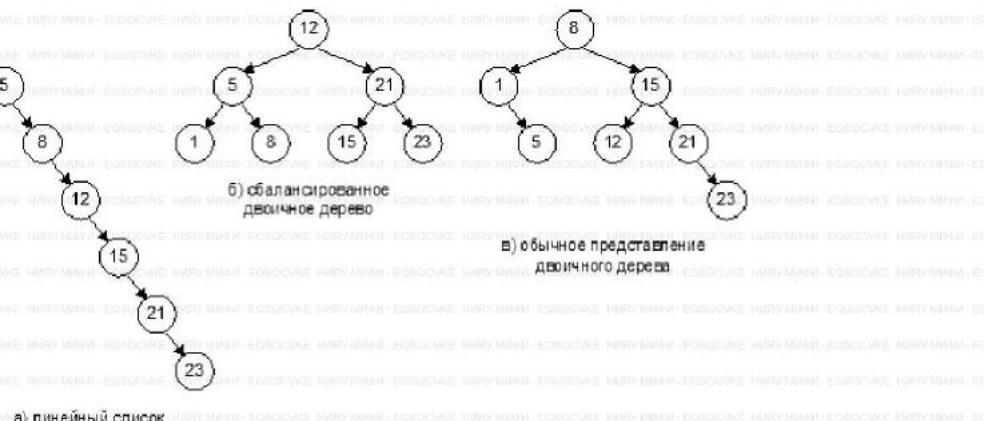
## Введем некоторые определения.

- Введем некоторые определения.

  1. Уровень вершины или листа  $i$ , обозначаемый  $L_i$ , определяется длиной пути от корневой вершины  $T_{NR}$  до вершины  $i$ . Корневая вершина, по определению, имеет уровень 0.
  2. Высота дерева определяется как максимальный уровень среди всех вершин дерева.
  3. Бинарное дерево называется сбалансированным, если разница уровней любых двух листьев не превышает 1.

Ниже приведены примеры бинарных деревьев:

- б) сбалансированное двоичное дерево**



в) обычное представление двоичного дерева

**а) линейный список**

Рис. 9.2. Примеры бинарных деревьев

Использование сбалансированного дерева минимизирует среднюю длину доступа.

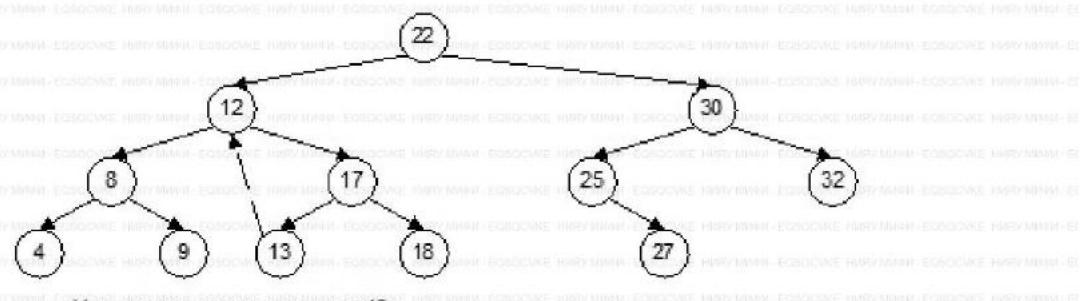
Неудобство бинарных деревьев поиска заключается в том, что они слишком «высокие»; требуется просмотреть достаточно большое количество вершин дерева, прежде чем будет найдена искомая.

Операции поиска и вставки (в несбалансированное дерево) очевидны.

С операцией удаления связаны определенные проблемы.

Удаление листа выполняется просто.

При удалении промежуточной вершины необходимо сохранить упорядоченность вершин дерева. Возможны разные алгоритмы удаления; например, на место удаляемой промежуточной вершины перемещается элемент с минимальным значением ключа из правого поддерева, подчиненного удаляемой вершине (рис. 9.3).



Удаляется элемент с ключом 12

Рис. 9.3. Пример удаления элемента из бинарного дерева

Удаление промежуточной вершины может потребовать реорганизации нескольких уровней бинарного дерева.

221

Ниже приведены примеры поиска в дереве с 4-х ходами:

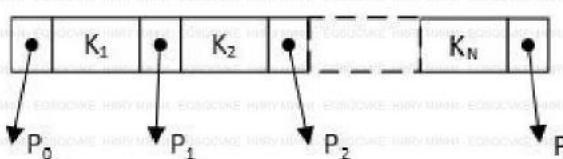
- Последовательность поиска: 42 -> 81 -> 24 -> 15.
- Последовательность поиска: 42 -> 81 -> 61 -> 50.
- Последовательность поиска: 42 -> 81 -> 107 -> 95.

На практике бинарные деревья поиска применяются крайне редко, так как из-за своей «высоты» требуют много обращений к внешней памяти при поиске.

### 9.3. МНОГОХОДОВЫЕ ДЕРЕВЬЯ

Для сокращения количества обращений к внешней памяти используется некоторое обобщение бинарных деревьев — так называемые многоходовые деревья поиска. Особенности данного типа дерева заключаются в следующем:

- каждая вершина дерева содержит  $N$  ключей и соответствующих им значений RowId и, соответственно,  $N + 1$  указатель на подчиненные вершины (рис. 9.4);
- ключи в вершине упорядочены по возрастанию:  $K_1 < K_2 < \dots < K_N$ ;
- ключи в поддеревьях упорядочены по такому же принципу, как и в бинарном дереве.



**Рис. 9.4.** Структура вершины многоходового дерева

Если обозначить указатели на подчиненные вершины через  $P_i$ , тогда вершина имеет вид

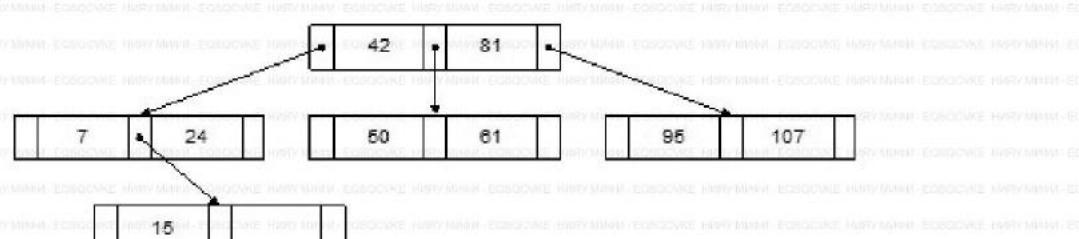
$$P_0 \ K_1 \ RowId_1 \ P_1 \ K_2 \ RowId_2 \ P_2 \dots P_{N-1} \ K_N \ RowId_N \ P_N$$

и для ключа  $K_i$  текущей вершины должно выполняться условие

$$\{K(P_{i-1})\} < K_i < \{K(P_i)\},$$

где  $\{K(P)\}$  определяет множество ключей в поддереве, заданном указателем  $P$ .

Пример многоходового дерева приведен на рис. 9.5.



**Рис. 9.5.** Пример многоходового дерева поиска

## 9.4. В-ДРЕВО

В-дерево — это сбалансированное многоходовое дерево. Особенностью В-дерева является то, что каждая вершина не должна содержать ровно  $N$  ключей; вершины В-дерева могут иметь свободное пространство, используемое для вставки новых элементов. Такая организация дерева упрощает операции вставки и удаления.

**Название «В-дерево» можно объяснить двумя способами:**

- а) от слова Balanced — сбалансированное дерево, в котором все листья имеют один и тот же уровень;
  - б) от Bayer — автора данной структуры.

3. от `base` — алгоритмической структуры.

**Определение.** В-деревом порядка  $n$  называется структура, обладающая следующими свойствами:

**1. Все пути от корня до любых листьев имеют одинаковую длину  $h$ , называемую также высотой  $B$ -дерева.**

2. В каждой вершине дерева, за исключением корня, должно располагаться минимум  $n$ , максимум  $2n$  ключей.

3. В корне В-дерева может располагаться минимум 1, максимум 21 ячеек.

4. Любая вершина дерева, за исключением листьев, имеющая

У ключей, должна иметь  $j + 1$  подчиненную вершину.

Последнее условие означает, что промежуточные вершины В-дерева имеют от  $n + 1$  до  $2n + 1$  указателей на подчиненные вершины, а корень дерева — от 2 до  $2n + 1$  указателей.

Таким образом, порядок В-дерева определяет степень его «пустоты» (или заполнения) — минимальное количество включенных в В-дерево элементов (или максимальное количество свободных позиций). Нижняя граница гарантирует, что вершины В-дерева заполнены по крайней мере наполовину. Верхняя граница позволяет определить регулярную структуру каждой вершины В-дерева.

Порядок дерева влияет на эффективность доступа: чем выше порядок дерева, тем ниже само дерево, а значит, тем меньше обращений к внешней памяти потребуется для поиска элемента.

#### 9.4.1. Структура вершины В-дерева

Обозначим записи, размещенные в одной вершине дерева, через  $R_1, R_2, \dots, R_j$ , а указатели на подчиненные вершины через  $P_0, P_1, P_2, \dots, P_j$ . Каждая запись  $R_i$  содержит соответствующие значения ключа  $K_i$  и  $RowId_i$ . Тогда структура вершины будет следующей:

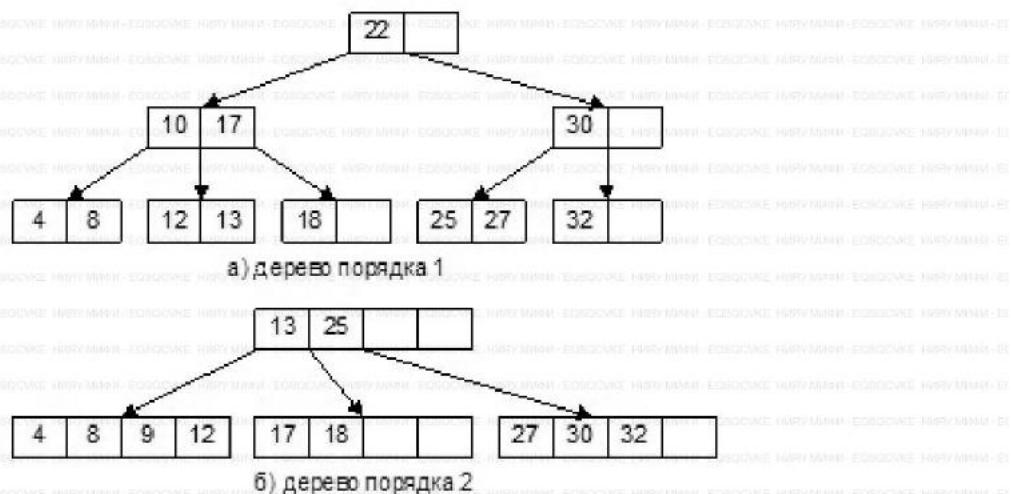
$P_0 R_1 P_1 R_2 P_2 \dots P_{j-1} R_j P_j$

Ниже приведены примеры B-деревьев порядка 1 и порядка 2 (рис. 9.6).

### Правила следования:

- Ключи записей в текущей вершине упорядочены по возрастанию, т.е. ключ записи  $R_1$  меньше ключа записи  $R_2$ , который, в свою очередь, меньше ключа записи  $R_3$ , и т.д.**
- Записи в подчиненной вершине, на которую указывает  $P_0$ , имеют ключи, меньшие, чем ключ записи  $R_1$ .**
- Записи в подчиненной вершине, на которую указывает  $P_j$ , имеют ключи, большие, чем ключ записи  $R_j$ .**
- Записи в подчиненной вершине, на которую указывает  $P_i$  ( $0 < i < j$ ), имеют ключи, большие, чем ключ записи  $R_i$ , и меньшие, чем ключ записи  $R_{i+1}$ .**

Ниже приведены примеры B-деревьев порядка 1 и порядка 2 (рис. 9.6).



**Рис. 9.6. Пример B-деревьев**

**Механизм поиска в B-дереве аналогичен механизму поиска в бинарном дереве и не требует дополнительных пояснений.**

**Операции включения и удаления должны:**

- сохранять сбалансированность B-дерева и степень заполнения его вершин;**
- не нарушать упорядоченности записей;**
- свести к минимуму перемещение информации.**

#### 9.4.2. Операция вставки

**Операции вставки предшествуют поиск, который должен завершиться неуспешно.**

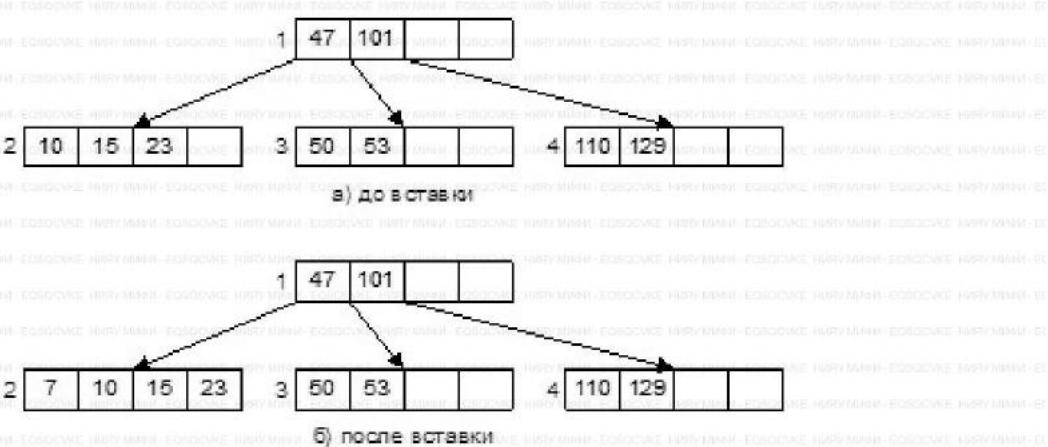
найти минимумы в листах, вставить новый элемент в лист с минимальным ключом, обновить минимальные и максимальные значения в листе, обновить минимальные и максимальные значения в родительской вершине, обновить минимальные и максимальные значения в корне дерева.

Следует иметь в виду (и это очень важно для понимания принципов использования B-дерева), что операция вставки позволяет включить новый элемент только в лист B-дерева. Поэтому прежде всего определяется целевая вершина — лист B-дерева, куда должен быть вставлен новый элемент, не нарушая упорядоченности записей.

Когда целевая вершина (лист) B-дерева будет найдена, можно столкнуться со следующими ситуациями.

1. Простейший случай, когда найденный лист имеет свободные позиции (не заполнен полностью). В этом случае новый элемент вставляется в найденный лист, не нарушая упорядоченности ключей.

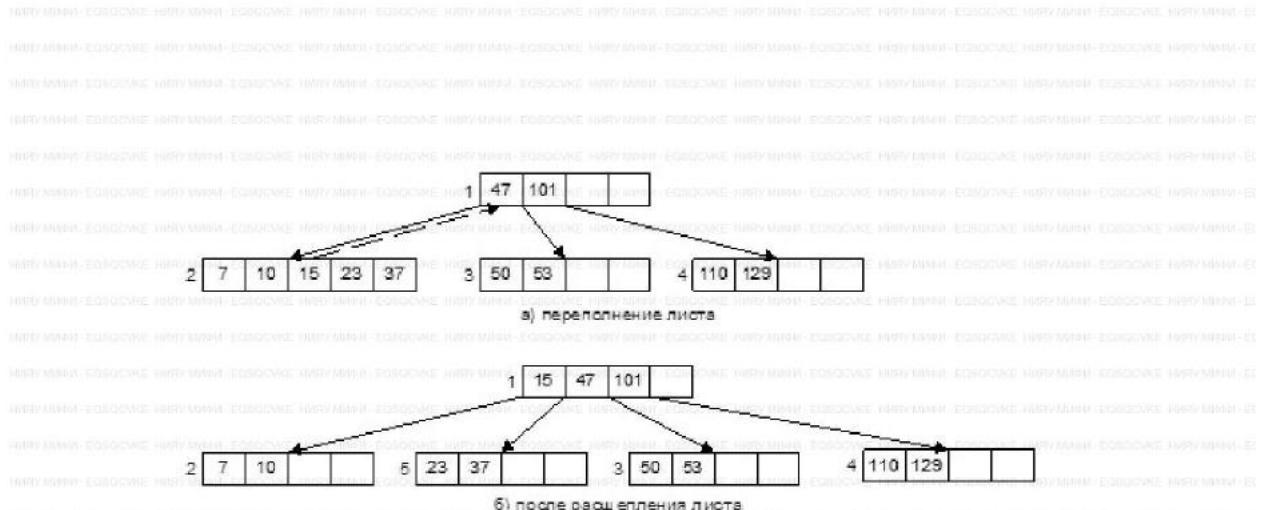
Пусть, например, имеется следующий фрагмент B-дерева порядка 2 (рис. 9.7, а; для удобства на рисунке листья B-дерева пронумерованы). Необходимо вставить элемент с ключом 7. Новый элемент должен быть размещен в листе с номером 2, в котором есть свободные позиции. В результате выполнения операции вставки получим новое B-дерево (рис. 9.7, б).



**Рис. 9.7. Реализация операции вставки в B-дерево**

2. Случай переполнения вершины: найденный целевой лист заполнен полностью. При вставке нового элемента в целевой лист получим последовательность из  $2n + 1$  ключей, тогда как в листе могут находиться максимум  $2n$  ключей. В этом случае выполняется операция расщепления листа: ключ из средней позиции переносится в родительскую вершину, а на уровне листьев появляются два новых листа.

Рассмотрим пример. Пусть в представленное выше дерево порядка 2 (рис. 9.8, а) вставляется элемент с ключом 37. Поэтому получим последовательность ключей: 7, 10, 15, 23, 37. В средней позиции находится элемент с ключом 15; он перемещается в родительскую вершину, и появляются два новых листа (рис. 9.8, б).

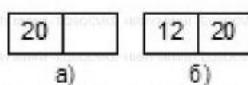
**Рис. 9.8.** Появление новых листьев при вставке в В-дерево

Когда элемент перемещается в родительскую вершину, для нее также рассматриваются эти же две ситуации. Если родительская вершина заполнена полностью, для нее также выполняется операция расщепления с перемещением элемента на расположенный выше уровень. В результате высота дерева может увеличиться на 1.

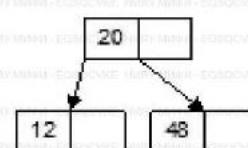
Рассмотрим пример вставки нескольких значений в В-дерево порядка 1.

Пусть вставляется следующая последовательность элементов: 20, 12, 48, 3, 5, 70, 101.

1. Первые два элемента заполняют первый лист, который является одновременно и корнем В-дерева (рис. 9.9).

**Рис. 9.9.** Вставка первых элементов в В-дерево

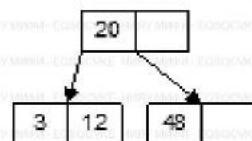
2. Вставляется следующий элемент — 48. Получаем переполнение — 12, 20, 48. Элемент из средней позиции поднимается вверх и создает новую вершину — корень В-дерева, которой подчинены два листа (рис. 9.10).

**Рис. 9.10.** Вставка элемента 48 в В-дерево

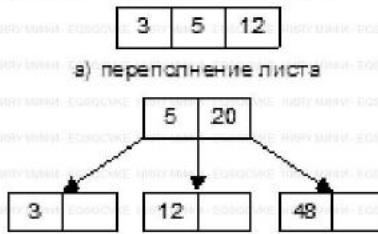
3. Элемент с ключом 3 вставляется в самый левый лист (рис. 9.11).

4. Элемент с ключом 5 также должен быть вставлен в самый левый лист; получаем переполнение — 3, 5, 12, и элемент из средней позиции перемещается в родительскую вершину, в которой есть свободное место (рис. 9.12).

**Рис. 9.11. Вставка элемента 3 в В-дерево**



**Рис. 9.11.** Вставка элемента 3 в В-дерево



**Рис. 9.12.** Вставка элемента 5 в В-дерево

```

graph TD
 Root[3] --> Node1[12]
 Root --> Node2[48]
 Root --> Node3[60]

```

6. DOCUMENTS

**5. Следующий элемент (70) вставляется в самый правый лист, в котором есть свободная позиция (рис. 9.13).**

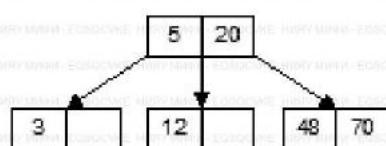
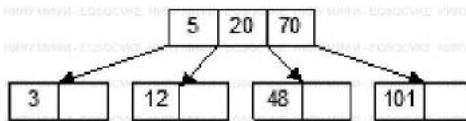


Рис. 9.13. Вставка элемента 70 в

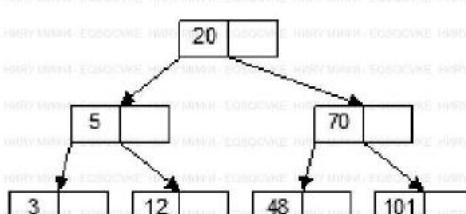
**В-дерево**

```
graph TD; Root[3, 5, 12, 14, 48, 70] --- Node1[3, 5]; Root --- Node2[12, 14]; Root --- Node3[48, 70]
```

6. В тот же лист должен быть вставлен следующий элемент с ключом 101; получаем переполнение — 48, 70, 101, и элемент из средней позиции перемещается в родительскую вершину. В родительской вершине также возникает переполнение — 5, 20, 70; в результате перемещения элемента из средней позиции образуется новая вершина — корень В-дерева (рис. 9.14) — и высота дерева увеличивается на 1.



а) результат расщепления листа



### B = 0.14 B<sub>0</sub> (magnetic field)

#### 14. Вставка элементов

②

у результат расщепления промежуточной Вершины

**Процесс вставок можно продолжать, включая в дерево новые элементы.**

Процесс вставок можно продолжать, включая в дерево новые элементы. Таким образом, при вставке элементов в дерево B-дерево растет вверх — появляется новый корень, хотя новый элемент вставляется в лист дерева.

#### 9.4.3. Удаление элемента

Ключ из B-дерева индексов удаляется из-за удаления записи из таблицы (из области данных). Операции удаления ключа предшествует успешный поиск вершины, в которой размещается удаляемый ключ.

Прежде всего рассмотрим следующие две ситуации.

1. Искомый ключ находится в листе дерева. В этом случае операция удаления не затрагивает глобально структуру B-дерева.

2. Искомый ключ находится в промежуточной вершине. Удаление ключа не должно разрушить структуру В-дерева, поэтому в такой ситуации обычно используется один из двух (равноправных) подходов:

а) удаляемый ключ замещается элементом с минимальным значением ключа из правого поддерева, подчиненного удаляемому элементу;

б) удаляемый ключ замещается элементом с максимальным значением ключа из левого поддерева, подчиненного удаляемому элементу.

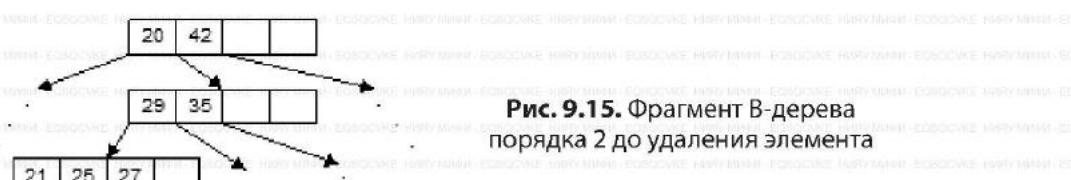
И в том и в другом случае в итоге приходим к необходимости удалить элемент из листа дерева. Поэтому в дальнейшем будем рассматривать все примеры, удаляющие элементы из листьев дерева.

Конкретный алгоритм удаления может использовать любой из двух подходов. Для определенности будем использовать подход а).

При удалении элемента из листа также можно столкнуться с другой ситуацией.

1. Простейшая ситуация, когда в листе находится более чем  $n$  элементов ( $n$  — порядок В-дерева). В этом случае удаление элемента не нарушает структуру В-дерева и заканчивается сразу же

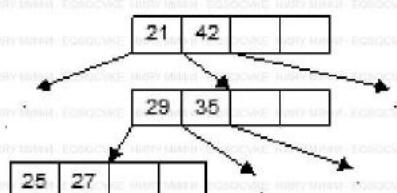
Рассмотрим пример. Пусть имеется следующий фрагмент В-дерева порядка 2 (рис. 9.15).



**Рис. 9.15.** Фрагмент В-дерева порядка 2 до удаления элемента

Пусть требуется удалить элемент с ключом 20. Удаляемый элемент находится в промежуточной вершине В-дерева. В правом поддереве, подчиненном удаляемому элементу, находим минимальный элемент — это 21 — и перемещаем его на место удаляемого элемента. Поскольку лист был заполнен более чем наполовину, после перемещения элемента в нем осталось необходимое количество элементов. В итоге получим следующее состояние В-дерева (рис. 9.16).

**Рис. 9.16.** Фрагмент В-дерева порядка 2 после удаления элемента



2. Случай антипереполнения вершины: в целевом листе находится минимально допустимое количество элементов. При удалении элемента в листе остается недостаточное количество элементов — возникает антипереполнение, которое должно быть устранено.

Антипереполнение устраняется одним из двух способов: с помощью перераспределения элементов или с помощью сцепления вершин. Рассмотрим эти способы.

## Перераспределение элементов

Рассматриваются соседние вершины дерева (слева или справа), подчиненные той же вершине и тому же ключу, что и целевая. Если хотя бы в одной из них имеется более  $n$  записей (где  $n$  — порядок В-дерева), выполняется перераспределение элементов.

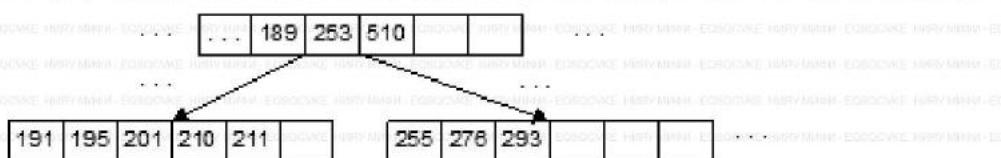
Для этого объединяются оставшиеся элементы из целевой вершины (в количестве  $n - 1$ ), выбранной соседней вершины (в количестве  $n + 1 + m$ , где  $m \geq 0$ ) и их общий корень. В результате объединения будет получено  $(n - 1) + (n + 1 + m) + 1 = 2n + 1 + m$  вершин, где  $m \geq 0$ . Эти элементы перераспределяются между целевой и соседней вершинами так, что в каждой из них появится минимум  $n$  элементов, и один элемент поднимется в родительскую вершину. Отсюда видно, что операция перераспределения обратна операции расщепления, выполняемой при вставке элементов.

Рассмотрим пример. Пусть дан следующий фрагмент В-дерева порядка 3 (рис. 9.17).

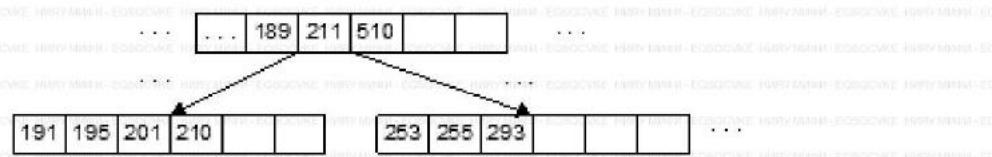
Пусть удаляется элемент с ключом 276. В целевой вершине остается 2 элемента, что недостаточно.

В соседней вершине имеется 5 элементов, поэтому можно выполнить

нить перераспределение элементов. В результате объединения полу-

**Рис. 9.17.** Фрагмент В-дерева порядка 3 до удаления элемента

Чим 5 (левый лист) + 1 (родительская вершина) + 2 (целевой лист) = 8 элементов. Эти элементы могут быть перераспределены между вершинами, например, так, как показано на рис. 9.18

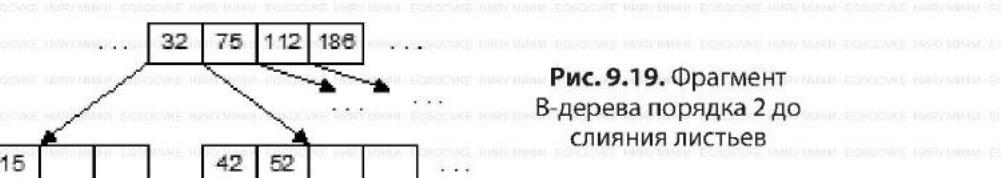
**Рис. 9.18.** Фрагмент В-дерева порядка 3 после удаления элемента

В результате получена структура, удовлетворяющая определению В-дерева.

#### Сцепление вершин

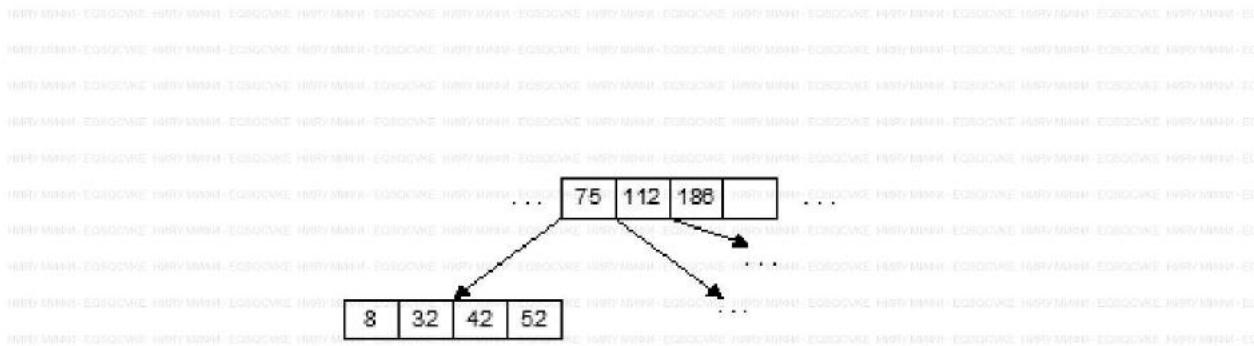
Если в соседних слева и справа вершинах находится только минимально допустимое количество элементов, перераспределение использовано быть не может. В этом случае используется сцепление: вместо двух вершин — целевой и какой-либо соседней — создается одна, в которую помещаются элементы из двух вершин и их общего корня. В результате в новой вершине окажется  $2n$  элементов — максимально возможное количество ключей в вершине В-дерева. Элемент из родительской вершины удаляется, а два указателя (левый и правый) для этого элемента объединяются в один, указывающий на вновь созданную вершину В-дерева.

Рассмотрим пример. Пусть дан следующий фрагмент В-дерева порядка 2 (рис. 9.19).

**Рис. 9.19.** Фрагмент В-дерева порядка 2 до слияния листьев

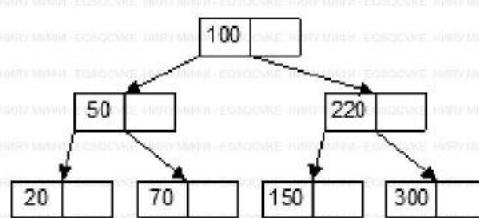
Удаляется элемент 15. В результате получим один лист и один указатель на него (рис. 9.20).

**230**



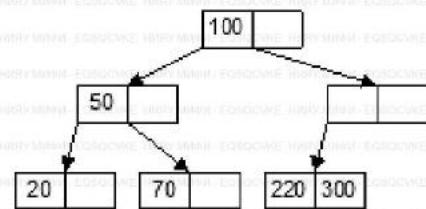
**Рис. 9.20.** Фрагмент В-дерева порядка 2 после слияния листьев

В результате такой операции сцепления количество элементов в родительской вершине уменьшается на единицу и в ней также может возникнуть антипереполнение, которое разрешается одним из двух рассмотренных способов. В результате распространения антипереполнения высота В-дерева может уменьшиться на 1. Рассмотрим пример. Пусть дано В-дерево порядка 1 (рис. 9.21).



**Рис. 9.21.** Пример В-дерева порядка 1

Удаляется элемент 150. В соседней вершине только один элемент, поэтому выполняется сцепление. В результате возникло антипереполнение в родительской вершине (рис. 9.22).



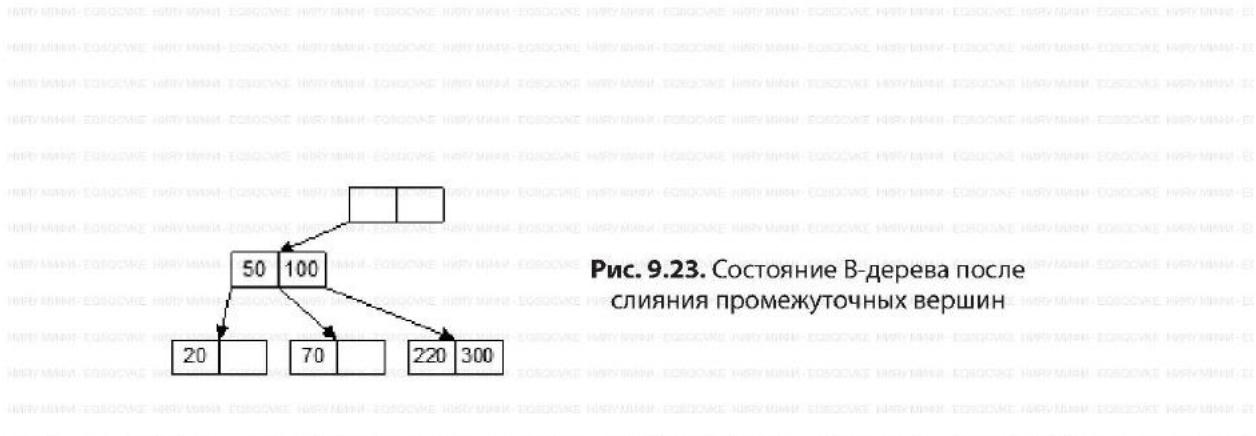
**Рис. 9.22.** Промежуточное состояние В-дерева после слияния листьев

Для данной вершины также выполняется сцепление, в результате которого в корне В-дерева не осталось ни одного элемента (рис. 9.23). В такой ситуации пустая вершина — корень В-дерева — удаляется и высота дерева уменьшается на 1.

Таким образом, можно указать следующие основные свойства В-дерева.

1. Ключи и ассоциированные с ними данные хранятся во всех вершинах В-дерева.

2. Произвольная выборка данных выполняется эффективно; максимальная длина пути равна высоте дерева, но может быть получена



**Рис. 9.23.** Состояние В-дерева после слияния промежуточных вершин

и меньшая длина пути, если искомый элемент расположен в какой-либо промежуточной вершине В-дерева.

3. Последовательная выборка данных, особенно требующая упорядоченности по значениям ключей, малоэффективна. Элемент с минимальным значением ключа получается при перемещении от корня В-дерева к левому листу. Чтобы получить следующие элементы, приходится подниматься от листа к корню, а затем вновь спускаться к листьям.

## 9.5. В<sup>+</sup>-ДЕРЕВО

Чтобы устранить недостатки, свойственные В-дереву при выполнении последовательной выборки данных, было введено В<sup>+</sup>-дерево.

В<sup>+</sup>-дерево во многом аналогично В-дереву. Как и В-дерево, В<sup>+</sup>-дерево является сбалансированным. Для В<sup>+</sup>-дерева устанавливаются те же ограничения на количество ключей в каждой вершине дерева, зависящие от порядка дерева.

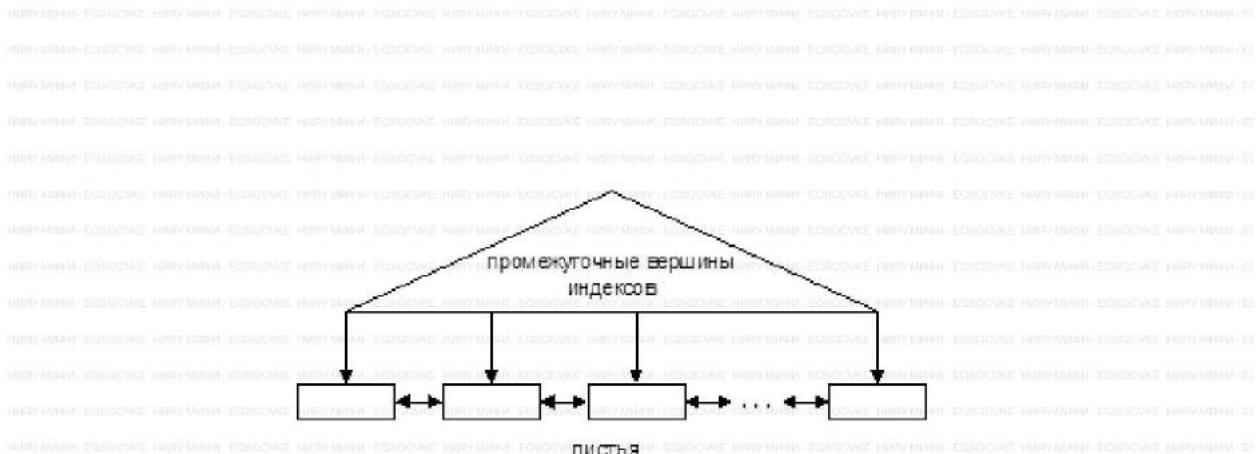
Различие между В<sup>+</sup>-деревом от В-деревом заключается в следующем.

В В-дереве все вершины дерева равноправны и содержат ключи и ассоциированные с ними данные.

В В<sup>+</sup>-дереве выделяются два типа вершин: промежуточные вершины индексов и листья. Ключи и ассоциированные с ними данные (RowId) размещаются только в листьях, тогда как в промежуточных вершинах индексов размещаются только ключи. Листья объединяются в связанное последовательное упорядоченное множество. Это позволяет эффективно выполнять последовательные запросы. Доступ к листьям осуществляется через промежуточные вершины В<sup>+</sup>-дерева, организованные в виде обычного В-дерева индексов (рис. 9.24).

Исходя из этого В<sup>+</sup>-дерево можно определить (используя введенное ранее определение В-дерева) следующим образом.

1. Для любой вершины В<sup>+</sup>-дерева, включая листья, выполняются те же ограничения на количество ключей и указателей (в зависимости от порядка дерева), что и в В-дереве.

**Рис. 9.24.** Структура B<sup>+</sup>-дерева

2. Все ключи в B<sup>+</sup>-дереве хранятся в листьях. Для обеспечения правильного доступа отдельные ключи могут дублироваться и в промежуточных вершинах индексов.

3. Доступ к информации в B<sup>+</sup>-дереве выполняется всегда за h шагов, где h — высота B<sup>+</sup>-дерева.

4. Выполнение операций включения и удаления в B<sup>+</sup>-дереве несколько отличается от выполнения соответствующих операций в B-дереве.

### 9.5.1. Операция включения

B<sup>+</sup>-дерево также растет от листьев к корню. Включение начинается с листьев. Также в случае переполнения выполняется расщепление вершины, но в B<sup>+</sup>-дереве по-разному выполняется расщепление листа и расщепление промежуточной вершины дерева. При расщеплении листа в B<sup>+</sup>-дереве ключ, расположенный в средней позиции листа, перемещается в родительскую вершину и остается в листе — в правом или левом, но в каком-то одном для всех операций вставки. Расщепление промежуточной вершины B<sup>+</sup>-дерева выполняется точно так же, как и в B-дереве.

Рассмотрим особенности выполнения операции включения на примере. Пусть в B<sup>+</sup>-дерево порядка 1 включаются те же элементы и в том же порядке, что и в B-дереве, рассмотренное ранее (см. рис. 9.9–9.14): 20, 12, 48, 3, 5, 70, 101.

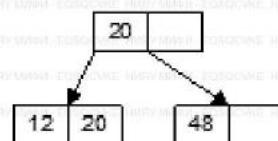
1. Включение первых двух элементов ничем не отличается от их включения в B-дерево (рис. 9.25).

|    |    |
|----|----|
| 12 | 20 |
|----|----|

**Рис. 9.25.** Включение первых двух элементов в B<sup>+</sup>-дерево

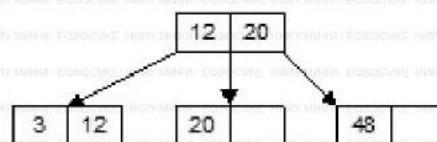
2. Включение третьего элемента вызывает переполнение листа — 12, 20, 48. Элемент из средней позиции перемещается в родитель-

скую вершину, в результате чего создается новый корень В<sup>+</sup>-дерева (рис. 9.26). Но так как в В<sup>+</sup>-дереве все ключи должны находиться в листьях, перемещаемый элемент остается и в одном из двух листьев, например в левом (примем для определенности).



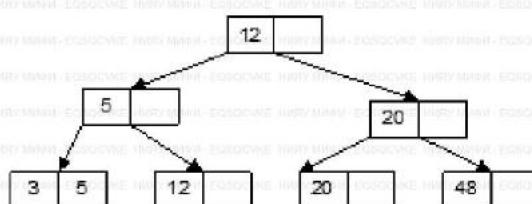
**Рис. 9.26.** Включение элемента 48 в  $B^+$ -дерево

3. Включение следующего элемента (3) также вызывает переполнение левого листа — 3, 12, 20. Элемент из средней позиции перемещается в родительскую вершину, в которой есть свободная позиция, и остается в левом листе (рис. 9.27).



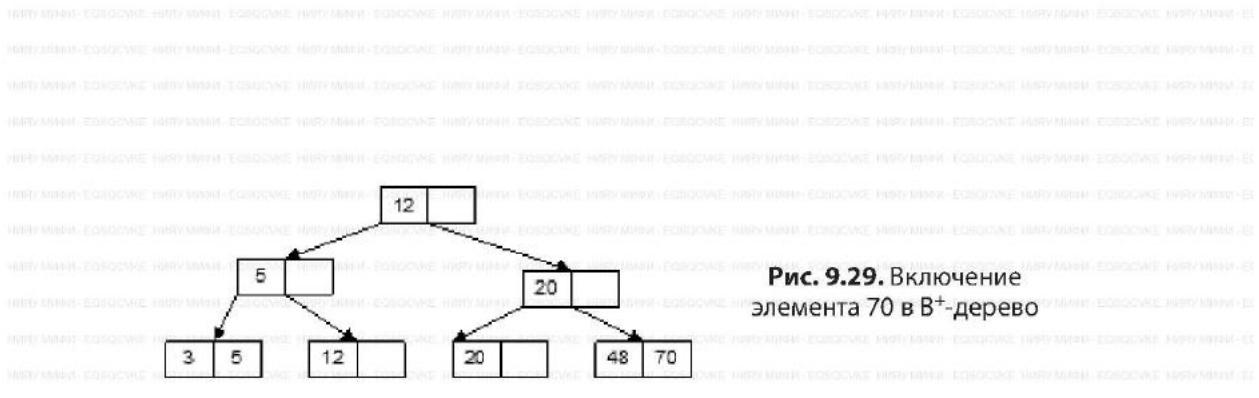
**Рис. 9.27.** Включение элемента 3 в  $B^+$ -дерево

4. Элемент 5 также должен быть размещен в левом листе: возникает переполнение — 3, 5, 12. В результате расщепления вершины элемент из средней позиции перемещается в родительскую вершину, в которой возникает переполнение — 5, 12, 20, и сохраняется в левом листе. Переполнение родительской вершины обрабатывается по правилам B-дерева — образуются две вершины с элементами 5 в левой и 20 в правой, а элемент 12 перемещается в родительскую вершину — создается новый корень B<sup>+</sup>-дерева (рис. 9.28).



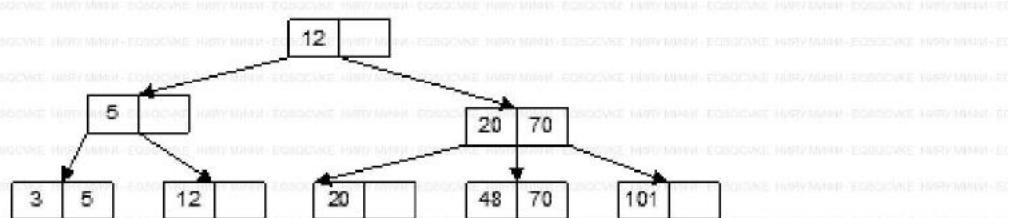
**Рис. 9.28.** Включение элемента 5 в  $B^+$ -дерево

**5. Элемент 70 должен быть размещен в самом правом листе — в нем есть свободная позиция, поэтому реорганизация дерева не проходит (рис. 9.29).**



**Рис. 9.29.** Включение элемента 70 в  $B^+$ -дерево

6. Включение последнего элемента (101) вызывает переполнение — 48, 70, 101. В результате расщепления вершины появляются два новых листа, элемент 70 сохраняется в левом листе и дублируется в родительской вершине, в которой есть свободная позиция (рис. 9.30).



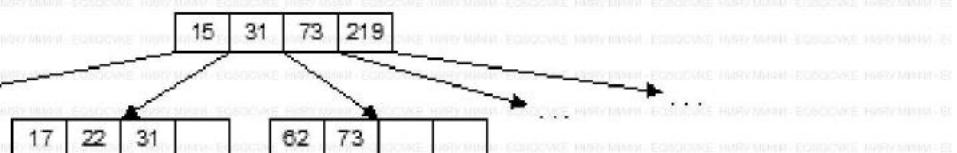
**Рис. 9.30.** Включение элемента 101 в  $B^+$ -дерево

### 9.5.2. Удаление элемента

Так как вся информация размещается в листьях, удаление выполняется только из листьев. При этом, если удаляемый ключ встречается еще и в промежуточных вершинах  $B$ -дерева индексов, он из них не удаляется (до поры до времени).

Если возникает антипереполнение, оно устраниется теми же методами, что и в  $B$ -дереве, при этом могут быть затронуты (и удалены) ключи из промежуточных вершин  $B$ -дерева индексов.

Рассмотрим пример. Пусть дан следующий фрагмент  $B^+$ -дерева порядка 2 (рис. 9.31).

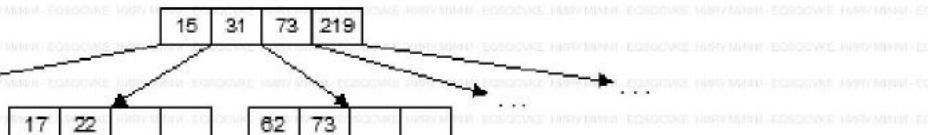


**Рис. 9.31.** Фрагмент  $B^+$ -дерева порядка 2 до удаления элемента

Если удаляется элемент 31, он просто удаляется из листа. Так как при этом антипереполнение в листе не возникает, операция удаления

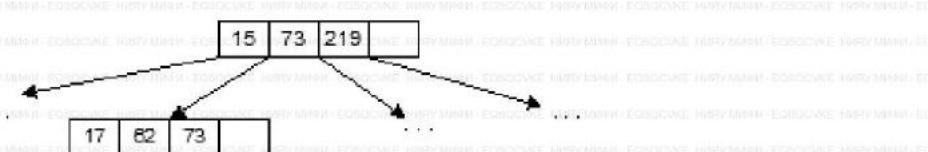
Ниже минимумы: 15, 31, 73, 219. Ключи: 17, 22, 62, 73. Данные: пустые.

заканчивается и ключ 31 сохраняется в промежуточной вершине В-дерева индексов (рис. 9.32).



**Рис. 9.32.** Фрагмент B<sup>+</sup>-дерева порядка 2 после удаления элемента

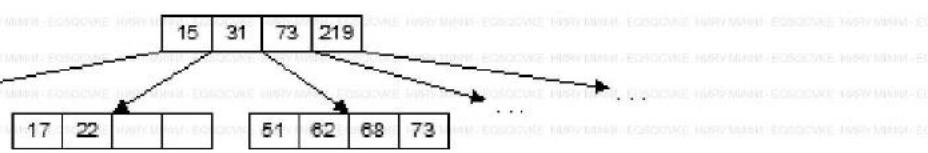
Если на следующем этапе удаляется элемент 22, в листе возникает антипереполнение. Поскольку соседний лист содержит минимально допустимое количество элементов, выполняется слияние двух листов. При этом ключ 31 из промежуточной вершины В-дерева индексов удаляется (рис. 9.33).



**Рис. 9.33.** Слияние листьев при удалении элемента из B<sup>+</sup>-дерева

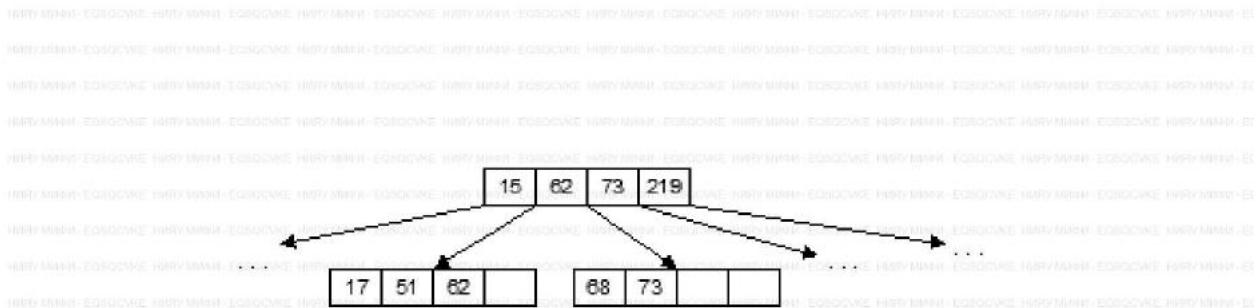
Если же соседний лист заполнен более чем наполовину (например, содержит максимально возможное количество элементов), тогда выполняется перераспределение.

Рассмотрим следующий фрагмент B<sup>+</sup>-дерева порядка 2 (рис. 9.34), полученный также после удаления элемента 31.



**Рис. 9.34.** Фрагмент B<sup>+</sup>-дерева порядка 2

Если теперь удаляется элемент 22, в листе возникает антипереполнение. Поскольку соседний лист содержит максимально возможное количество элементов, выполняется перераспределение: объединяются данные из левого листа (17), родительской вершины (31) и правого листа (51, 62, 68, 73), при этом ранее удаленный элемент 31 удаляется. Для полученной последовательности ключей (17, 51, 62, 68, 73) средний элемент перемещается в родительскую вершину и остается в левом листе. В результате получаем следующий фрагмент B<sup>+</sup>-дерева (рис. 9.35).



**Рис. 9.35.** Перераспределение элементов при удалении из  $B^+$ -дерева

## 9.6. ХЭШ-ТАБЛИЦЫ

Еще один способ организации данных, используемый в базах данных, — использование хэш-таблиц. Хэш-таблицы получили свое название от метода, используемого для организации доступа, — хешированный индекс. Этот метод обеспечивает быструю произвольную выборку и обновление записей.

Этот метод принципиально отличается от  $B^+$ -деревьев, т.е. имеет принципиально другой алгоритм размещения строк в таблице. Рассмотрим особенности построения и использования хэш-индексов.

### 9.6.1. Идея хэширования

Символический ключ (или идентификатор) — это совокупность атрибутов, на которых строится хэш-индекс; должен уникально идентифицировать каждую запись таблицы. Он преобразуется в физический адрес. Функция преобразования — это и есть хэш-функция.

Принцип отображения:  $m:1$ , т.е.  $m$  разных идентификаторов могут быть преобразованы в один и тот же хэш-индекс (или собственно адрес). Такие идентификаторы называются синонимами, а ситуация, в которой возникают синонимы, — коллизией.

При возникновении коллизий необходимы специальные меры для их разрешения, т.е. для размещения синонима в пространстве таблицы так, чтобы его можно было найти. Обычно поступают так: все адресное пространство, непосредственно доступное функции хэширования, делится на несколько областей фиксированного размера — бакеты (buckets). Это может быть страница (блок) физической памяти или несколько страниц, т.е. любой участок памяти, адресуемый как одно целое. Хэш-функция отображает идентификатор в номер бакета. В одном бакете может размещаться несколько записей (строк таблицы).

Процесс разрешения коллизий состоит из двух шагов.

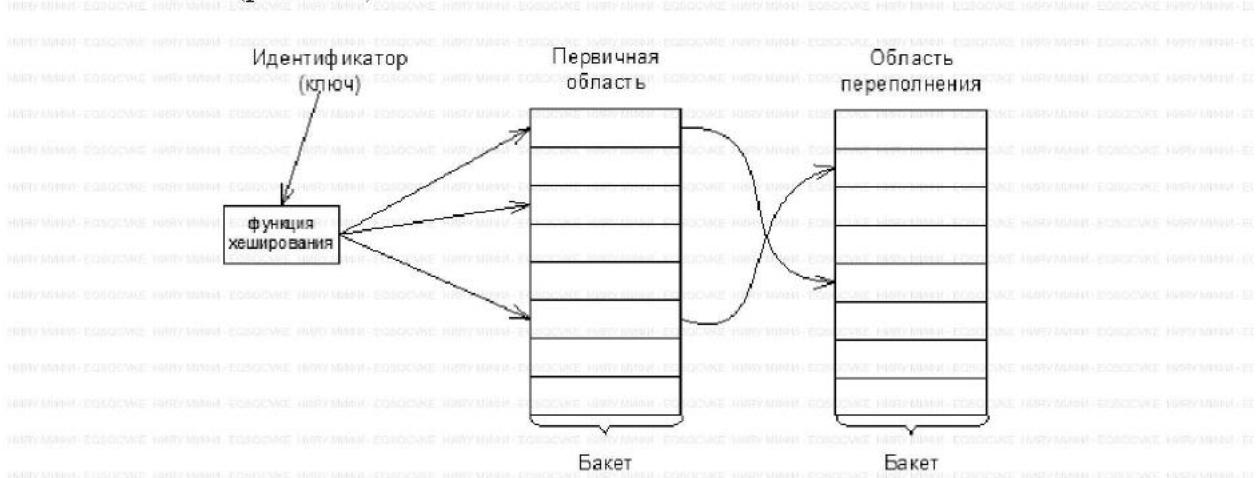
- Выполняется просмотр бакета с целью выявления в нем свободного пространства для новой записи. Если оно есть, новая запись размещается в этом бакете. Если свободного пространства нет, тогда выполняется шаг 2.

- Обрабатывается переполнение бакета.

Ниже приведены основные методы обработки переполнения:

- Метод хеш-функции**. Использование пространства памяти для хранения информации о записих в виде бакетов.
- Метод хеш-индексов**. Использование пространства памяти для хранения информации о записих в виде бакетов.
- Метод хеш-таблиц**. Использование пространства памяти для хранения информации о записих в виде бакетов.

**Область памяти (совокупность бакетов), на которую хэш-функция отображает символический ключ, обычно называют первичной областью, а остальную часть доступной памяти — областью переполнения (рис. 9.36).**



**Рис. 9.36. Использование пространства**

**Методы обработки переполнения призваны обеспечить эффективное хранение записей переполнения.**

### 9.6.2. Принципы построения хэш-индекса

Хэш-индекс строится (определяется) на основании некоторой хэш-функции. Хэш-функция отображает значение заданных атрибутов в конкретное размещение в памяти (т.е. в физический адрес — обычно относительный). При использовании хэш-функции важно учитывать следующее.

1. Адрес, получаемый в результате, должен попасть в диапазон адресов, соответствующий выделенному пространству. Поэтому пространство под таблицу резервируется при создании таблицы, и в хэш-функции часто используется операция вычисления остатка от деления на размер таблицы.

2. Хэш-функция должна давать достаточно однородные значения хэш-индексов, чтобы не было скопления синонимов (или коллизий), требующих специальных методов обработки.

Поэтому, поскольку на качество хэширования может существенно повлиять выбор функции хэширования, обычно учитываются следующие правила построения хэш-функции:

- a) первый ключ обязательно входит в атрибуты, на которых строится хэш-индекс;

б) хэш-функция должна использовать значения всех атрибутов первичного ключа.

Наилучшей функцией хэширования является функция, отображающая NR значений ключа в точности в NR значений собственных адресов (хэш-индексов) без синонимов. Теоретически существует  $NR!$  (факториал) таких способов отображения. Однако если учесть, что существует  $NR^{NR}$  способов присвоения NR ключам NR собственных адресов, вероятность такого идеального отображения ничтожно мала [13]. Поэтому тратятся усилия на выбор хэш-функции, достаточно равномерно отображающей ключи на адресное пространство.

Вероятность возникновения коллизий можно уменьшить за счет:

- а) выбора хорошей хэш-функции;  
б) выделения избыточного адресного пространства.

Если же коллизии, тем не менее, возникают, необходимо их разрешать, используя тот или иной метод обработки переполнения.

### 9.6.3. Методы обработки переполнения

## Метод открытой адресации

При использовании данного метода обработки переполнения основная область и область переполнения объединяются в одно общее пространство.

Изначально все бакеты пусты. Хэш-функция отображает ключи в номера бакетов, в которых сохраняются соответствующие записи. Когда какой-нибудь бакет, например А, будет заполнен и хэш-функция отображает новую запись в этот же бакет А, тогда запись переполнения заносится в первую свободную позицию ближайшего не-заполненного бакета, например В (рис. 9.37). Неуспешный поиск продолжается до тех пор, пока не будет обнаружена свободная позиция.



**Рис. 9.37.** Метод открытой адресации

**ция или же в результате просмотра всех бакетов не будет получен исходный бакет. Выполняется линейный поиск.**

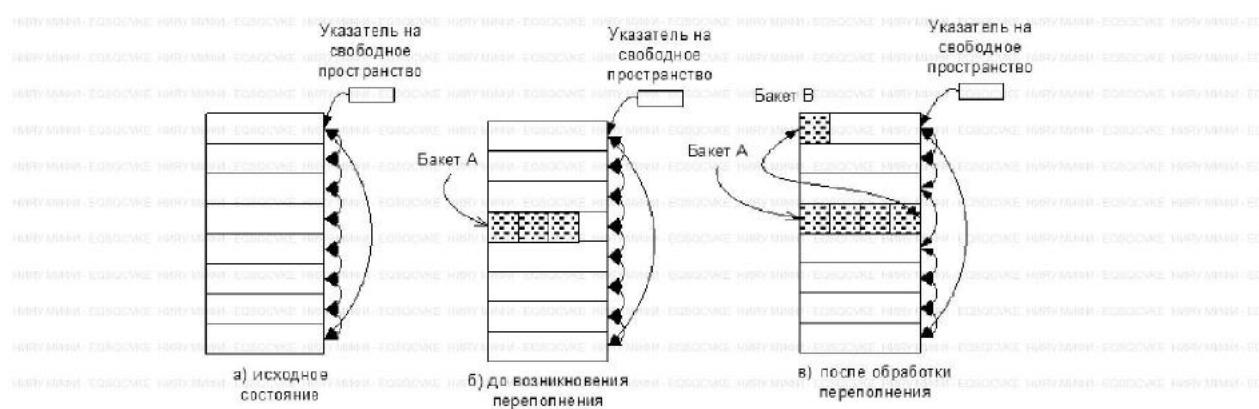
ция или же в результате просмотра всех бакетов не будет получен исходный бакет. Выполняется линейный поиск.

Один и тот же бакет может использоваться и для размещения переполняющих записей, и для собственных записей. Количество бакетов, которые надо просмотреть, прежде чем запомнить запись, называется смещением записи. Для синонимов имеет место значительное увеличение смещения, что плохо.

Есть разновидности с нелинейным поиском, но все равно это не лучший способ обработки переполнения.

#### Метод срастаний цепочек

Выделяется специальная группа свободных бакетов с учетом возможности переполнения, но они все еще входят в первичную область. Область организуется в двухсвязный список (рис. 9.38, а).



**Рис. 9.38.** Метод срастанийющихся цепочек

Изначально все бакеты пусты и связаны двусвязным списком; указатель на свободное пространство определяет первый незаполненный бакет.

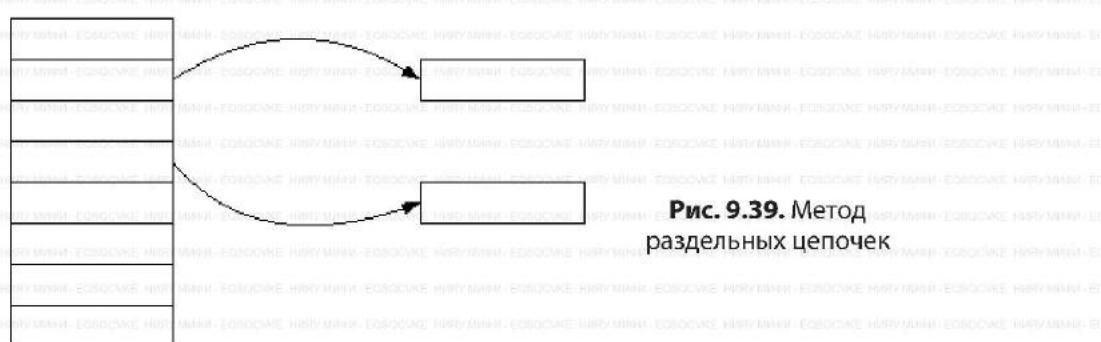
Хэш-функция отображает ключи в номера бакетов, в которых сохраняются соответствующие записи (рис. 9.38, б). Когда какой-нибудь бакет, например А, будет заполнен, он исключается из списка свободных бакетов. Если хэш-функция отображает новую запись в бакет А, тогда из списка свободных бакетов выбирается первый незаполненный бакет (по указателю на свободное пространство), в котором и размещается новая запись (например, бакет В). При этом бакеты А и В связываются между собой указателем (рис. 9.38, в).

Когда и бакет В будет заполнен, он также будет исключен из списка свободных бакетов; при этом указатель на свободное пространство переместится на следующий незаполненный бакет.

Бакет В может использоваться и для размещения записей, переполняющих бакет А, и для собственных записей. В случае переполнения бакета В происходит срашивание цепочек синонимов для бакетов А и В.

### Метод раздельных цепочек

Для каждого бакета первичной области выделяются локальные бакеты для области переполнения. Все бакеты переполнения выделяются в отдельную область переполнения, и их адреса не могут быть использованы в качестве собственных адресов записей (рис. 3.39).



**Рис. 9.39.** Метод  
раздельных цепочек

## Основная область

## Область переполнения

## **9.7. БИТОВЫЕ ИНДЕКСЫ ИЛИ ИНДЕКСЫ НА ОСНОВЕ БИТОВЫХ КАРТ**

В СУБД Oracle и PostgreSQL существует еще один способ произвольного доступа к данным, основанный на использовании последовательности, состоящей из одного или нескольких бит, формирующих битовую карту. Основная идея индекса на основе битовых карт (bitmap-индекса) состоит в том, что одна и та же битовая карта (последовательность бит) может указывать на диапазон строк, у которых значение bitmap-индекса соответствует значению данной битовой карты. Таким образом, чем больше одна и та же битовая карта содержит указателей назначающие строки и эти указатели максимально компактно упорядочены, тем меньше размер битового индекса и, соответственно, использование данного индекса является более эффективным при доступе к данным с низкой селективностью. То есть битовый индекс не избирателен в отличие от индекса на основе B-дерева.

Понятие избирательности (или селективности) индекса позволяет определить количественные характеристики запроса, при выполнении которого используется соответствующий индекс, т.е. какая часть строк возвращается по запросу, использующему индекс при некотором условии отбора. Избирательность, или селективность, индекса можно определить как отношение количества уникальных значений в индексируемой колонке к общему количеству строк таблицы. Идеальное значение селективности индекса равно 1. Например, пусть имеется таблица tab, в которой общее количество строк равно 120, при этом одна из индексированных колонок n1 содержит 90 уникальных значений. Тогда селективность индекса i\_n1 будет равна  $0,75 = \frac{90}{120}$ . Таким образом, колонки, к которым применяются ограничения первичного ключа и на которых, как правило, создается уникальный индекс, являются наиболее избирательными, или селективными, поскольку используются для чтения только необходимого количества строк. Говорят, что индекс неизбирателен, или неселективен, когда его селективность стремится к нулю. Например, пусть имеется таблица tab, в которой общее количество строк равно 120, при этом одна из индексированных колонок n2 содержит только 5 уникальных значений. Тогда селективность индекса i\_n2 будет равна  $0,042 = \frac{5}{120}$ . Таким образом, SQL-запрос, который использует индекс i\_n2, будет возвращать 24 =  $\frac{120}{5}$  строки для каждого уникального значения индекса i\_n2. Соответственно, использование такого индекса неэффективно с точки зрения доступа к данным, и выгодней не использовать подобный индекс совсем.

Пример использования bitmap-индекса представлен на рис. 9.40

| BITMAP INDEX |                  |                  |                                         |                  |                  |
|--------------|------------------|------------------|-----------------------------------------|------------------|------------------|
| Номер строки | Номер автомобиля | Марка автомобиля | Признак иностранный/отечественный (И/О) | Признак=0 Bitmap | Признак=1 Bitmap |
| 1            | A001БВ           | FORD             | И                                       | 0                | 1                |
| 2            | Г001ДЕ           | LADA             | О                                       | 1                | 0                |
| 3            | Ж001ЗИ           | GAZ              | О                                       | 1                | 0                |
| 4            | К001ЛМ           | LEXUS            | И                                       | 0                | 1                |
| 5            | Н001ОП           | AUDI             | И                                       | 0                | 1                |

Рис. 9.40. Пример использования bitmap-индекса

В данном примере bitmap-индекс состоит из двух битовых карт. В первой битовой карте значение единица установлено для строк, в которых значение колонки «иностранный/отечественный» равно «О». Для строк, в которых значение колонки «иностранный/отечественный» равно «И», в первой битовой карте установлено зна-

Чтение ноль. Во второй битовой карте единица установлена для строк, в которых значение колонки «иностранный/отечественный» равно «И». Для строк, в которых значение колонки «иностранный/отечественный» равно «О», во второй битовой карте установлен ноль. Так же как и индексы на основе B-дерева, индексы на основе битовых карт могут быть составными, т.е. в основе одного bitmap-индекса используется нескольких колонок. Предпосылки к использованию составного bitmap-индекса такие же, как и при использовании bitmap-индекса на основе одной колонки: каждая колонка составного bitmap-индекса должна быть неизбирательна, т.е. данные в этих колонках должны быть неселективными. Пример использования составного bitmap-индекса представлен на рис. 9.41.

| Составной BITMAP INDEX |                  |                                         |                  |                  |               |            |               |                |
|------------------------|------------------|-----------------------------------------|------------------|------------------|---------------|------------|---------------|----------------|
| Номер строки           | Марка автомобиля | Признак иностранный/отечественный (И/О) | Признак О Bitmap | Признак И Bitmap | Москва Bitmap | Уфа Bitmap | Рязань Bitmap | Иркутск Bitmap |
| 1                      | FORD             | И                                       | 0                | 1                | 1             | 0          | 1             | 1              |
| 2                      | LADA             | О                                       | 1                | 0                | 1             | 1          | 1             | 0              |
| 3                      | GAZ              | О                                       | 1                | 0                | 0             | 1          | 1             | 1              |
| 4                      | LEXUS            | И                                       | 0                | 1                | 1             | 0          | 0             | 1              |
| 5                      | AUDI             | И                                       | 0                | 1                | 0             | 1          | 0             | 0              |

Рис. 9.41. Пример использования составного bitmap-индекса

Составные индексы на основе битовых карт можно эффективно комбинировать, используя бинарные операции AND, OR, NOT (рис. 9.42).

| A | B | A AND B | A OR B | NOT A |
|---|---|---------|--------|-------|
| 0 | 0 | 0       | 0      | 1     |
| 0 | 1 | 0       | 1      | 1     |
| 1 | 0 | 0       | 1      | 0     |
| 1 | 1 | 1       | 1      | 0     |

Рис. 9.42. Комбинирование bitmap-индексов бинарными операциями

### 9.7.1. Внутренняя структура bitmap-индекса

Индексы на основе битовых карт имеют древовидную структуру. Так же как и у индексов на основе B<sup>+</sup>-дерева, у bitmap-индексов выделяются два типа вершин: промежуточные вершины и листовые. Битовая карта и диапазоны RowId располагаются в листовых вершинах дерева bitmap-индекса. Рассмотрим структуру листовой вершины

Ниже приведены основные положения листовой вершины bitmap-индекса:

- Каждое значение bitmap-индекса представлено одной или несколькими битовыми картами.
- Каждая битовая карта указывает на диапазон значений RowId.
- Промежуточные вершины соответствуют внутренним узлам  $B^+$ -дерева.
- Каждая строка листовой вершины bitmap-индекса имеет структуру, представленную в табл. 9.1.

Таблица 9.1

## Структура строки листовой вершины bitmap-индекса

| Колонка | Описание                 |
|---------|--------------------------|
| 1       | Индексируемое значение   |
| 2       | Начальное значение RowId |
| 3       | Конечное значение RowId  |
| 4       | Значение битовой карты   |

Рассмотрим пример организации листовой вершины bitmap-индекса. В данном примере bitmap-индекс построен на колонке «Год выпуска» таблицы, содержащей информацию об автомобилях (рис. 9.43).

| Номер автомобиля | Марка автомобиля | Год выпуска | Цвет    |
|------------------|------------------|-------------|---------|
| A001БВ           | FORD             | 2012        | Синий   |
| Г001ДЕ           | LADA             | 2013        | Зеленый |
| Ж001ЗИ           | GAZ              | 2014        | Синий   |
| K001ЛМ           | LEXUS            | 2012        | Белый   |
| H001ОП           | AUDI             | 2012        | Белый   |
| A002БВ           | BMW              | 2013        | Черный  |
| Г002ДЕ           | MERCEDES         | 2012        | Красный |
| Ж002ЗИ           | KAMAZ            | 2012        | Желтый  |

BITMAP INDEX

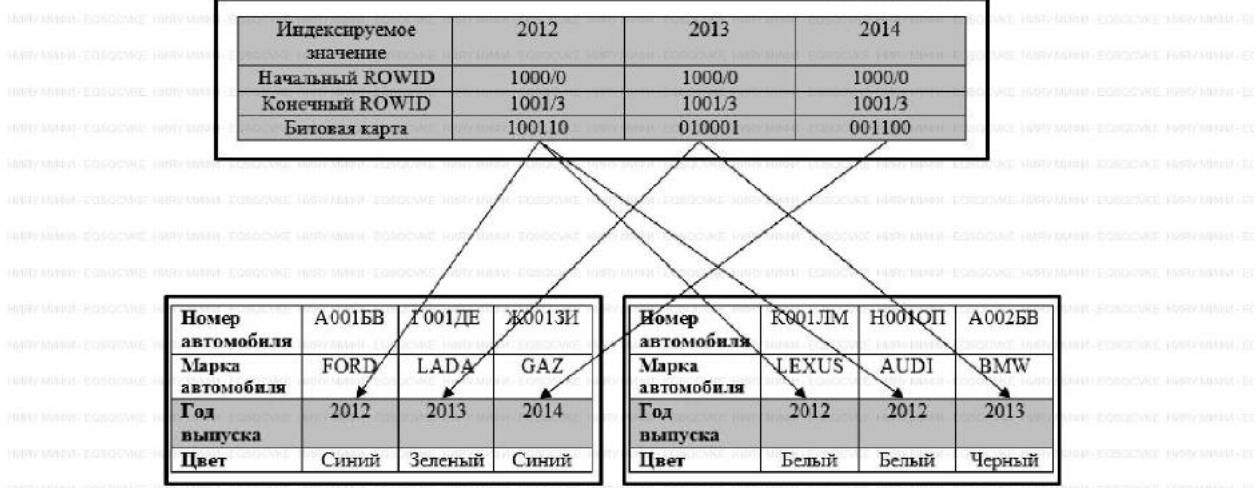
Рис. 9.43. Фрагмент таблицы «Автомобиль». Битовый индекс по колонке «Год выпуска»

Пример структуры листовой вершины bitmap-индекса на уровне блоков данных представлен на рис. 9.44.

Структура дерева индекса на основе битовых карт является сбалансированной, т.е. все вершины дерева равноправны, и листья объединяются в связанное последовательное упорядоченное множество. Отличие bitmap-индекса от индекса на основе  $B^+$ -дерева определяется структурой листовой вершины, которая включает в себя бито-

Ниже приведены примеры структуры листовой вершины bitmap-индекса для различных типов данных.

### BITMAP-индекс



Блок данных №1000

Блок данных №1001

Рис. 9.44. Пример структуры листовой вершины bitmap-индекса

ую карту для определенного диапазона RowId (рис. 9.45). Соответственно, операции поиска, добавления и удаления элементов в  $B^+$ -дереве применимы и для bitmap-индекса.

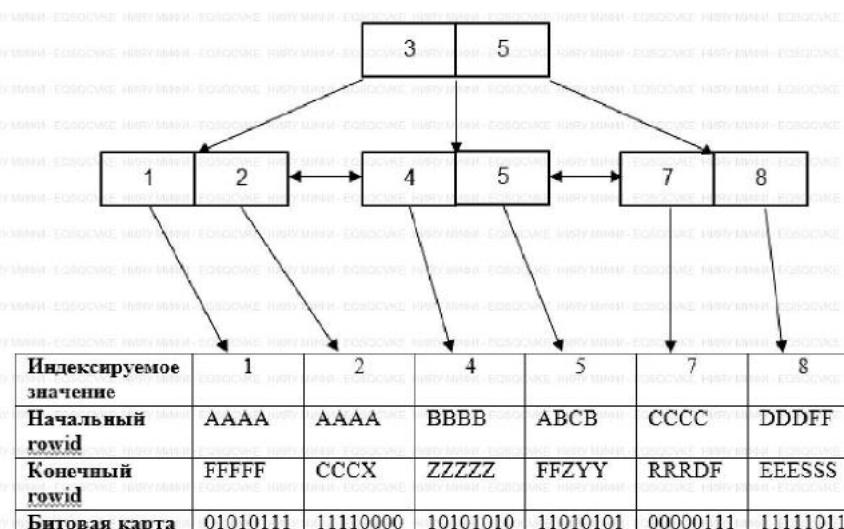


Рис. 9.45. Структура дерева и листовой вершины bitmap-индекса

Каждая строка листовой вершины состоит из четырех значений:

- индексируемое значение — значение, для которого построен bitmap-индекс;

- начальный RowId — значение RowId, которое определяет начальную границу диапазона, в рамках которого применяется битовая карта;
- конечный RowId — значение RowId, которое определяет конечную границу диапазона, в рамках которого применяется битовая карта;
- битовая карта — последовательность 1 или 0; для RowId устанавливается 1, если значение равно индексируемому значению, иначе устанавливается 0.

Рассмотрим пример реализации bitmap-индекса на основе примера, представленного на рис. 9.43–9.44. Добавим в представление на рис. 9.43 колонку RowId (табл. 9.2). Битовый индекс построен на колонке «Год выпуска».

Таблица 9.2

Фрагмент таблицы «Автомобиль» со значением RowId

| RowId | Номер автомобиля | Марка автомобиля | Год выпуска | Цвет    |
|-------|------------------|------------------|-------------|---------|
| AAA1  | A001БВ           | FORD             | 2012        | Синий   |
| AAA2  | Г001ДЕ           | LADA             | 2013        | Зеленый |
| AAA3  | Ж001ЗИ           | GAZ              | 2014        | Синий   |
| BBB1  | К001ЛМ           | LEXUS            | 2012        | Белый   |
| BBB2  | Н001ОП           | AUDI             | 2012        | Белый   |
| BBB3  | A002БВ           | BMW              | 2013        | Черный  |
| CCC1  | Г002ДЕ           | MERCEDES         | 2012        | Красный |
| CCC2  | Ж002ЗИ           | KAMAZ            | 2014        | Желтый  |

Соответственно, листовые вершины для каждого индексированного значения колонки «Год выпуска» будут иметь значения, которые представлены в табл. 9.3. Определение диапазонов RowId зависит от объемов данных и степени их распределения по блокам данных. В текущем примере демонстрируется основной принцип заполнения листовой вершины битового индекса.

Таблица 9.3

## Значение листовых вершин bitmap-индекса

| Индексируемое значение | 2012     | 2012     | 2013     | 2014     |
|------------------------|----------|----------|----------|----------|
| Начальный RowId        | AAA1     | BBB2     | AAA2     | AAA3     |
| Конечный RowId         | BBB1     | CCC2     | BBB3     | CCC2     |
| Битовая карта          | 10011000 | 00000010 | 01000100 | 00100001 |

### 9.7.2. Операция обновления bitmap-индекса

В силу специфики структуры bitmap-индекса не исключены предположения, что использование bitmap-индексов в транзакционных системах накладывает серьезные ограничения на их работу. Это связано с тем, что при выполнении DML-операций, например обновление (UPDATE) одного из значений, на котором построен bitmap-индекс, может возникнуть такая ситуация, при которой на все строки таблицы (даже на те, которые не попадают под условие обновления) будет установлена X-блокировка и, соответственно, доступ к таблице будет ограничен. Такие предположения связаны с тем, что битовые карты для некоторых значений могут применяться для диапазона RowId, охватывающего все строки таблицы. Этот случай представлен в табл. 9.3, где для значения «2012» битовая карта в совокупности применяется для RowId в диапазоне от первого до последнего. В реальности блокируется только часть данных, относящихся к конкретной битовой карте.

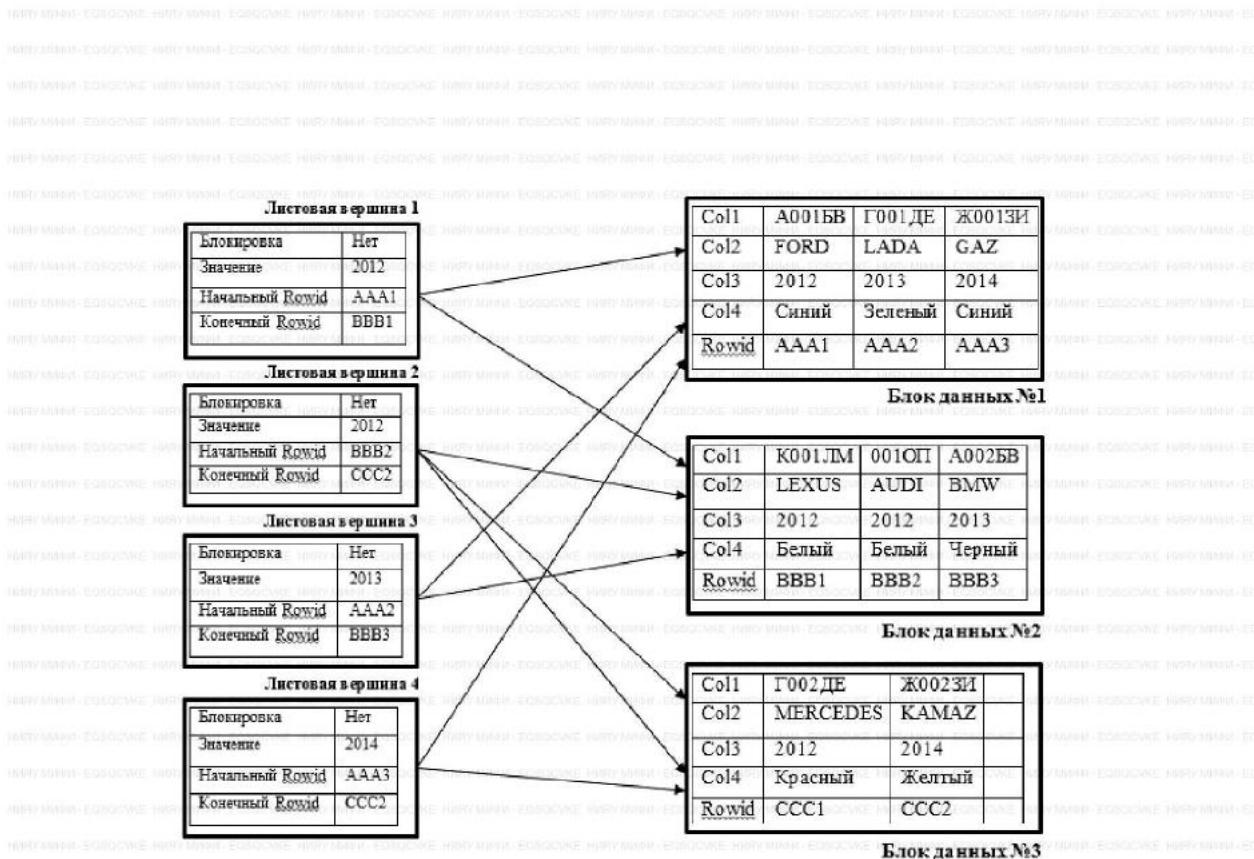
Как и в случае с  $B^+$ -деревом, обновление предполагает удаление старого и добавление нового значения. В процесс обновления вовлечены как минимум две части битовой карты, соответствующие удаляемому и добавляемому значениям. Сформулируем ряд положений, связанных с операцией обновления:

- операция обновления выполняется через удаление и последующую вставку: удаляется старое значение из части битовой карты, добавляется новое значение части битовой карты;
  - как минимум две части битовой карты вовлечены в процесс каждого обновления — для старого и нового значений соответственно;
  - когда bitmap-индекс обновляется, устанавливается блокировка на запись листовой вершины;
  - обновляемая запись в листовой вершине может содержать битовую карту, которая охватывает несколько строк в других блоках данных;
  - ни одна другая транзакция не может обновлять строку, вовлеченную в процесс обновления другой строкой до завершения транзакции.

Рассмотрим подробнее, как происходит процесс обновления на уровне блоков данных и листовых вершин bitmap-индекса. В качестве примера используем данные табл. 9.2 и 9.3. Введем обозначения для колонок табл. 9.2: «Номер автомобиля» — Col1, «Марка автомобиля» — Col2, «Год выпуска» — Col3, «Цвет» — Col4.

На рис. 9.46 представлены фрагменты листовых вершин (левая часть рис. 9.46) и фрагменты блоков данных (правая часть рис. 9.46.).

Битовый индекс построен для колонки «Год выпуска» (Col3). Листо-



**Рис. 9.46.** Распределение листовых вершин bitmap-индекса по блокам данных

Листовые вершины bitmap-индекса содержат диапазоны значений RowId, соответствующие строки которых располагаются в блоках данных. Так, например, для индексированного значения 2012 существуют две листовые вершины, которые содержат два разных диапазона значений RowId. Листовая вершина 1 указывает на данные в блоке 1 и 2, так как строки из диапазона значений RowId попадают в блоки данных № 1 и № 2, и в них присутствует индексированное значение 2012. Листовая вершина 2 указывает на данные в блоке 2 и 3, так как строки из диапазона значений RowId попадают в блоки данных № 2 и № 3, и в них присутствует индексированное значение 2012 и т.д.

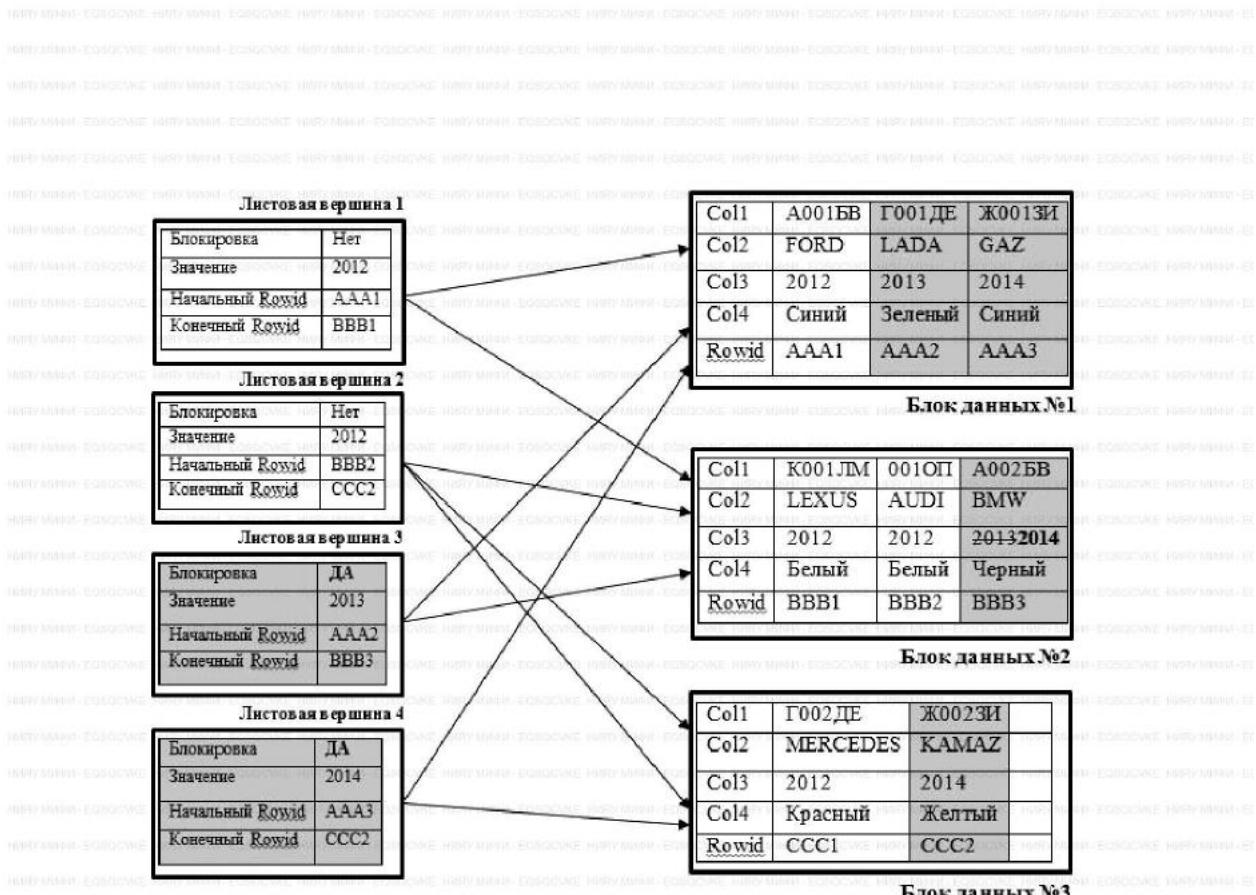
Выполним обновление одной из строк таблицы:

UPDATE Автомобиль SET Col3='2014' WHERE Col1='A002БВ'

Таким образом, в процессе обновления чувствуют два значения: 2013 – старое, удаляемое значение и 2014 – новое значение. На листовые вершины нового и старого значений устанавливается блокировка на запись (рис. 9.47).

Соответственно, в результате обновления блокировка на запись устанавливается на все строки, указанные в диапазоне значений RowId листовой вершины, вовлеченнной в процесс обновления. Рассматривая пример на рис. 9.47, блокировки на запись для операции обновления

248



**Рис. 9.47.** Блокировка строк при обновлении данных bitmap-индекса

**UPDATE Автомобиль SET Col3=2014 WHERE Col1='A002БВ'**

будут установлены следующим образом:

1) обновляемая строка — третья строка в блоке данных № 2, соответствующая условию  $Col1='A002БВ'$ ;

2) обновляемая колонка —  $Col3$ , текущее значение которой равно 2013;

3) листовая вершина bitmap-индекса для значения 2013 в блоке данных № 2 — это листовая вершина № 3;

4) блокировка на запись устанавливается не только на обновляемую строку ( $RowId = BBB3$ ), но и на строки, которые находятся в диапазоне значений  $RowId$  листовой вершины № 3. То есть в процессе обновления вовлечены строки, не имеющие явного отношения к обновляемой строке, но присутствующие в диапазоне  $RowId$  листовой вершины bitmap-индекса обновляемой строки. В примере на рис. 9.47 помимо обновляемой строки с  $RowId$ , равным BBB3, блокировка на запись устанавливается для строк с  $RowId$ , равными AAA2 и AAA3;

5) для нового значения 2014 ситуация аналогична приведенной в п. 4;

6) листовая вершина bitmap-индекса для значения 2014 — это листовая вершина № 4. Соответственно, все строки в диапазоне RowId, которые указаны в листовой вершине № 4, вовлечены в процесс обновления, и для них устанавливается блокировка на запись.

Таким образом, рассмотренный пример показывает, что обновление данных, входящих в состав bitmap-индекса, затрагивает не относящиеся к текущему обновлению строки в соответствии с их представлением в листовой вершине bitmap-индекса, но не блокирует таблицу полностью.

## 9.8. СРАВНЕНИЕ МЕТОДОВ ИНДЕКСИРОВАНИЯ

Все СУБД используют индексы, организованные по принципу  $B^+$ -дерева индексов (хотя и имеют свои особенности в организации работы с ними), но не все СУБД используют хэш-индексы и bitmap-индексы. Например, в MS SQL Server используются только индексы на основе  $B^+$ -дерева, тогда как в Oracle доступны все три типа индексов. В случае когда СУБД предоставляет возможность использования разных типов индексов, следует руководствоваться какими-то рекомендациями, позволяющими выбрать тот или иной конкретный способ организации индекса.

Проведем сравнение разных методов индексирования на основе следующих двух критериев:

- возможно ли, используя соответствующий подход, создавать не-уникальные индексы;
  - существуют ли рекомендации по использованию тех или иных методов индексирования.

В соответствии с рассмотренными выше принципами организации методов индексирования можно утверждать, что хэш-индексы целесообразно создавать только на уникальных атрибутах, в то время как bitmap-индексы, напротив, целесообразно создавать на неуникальных атрибутах, значения которых повторяются достаточно часто. Индексы на основе  $B^+$ -дерева могут быть созданы как на уникальных, так и на неуникальных атрибутах. В соответствии с этим рассмотрим сначала особенности создания и использования уникальных индексов (хэш-индекс и  $B^+$ -дерево индексов), а затем — неуникальных индексов ( $B^+$ -дерево индексов и bitmap-индексы).

## Сравнение В<sup>+</sup>-дерева индексов и хэш-индексов

$B^+$ -деревья используют индексы, которые существуют как объекты базы данных, занимают дисковое пространство и содержат в себе указатели на реальные строки таблицы. В общем случае  $B^+$ -дерево не изменяет способ хранения самой таблицы и может быть создано для таблицы, в которую уже включены какие-то строки данных

(т.е. это некоторая надстройка над таблицей). Поэтому индекс на основе  $B^+$ -дерева может быть создан уже после того, как создана таблица. Обычно, когда создается таблица (предложение **CREATE TABLE**) и в ней задаются ограничения **PRIMARY KEY** и **UNIQUE**, СУБД автоматически создает на колонках первичного и альтернативных ключей таблицы уникальные индексы на основе  $B^+$ -дерева.

Хэш-индексы не представляются значимыми объектами, отображаются на реальные строки таблицы, причем это отображение существенно зависит от размеров таблицы, а также влияют на способ хранения строк таблицы и могут его изменить, поэтому хэш-индекс должен быть создан до включения какой-либо строки в таблицу. Поэтому, если для какой-либо таблицы нужно использовать хэш-индекс, сначала должен быть создан этот индекс, а затем на его основе создается таблица, использующая этот индекс. Например, в СУБД Oracle это можно сделать так.

Сначала, с помощью следующего предложения, создается кластерный индекс:

**CREATE CLUSTER** *hd* (  
    *did* NUMBER (5,0) **SIZE** 1K — размер строк с одним и тем же значением индекса (бакета),  
    **HASHKEYS** 200 — число различных значений хэш-индекса  
    **HASH IS** *did* — на каком атрибуте создается хэш-индекс

Здесь **SIZE** — указывает количество пространства в байтах, зарезервированного для хранения строк с одним и тем же хэш-значением (т.е. размер бакета). Это пространство определяет максимальное количество хэш-значений, сохраняемых в блоке;

**HASHKEYS** — указывает количество хэш-значений для хэш-кластера. Хэш-значение для строки — это значение, возвращаемое хэш-функцией. Должно обязательно указываться при создании хэш-кластера (для хэш-таблиц). При создании хэш-кластера СУБД сразу же выделяет пространство для кластера на основе значений параметров **SIZE** и **HASHKEYS**:

**HASH IS** *expr* — указывает выражение, которое будет использоваться хэш-функцией для этого хэш-кластера. Выражение должно содержать по крайней мере одну колонку.

После создания кластерного хэш-индекса можно создавать таблицы, использующие этот индекс, например:

```
CREATE TABLE tab (
 mdid NUMBER (5) NOT NULL PRIMARY KEY,
 ... -- другие колонки таблицы
) CLUSTER hd(mdid) -- на каком хэш-индексе строится таб-
лика
```

В соответствии с этим можно сформулировать следующие важные отличия хэш-индексов от индексов  $B^+$ -дерева.

1.  $B^+$ -дерево имеет реальные объекты-индексы. С хэш-таблицей реальные объекты-индексы не связаны.
  2.  $B^+$ -дерево не влияет на структуру таблицы, особенности ее отображения в памяти; доступ к строкам таблицы осуществляется через дерево индексов. Хэш-таблица имеет специальную организацию, влияющую на доступ к ее строкам.
  3. В  $B^+$ -дереве строки таблицы упорядочены (на уровне листьев) по ключу. В хэш-таблице никакой речи об упорядоченности строк не может быть, поэтому выборки, требующие упорядоченности (**ORDER BY**), очень неэффективны.
  4. При поиске в  $B^+$ -дереве определены понятия «меньше» и «больше» для ключей (выбирается соответствующее поддерево или индекс  $B^+$ -дерева). Для хэш-таблиц такие понятия отсутствуют, эффективный поиск осуществляется только по условию «равно». Поиск по условиям «меньше» и «больше» реализуется с помощью полного просмотра (сканирования) всей таблицы.
  5. Пространство для таблицы, использующей  $B^+$ -дерево, выделяется в процессе вставки новых строк. Пространство для хэш-таблиц должно быть выделено предварительно, при создании таблицы. Выделение недостаточного объема памяти приводит к низкой эффективности доступа. Если выделить слишком много памяти, можно столкнуться с неэффективным использованием пространства.

Отсюда можно дать следующие рекомендации:

  - если размер таблицы сильно изменяется — не хэш-таблица;
  - если часто организуется поиск по критериям «меньше» и/или «больше» — не хэш-таблица;
  - если используются справочники (мало изменяемые таблицы, поиск в них по условию «равно») — хэш-таблицы.

## Сравнение B<sup>+</sup>-дерева индексов и bitmap-индексов

Bitmap-индексы, как и индексы на основе B<sup>+</sup>-дерева, существуют как объекты базы данных, занимают дисковое пространство и содержат в себе указатели на реальные строки таблицы. Так же как и индекс на основе B<sup>+</sup>-дерева, битовые индексы не изменяют способы хранения самой таблицы и могут быть созданы для таблицы, в которую включены строки данных. Bitmap-индексы создаются достаточно быстро и занимают относительно мало места при условии их использования на неселективных данных.

Отличия индексов на основе  $B^+$ -дерева и индексов на топовых карт можно сформулировать следующим образом.

1. Битовые индексы, в отличие от индексов на основе  $B^+$ -дерева, не могут быть использованы для обеспечения ограничения уникальности.
  2. Битовые индексы достаточно быстро создаются и являются компактными при условии их использования на неселективных данных.
  3. Использование битовых индексов в транзакционных системах с большим числом DML-операций снижает степень коллективной работы с данными, не вовлеченными в процесс изменения, но расположенными в тех же блоках данных, что и изменяемые.
  4. Комбинирование нескольких битовых индексов эффективно, особенно при поиске по условию OR.

## Вопросы

- Укажите, какие компоненты двухуровневой системы организации доступа к данным обеспечивают последовательный и произвольный доступ к данным и почему.
  - Сформулируйте основные отличия бинарных и многоходовых деревьев, а также их достоинства и недостатки.
  - Выполните операцию вставки в В-дерево степени 2 следующей последовательности ключей: 12, 8, 32, 20, 15, 5, 6, 4, 45, 80, 18.
  - Выполните операцию удаления ключа 18 из В-дерева степени 2, полученного в п. 3.
  - Сформулируйте основные отличия В-дерева и  $B^+$ -дерева.
  - Выполните операцию вставки в  $B^+$ -дерево степени 2 той же последовательности ключей, что и в п. 3.
  - Выполните операцию удаления ключа 18 из построенного в п. 6  $B^+$ -дерева степени 2.
  - Сформулируйте сравнительные характеристики методов хэширования. Определите их достоинства и недостатки.
  - Дайте развернутое объяснение и приведите пример, почему использование bitmap-индексов невыгодно в системах с большим числом коротких транзакций DML-операций.

ГЛАВА 10

# ОСНОВЫ ПОСТРОЕНИЯ И АНАЛИЗА ПЛАНОВ ВЫПОЛНЕНИЯ SQL-ЗАПРОСОВ

В данной главе рассматриваются возможные подходы, связанные с выявлением проблем производительности SQL-запросов на основе анализа плана выполнения запроса, который строится оптимизатором SQL-запросов конкретной СУБД. Рассматриваются ключевые компоненты оптимизатора SQL-запросов, интерпретация основных операций плана выполнения SQL-запросов, а также практические примеры анализа проблем производительности SQL-запросов и возможные варианты их решения.

Материал, представленный в данной главе, изложен с учетом уже существующих навыков написания SQL-запросов, понимания методов доступа к данным, а также представления об архитектуре современных реляционных СУБД.

Язык SQL в силу своей декларативности представляет возможность получить один и тот же результат множеством различных способов. То, какие подходы и методы доступа к данным ядро СУБД будет использовать для выполнения SQL-запроса, определяет степень вовлеченности ресурсов системы, в рамках которой развернута используемая СУБД, в процесс исполнения каждой SQL-инструкции. Предложения языка SQL, используемые для решения тех или иных задач, на уровне операционной системы представляются набором процессов, которые выполняют операции чтения и (или) записи. Ресурсы даже самой производительной системы ограничены. Особенно этот факт критически важен для высоконагруженных транзакционных систем, которые характеризуются множеством коротких, но достаточно частых операций. Соответственно, неэффективное использование ресурсов системы, причиной которого часто являются, в том числе, и недостаточно оптимально с точки зрения производительности написанные SQL-запросы, часто приводит к снижению производительности приложений, работающих с базой данных. Большинство современных СУБД имеют в своем составе набор средств, направленных на улучшение производительности выполнения SQL-инструкций за счет использования различных оценочных и статистических методов. Однако, ввиду тех или иных причин, эти методы не всегда используют оптимальное с точки зрения производительности решение, позволяющее, с одной стороны, достаточно эффективно использовать ресурсы системы, а с другой — предоставлять требуемый результат SQL-инструкции в пределах ожидаемого промежутка времени. В некоторых случаях возможно даже ухудше-

**ние производительности как результат, например, ошибочно выбранного метода доступа к данным — из-за того, что информация об объектах и (или) данных, используемых в текущей SQL-инструкции, некорректна или устарела. В связи с этим можно отметить, что не только знание синтаксиса и сложных конструкций языка SQL обеспечивает стабильность и устойчивость работы приложений баз данных, но и понимание того, как СУБД выполняет или будет выполнять написанное SQL-предложение. Понимание механизмов обработки SQL-предложений и возможность анализировать предложенные СУБД путем доступа к данным значительно расширяют для разработчика комплекс мер, позволяющих обеспечить СУБД условия, в которых будут приниматься оптимальные решения с точки зрения производительности, для более эффективного выполнения SQL-предложений.**

В состав большинства современных СУБД входят программные компоненты, отвечающие за подготовку и реализацию SQL-запросов. Работа любого SQL-запроса предполагает, что сервер СУБД выполняет последовательность действий (шагов), направленных на получение результата, описанного в декларативной части SQL-запроса. Порядок действий и методов, которые должен выполнить сервер СУБД для получения доступа к требуемым данным, определяется одним из встроенных в СУБД программных компонентов — оптимизатором запроса (query optimizer), или просто оптимизатором.

**Определение.** Оптимизатор запроса (оптимизатор) — встроенное в

СУБД программное обеспечение, которое определяет наиболее эффективный способ выполнения SQL-выражения.

**Определение.** План выполнения запроса — последовательность шагов или инструкций СУБД, необходимых для выполнения SQL-выражения.

**Определение.** Стоимость выполнения запроса — наилучшая оценка времени, необходимого для выполнения оператора, полученная оптимизатором [14].

Оптимизатор разрабатывает множество потенциальных планов выполнения SQL-запросов, оценивается стоимость каждого плана, сравниваются затраты на их выполнение и затем выбирается наиболее подходящий с точки зрения оптимизатора план выполнения запроса. Выбор оптимального плана выполнения запроса зависит от множества условий, включая объем предполагаемого набора данных, расположение данных, а также структуры доступа к данным. Оптимизатор определяет наилучший с его точки зрения план выполнения SQL-запроса, исследуя различные методы доступа к данным, такие как полное сканирование (просмотр) таблицы, или использование

**индекса(ов), различные типы соединения, такие как вложенные циклы, хэш-соединения, сортировка и слияние.**

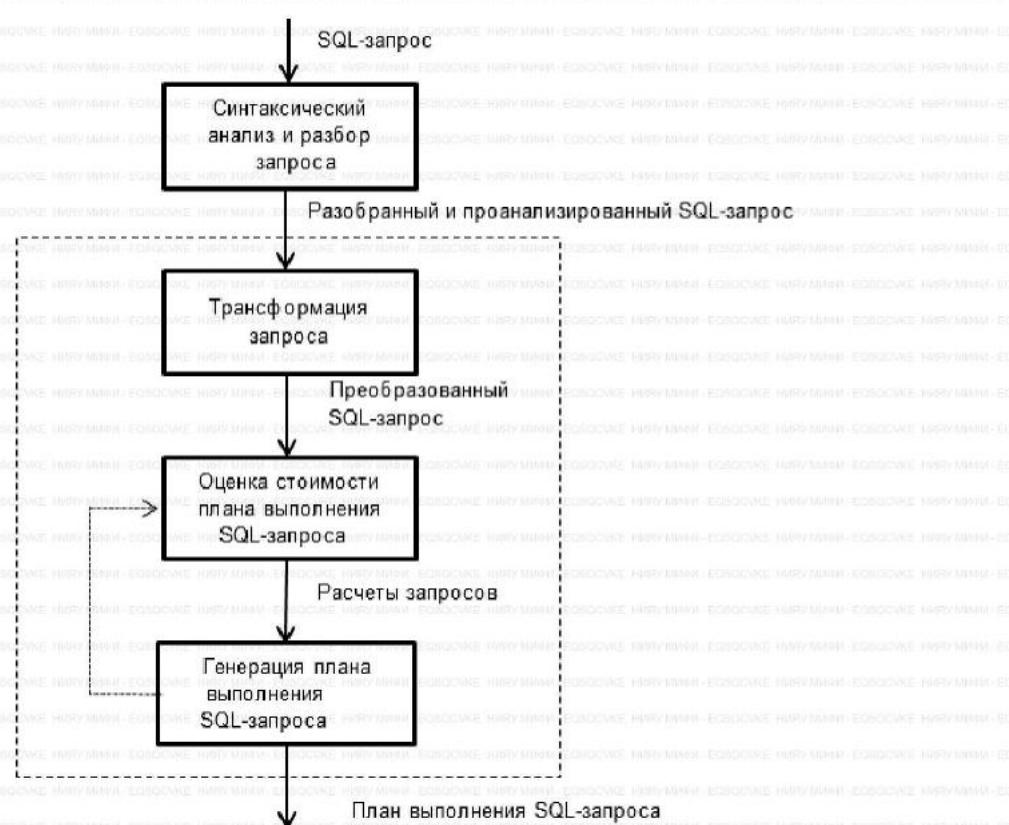
В процессе своей работы оптимизатор решает следующие задачи:

- преобразование SQL-операторов;

- преобразование SQL-операторов;
  - выбор способа оптимизации;
  - выбор путей доступа;
  - выбор порядка соединений таблиц;
  - выбор метода соединений таблиц;
  - определение наиболее эффективного плана выполнения;
  - вычисление выражений.

Процессу работы оптимизатора предшествует синтаксический анализ запроса, результат которого гарантирует, что запрос имеет допустимый синтаксис. Далее в зависимости от СУБД SQL-запрос может быть преобразован, например, в блочную или древовидную структуру, которая затем поступает на вход оптимизатора.

Основные компоненты оптимизатора представлены на рис. 10.1.



**Рис. 10.1.** Основные компоненты оптимизатора

Ниже приведены примеры оптимизации запросов на языке SQL:

```

SELECT * FROM owner_auto WHERE mark='Volvo' OR owner_id=125842

```

Таким образом, оптимизатор выполняет следующие операции.

**1. Трансформация запроса.** Поскольку в языке SQL один и тот же результат может быть достигнут множеством различных способов, для некоторых запросов оказывается более предпочтительным переписать первоначальный вариант SQL-запроса в семантически эквивалентный вариант с меньшими затратами ресурсов СУБД. Пример трансформации запроса, представленный на рис. 10.2, показывает, что объединение результатов двух запросов с помощью теоретико-множественной операции объединения (**UNION**) может оказаться более предпочтительным с точки зрения уменьшения стоимости запроса, чем один запрос с использованием в условии поиска операции **OR**.

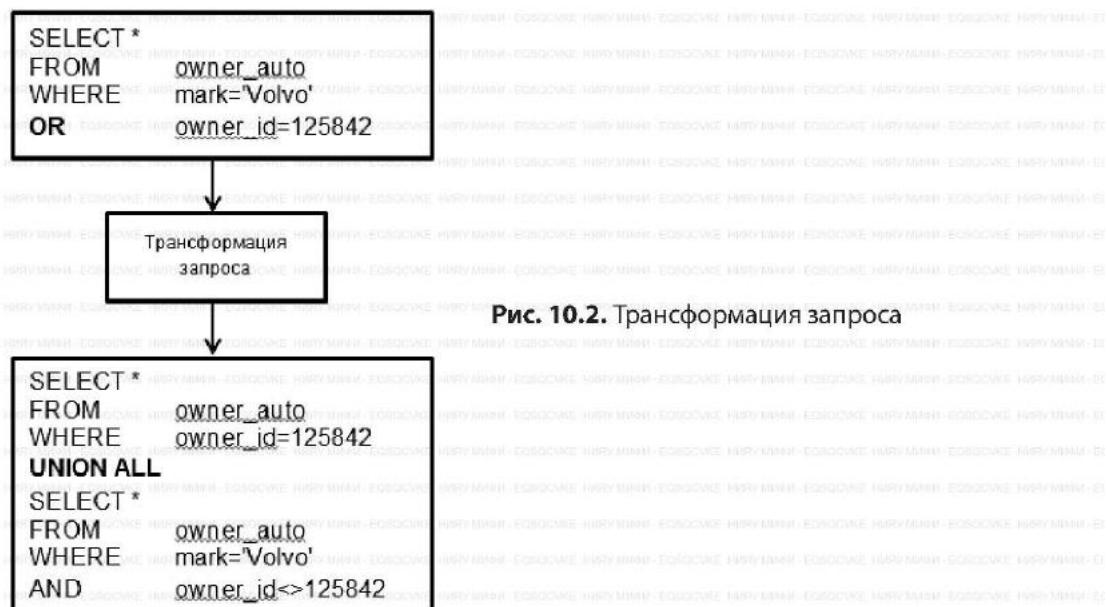


Рис. 10.2. Трансформация запроса

**2. Оценка стоимости плана выполнения запроса.** Для каждого потенциального плана выполнения запроса оптимизатор оценивает стоимость его выполнения. Для оценки применяется различная информация о требуемых ресурсах, таких как дисковый ввод/вывод, загрузка центрального процессора, используемая оперативная память. Кроме информации о ресурсах системы, для оценки стоимости текущего плана выполнения SQL-запроса используются два типа мер: селективность (избирательность) и кардинальность. В общем случае селективность представляется двумя типами: селективностью таблицы и селективностью индекса.

**Определение.** Селективность (избирательность) таблицы — значение

от нуля до единицы, представляющее долю строк таблицы, удовлетворяющих определенному условию выбора. Селективность таблицы связана с условием выбора строк или предикатами. Говорят, что предикат является избирательным, когда его селективность приближается к нулю, и неизбирательным, или менее селективным, когда его селективность приближается к единице.

Например, если общее число строк в таблице равно 1000, а при выполнении предложения **SELECT** в соответствии с указанным в нем условием поиска в качестве результата будет возвращено только 12 строк, селективность таблицы для такого запроса будет равна  $0,012 = 12/1000$ .

**Определение.** Селективность (избирательность) индекса — значение

**Селективность (избирательность) индекса** – значение от нуля до единицы, представляющее отношение количества уникальных значений индексируемых колонок или колонки к общему числу строк таблицы. То есть селективность индекса показывает долю строк от общего числа строк в таблице, которое приходится на одно значение индекса. Индекс называют селективным, или избирательным, если его селективность стремится к единице. Колонка или колонки первичного ключа обеспечивают высокую селективность индекса, построенного на их основе.

Например, если общее число строк в таблице равно 1000 и одна из индексируемых колонок содержит 800 уникальных значений, тогда селективность этого индекса будет равна  $0,8 = 800/1000$ . Данный индекс является избирательным, поскольку одному значению индекса соответствует  $1,25 = 1000/800$  строк таблицы, т.е. использование данного индекса не приведет к чрезвычайно избыточному чтению данных.

**Определение.** Кардинальность — количество строк, возвращаемых

**Картинаность** — количество строк, возвращаемых после каждой операции плана выполнения запроса.

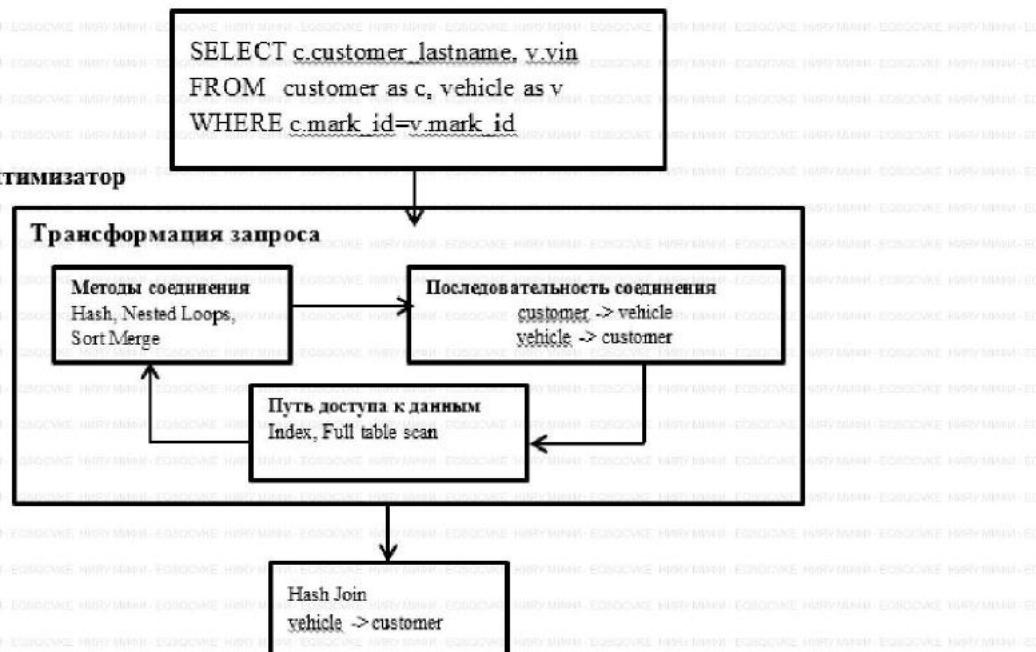
Значение кардинальности равно произведению селективности на количество обработанных строк:

**Кардинальность = Селективность × Количество обработанных строк**

3. Генератор выполняет исследование разных планов выполнения SQL-запроса, используя различные пути доступа к данным, методы и порядок соединения. Оптимизатор выбирает план с наименьшими стоимостными затратами. Так, на рис. 10.3 представлен пример выбора оптимизатором плана выполнения запроса с низкой стоимостью: в этом плане наиболее предпочтительным является хэширование.

**соединение таблиц (HASH JOIN), а в качестве внешней (зондируемой) таблицы используется таблица customer.**

соединение таблиц (**HASH JOIN**), а в качестве внешней (зондируемой) таблицы используется таблица *customer*.



**Рис. 10.3.** Генератор плана выполнения SQL-запроса

## 10.1. МЕТОДЫ СОЕДИНЕНИЯ ИСТОЧНИКОВ ДАННЫХ SQL-ЗАПРОСА

Всякий раз, когда в конструкции **FROM** SQL-запроса указываются имена нескольких (более одного) источников данных — таблиц, представлений или других, СУБД выполняет их соединение. Оптимизатор на этапе трансформации определяет методы соединения источников данных SQL-запроса. Как правило, соединение источников данных представляется в виде древовидной структуры. Цель оптимизатора — свести к минимуму процесс обработки данных, отфильтровывая ненужные данные как можно скорее. На этапе анализа оптимизатор должен определить, какая операция соединения пары таблиц является наиболее эффективной. Чтобы выбрать метод соединения, оптимизатор должен оценить стоимость выполнения каждого соединения. В большинстве СУБД используются четыре метода соединения: соединение с использованием вложенных циклов (**NESTED LOOP JOIN**), хэш-соединение (**HASH JOIN**), соединение сор-

Ниже минимумы, которые определяют минимальные значения для каждого столбца в таблице. Ключевые слова `MIN` и `MAX` указывают на то, что возвращаются максимальные и минимальные значения для каждого столбца. Ключевое слово `ALL` указывает на то, что возвращаются все строки из таблицы.

**Соединение (JOIN).** Соединение — это оператор SQL, который объединяет две или более таблиц в одну. Оператор `JOIN` имеет следующий синтаксис:

```
SELECT * FROM table1 JOIN table2 ON table1.common_column = table2.common_column;
```

где `table1` и `table2` — это имена таблиц, а `common_column` — это имя общего столбца, по которому происходит соединение.

**Соединение с фильтрацией (WHERE).** Соединение с фильтрацией — это оператор SQL, который объединяет две или более таблиц в одну, но только для тех строк, которые удовлетворяют заданным условиям. Оператор `WHERE` имеет следующий синтаксис:

```
SELECT * FROM table1 JOIN table2 ON table1.common_column = table2.common_column WHERE condition;
```

где `condition` — это условие, которое определяет, какие строки должны быть включены в результат.

**Соединение с группировкой (GROUP BY).** Соединение с группировкой — это оператор SQL, который объединяет две или более таблиц в одну, но только для тех строк, которые относятся к определенным группам. Оператор `GROUP BY` имеет следующий синтаксис:

```
SELECT * FROM table1 JOIN table2 ON table1.common_column = table2.common_column GROUP BY group_column;
```

где `group_column` — это имя столбца, по которому производится группировка.

**Соединение с сортировкой (ORDER BY).** Соединение с сортировкой — это оператор SQL, который объединяет две или более таблиц в одну, но только для тех строк, которые отсортированы по определенным столбцам. Оператор `ORDER BY` имеет следующий синтаксис:

```
SELECT * FROM table1 JOIN table2 ON table1.common_column = table2.common_column ORDER BY sort_column;
```

где `sort_column` — это имя столбца, по которому производится сортировка.

**Соединение с ограничением (LIMIT).** Соединение с ограничением — это оператор SQL, который объединяет две или более таблиц в одну, но только для тех строк, количество которых ограничено. Оператор `LIMIT` имеет следующий синтаксис:

```
SELECT * FROM table1 JOIN table2 ON table1.common_column = table2.common_column LIMIT number;
```

где `number` — это количество строк, которое нужно вернуть.

### тировкой и слиянием (**MERGE JOIN**), перекрестное соединение (**CROSS JOIN** или **CARTESIAN JOIN**).

Концептуально все методы соединения в конкретный момент времени предполагают наличие двух источников. Как правило, один источник является ведущим, или внешним, второй — ведомым, или внутренним (рис. 10.4). Колонки источников данных, по которым выполняется их соединение, определяются как атрибуты соединения.



Рис. 10.4. Генератор плана выполнения SQL-запроса

#### 10.1.1. Соединение с использованием вложенных циклов (**NESTED LOOPS JOIN**)

Основной идеей данного метода соединения является то, что для каждой строки внешнего набора данных выполняется поиск соответствующих данных во внутреннем наборе данных (рис. 10.5).

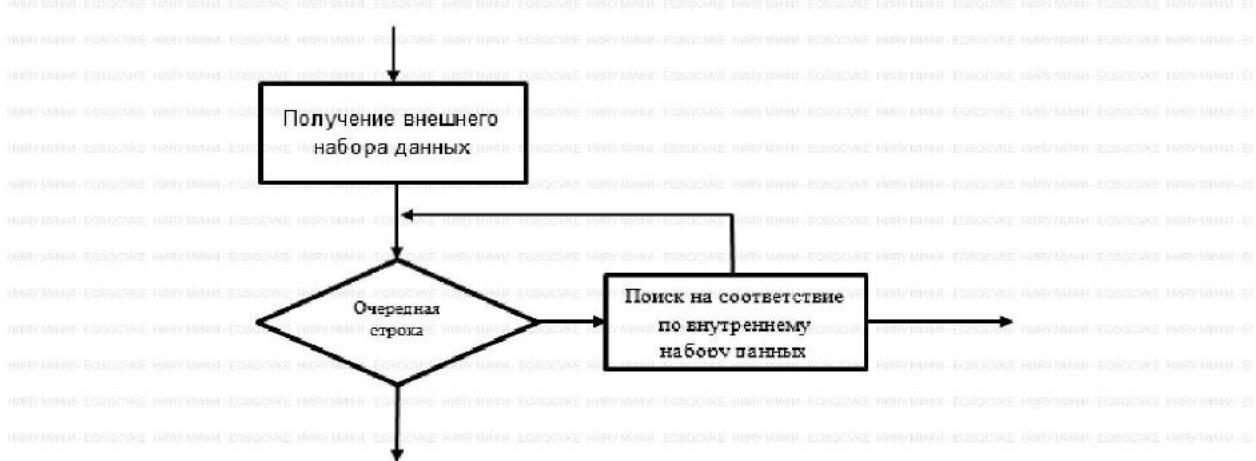


Рис. 10.5. Общий алгоритм работы соединения с использованием вложенных циклов

**Соединение двух источников данных с использованием вложенных циклов** включает в себя следующие этапы.

1. Определяется ведущая таблица. Применяется условие выбора необходимых строк ведущей таблицы.
2. Результирующий набор строк ведущей таблицы передается для дальнейшей обработки.
3. Каждая запись ведущей таблицы сравнивается по заданному в запросе условию отбора со всеми строками первой присоединяемой (ведомой) таблицы.
4. Строки, не удовлетворяющие условию отбора, отбрасываются.
5. В зависимости от количества таблиц или источников, вовлеченных в процесс соединения, строки, удовлетворяющие условию отбора, передаются дальше для проверки на соответствие с другими присоединяемыми таблицами.

Рассмотрим следующий запрос:

```
SELECT t1.x, t1.y
```

```
SELECT city, city
 FROM table1 t1, table2 t2
 WHERE t1.z = 'ABC'
 AND t1.id = t2.id
```

Используя некоторый псевдокод для описания алгоритма выполнения запроса, данный запрос можно представить следующим образом:

```
FOR t1row IN (select * from t1 where t1.z='ABC') LOOP
-- 1
 FOR t2row IN (select * from t2 которые совпадают с
текущей строкой t1row таблицы t1) LOOP -- 2
 вывести текущие значения строк t1row и t2row таблиц
t1 и t2
 END LOOP;
```

Данный код демонстрирует две циклические конструкции. Внешний цикл (1) читает по одной строке из таблицы table1, удовлетвроящей условию отбора t1.z = 'ABC'. Внутренний цикл (2) выполняет поиск строк в таблице table2, соответствующих текущей строке таблицы table1.

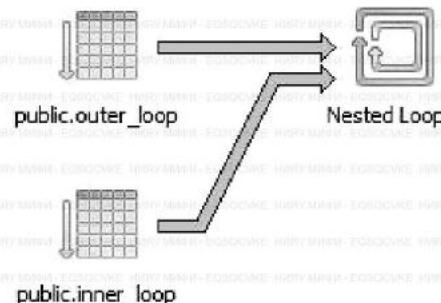
В текстовой интерпретации плана выполнения SQL-запроса соединение с использованием вложенных циклов представляется следующим образом:

```
NESTED LOOPS
outer_loop
inner loop
```

где `outer_loop` — определяет ведущую таблицу; `inner_loop` — определяет ведомую таблицу.

Графическая интерпретация (на примере СУБД PostgreSQL) фрагмента плана выполнения SQL-запроса, содержащего соединение с использованием вложенных циклов, представлена на рис. 10.6.

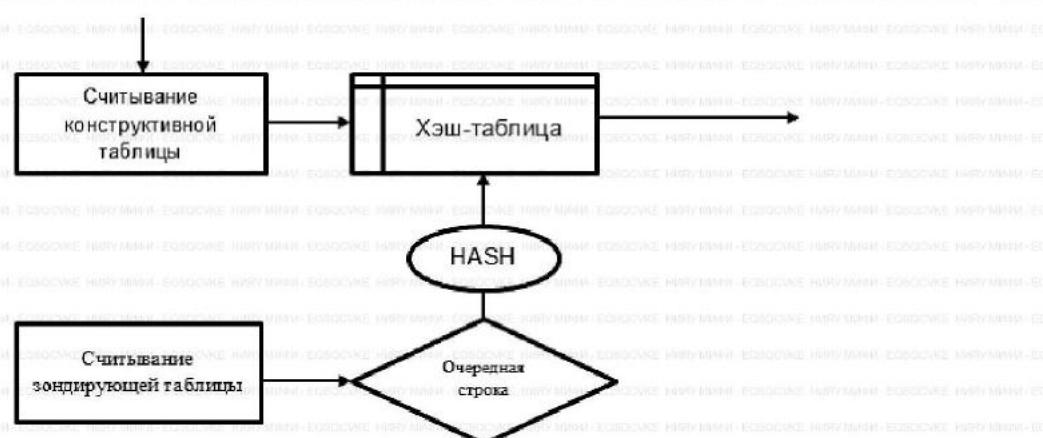
Графическая интерпретация (на примере СУБД PostgreSQL) фрагмента плана выполнения SQL-запроса, содержащего соединение с использованием вложенных циклов, представлена на рис. 10.6.



**Рис. 10.6.** Соединение с использованием вложенных циклов. Фрагмент плана запроса

#### 10.1.2. Хэш-соединение (HASH JOIN)

Метод соединения с использованием хэширования предполагает последовательный перебор всех строк таблиц, участвующих в соединении и удовлетворяющих условию поиска, заданному в конструкции **WHERE** (если она присутствует). В основе данного метода соединения используется идея зондирования (исследования) хэш-таблицы, сформированной на основе одной из таблиц, участвующих в соединении (рис. 10.7). Таблица, на основе которой в памяти формируется область хэширования, называется конструктивной, или создающей. Вторую таблицу называют зондирующей (так как для каждой ее строки выполняется зондирование конструктивной хэш-таблицы), или пробной. Оптимизатор запросов распределяет роли



**Рис. 10.7.** Общий алгоритм выполнения хэш-соединения

таким образом, что в качестве конструктивной выбирается таблица с меньшим количеством строк. Если обе таблицы имеют одинаковый размер, оптимизатор распределяет роли между таблицами произвольным образом.

**Соединение методом хэширования двух источников данных** включает в себя следующие этапы.

1. Таблица с меньшим количеством строк определяется как конструктивная.
2. Таблица с большим количеством строк определяется как зондирующая.
3. Формируется хэш на атрибуте соединения и отображается в области хэширования (как упоминалось выше, атрибут соединения — это колонки таблицы, по которым выполняется операция соединения).
4. Область хэширования хранит небольшой результирующий набор.
5. Для каждой строки зондирующей таблицы выполняется та же функция хэширования на атрибуте соединения.
6. Результат работы хэш-функции используется для перехода в соответствующую область хэширования конструктивной таблицы.
7. Выполняется поиск на соответствие в небольшой части области хэширования.
8. Результат успешного поиска возвращается или отправляется на следующее соединение.

Рассмотрим следующий запрос:

```
SELECT t1.x, t1.y
 FROM table1 t1, table2 t2
```

```
WHERE t1.z = 'ABC'
```

```
AND t1.id = t2.id
```

Соединение с использованием хэширования состоит из двух этапов: построение хэш-таблицы и зондирование.

Используя псевдокод, рассмотрим каждый из этих этапов.

#### Этап построения хэш-таблицы

На основе конструктивной (меньшей) таблицы формируется хэш-таблица по атрибутам соединения в области хэширования:

```
FOR t1row_small_table IN (select * from t1 where
t1.z='ABC')
 LOOP
 Номер_Бакета:=HASH(t1row_small_table.join_colmn)
 ДОБАВИТЬ_В_ХЭШ_ТАБЛИЦУ(Номер_Бакета,
 t1row_small_table)
```

```
END LOOP
```

Ниже приведен фрагмент плана выполнения SQL-запроса с использованием хэширования:

#### Этап зондирования

Для каждой строки зондирующей (большой) таблицы вычисляется хэш-функция по атрибуту соединения. Затем выполняется поиск соответствия в хэш-таблице. Совпадение выводится в качестве результата:

```
FOR t2row_large_table IN (select * from t2)
LOOP
```

```
 Номер_Бакета:=HASH(t2row_large_table. join_colmn)
 t1row_small_table = ПРОСМОТР_ХЭШ_ТАБЛИЦЫ(номер_
 Бакета,t2row_large_table.join_colmn)
```

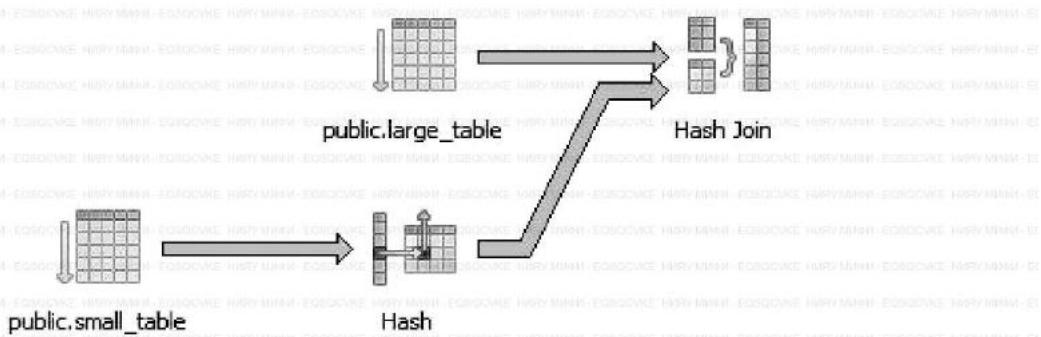
```
 IF t1row_small_table FOUND
```

```
 THEN output t1row_small_table +output t2row_large_table
```

```
 END IF
```

```
 END LOOP
```

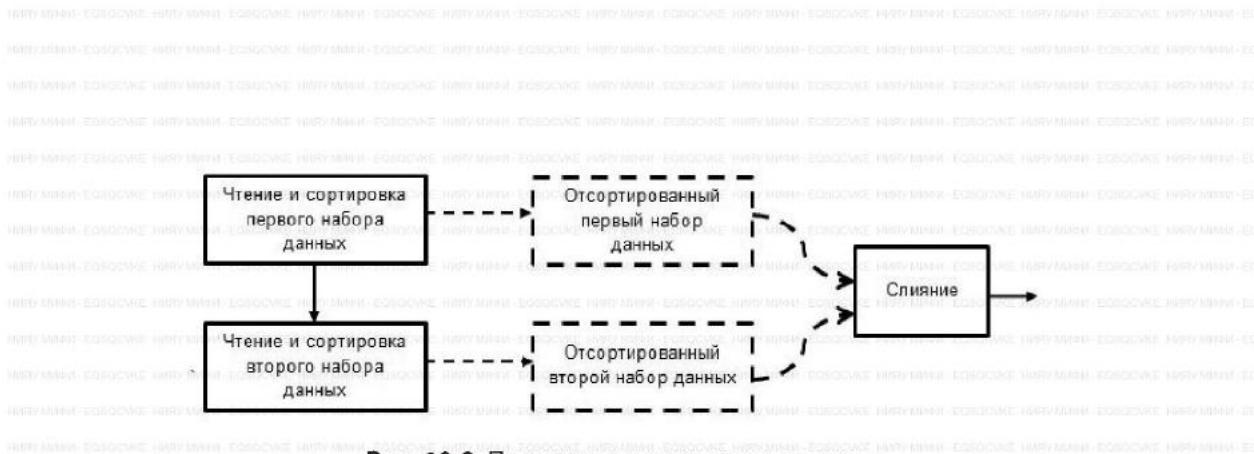
Графическая интерпретация (на примере СУБД PostgreSQL) фрагмента плана выполнения SQL-запроса, содержащего соединение с использованием хэширования, представлена на рис 10.8.



**Рис. 10.8.** Соединение с использованием хэширования. Фрагмент плана запроса

#### 10.1.3. Соединение сортировки и слияния (SORT MERGE JOIN)

Метод соединения сортировки и слияния предполагает наличие двух независимых наборов данных. Каждый набор данных считывается в соответствии с условиями отбора и сортируется по ключу соединения. Основная идея данного метода состоит в том, что для каждой строки из первого набора данных находится стартовая точка во втором наборе данных. После этого в качестве результата возвращаются строки, удовлетворяющие условию соединения первого и второго наборов, до тех пор пока не будет найдена строка, не удовлетворяющая условию соединения (рис. 10.9).

**Рис. 10.9. Процесс сортировки и слияния**

Соединение методом сортировки и слияния двух источников данных включает в себя следующие этапы.

- Выполняется выборка набора строк из источников данных.
- Выполняется формирование двух независимых наборов данных.
- Выполняется сортировка первого и второго наборов данных по ключу соединения в буферную область.
- Выполняется проверка на соответствие (слияние) отсортированных строк первого и второго наборов данных.
- Возвращается результат успешного слияния отсортированных наборов строк.

Используя псевдокод, процесс соединения сортировки и слияния можно представить следующим образом:

```

READ Набор_данных_1 SORT BY JOIN KEY TO Временный_на-
бор_данных_1
READ Набор_данных_2 SORT BY JOIN KEY TO Временный_на-
бор_данных_2
READ Временный_набор_данных_1.Строка FROM Временный_на-
бор_данных_1
READ Временный_набор_данных_2.Строка FROM Временный_на-
бор_данных_2
WHILE NOT eof Временный_набор_данных_1, Временный_на-
бор_данных_2
LOOP
IF (Временный_набор_данных_1.key = Временный_набор_-
данных_2.key)
 OUTPUT JOIN Временный_набор_данных_1.Строка
 Временный_набор_данных_2.Строка
ELSIF Временный_набор_данных_1.key <= Временный_на-
бор_данных_2.key)
 READ Временный_набор_данных_1.Строка FROM Временный_на-
бор_данных_1
ELSIF (Временный_набор_данных_1.key => Временный_на-
бор_данных_2.key.key)
 READ Временный_набор_данных_2.Строка FROM Временный_на-
бор_данных_2

```

**READ Временный\_набор\_данных\_2.Строка FROM Временный\_на-  
бор\_данных\_2**  
**END LOOP**

**Рассмотрим пример. Набор данных №1: 1,2,3,4,5,6; набор данных  
№2: 2,4,4,4,5,6,7,8.**

**Шаг 1.** Считываем первое значение (1) набора данных № 1 и пы-  
таемся найти стартовую точку в наборе № 2. Оцениваем соответствие  
текущего значения набора № 1 и очередного значения набора № 2:  
 $1 < 2$ . Результат шага1: читаем следующее значение набора данных  
№ 1.

**Шаг 1.** Считываем первое значение (1) набора данных № 1 и пытаемся найти стартовую точку в наборе № 2. Оцениваем соответствие текущего значения набора № 1 и очередного значения набора № 2:  $1 < 2$ . Результат шага 1: читаем следующее значение набора данных № 1.

**Шаг 2.** Считываем второе значение (2) набора данных № 1 и пытаемся найти стартовую точку в наборе № 2. Оцениваем соответствие текущего значения набора № 1 и очередного значения набора № 2:  $2 = 2$ . Результат шага 2: возвращение результата слияния строк наборов № 1 и № 2 до первого несовпадения.

**Шаг 3.** Считываем третье значение (3) набора данных № 1 и пытаемся найти стартовую точку в наборе № 2. Оцениваем соответствие текущего значения набора № 1 и очередного значения набора № 2:  $3 > 2$ . Результат шага 3: читаем следующее значение набора данных № 2.

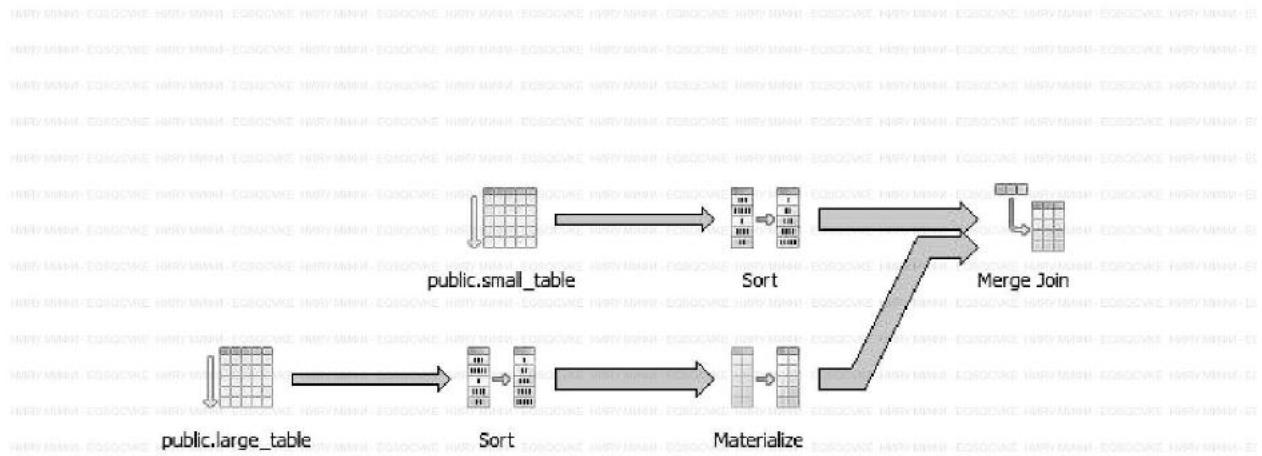
**Шаг4:** Используя третье значение (3) набора данных № 1, пытаемся найти стартовую точку в наборе № 2. Оцениваем соответствие текущего значения набора № 1 и очередного значения набора № 2:  $3 < 4$ . Результат шага 4: читаем следующее значение набора данных № 1.

**Шаг 5.** Считываем четвертое значение (4) набора данных № 1 и пытаемся найти стартовую точку в наборе № 2. Оцениваем соответствие текущего значения набора № 1 и очередного значения набора № 2:  $4 = 4$ . Результат шага 5: возвращение результата слияния строк наборов № 1 и № 2 до первого несовпадения.

**Шаг 6.** Оцениваем соответствие текущего значения (4) набора № 1 и очередного значения (5) набора № 2:  $4 < 5$ . Результат шага 6: читаем следующее значение набора данных № 1.

Остальные шаги выполняются аналогично рассмотренным ранее.

Графическая интерпретация (на примере СУБД PostgreSQL) фрагмента плана выполнения SQL-запроса, содержащего соединение с использованием сортировки и слияния, представлена на рис. 10.10.



**Рис. 10.10.** Соединение сортировки и слияния. Фрагмент плана запроса

#### 10.1.4. Перекрестное соединение (CORTASIAN JOIN)

Перекрестное соединение представляет собой операцию, когда оптимизатор соединяет каждую строку из одного источника данных с каждой строкой из другого источника данных, создавая декартово произведение двух множеств. Это происходит, когда отсутствует какое-либо условие соединения двух или более наборов данных. Таким образом, общее количество строк в результате соединения рассчитывается по формуле

$$A \times B = \text{набор всех возможных сочетаний } (a, b),$$

где **A** — количество строк в первом наборе; **B** — количество строк во втором наборе.

Перекрестные соединения используются достаточно редко и характеризуются большими значениями кардинальности и стоимости SQL-запроса.

Используя псевдокод, процесс перекрестного соединения можно представить следующим образом:

```

FOR Набор_данных_1.Строка IN Набор_данных_1 LOOP
 FOR Набор_данных_2.Строка IN Набор_данных_2 LOOP
 OUTPUT Набор_данных_1.Строка and Набор_данных_2.
 Стока
 END LOOP
END LOOP

```

## 10.2. ПЛАН ВЫПОЛНЕНИЯ SQL-ЗАПРОСА.

### ИНТЕРПРЕТАЦИЯ ОСНОВНЫХ ОПЕРАЦИЙ

#### План выполнения SQL-запроса

План выполнения SQL-запроса, или план запроса, — это последовательность шагов или инструкций СУБД, необходимых для выполнения SQL-запроса. На каждом шаге операция, инициировавшая

данный шаг выполнения SQL-запроса, извлекает строки данных, которые могут формировать конечный результат или использоваться для дальнейшей обработки. Инструкции плана выполнения SQL-запроса представляются в виде последовательности операций, которые выполняются СУБД для предложений **SQL SELECT, INSERT, DELETE и UPDATE**. Содержимое плана запроса, как правило, представляется древовидной структурой и включает в себя следующую информацию:

- порядок соединения источников данных (таблиц, представлений и т.п.);
  - метод доступа для каждого источника данных;
  - методы соединения источников данных;
  - операции ограничения выбора данных, сортировки и агрегирования;
  - стоимость и кардинальность каждой операции;
  - возможное использование секционирования и параллелизма.

Информация, предоставляемая планом выполнения SQL-запроса, позволяет разработчику увидеть, какие подходы и методы выбирает оптимизатор для выполнения SQL-операций.

## Интерпретация плана выполнения SQL-запроса

Визуализация плана выполнения SQL-запроса зависит от инструментов и средств разработки, которые могут как входить в состав СУБД, запрос которой представляет интерес для анализа, так и являться отдельными коммерческими или свободно распространяемыми программными продуктами, не имеющими прямого отношения к конкретному производителю СУБД. Использование того или иного инструмента визуализации плана выполнения запроса, как правило, существенно не влияет на восприятие того, что описывает представленный план запроса. Определяющей в процессе анализа того, каким путем пойдет оптимизатор при выполнении конкретного запроса, является способность верно интерпретировать информацию, которая представлена в плане запроса.

Как уже упоминалось, план SQL-запроса имеет древовидную структуру, которая описывает не только последовательность выполнения SQL-операций, но также и связь между этими операциями. Каждый узел дерева плана запроса — это операция, например сортировка, или метод доступа к таблице. Между узлами существует взаимосвязь родитель–потомок. Отношения родитель–потомок регулируются по следующим правилам:

- родитель может иметь одного или нескольких потомков;
  - потомок имеет только одного родителя;
  - операция, не имеющая родительской операции, является вершиной дерева;

Ниже приведены примеры планов выполнения SQL-запросов в Oracle SQL Developer.

- в зависимости от метода визуализации плана SQL-запроса потомок располагается с некоторым отступом относительно родителя. Потомки одного родителя располагаются на одинаковом расстоянии от своего родителя.

Рассмотрим более подробно информацию, представляемую планом выполнения SQL-запроса. Приведенные примеры выполнены в среде СУБД Oracle. В качестве инструмента выполнения запросов и визуализации плана SQL-запросов был использован Oracle SQL Developer. Фрагмент плана SQL-запроса представлен на рис. 10.11.

| <b>  Id   Operation</b>         | <b>  Name</b>          |
|---------------------------------|------------------------|
| 0   SELECT STATEMENT            |                        |
| 1   SORT ORDER BY               |                        |
| 2   NESTED LOOPS                |                        |
| 3   NESTED LOOPS                |                        |
| 4   TABLE ACCESS FULL           | ORDER_ITEMS            |
| 5   INDEX UNIQUE SCAN           | PRODUCT_INFORMATION_PK |
| 6   TABLE ACCESS BY INDEX ROWID | PRODUCT_INFORMATION    |

**Рис. 10.11.** Фрагмент плана выполнения SQL-запроса в среде СУБД Oracle

Используя правила отношения операций плана запроса, можно определить следующее их формальное описание.

Операция 0 — корень дерева плана запроса. Корень имеет одного потомка: операция 1.

Операция 1 — операция имеет одного потомка: операция 2.

Операция 2 — операция имеет двух потомков: операция 3 и операция 6.

Операция 3 — операция имеет двух потомков: операция 4 и операция 5.

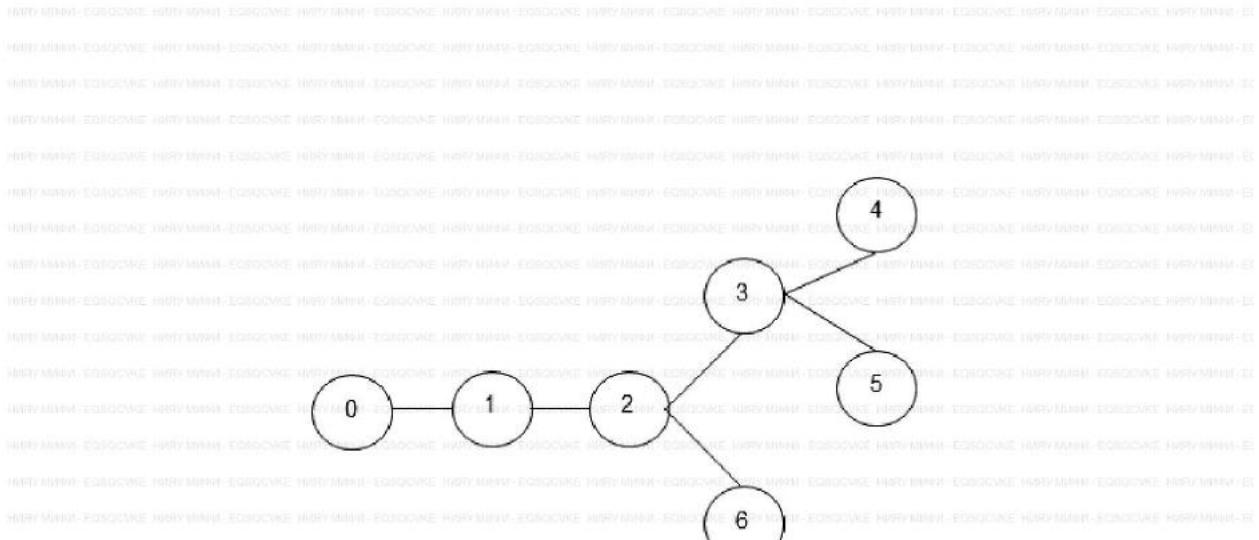
Операция 4 — операция не имеет потомков.

Операция 5 — операция не имеет потомков.

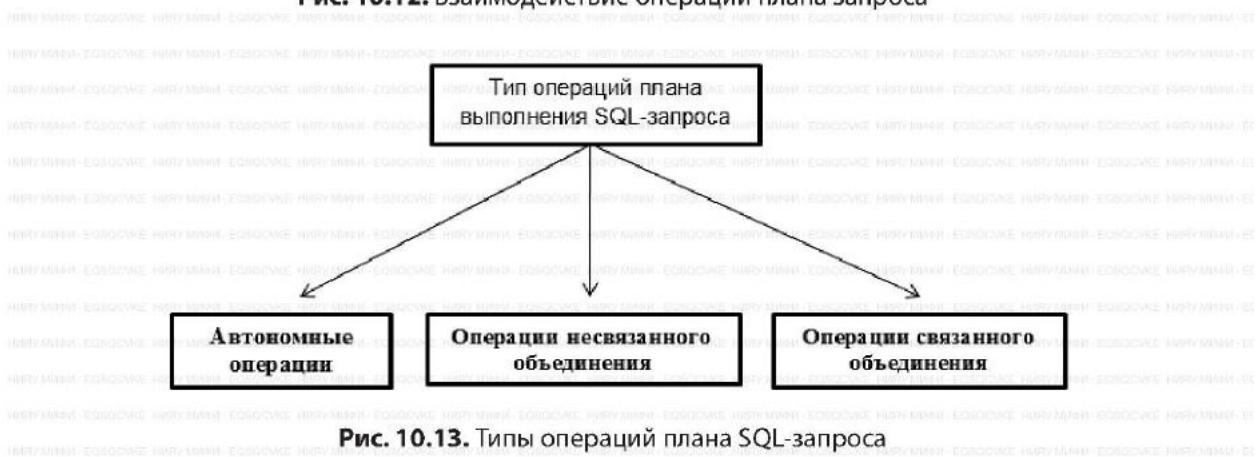
Операция 6 — операция не имеет потомков.

Взаимодействие родитель–потомок между операциями плана запроса представлено на рис. 10.12.

Операции, выполняемые в плане запроса, можно разделить на три типа: автономные, операции не связанного объединения и операции связанного объединения [15] (рис. 10.13).



**Рис. 10.12.** Взаимодействие операций плана запроса



**Рис. 10.13.** Типы операций плана SQL-запроса

#### Автономные операции

**Автономные операции** — это операции, которые имеют не более одной дочерней операции.

Правила следования, по которым выполняются автономные операции, можно сформулировать следующим образом.

1. Дочерняя операция выполняется перед родительской операцией.
2. Каждая дочерняя операция выполняется только один раз.
3. Каждая дочерняя операция возвращает свой результат родительской операции.

На рис. 10.14 представлен план следующего запроса:

```

SELECT o.order_id , o.order_status FROM orders o ORDER
BY o.order_status

```

Данный запрос содержит только автономные операции.

| ID | Operation         | Name   |
|----|-------------------|--------|
| 0  | SELECT STATEMENT  |        |
| 1  | SORT ORDER BY     |        |
| 2  | TABLE ACCESS FULL | ORDERS |

**Рис. 10.14.** Автономные операции, план запроса

Учитывая правила следования автономных операций, последовательность их выполнения будет следующая.

- В соответствии с правилом следования автономных операций № 1 первой будет выполнена операция с ID = 2. Выполняется последовательное чтение всех строк таблицы orders.
- Далее выполняется операция с ID = 1. Выполняется сортировка строк, возвращаемых операцией с ID = 2, по условию предложения сортировки ORDER BY.
- Выполняется операция с ID = 0. Возвращается результирующий набор данных.

#### Операции несвязанного объединения

**Операции несвязанного объединения** — это операции, которые имеют более одной независимо выполняемой дочерней операции. Пример: HASH JOIN, MERGE JOIN, INTERSECTION, MINUS, UNION ALL.

Правила следования, по которым работают операции несвязанного объединения, можно сформулировать следующим образом.

- Дочерняя операция выполняется перед родительской операцией.
- Дочерние операции выполняются последовательно, начиная с наименьшего значения ID операции в порядке возрастания этих значений.
- Перед началом работы каждой следующей дочерней операции текущая операция должна быть выполнена полностью.
- Каждая дочерняя операция выполняется только один раз независимо от других дочерних операций.
- Каждая дочерняя операция возвращает свой результат родительской операции.

На рис. 10.15 представлен план следующего запроса:

```
SELECT o.order_id from orders o
UNION ALL
SELECT oi.order_id from order_items oi
```

Данный запрос содержит операцию несвязанного объединения UNION ALL. Остальные две операции являются автономными.

**Рис. 10.15.** Операции

## **несвязанного объединения, план запроса**

Учитывая правила следования операций несвязанного объединения, последовательность их выполнения будет следующей:

1. В соответствии с правилами 1 и 2 следования операций несвязанного объединения первой будет выполнена операция с ID = 2. Выполняется последовательное чтение всех строк таблицы orders.
  2. В соответствии с правилом 5 операция с ID = 2 возвращает считанные на шаге 1 строки родительской операции с ID = 1.
  3. Операция с ID = 3 начнет выполняться, только когда закончится операция с ID = 2.
  4. После окончания выполнения операции с ID = 2 начинает выполняться операция с ID = 3. Выполняется последовательное чтение всех строк таблицы order\_items.
  5. В соответствии с правилом 5 операция с ID = 3 возвращает считанные на шаге 4 строки родительской операции с ID = 1.
  6. Операция с ID = 1 формирует результирующий набор данных на основе данных, полученных от всех ее дочерних операций (с ID = 2 и ID = 3).
  7. Выполняется операция с ID = 0. Возвращается результирующий набор данных.

Таким образом, можно отметить, что операция независимого объединения последовательно выполняет свои дочерние операции.

## Операции связанныго объединения

*Операции связанныго объединения* — это операции, которые имеют более одной дочерней операции, причем одна из операций контролирует выполнение остальных. Пример: `NESTED LOOPS UPDATE`.

Правила следования, по которым работают операции связанного объединения, можно сформулировать следующим образом.

1. Дочерняя операция выполняется перед родительской операцией.
  2. Дочерняя операция с наименьшим номером операции (ID) контролирует выполнение остальных дочерних операций.
  3. Дочерние операции, имеющие общую родительскую операцию, выполняются, начиная с наименьшего значения ID операции в порядке возрастания этих значений. Остальные дочерние операции выполняются НЕ последовательно.

Ниже приведен план выполнения запроса, полученный с помощью инструмента Explain Plan.

- Только первая дочерняя операция выполняется один раз. Все остальные дочерние операции выполняются несколько раз либо не выполняются совсем.

На рис. 10.16 представлен план следующего запроса:

```
SELECT *
 FROM order_items oi, orders o
 WHERE o.order_id= oi.order_id
 AND oi.product_id>100
 AND o.customer_id between 100 and 1000
```

Данный запрос содержит операцию связанныго объединения

#### NESTED LOOPS.

**Рис. 10.16. Операции**

**связанного объединения, план запроса**

| <b>ID</b> | <b>Operation</b>            | <b>Name</b>                        |
|-----------|-----------------------------|------------------------------------|
| 0         | SELECT STATEMENT            | ONE-Nested Loops                   |
| 1         | NESTED LOOPS                | ORDERS->ORD_CUSTOMER_ID            |
| 2         | TABLE ACCESS BY INDEX ROWID | ORDERS                             |
| 3         | INDEX RANGE SCAN            | ORD_CUSTOMER_ID -> ORD_CUSTOMER_IX |
| 4         | TABLE ACCESS FULL           | ORDER_ITEMS                        |

Учитывая правила следования операций связанныго объединения, последовательность их выполнения будет следующей.

1. В соответствии с правилами 1 и 2 следования операций связанныго объединения первой должна быть выполнена операция с ID = 2. Однако операции с ID = 2 и ID = 3 являются автономными, и в соответствии с правилом 1 следования автономных операций первой будет выполнена операция с ID = 3. Выполняется просмотр диапазона индекса ORD\_CUSTOMER\_IX по условию: o.customer\_id between 100 and 1000.

2. Операция с ID=3 возвращает родительской операции (с ID=2) список идентификаторов строк RowId, полученных на шаге 1.

3. Операция с ID = 2 выполняет чтение строк в таблице orders, в которых значение RowId соответствует списку значений RowId, полученных на шаге 2.

4. Операция с ID = 2 возвращает считанные строки родительской операции (с ID = 1).

5. Для каждой строки, возвращаемой операцией с ID = 2, выполняется вторая дочерняя операция (с ID = 4) операции NESTED LOOPS. То есть для каждой строки, возвращаемой операцией с ID = 2, выполняется полный последовательный просмотр таблицы order\_items с целью найти соответствие по атрибуту соединения.

6. Шаг 5 повторяется столько раз, сколько строк возвращает операция с ID = 2.

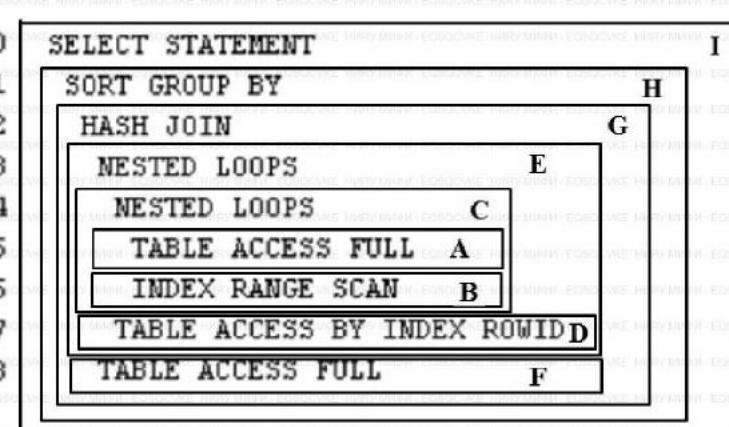
7. Операция с ID = 1 возвращает результаты работы родительской операции (с ID = 0).

8. Выполняется операция с ID = 0. Возвращается результирующий набор данных.

В зависимости от сложности анализируемого запроса план его выполнения может иметь достаточно сложную структуру, что на первый взгляд кажется затруднительным для его интерпретации. Методичное выполнение описанных выше правил и декомпозиция операций позволяют достаточно эффективно проанализировать план выполнения SQL-запроса любой сложности. Рассмотрим пример запроса, который формирует список клиентов, количество купленных ими товаров и их общую стоимость:

```
SELECT c.cust_first_name customer_name,
 COUNT(DISTINCT oi.product_id) as product_qty,
 SUM(oi.quantity* oi.unit_price) as total_cost
 FROM oe.orders o INNER JOIN customers c ON
o.customer_id=c.customer_id
 INNER JOIN oe.order_items oi ON o.order_id= oi.order_id
 GROUP BY c.cust_first_name
```

Последовательность операций плана данного запроса представлена на рис. 10.17.



**Рис. 10.17** План запроса, последовательность выполнения операций

Опишем возможный подход к интерпретации плана выполнения SQL-запроса, представленного на рис. 10.17. Данный подход включает в себя два основных этапа: декомпозиция операций на блоки и определение порядка выполнения операций.

На первом этапе необходимо выполнить декомпозицию выполняемых операций на блоки. Для этого находим все операции объединения, т.е. операции, которые имеют более одной дочерней операции (на рис. 10.17 это операции 2, 3 и 4), и выделяем эти дочерние опе-

рации в блоки. В результате, используя пример на рис. 10.17, получим три операции обединения и семь блоков операций.

рации в блоки. В результате, используя пример на рис. 10.17, получаем три операции объединения и семь блоков операций.

На втором этапе определяется последовательность выполнения блоков операций. Для этого необходимо применить правила следования операций, описанные выше. Выполним ряд рассуждений по вопросу выполнения каждой операции относительно ее идентификационного номера (ID).

Операция ID = 0 — автономная и является родительской для операции с ID = 1.

Операция ID = 1 тоже автономная; является родительской для операции ID = 2 и выполняется перед операцией ID = 0.

Операция ID = 2 выполняется перед операцией ID = 3.

Операция ID = 3 — операция связанныго объединения, является родительской для операций ID = 4, ID = 7. Операция ID = 3 выполняется перед операцией ID = 2.

Операция ID = 4 — операция связанных объединений, является родительской для операций ID = 5, ID = 6. Операция ID = 4 выполняется перед операцией ID = 3.

**Операция ID = 5 — автономная операция, выполняется перед операцией ID = 4**

Операция ID = 6 — автономная операция, выполняется перед операцией ID = 5.

**Операция ID = 7 – автономная операция, выполняется после выполнения блока операций «С».**

Операция ID = 8 — автономная операция, выполняется после  
блока операции «Г».

На основе проведенных рассуждений и правил следования сфор-

**1. Первой выполняется автономная операция ID = 5, см. правила**

следования операций связанного объединения. Выполняется последовательное чтение всей таблицы.

2. Результат операции ID = 5 — считанные строки таблицы — передается операции ID = 4.

3. Выполняется операция ID = 4: для каждой строки, возвращенной операцией ID = 5, выполняется операция ID = 6. То есть выпол-

4. Результат операции ID = 4 передается операции ID = 3. То есть

5. Выполняется операция  $ID = 3$ ; для каждого значения  $RowId$ ,

При выполнении операции ID = 7 для каждого блока таблицы, возвращенного в результате работы блока операций «С», выполняется операция ID = 7, т.е. выполняется чтение строк таблицы по за-

**данному списку идентификаторов строк RowId, полученных после выполнения операции ID = 4.**

данному списку идентификаторов строк RowId, полученных после выполнения операции ID = 4.

6. Выполняется автономная операция ID = 8 — последовательное чтение всей таблицы.

7. Выполняется операция несвязанного объединения ID = 2: выполняется соединение хэшированием результатов работы блоков операций «E» и «F».

8. Результат операции ID = 2 передается операции ID = 1.

9. Выполняется операция несвязанного объединения ID = 1: выполняется агрегирование и сортировка данных, полученных в результате операции ID = 2.

10. Выполняется операция ID = 0. Возвращается результирующий набор данных

Правила следования, сформулированные для основных типов операций, применимы для большинства планов выполнения SQL-запроса. Однако существуют конструкции, используемые в SQL-запросах, которые предполагают нарушение порядка выполнения операций, описанных в правилах следования. Такие ситуации могут появляться в результате использования, например, подзапросов или предикатов антисоединения. В любом случае процесс интерпретации плана выполнения SQL-запроса не предполагает только использование ряда правил, которые обеспечивают именно максимально верный анализ того, что собирается делать оптимизатор при выполнении SQL-запроса. Очередной SQL-запрос — это всегда индивидуальный случай; и то, как он будет выполнен в СУБД, зависит от множества факторов, среди которых версия СУБД, версия и тип операционной системы, на которой развернут экземпляр СУБД, используемая аппаратная часть, квалификация автора SQL-запроса и т.д.

## **10.3. АНАЛИЗ ЭФФЕКТИВНОСТИ ВЫПОЛНЕНИЯ SQL-ЗАПРОСОВ. ВОЗМОЖНЫЕ ПОДХОДЫ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ SQL-ЗАПРОСОВ**

Цель оптимизации SQL-запроса заключается в том, чтобы получить требуемый результат за приемлемое время при наименьших затратах ресурсов на его выполнение. В зависимости от специфики приложения цели оптимизации могут иметь различный характер. Так, например, для пакетных приложений, которые формируют отчеты, одним из важных критериев оптимизации является пропускная способность. С другой стороны, в интерактивных приложениях такой критерий, как лучшее время отклика, является основным с точки зрения оптимизации работы приложения, поскольку пользователь

**такого приложения ожидает максимально быстро увидеть результат SQL-запроса или хотя бы его часть.**

такого приложения ожидает максимально быстро увидеть результат SQL-запроса или хотя бы его часть.

Несмотря на относительно ограниченный набор операторов, SQL является достаточно гибким языком программирования. Такая особенность позволяет получить один и тот же результат множеством различных способов. Соответственно, не исключены ситуации, когда, например, написанные по-разному два SQL-запроса, представляющие один и тот же результат, выполняются за разные промежутки времени, причем разница во времени может быть достаточно существенной. Таким образом, эффективность выполнения SQL-запроса определяется в том числе и тем, насколько качественно он написан. В рамках этой проблемы можно отметить следующие рекомендации.

1. Колонки, участвующие в соединении таблиц или в условии поиска конструкции `WHERE`, не должны быть преобразованы какими-либо функциями. В противном случае оптимизатор SQL-запроса будет игнорировать наличие индекса, который, возможно, мог бы быть использован.

Рассмотрим пример запроса, который выбирает всех клиентов с фамилией Петров из таблицы client. Для колонки lastName (фамилия клиента) построен неуникальный индекс. Сначала выполним запрос без применения каких-либо функций, использующих в качестве аргумента колонку, указанную в условии поиска, и проанализируем план выполнения SQL-запроса:

```
SELECT * FROM client cl WHERE cl.lastName = 'ПЕТРОВ'
```

Рассмотрим план выполнения запроса (рис. 10.18).

| <b>Id</b> | <b>Operation</b>            | <b>Name</b>       |
|-----------|-----------------------------|-------------------|
| 0         | <b>SELECT STATEMENT</b>     |                   |
| 1         | TABLE ACCESS BY INDEX ROWID | CLIENT            |
| 2         | INDEX RANGE SCAN            | CLIENT_LASTNAME_I |

**Рис. 10-18.** План SQL-запроса с использованием индекса

Приведенный план выполнения SQL-запроса показывает, что для доступа к данным используется существующий индекс **CLIENT\_LASTNAME\_I**.

Добавим в условие поиска функцию `UPPER`, которая преобразует данные в колонке `Last Name` в верхний регистр:

```
SELECT * FROM client cl WHERE UPPER(lastName) = 'NETEROV'
```

План выполнения данного запроса приведен на рис. 10.19.

| <b>Id</b> | <b>Operation</b>                | <b>Name</b> |
|-----------|---------------------------------|-------------|
| 0         | <b>SELECT STATEMENT</b>         |             |
| 1         | <b>TABLE ACCESS FULL CLIENT</b> |             |

**Рис. 10.19.** План SQL-запроса без использования индекса

Приведенный план выполнения SQL-запроса показывает, что теперь для доступа к данным используется последовательный перебор всех строк таблицы client, т.е. использование существующего индекса CLIENT\_LASTNAME\_I не рассматривается оптимизатором как один из способов доступа к данным.

## 2. Необходимо избегать неявного преобразования типов данных.

Так, например, если существует конструкция WHERE col1=col2, где col1 имеет тип данных INT, а col2 имеет тип данных VARCHAR2(10), но в col2 хранятся только числовые данные. В этом случае оптимизатор СУБД будет выполнять неявное преобразование данных, которое представляется в виде условия WHERE col1=cast(col2 as INT), что исключает для оптимизатора возможность рассматривать использование индекса по col2.

3. Необходимо представлять альтернативный синтаксис написания SQL-запросов для получения одного и того же результата.

Рассмотрим два SQL-запроса, которые возвращают один и тот же результат, но с использованием разных подходов. В примере, приведенном ниже, запрос должен возвращать названия стран, в которых есть клиенты с фамилией Петров. В первом варианте используется обычный подзапрос в предикате IN:

```
SELECT ct.countryname FROM country ct WHERE
ct.countryid IN
 (SELECT ct.countryid FROM client cl WHERE
cl.lastname='Петров')
```

Во втором варианте используется коррелированный подзапрос в предикате EXISTS:

```
SELECT ct.countryname FROM country ct WHERE EXISTS
 (SELECT ct.countryId FROM client cl
 WHERE cl.countryId=ct.countryId AND
 cl.lastName='Петров')
```

Пусть таблица client содержит 220 000 строк, таблица country содержит 3 строки. План выполнения варианта запроса, использующего предикат EXISTS, приведен на рис. 10.20.

| Execution Plan for SELECT Statement |     |                             |                                    |       |       |             |
|-------------------------------------|-----|-----------------------------|------------------------------------|-------|-------|-------------|
|                                     | Id  | Operation                   | Name                               | Rows  | Bytes | Cost (%CPU) |
| HARRY.MINI..EDGOCVRE                | 0   | SELECT STATEMENT            |                                    | 3     | 417   | 15 (0%)     |
| HARRY.MINI..EDGOCVRE                | 1   | NESTED LOOPS SEMI           |                                    | 3     | 417   | 15 (0%)     |
| HARRY.MINI..EDGOCVRE                | 2   | TABLE ACCESS FULL           | HARRY.MINI..EDGOCVRE.COUNTRY       | 3     | 345   | 3 (0%)      |
| HARRY.MINI..EDGOCVRE                | * 3 | TABLE ACCESS BY INDEX ROWID | HARRY.MINI..EDGOCVRE.CLIENT        | 12992 | 304K  | 4 (0%)      |
| HARRY.MINI..EDGOCVRE                | 4   | INDEX RANGE SCAN            | HARRY.MINI..EDGOCVRE.CLIENT_COUNTR | 17    | 2     | 2 (0%)      |

**Predicate Information (identified by operation id):**

**Рис. 10.20.** План SQL-запроса. Использование EXISTS

Рассмотрим подробнее план выполнения данного запроса. В соответствии с правилами следования автономных операций первой будет выполняться операция ID = 2, т.е. сначала выполняется полное последовательное чтение таблицы country. Затем полученный результат передается операции ID = 1. После этого в цикле для каждой строки таблицы country выполняется поиск в таблице client по диапазону индекса **CLIENT\_COUNTRY\_I** (колонка countryId в таблице client) и получение нужных строк из данной таблицы с фильтрацией по значению «Петров» в колонке lastName. Эти действия выполняются операциями ID = 4 и ID = 3. По окончании работы цикла результат передается операции ID = 0.

Отметим некоторые затраты ресурсов, которые предположительно будут сделаны СУБД при выполнении приведенного плана запроса:

- предполагаемое количество считанных строк из таблицы client (Rows) = 12 922;
  - предполагаемое количество считанных строк из таблицы country (Rows) = 3;
  - предполагаемое количество данных, возвращаемых при чтении таблицы client (Bytes) = 304 Кбайт;
  - стоимость выполнения запроса Cost = 15.

План выполнения запроса, использующего предикат IN, приведен на рис. 10.21.

Рассмотрим подробнее план выполнения данного запроса. В соответствии с правилами следования автономных операций первой будет выполняться операция ID = 2, т.е. сначала выполняется полное, последовательное чтение таблицы country. Затем полученный результат передается операции ID = 1. После этого выполняется операция ID = 4, выполняющая поиск по диапазону индекса CLIENT\_LASTNAME\_1 (колонка lastName) в таблице client с фильтрацией по

|                                                                                                                                                                                                                                      |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Ниже приведен план SQL-запроса, полученный с помощью инструмента Explain Plan в Oracle Database 11g. План показывает, что для выполнения запроса используется индексный сканирование по индексу CLIENT_LASTNAME_I на таблице CLIENT. |
| -----                                                                                                                                                                                                                                |
| Id   Operation   Name   Rows   Bytes   Cost (%CPU)                                                                                                                                                                                   |
| -----                                                                                                                                                                                                                                |
| 1 0   SELECT STATEMENT   /*+ */   3   345   6 (0)                                                                                                                                                                                    |
| * 1   FILTER     3   345   3 (0)                                                                                                                                                                                                     |
| 2   TABLE ACCESS FULL   COUNTRY   3   345   3 (0)                                                                                                                                                                                    |
| * 3   FILTER     2   42   3 (0)                                                                                                                                                                                                      |
| * 4   INDEX RANGE SCAN   CLIENT_LASTNAME_I   2   42   3 (0)                                                                                                                                                                          |
| -----                                                                                                                                                                                                                                |

#### Predicate Information (identified by operation id):

```

1 - filter(EXISTS (SELECT 0 FROM "CLIENT" "CL" WHERE :B1=:B2 AND
 "CL"."LASTNAME"='Петров'))
 3 - filter(:B1=:B2)
 4 - access("CL"."LASTNAME"='Петров')

```

**Рис. 10.21.** План SQL-запроса. Использование предиката IN

значению «Петров» в колонке lastName — операция ID = 3. При этом на этапе выполнения операции ID = 4 значения колонки lastName выбираются непосредственно из индекса, а не из таблицы. Операции ID = 3 и ID = 4 выполняются для каждой строки операции ID = 1. Также необходимо обратить внимание на тот факт, что на шаге выполнения операции ID = 1 оптимизатор выполнил преобразование запроса и использует конструкцию EXISTS вместо предиката IN с соответствующими переменными связывания.

Отметим некоторые затраты ресурсов, которые предположительно будут сделаны СУБД при выполнении приведенного плана запроса:

- предполагаемое количество считанных строк индекса CLIENT\_LASTNAME\_I = 2;
- предполагаемое количество считанных строк из таблицы country (Rows) = 3;
- максимальное предполагаемое количество данных, возвращаемых при чтении (Bytes) = 345 байт;
- стоимость выполнения запроса Cost = 6.

Рассмотренные примеры показывают, что различные подходы при написании запросов для получения одного и того же результата могут определять и разные методы доступа к данным, что, в свою очередь, влияет на производительность выполнения SQL-запроса. Таким образом, можно отметить, что проблема дизайна SQL-кода является достаточно важной и требует понимания не только синтаксиса SQL, но и особенностей организации работы аналогичных на первый взгляд конструкций.

Обычно на начальных этапах жизненного цикла систем хранения и обработки данных проблемы производительности SQL-запросов не всегда имеют явный характер. Это связано с тем, что некоторое время приложения, использующие SQL-запросы, работают с относительно небольшими объемами данных. В этом случае работа даже неэффективно с точки зрения производительности написанного SQL-запроса компенсируется ресурсами инфраструктуры, в рамках которой функционирует СУБД. Увеличение количества данных выявляет «узкие» места, в том числе и то, насколько тот или иной SQL-запрос эффективно расходует ресурсы системы, а также то, как осуществляется доступ к данным. При увеличении объемов данных проблема доступа к данным является достаточно важной; и от того, насколько автор SQL-запросов представляет, как СУБД осуществляет доступ к данным, зависит эффективность выполнения SQL-запросов с точки зрения производительности. Методы доступа при относительно больших объемах данных в большинстве случаев предполагают использование индексов. Часто (но не всегда) оптимизатор SQL-запросов выбирает метод доступа с использованием индекса, когда применяется ограничение выбора строк по условию, указанному в конструкции **WHERE**. Однако не только наличие индекса на колонке обеспечивает его эффективное использование, но и понимание того, какие условия необходимо создать для того, чтобы индекс был избирательным, т.е. понимание того, как оптимизатор SQL-запроса будет использовать тот или иной индекс.

В процессе написания SQL-запросов часто возникают ситуации, когда необходимо принимать решения о необходимости использования того или иного метода доступа к данным. Если, например, рассматривается произвольный метод доступа к данным, который предполагает использование индексов, возникает необходимость определить, каким будет этот индекс: будет ли это один — составной — индекс, построенный на нескольких колонках таблицы, или же это будут несколько индексов, каждый из которых построен на одной колонке таблицы, и т.д. В любом случае получить однозначный ответ на то, какой подход реализует оптимальный с точки зрения производительности доступ к данным, можно, только проведя серию экспериментов. Однако то, насколько эти эксперименты будут эффективны, характеризуется степенью понимания фундаментальных принципов функционирования СУБД, в том числе и в области методов доступа к данным.

Рассмотрим некоторые возможные подходы к процессу выбора метода доступа к данным на основе сравнения использования индексов на основе древовидной структуры и индексов на основе битовых карт. В табл. 10.1 приведены основные характеристики bitmap-индекса и индекса на основе  $B^+$ -дерева.

Ниже приведены основные характеристики индексов bitmap/B<sup>+</sup>-дерева:

Таблица 10.1

Основные характеристики индексов bitmap/B<sup>+</sup>-дерево

| Характеристики                                | B <sup>+</sup> -дерево | Bitmap      |
|-----------------------------------------------|------------------------|-------------|
| Ограничение уникальности Primary key / Unique | Да                     | Нет         |
| Блокировка на уровне строк                    | Да                     | В том числе |
| Использование комбинации нескольких индексов  | Нет                    | Да          |

Представленная табл. 10.1 отражает некоторые основные ограничения использования каждого из типов индекса. Первое, что можно отметить, — то, что индекс на основе B<sup>+</sup>-дерева может использоваться как ограничение уникальности первичного или альтернативного ключа. Второе — индекс на основе B<sup>+</sup>-дерева поддерживает уровень блокировки данных на уровне строк, в то время как блокировки, установленные bitmap-индексом, могут затрагивать в том числе и строки, не участвующие в процессе модификации данных. Третье — комбинация нескольких bitmap-индексов достаточно эффективна, особенно при использовании условия OR, в отличие от индекса на основе B<sup>+</sup>-дерева, в котором не поддерживаются комбинации индексов.

Рассмотрим примеры того, как в зависимости от выбранного способа индексирования оптимизатор запросов выбирает метод доступа к данным и выполняет соответствующие операции в одинаковых условиях. В примерах используются таблицы предметной области проката автотранспортных средств.

Рассмотрим запрос поиска клиента в таблице client по значению первичного ключа:

```
SELECT * FROM client WHERE clientId=14;
```

## Исходные данные:

Таблица: client

Количество строк: 220 000

Проверка по условию равенства (=)

Индекс на основе B<sup>+</sup>-дерева SYS\_C0030342

Индекс на основе bitmap SYS\_BITMAP\_1

План выполнения запроса представлен на рис. 10.22.

В данном запросе используется уникальный индекс, который обеспечивает ограничения первичного ключа. Имя ограничения первичного ключа SYS\_C0030342. Представленный запрос выполняется с использованием трех операций. Первой выполняется операция ID = 2 доступа по уникальному значению индекса clientId = 14. Во второй операции ID = 1 выполняется поиск нужной строки в таб-

| План выполнения запроса по clientId |              |                          |  |  |  |
|-------------------------------------|--------------|--------------------------|--|--|--|
| Id   Operation                      | Name         | Starts   E-Rows   A-Rows |  |  |  |
| 0   SELECT STATEMENT                |              | 1   1   1                |  |  |  |
| 1   TABLE ACCESS BY INDEX ROWID     | CLIENT       | 1   1   1                |  |  |  |
| * 2   INDEX UNIQUE SCAN             | SYS_C0030342 | 1   1   1                |  |  |  |

**Рис. 10.22.** План запроса поиска клиента по clientId. Индекс —  $B^+$ -дерево

лице client по полученному операцией с ID = 2 RowId. Третьей выполняется операция с ID = 0, которая возвращает результирующие строки. Каждая из этих операций выполняется один раз (Starts = 1); кроме того, предполагаемое (E-Rows) и актуальное (A-Rows) количество считанных строк равно единице.

Рассмотрим тот же запрос поиска клиента в таблице client с использованием bitmap-индекса, построенного на колонке с уникальными значениями clientId.

Исходные данные:

Таблица: client

Количество строк: 220 000

Проверка по условию равенства (=)

Индекс на основе битовых карт CLIENT\_CLIENTID\_BITMAP\_I

План выполнения запроса представлен на рис. 10.23.

| План выполнения запроса по clientId |                          |                          |  |  |  |
|-------------------------------------|--------------------------|--------------------------|--|--|--|
| Id   Operation                      | Name                     | Starts   E-Rows   A-Rows |  |  |  |
| 0   SELECT STATEMENT                |                          | 1   1   1                |  |  |  |
| 1   TABLE ACCESS BY INDEX ROWID     | CLIENT                   | 1   1   1                |  |  |  |
| 2   BITMAP CONVERSION TO ROWIDS     |                          | 1   1   1                |  |  |  |
| * 3   BITMAP INDEX SINGLE VALUE     | CLIENT_CLIENTID_BITMAP_I | 1   1   1                |  |  |  |

**Рис. 10.23.** План запроса поиска клиента по clientId. Индекс — bitmap

В данном запросе используется bitmap-индекс, который построен на колонке clientId. Представленный запрос выполняется с использованием четырех операций. Первой выполняется операция с ID = 3, которая сканирует индекс и применяет ограничение clientId = 14. Во второй операции ID = 2 выполняется преобразование полученных битовых карт в список RowId. В третьей операции ID = 1 выполняется поиск нужной строки в таблице client по полученному в операции с ID = 2 RowId. Четвертой выполняется операция с ID = 0, которая возвращает результирующие строки. Каждая из этих операций выполняется один раз (Starts = 1), и предполагаемое (E-Rows) и актуальное (A-Rows) количество считанных строк также равно единице.

Планы запроса, представленные на рис. 10.22 и 10.23, показывают, что в случае использования bitmap-индекса, построенного на

основе атрибута с уникальными значениями, выполняемых операций на одну больше. Однако в данном случае эта информация не является весомым аргументом в пользу того, чтобы не использовать bitmap-индекс. Если, например, проанализировать объемы внешней памяти (табл. 10.2), которые занимают оба индекса, станет более очевидным, что использование bitmap-индекса является менее эффективным в условиях высокой селективности; и это вполне предсказуемо, учитывая принципы построения bitmap-индексов.

Таблица 10.2

## Размер используемых индексов

| Название индекса         | Тип индекса            | Размер индекса, Мбайт |
|--------------------------|------------------------|-----------------------|
| SYS_C0030342             | B <sup>+</sup> -дерево | 4                     |
| CLIENT_CLIENTID_BITMAP_I | Bitmap                 | 7                     |
|                          |                        |                       |

Рассмотрим запрос поиска тех клиентов в той же таблице client, у которых отсутствует отчество, с использованием неуникального индекса на основе B<sup>+</sup>-дерева, построенного на колонке middleName (имя построенного индекса — CLIENT\_MIDDLENAME\_I):

```
SELECT * FROM client WHERE middleName IS NULL;
```

Исходные данные:

Таблица: client

Количество строк: 220 000

Проверка по условию IS NULL

Индекс на основе B<sup>+</sup>-дерева CLIENT\_MIDDLENAME\_I

План выполнения запроса представлен на рис. 10.24.

| ID | Operation         | Name   | Starts | E-Rows | A-Rows |
|----|-------------------|--------|--------|--------|--------|
| 0  | SELECT STATEMENT  |        | 1      | 1500   | 1500   |
| 1  | TABLE ACCESS FULL | CLIENT | 1      | 1      | 1500   |

Рис. 10.24. План запроса поиска клиентов с пустым отчеством.

Индекс — B<sup>+</sup>-дерево

Представленный запрос выполняется с использованием двух операций. В первой операции ID = 1 выполняется полное табличное сканирование и выбираются строки, у которых отсутствует значение в колонке middleName. Несмотря на то, что на колонке middleName построен неуникальный индекс CLIENT\_MIDDLENAME\_I, оптимизатор его не использует. Вторая операция (ID = 0) возвращает ре-

Ниже приведен план выполнения запроса поиска клиентов с пустым отчеством. Использование индекса на основе битовых карт (CLIENT\_MIDDLENAME\_BM\_I) показывает, что операция SELECT выполняется с использованием индекса на основе битовых карт.

**зультирующие строки. Каждая из этих операций выполняется один раз ( $Starts = 1$ ), и предполагаемое количество считанных строк (E-Rows) равно единице, в то время как актуальное количество считанных строк (A-Rows) равно 1500.**

Рассмотрим тот же запрос поиска клиентов в таблице client с использованием индекса на основе битовых карт, построенного на колонке с неуникальными значениями middleName (имя построенного индекса — CLIENT\_MIDDLENAME\_BM\_I).

Исходные данные:

Таблица: client

Количество строк: 220 000

Проверка по условию IS NULL

Индекс на основе битовых карт CLIENT\_MIDDLENAME\_BM\_I

План выполнения запроса представлен на рис. 10.25.

| <b>  Id   Operation</b>                                  | <b>  Name</b> | <b>  Starts   E-Rows   A-Rows  </b> |
|----------------------------------------------------------|---------------|-------------------------------------|
| 0   SELECT STATEMENT                                     |               | 1   1500   1500                     |
| 1   TABLE ACCESS BY INDEX ROWID   CLIENT                 |               | 1   1   1500                        |
| 2   BITMAP CONVERSION TO ROWIDS                          |               | 1   1   1500                        |
| * 3   BITMAP INDEX SINGLE VALUE   CLIENT_MIDDLENAME_BM_I |               | 1   1   1                           |

**Рис. 10.25.** План запроса поиска клиентов с пустым отчеством. Индекс —

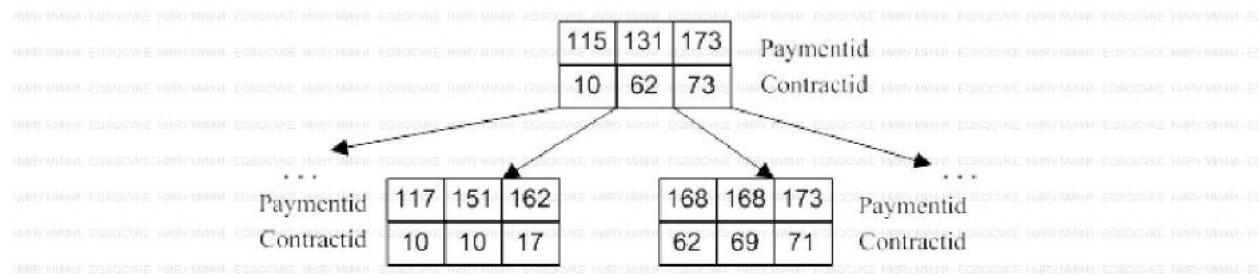
**bitmap**

В данном запросе используется bitmap-индекс, который построен на колонке middleName. Представленный запрос выполняется с использованием четырех операций. Первой выполняется операция ID = 3, которая сканирует индекс и применяет ограничение middleName IS NULL. Во второй операции ID = 2 выполняется преобразование полученных битовых карт в список RowId. В третьей операции (ID = 1) выполняется поиск нужных строк в таблице client по полученным в операции с ID = 2 диапазонам значений RowId. Четвертой выполняется операция с ID = 0, которая возвращает результаты, полученные в результате выполнения операции ID = 1.

План запроса, представленный на рис. 10.24, показывает, что наличие индекса на основе  $B^+$ -дерева по колонке, используемой в условии запроса в конструкции WHERE, не рассматривается оптимизатором как возможный метод доступа к данным. В данном случае причина такого поведения оптимизатора является объяснимой, поскольку индекс на основе  $B^+$ -дерева не хранит NULL-значения. То есть при требуемом условии поиска middleName IS NULL наличие

индекса на основе  $B^+$ -дерева является неоправданным. Другая ситуация — с планом запроса на рис. 10.25, который показывает, что наличие bitmap-индекса на колонке, используемой в условии запроса в конструкции **WHERE**, рассматривается оптимизатором как возможный метод доступа к данным, поскольку bitmap-индекс хранит информацию о строках с NULL-значениями.

Достаточно часто в SQL-запросах в условии выбора строк таблицы используется более одной колонки. До сих пор рассматривались случаи, когда один индекс строился на одной колонке, т.е. соотношение индекса и колонки таблицы представляется как 1:1. Однако в современных СУБД существует возможность использовать индекс, построенный на нескольких колонках. Такой индекс называется составным. Помимо того что составной индекс может эффективно использоваться, когда в конструкции **WHERE** SQL-запроса содержится сочетание нескольких колонок по условию **AND**, он также используется как один из способов ограничения целостности составного первичного или альтернативного ключа. Пример составного индекса на основе  $B^+$ -дерева представлен на рис. 10.26.



**Рис. 10.26.** Составной индекс на основе  $B^+$ -дерева

Рассмотрим примеры SQL-запросов и соответствующие им планы запроса, в которых используется составной индекс. В таблице client выполним поиск клиента по заданным значениям фамилии, имени и отчества; при этом могут использоваться индексы как на основе одной колонки, так и составные:

```
SELECT * FROM client WHERE lastName='Иванов' AND
firstName='Петр'
AND middleName='Сергеевич';
```

Пусть для заданных условий поиска запрос возвращает одну строку.

Пусть на текущий момент существует один неуникальный индекс **CLIENT\_LASTNAME\_I**, построенный на колонке *lastName*. Проанализируем полученный план SQL-запроса.

Исходные данные:  
 Таблица: client  
 Количество строк: 220 000  
 Поиск клиента по фамилии, имени и отчеству  
 Индекс на основе  $B^+$ -дерева CLIENT\_LASTNAME\_I

План выполнения запроса представлен на рис. 10.27.

| <b>Id</b> | <b>Operation</b>            | <b>Name</b>       | <b>Starts</b> | <b>E-Rows</b> | <b>A-Rows</b> |
|-----------|-----------------------------|-------------------|---------------|---------------|---------------|
| 0         | SELECT STATEMENT            |                   | 1             | 1             | 1             |
| 1         | TABLE ACCESS BY INDEX ROWID | CLIENT            | 1             | 1             | 1             |
| 2         | INDEX RANGE SCAN            | CLIENT_LASTNAME_I | 1             | 1             | 9757          |

**Predicate Information (identified by operation id):**

Рис. 10.27. План запроса поиска клиентов. Индекс —  $B^+$ -дерево CLIENT\_LASTNAME\_I

План запроса, представленный на рис. 10.27, показывает, что операция с ID=2 использует индекс CLIENT\_LASTNAME\_I для поиска всех строк, удовлетворяющих условию lastName='Иванов'. Остальные два предиката: firstName='Петр' AND middleName='Сергеевич' применяются как фильтр на результатирующем наборе строк, который возвращает операция с ID = 1. Актуальное (A-Rows) количество считанных строк на этапе операции с ID = 2 равно 9757, что является неприемлемым для получения одной строки.

Рассмотрим ситуацию, когда создан еще один неуникальный индекс на колонке firstName. Таким образом, на текущий момент существуют два индекса: CLIENT\_LASTNAME\_I на колонке lastName и CLIENT\_FIRSTNAME\_I на колонке firstName. Проанализируем полученный план SQL-запроса.

Исходные данные:

Таблица: client

Количество строк: 220 000

Поиск клиента по фамилии, имени и отчеству

Индексы на основе  $B^+$ -дерева CLIENT\_LASTNAME\_I, CLIENT\_FIRSTNAME\_I

План выполнения запроса представлен на рис. 10.28.

Ниже приведен план выполнения запроса на языке SQL-анализатора Oracle SQL Developer.

| <b>  Id   Operation</b>                    | <b>  Name</b> | <b>  Starts   E-Rows   A-Rows  </b> |
|--------------------------------------------|---------------|-------------------------------------|
| 0   SELECT STATEMENT                       |               | 1   1   1                           |
| * 1   TABLE ACCESS BY INDEX ROWID  CLIENT  |               | 1   1   1                           |
| * 2   INDEX RANGE SCAN  CLIENT_FIRSTNAME_I |               | 1   5002   5002                     |

**Predicate Information (identified by operation id):**

- 1 - filter(("MIDDLENAME"='Сергеевич' AND "LASTNAME"='Иванов'))
- 2 - access("FIRSTNAME"='Петр')

**Рис. 10.28.** План запроса поиска клиентов. Индексы —  $B^+$ -дерево `CLIENT_`

`LASTNAME_I, CLIENT_FIRSTNAME_I`

План запроса, представленный на рис. 10.28, показывает, что операция с ID = 2 использует индекс `CLIENT_FIRSTNAME_I` для поиска всех строк, удовлетворяющих условию `firstName='Петр'`, несмотря на то что существует индекс и для фамилии. В данном случае оптимизатор считает более эффективным использовать индекс `CLIENT_FIRSTNAME_I`.

Остальные два предиката: `lastName='Иванов'` AND `middleName='Сергеевич'` — применяются как фильтр на результатирующем наборе строк, который возвращает операция с ID = 1. Актуальное (A-Rows) количество считанных строк на этапе операции с ID = 2 равно 5002, что также неприемлемо для получения одной строки.

Рассмотрим ситуацию, когда создается еще один неуникальный индекс на колонке `middleName`. Таким образом, на текущий момент существуют три индекса: `CLIENT_LASTNAME_I` на колонке `lastName`, `CLIENT_FIRSTNAME_I` на колонке `firstName` и `CLIENT_MIDDLENAME_I` на колонке `middleName`. Проанализируем полученный план SQL-запроса.

Исходные данные:

Таблица: `client`

Количество строк: 220 000

Поиск клиента по фамилии, имени и отчеству

Индексы на основе  $B^+$ -дерева `CLIENT_LASTNAME_I`, `CLIENT_FIRSTNAME_I`, `CLIENT_MIDDLENAME_I`

План выполнения запроса представлен на рис. 10.29.

План запроса, представленный на рис. 10.29, показывает, что операция с ID=2 использует индекс `CLIENT_MIDDLENAME_I` для поиска всех строк, удовлетворяющих условию `middleName='Сергеевич'`, несмотря на то что существуют еще два индекса (для фамилии и имени). В данном случае оптимизатор считает более эффективным использовать индекс `CLIENT_MIDDLENAME_I`.

План выполнения запроса представлен на рис. 10.29.

План запроса, представленный на рис. 10.29, показывает, что операция с ID=2 использует индекс `CLIENT_MIDDLENAME_I` для поиска всех строк, удовлетворяющих условию `middleName='Сергеевич'`, несмотря на то что существуют еще два индекса (для фамилии и имени). В данном случае оптимизатор считает более эффективным использовать индекс `CLIENT_MIDDLENAME_I`.

| Handle            | Event                            | Object              | Starts | E-Rows | A-Rows | Ends              |
|-------------------|----------------------------------|---------------------|--------|--------|--------|-------------------|
| HHRV-MNNH-EQGKQWE | Id    Operation                  | Name                | Starts | E-Rows | A-Rows | End               |
| HHRV-MNNH-EQGKQWE | 0    SELECT STATEMENT            |                     | 1      | 1      | 1      | HHRV-MNNH-EQGKQWE |
| HHRV-MNNH-EQGKQWE | 1    TABLE ACCESS BY INDEX ROWID | CLIENT              | 1      | 1      | 1      | HHRV-MNNH-EQGKQWE |
| HHRV-MNNH-EQGKQWE | 2    INDEX RANGE SCAN            | CLIENT_MIDDLENAME_I | 1      | 302    | 302    | HHRV-MNNH-EQGKQWE |

ИМЯ МИНН - ГОДОСЧЕ

```
HIBERNATE_EAGER_LOADING = true
1 - filter("FIRSTNAME='Петр' AND LASTNAME='Иванов')
2 - access("MIDDLENAME='Сергеевич') + lazy loading
```

**Рис. 10.29.** План запроса поиска клиентов. Индексы — B<sup>+</sup>-дерево CLIENT\_LASTNAME | CLIENT\_FIRSTNAME | CLIENT\_MIDDLENAME |

Остальные два предиката: `lastName='Иванов' AND firstName='Петр'` применяются как фильтр на результирующем наборе строк, который возвращает операция с `ID = 1`. Актуальное (A-Rows) количество считанных строк на этапе операции с `ID = 2` равно 302, что также недостаточно хорошо для получения одной строки.

Таким образом, из приведенных примеров видно, что, несмотря на наличие трех отдельно существующих индексов для трех колонок, которые присутствуют в условии поиска в конструкции **WHERE SQL-запроса**, используется только один индекс, что не всегда является достаточно эффективным с точки зрения доступа к данным в рамках текущего запроса.

Рассмотрим ситуацию, когда на колонках lastName, firstName и middleName создан один составной индекс. Таким образом, на текущий момент существуют три разных индекса, каждый из которых создан на одной колонке таблицы: CLIENT\_LASTNAME\_I, CLIENT\_FIRSTNAME\_I, CLIENT\_MIDDLENAME\_I, и один составной индекс CLIENT\_COMPOSITE\_I на совокупности колонок lastName, firstName и middleName. Проанализируем полученный план SQL-запроса.

## Исходные данные:

Таблица: client

Количество строк: 220 000 из 220 000. ЕГЭ по математике. ЕГЭ по математике. ЕГЭ по математике. ЕГЭ по математике.

## Поиск клиента по фамилии

Составной индекс на основе B<sup>+</sup>-дерева CLIENT\_COMPOSITE\_I

## Индексы на основе B<sup>+</sup>-дерева CLIENT LAST

**FIRSTNAME** I, **CLIENT** **MIDDLENAME** I

План выполнения запроса представлен на рис. 10.30.

План запроса, представленный на рис. 10.30

рация с ID = 2 использует индекс CLIENT COMPOSITE I для по-

ОС НИЖЕМЯННОЕ ЕДИСТАВЛЕНИЕ КОМПАНИИ - ГОСУДАРСТВЕННОЕ НАЧАЛО МИНИСТЕРСТВО ПО РЕГИОНАЛЬНОМУ РАЗВИТИЮ И ТЕХНОЛОГИЧЕСКОМУ РАЗВИТИЮ РОССИЙСКОЙ ФЕДЕРАЦИИ

| ID  | Operation Name              | Name               | Starts | E-Rows | A-Rows |
|-----|-----------------------------|--------------------|--------|--------|--------|
| 0   | SELECT STATEMENT            |                    | 1      | 1      | 1      |
| 1   | TABLE ACCESS BY INDEX ROWID | CLIENT             | 1      | 1      | 1      |
| * 2 | INDEX RANGE SCAN            | CLIENT_COMPOSITE_I | 1      | 1      | 1      |

-----  
2 - access("LASTNAME"='Измаков' AND "FIRSTNAME"='Петр' AND "MIDDLENAME"='Сергеевич')

**Рис. 10.30.** План запроса поиска клиентов. Составной индекс —  $B^+$ -дерево

иска всех строк, удовлетворяющих полному условию **WHERE**  
**lastname='Иванов' AND firstname='Петр' AND middlename='Сергеевич'**. В данном случае оптимизатор считает более эффективным использовать индекс **CLIENT\_COMPOSITE\_I**. Актуальное (A-Rows) количество считанных строк на этапе операции с **ID = 2** равно 1, что является вполне удовлетворительным для получения одной строки.

В рассмотренном примере (см. рис. 10.30) метод доступа к данным с использованием составного индекса является более эффективным по сравнению с SQL-запросами, в которых используются несколько индексов, построенных на одной колонке, поскольку в данном случае количество считанных строк на этапе доступа к данным равно количеству строк полученного результата.

В отличие от индексов на основе  $B^+$ -дерева индексы на основе битовых карт, построенные по одной колонке, можно достаточно эффективно комбинировать, особенно при использовании условия OR. Рассмотрим пример запроса и соответствующий план с использованием операции OR для индексов на основе  $B^+$ -дерева и bitmap-индексов:

```
SELECT COUNT(1) FROM client WHERE lastName='Иванов' OR
firstName='Петр'
OR middleName='Сергеевич';
```

## Исходные данные:

## Таблица: client

Количество строк: 220 000

Поиск клиента по фамилии, или имени, или отчеству  
Индексы на основе B<sup>+</sup>-дерева CLIENT\_COMPOSITE\_I, CLIENT\_LASTNAME\_I, CLIENT\_FIRSTNAME\_I, CLIENT\_MIDDLENAME\_I

~~MIDDLENAME\_1~~

План выполнения запроса представлен на рис. 10.31.

**Рис. 10.31.** План запроса поиска количества клиентов, используя условие OR и индексы на основе B<sup>+</sup>-дерева

План запроса, представленный на рис. 10.31, показывает, что, несмотря на наличие как составного, так и несоставных индексов на основе  $B^+$ -дерева, оптимизатор запросов их не использует, что приводит к полному сканированию таблицы (операция с ID = 2). Актуальное (A-Rows) количество считанных строк на этапе операции с ID = 2 равно 14 547. Стоимость запроса (Cost(%CPU)) равна 1914.

Рассмотрим ситуацию, когда на каждой из колонок lastName, firstName, middleName созданы индексы на основе битовых карт: CLIENT\_LASTNAME\_BM\_I, CLIENT\_FIRSTNAME\_BM\_I, CLIENT\_MIDDLENAME\_BM\_I. Проанализируем полученный план SQL-запроса.

План выполнения запроса представлен на рис. 10.32.

**Рис. 10.32.** План запроса поиска количества клиентов, используя условие OR и bitmap-индексы

План запроса, представленный на рис. 10.32, показывает, что оптимизатор запросов считает использование существующих bitmap-индексов эффективным методом доступа к данным, условие отбора

которых использует операцию OR. Учитывая специфику и принципы построения bitmap-индексов, выбранный оптимизатором метод доступа к данным в текущих условиях является вполне предсказуемым. Максимальное актуальное (A-Rows) количество считанных строк на этапе выполнения различных операций не превышает 6. Стоимость запроса (Cost(%CPU)) равна 3. Таким образом, если рассматривать полученные значения стоимости и количества считанных строк в планах выполнения запроса, представленных на рис. 10.31 и 10.32, можно отметить, что в данном случае подход с использованием нескольких bitmap-индексов является более предпочтительным с точки зрения использования ресурсов СУБД при выполнении представленного SQL-запроса.

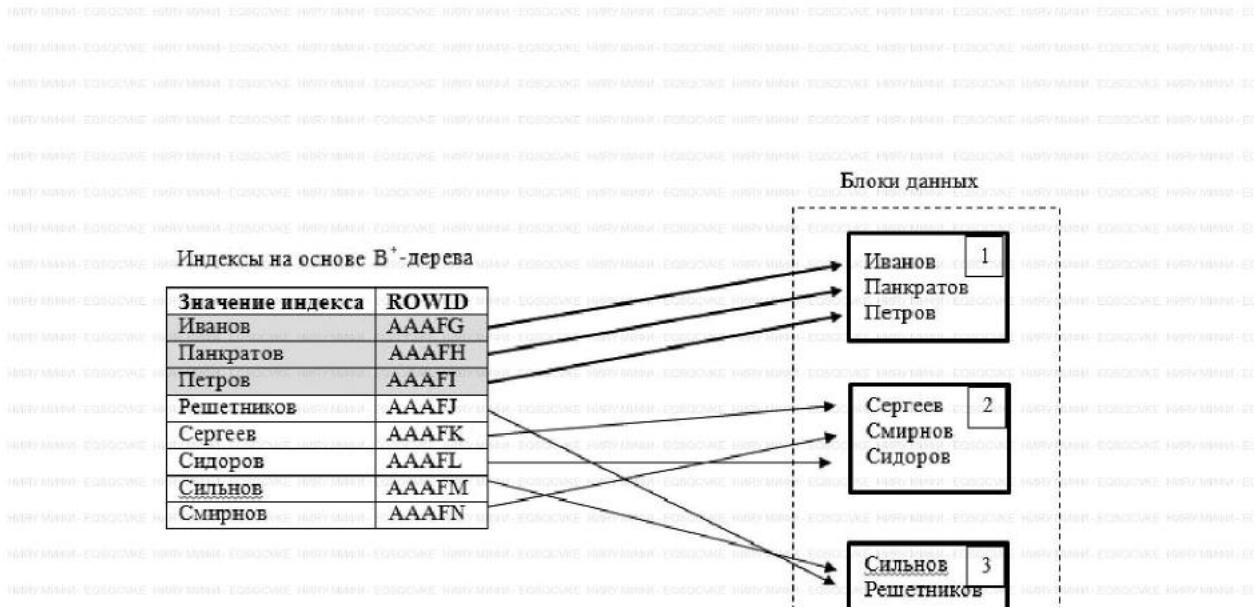
Использование различных типов индексов, для того чтобы обеспечить оптимизатору SQL-запросов возможность выбора более эффективного, на его взгляд, метода доступа к данным, является важной и не всегда простой задачей при разработке приложений, использующих SQL-запросы. В процессе принятия решения — использовать или не использовать тот или иной индекс — оптимизатор SQL-запросов, кроме наличия или отсутствия самого индекса, учитывает и тот факт, насколько последовательно распределены данные относительно их индексов. В СУБД значение, которое показывает степень случайности распределения данных в таблице, называется фактором кластеризации [16].

### **Определение. Фактор кластеризации** — мера того, как синхронизированы индексы с данными в таблице.

С точки зрения производительности процессы ядра СУБД, которые обеспечивают чтение данных, должны избегать построчного чтения. В том числе и по этой причине в современных СУБД чтение выполняется сразу блоками или страницами данных, которые затем располагаются в оперативной памяти в различных областях кэширования. Такой подход сокращает число обращений к внешней памяти. И соответственно от того, как данные распределены по блокам или страницам внешней памяти, и зависит выбор того или иного метода доступа. На рис. 10.33 представлен пример того, как данные могут быть распределены относительно своих индексов.

Например, расположение фамилий Иванов, Панкратов, Петров в одном и том же блоке данных 1 соответствует последовательности, определенной в индексе. Рассмотрим пример запроса, который выбирает список фамилий, определенных предикатом IN:

```
SELECT lastName FROM client c
WHERE lastName IN ('Иванов', 'Панкратов', 'Петров',
'Решетников', 'Сергеев');
```

**Рис. 10.33.** Отношение индексов и блоков данных

План данного запроса, приведенный на рис. 11.34, включает операцию **INDEX RANGE SCAN** — операцию сканирования диапазона индекса **CLIENT\_LASTNAME\_I**.

| Id   Operation                            | Name     | Rows            | Bytes | Cost (%CPU) |
|-------------------------------------------|----------|-----------------|-------|-------------|
| 0   SELECT STATEMENT                      | :<query> | 6 (100)         | 6000  | 6 (100)     |
| 1   INLIST ITERATOR                       | :<query> | 5   105   6 (0) |       |             |
| * 2   INDEX RANGE SCAN  CLIENT_LASTNAME_I |          |                 |       |             |

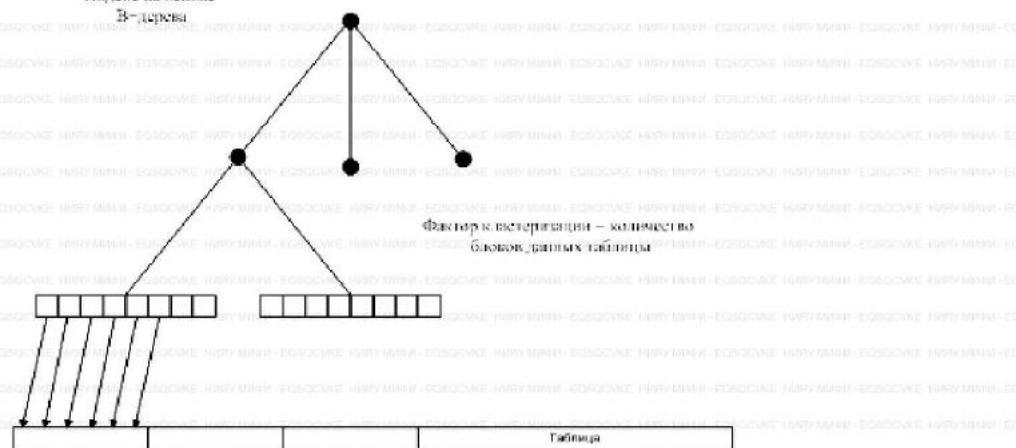
**Рис. 10.34.** План запроса поиска фамилий из списка

По полученным значениям RowId выполняется побочное чтение необходимых данных. Таким образом, для получения фамилии «Иванов» будет полностью считан блок данных 1 и все его содержимое помещается в область кэширования, что в дальнейшем сократит количество медленных операций физического ввода-вывода для получения информации из этого блока данных. Соответственно, фамилии «Иванов», «Панкратов», «Петров» будут получены за одну операцию физического чтения. Далее для получения фамилии «Решетников» будет полностью считан блок данных 3, а для получения фамилии «Сергеев» будет полностью считан блок данных 2.

В итоге для получения  $\frac{2}{3}$  информации в данном примере процессы СУБД должны будут прочитать все блоки данных, что при определенных объемах данных может оказаться на производительности выполняемых SQL-запросов. Текущая ситуация возникла по причине недостаточной степени синхронизации индексов и данных.

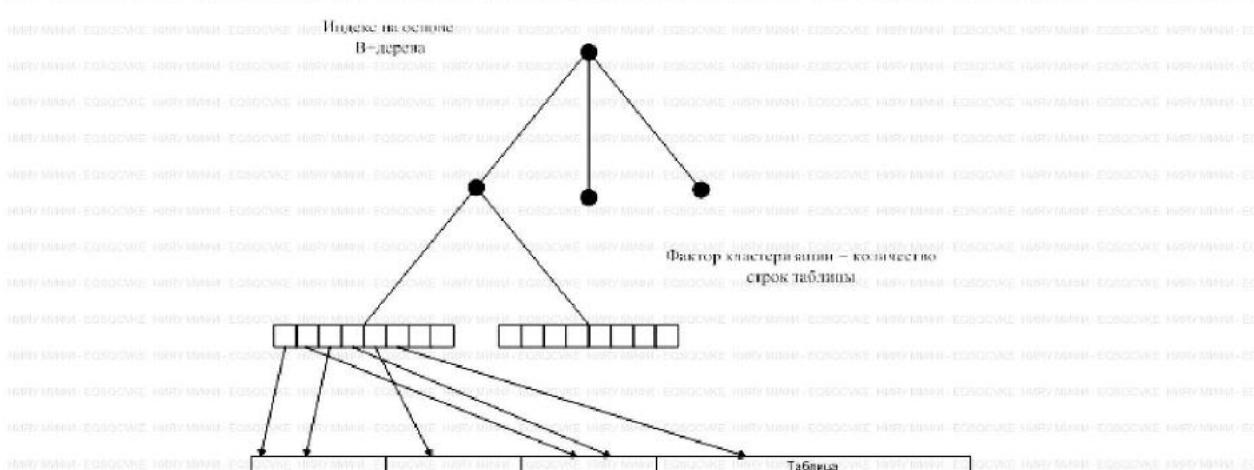
в таблице, что и определяет фактор кластеризации. Фактор кластеризации является оптимальным, когда его значение близко к количеству блоков или страниц данных (рис. 10.35).

в таблице, что и определяет фактор кластеризации. Фактор кластеризации является оптимальным, когда его значение близко к количеству блоков или страниц данных (рис. 10.35).



**Рис. 10.35.** Оптимальная синхронизация индексов и данных в таблице

Фактор кластеризации является неоптимальным, когда его значение близко к количеству строк таблицы (рис. 10.36).



**Рис. 10.36.** Неоптимальная синхронизация индексов и данных в таблице

На основе рассмотренных примеров и представленных подходов по увеличению эффективности выполнения SQL-запросов можно сформулировать следующие рекомендации для повышения производительности SQL-запросов:

*1. Повышение производительности начинается с хорошего дизайна.*

Корректность и точность применяемых конструкций при написании SQL-запросов в значительной степени определяют, насколько эффективно оптимизатор SQL-запросов будет понимать и, как следствие, формировать оптимальный, по его оценкам, план выполнения SQL-запроса. Применение без определенной необходимости функций к атрибутам связи при соединении таблиц, использование неявного преобразования данных, а также отказ от рассмотрения альтернативных вариантов написания запросов для получения одного и того же результата ограничивают возможности СУБД принимать ряд, возможно, более эффективных решений для получения требуемого уровня производительности.

**2. Понимание методов доступа к данным.** Представление о том, как СУБД будет или уже выполняет доступ к данным, является одним из важных принципов формирования эффективных с точки зрения производительности SQL-запросов. Принятие решения о том, какие методы доступа к данным необходимо обеспечить процессам СУБД при выполнении того или иного SQL-запроса, предполагает однозначные и обоснованные ответы на вопросы типа: использовать ли индексы? если индексы используются, то на основе какой структуры? использовать ли составные или несоставные индексы? И так далее.

*3. Только наличия индексов может быть недостаточно.* То, как данные распределены на уровне логических структур, определяет, какие методы доступа к данным будут использоваться СУБД. Даже при наличии индексов, сформированных в соответствии с очевидной их необходимостью, оптимизатор СУБД может принять решение о том, что их использование будет более затратным с точки зрения использования ресурсов системы. И это решение, возможно, будет обосновано именно тем, что синхронизация существующих индексов и данных в таблице является неоптимальной.

**4. Понимание данных очень важно!** Любой процесс, направленный на улучшение производительности, не будет достаточно эффективен без понимания того, к каким данным, в том числе с точки зрения семантики, будут применяться различные методы и подходы для увеличения эффективности выполнения того или иного SQL-запроса. Недостаточное понимание данных может привести к неверной трактовке проблемы, связанной с эффективностью выполнения SQL-запроса и, как следствие, к принятию ошибочных мер по устранению данной проблемы, а возможно, даже усугубит ее. Понимание данных является критически важным этапом процесса повышения производительности SQL-запросов.

## Вопросы

1. Перечислите методы соединения источников данных SQL-запроса и опишите принципы работы каждого из них.
  2. Сформулируйте правила следования основных операций плана выполнения SQL-запроса.
  3. Напишите SQL-запрос, который возвращает один и тот же результат, всеми известными вам способами. Проанализируйте план выполнения каждого из них на разных объемах данных.
  4. Напишите SQL-запрос, который возвращает один и тот же результат, тремя разными способами так, чтобы план выполнения каждого SQL-запроса использовал для соединения двух одних и тех же таблиц соединения типа: соединение вложенных циклов, соединение хэшированием, соединения сортировки и слияния.
  5. Используйте таблицы с несколькими тысячами строк. Напишите SQL-запросы, которые используют как одну, так и несколько колонок в качестве атрибута соединения, а также применяйте предикаты и подзапросы. Проанализируйте план выполнения запроса и постройте необходимые индексы в различных вариантах: на одной колонке и составные, на основе  $B^+$ -дерева и Bitmap-индексы. Проанализируйте полученные планы SQL-запроса и оцените эффективность использования индексов.

ЛИТЕРАТУРА

1. Коннолли Т., Бэгг К., Страчан А. Базы данных: проектирование, реализация, сопровождение. Теория и практика: Пер. с англ. М.: Вильямс, 2000.
  2. Дейт К.Дж. Введение в системы баз данных. 8-е изд.: Пер. с англ. М.: Вильямс, 2005.
  3. Ульман Дж. Основы систем баз данных: Пер. с англ. М.: Финансы и статистика, 1983.
  4. Мейер Д. Теория реляционных баз данных: Пер. с англ. М.: Мир, 1987.
  5. Мартин Дж. Организация баз данных в вычислительных системах: Пер. с англ. М.: Мир, 1998.
  6. Чен П. Модель сущность–связь — шаг к единому представлению данных // СУБД. 1995. № 3. С. 137–158.
  7. Кодд Е.Ф. Реляционная модель данных для больших совместно используемых банков данных // СУБД. 1995. № 1. С. 145–160.
  8. Тиори Т., Фрай Дж. Проектирование структур баз данных. Т. 2.: Пер. с англ. М.: Мир, 1985.
  9. Стандарт SQL-92: ISO/IEC 9075:1992. <http://www.iso.org/>
  10. Спецификации на стандарт IDEF. <http://www.idef.com>
  11. Ильиных Т.Е., Шустова Л.И. Проектирование реляционных баз данных в нотациях IDEF1X. М.: МИФИ, 2000.
  12. Документация по MS SQL Server. <http://msdn.microsoft.com/en-us/library – Servers and Enterprise Development – SQL Server>.
  13. Кнут Д. Искусство программирования. Т. 3. Сортировка и поиск: Пер. с англ. М.: Вильямс, 2007.
  14. Джонатан Льюис. Oracle. Основы стоимостной оптимизации. СПб.: Питер, 2007.
  15. Antognini Ch. Troubleshooting Oracle Performance. Apress, 2008.
  16. Документация по Oracle Database <http://docs.oracle.com/ Database Performance Tuning Guide>.
  17. Dyke J. Bitmap Index Internals, 2005. <http://www.juliandyke.com>



|                                                        |     |
|--------------------------------------------------------|-----|
| <b>ОГЛАВЛЕНИЕ</b>                                      |     |
| <b>Введение</b>                                        | 3   |
| <b>Предисловие</b>                                     | 5   |
| <b>ГЛАВА 1<br/>ОСНОВНЫЕ ПОНЯТИЯ</b>                    | 7   |
| 1.1. Понятие данных                                    | 7   |
| 1.2. Файловые системы                                  | 8   |
| 1.3. Системы баз данных                                | 9   |
| 1.4. История развития СУБД                             | 10  |
| 1.5. Трехуровневая архитектура ANSI/SPARC              | 11  |
| Вопросы                                                | 14  |
| <b>ГЛАВА 2<br/>ОБЩАЯ ХАРАКТЕРИСТИКА МОДЕЛЕЙ ДАННЫХ</b> | 15  |
| 2.1. Основные понятия модели данных                    | 15  |
| 2.2. Представление статических и динамических свойств  | 16  |
| 2.3. Общая характеристика структурных компонентов.     | 17  |
| Множества: домены и атрибуты                           | 17  |
| 2.4. Общая характеристика структурных компонентов.     | 20  |
| Отношения: сущности                                    | 20  |
| 2.5. Общая характеристика структурных компонентов.     | 21  |
| Отношения: связи                                       | 21  |
| 2.6. Общая характеристика ограничений целостности      | 24  |
| Вопросы                                                | 27  |
| <b>ГЛАВА 3<br/>МОДЕЛЬ ДАННЫХ «СУЩНОСТЬ-СВЯЗЬ»</b>      | 28  |
| 3.1. Уровни представления информации                   | 28  |
| 3.2. Уровень 1. Информация о сущностях и связях        | 28  |
| 3.2.1. Сущности: тип сущности и множество сущностей    | 29  |
| 3.2.2. Связи, роли и множество связей                  | 30  |
| 3.2.3. Атрибут, значение и множество значений          | 31  |
| 3.3. Уровень 2. Структура информации                   | 33  |
| 3.3.1. Представление сущностей                         | 33  |
|                                                        | 299 |

|                                                                        |    |
|------------------------------------------------------------------------|----|
| <b>3.3.2. Представление связи</b>                                      | 35 |
| <b>3.3.3. Некоторые особенности представления «сущности и связи»</b>   | 35 |
| <b>3.3.4. Язык описания данных: диаграмма «сущность–связь»</b>         | 36 |
| <b>3.3.5. Общая характеристика связей</b>                              | 37 |
| <b>3.4. Расширенная модель данных «сущность–связь»: нотация IDEF1x</b> | 39 |
| <b>3.5. Ограничения целостности в модели «сущность–связь»</b>          | 44 |
| <b>Вопросы</b>                                                         | 46 |
| <b>ГЛАВА 4</b>                                                         |    |
| <b>ИЕРАРХИЧЕСКАЯ И СЕТЕВАЯ МОДЕЛИ ДАННЫХ</b>                           |    |
| <b>4.1. Иерархическая модель данных</b>                                | 47 |
| <b>4.2. Сетевая модель данных</b>                                      | 48 |
| <b>Вопросы</b>                                                         | 49 |
| <b>ГЛАВА 5</b>                                                         |    |
| <b>РЕЛЯЦИОННАЯ МОДЕЛЬ ДАННЫХ</b>                                       |    |
| <b>5.1. Базовые структурные компоненты реляционной модели данных</b>   | 51 |
| <b>5.1.1. Домены, атрибуты и отношения</b>                             | 52 |
| <b>5.1.2. Представление сущности</b>                                   | 55 |
| <b>5.1.3. Связи</b>                                                    | 56 |
| <b>5.2. Целостная часть реляционной модели данных</b>                  | 59 |
| <b>5.2.1. Целостность сущностей</b>                                    | 60 |
| <b>5.2.2. Ссылочная целостность</b>                                    | 60 |
| <b>5.3. Манипуляционная часть реляционной модели данных</b>            | 62 |
| <b>5.3.1. Общая характеристика</b>                                     | 62 |
| <b>5.3.2. Реляционная алгебра. Общая характеристика</b>                | 63 |
| <b>5.3.3. Теоретико-множественные операции</b>                         | 65 |
| <b>5.3.4. Специальные операции</b>                                     | 70 |
| <b>5.3.5. Реляционное исчисление</b>                                   | 76 |
| <b>5.3.6. Общая характеристика языков манипулирования данными</b>      | 85 |
| <b>Вопросы</b>                                                         | 87 |
| <b>ГЛАВА 6</b>                                                         |    |
| <b>ТЕОРИЯ ПРОЕКТИРОВАНИЯ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ</b>                    |    |
| <b>6.1. Цели проектирования</b>                                        | 89 |
| <b>6.2. Функциональные зависимости</b>                                 | 91 |
| <b>6.2.1. Основные понятия</b>                                         | 91 |
| <b>6.2.2. Замыкание множества функциональных зависимостей</b>          | 94 |
| <b>6.2.3. Правила вывода Армстронга</b>                                | 94 |

|                                                                          |            |
|--------------------------------------------------------------------------|------------|
| <b>6.2.4. Определение ключа.....</b>                                     | <b>97</b>  |
| <b>6.2.5. Декомпозиция с соединением без потерь.....</b>                 | <b>98</b>  |
| <b>6.3. Нормализация отношений.....</b>                                  | <b>102</b> |
| <b>6.3.1. Первая нормальная форма .....</b>                              | <b>103</b> |
| <b>6.3.2. Вторая нормальная форма .....</b>                              | <b>106</b> |
| <b>6.3.3. Третья нормальная форма.....</b>                               | <b>108</b> |
| <b>6.3.4. Нормальная форма Бойса–Кодда.....</b>                          | <b>109</b> |
| <b>6.3.5. Многозначные зависимости .....</b>                             | <b>113</b> |
| <b>6.3.6. Четвертая нормальная форма .....</b>                           | <b>116</b> |
| <b>6.3.7. Пятая нормальная форма.....</b>                                | <b>117</b> |
| <b>Вопросы .....</b>                                                     | <b>118</b> |
| <b>ГЛАВА 7</b>                                                           |            |
| <b>ЯЗЫК SQL.....</b>                                                     | <b>119</b> |
| <b>7.1. Общая характеристика языка SQL.....</b>                          | <b>121</b> |
| <b>7.2. Описание предметной области<br/>«Прокат автомобилей» .....</b>   | <b>121</b> |
| <b>7.3. Представление данных.....</b>                                    | <b>123</b> |
| <b>7.3.1. Способы именования объектов .....</b>                          | <b>123</b> |
| <b>7.3.2. Основные типы данных.....</b>                                  | <b>125</b> |
| <b>7.3.3. Выражения.....</b>                                             | <b>133</b> |
| <b>7.3.4. Предикаты .....</b>                                            | <b>140</b> |
| <b>7.3.5. Условие поиска.....</b>                                        | <b>144</b> |
| <b>Вопросы .....</b>                                                     | <b>145</b> |
| <b>7.4. Средства SQL для описания объектов базы данных .....</b>         | <b>146</b> |
| <b>7.4.1. Создание таблицы.....</b>                                      | <b>147</b> |
| <b>7.4.2. Примеры создания таблиц .....</b>                              | <b>155</b> |
| <b>7.4.3. Удаление и модификация таблиц .....</b>                        | <b>157</b> |
| <b>7.4.4. Создание и использование последовательности .....</b>          | <b>163</b> |
| <b>7.4.5. Язык управления доступом DCL (Data Control Language) .....</b> | <b>165</b> |
| <b>Вопросы .....</b>                                                     | <b>166</b> |
| <b>7.5. Подмножество SQL для манипулирования данными .....</b>           | <b>168</b> |
| <b>7.5.1. Предложение INSERT .....</b>                                   | <b>168</b> |
| <b>7.5.2. Предложение DELETE .....</b>                                   | <b>173</b> |
| <b>7.5.3. Предложение UPDATE .....</b>                                   | <b>175</b> |
| <b>7.5.4. Формирование запросов к базе данных.....</b>                   | <b>176</b> |
| <b>7.5.5. Предложение SELECT .....</b>                                   | <b>178</b> |
| <b>7.5.6. Использование теоретико-множественных операций .....</b>       | <b>191</b> |
| <b>7.5.7. Формирование расширенных запросов.....</b>                     | <b>194</b> |

|                                                                                                                             |     |
|-----------------------------------------------------------------------------------------------------------------------------|-----|
| <b>7.5.8. Создание представлений — CREATE VIEW</b>                                                                          | 203 |
| <b>Вопросы</b>                                                                                                              | 205 |
|  <b>7.6. Примеры написания запросов</b>    | 206 |
|  <b>7.7. Реализация процедурной логики</b> | 206 |
| <b>ГЛАВА 8 УПРАВЛЕНИЕ ПАРАЛЛЕЛИЗМОМ В СУБД</b>                                                                              | 207 |
| <b>8.1. Понятие транзакции</b>                                                                                              | 207 |
| <b>8.2. Проблемы параллелизма</b>                                                                                           | 210 |
| <b>8.2.1. Потеря результатов обновления</b>                                                                                 | 210 |
| <b>8.2.2. Незафиксированные зависимости</b>                                                                                 | 210 |
| <b>8.2.3. Несовместимый анализ</b>                                                                                          | 211 |
| <b>Вопросы</b>                                                                                                              | 216 |
| <b>ГЛАВА 9 ВНУТРЕННИЕ СТРУКТУРЫ ХРАНЕНИЯ</b>                                                                                | 217 |
| <b>9.1. Структурная схема обработки запроса</b>                                                                             | 217 |
| <b>9.2. Бинарные деревья</b>                                                                                                | 220 |
| <b>9.3. Многоходовые деревья</b>                                                                                            | 222 |
| <b>9.4. B-дерево</b>                                                                                                        | 223 |
| <b>9.4.1. Структура вершины B-дерева</b>                                                                                    | 223 |
| <b>9.4.2. Операция вставки</b>                                                                                              | 224 |
| <b>9.4.3. Удаление элемента</b>                                                                                             | 228 |
| <b>9.5. B+ дерево</b>                                                                                                       | 232 |
| <b>9.5.1. Операция включения</b>                                                                                            | 233 |
| <b>9.5.2. Удаление элемента</b>                                                                                             | 235 |
| <b>9.6. Хэш-таблицы</b>                                                                                                     | 237 |
| <b>9.6.1. Идея хэширования</b>                                                                                              | 237 |
| <b>9.6.2. Принципы построения хэш-индекса</b>                                                                               | 238 |
| <b>9.6.3. Методы обработки переполнения</b>                                                                                 | 239 |
| <b>9.7. Битовые индексы или индексы на основе битовых карт</b>                                                              | 241 |
| <b>9.7.1. Внутренняя структура bitmap-индекса</b>                                                                           | 243 |
| <b>9.7.2. Операция обновления bitmap-индекса</b>                                                                            | 247 |
| <b>9.8. Сравнение методов индексирования</b>                                                                                | 250 |
| <b>Вопросы</b>                                                                                                              | 253 |

ГЛАВА 10

## **ОСНОВЫ ПОСТРОЕНИЯ И АНАЛИЗА ПЛАНОВ ВЫПОЛНЕНИЯ SQL-ЗАПРОСОВ**

..254

|                                                                                                      |        |
|------------------------------------------------------------------------------------------------------|--------|
| <b>10.1. Методы соединения источников данных SQL-запроса</b>                                         | 259    |
| 10.1.1. Соединение с использованием вложенных циклов (NESTED LOOPS JOIN)                             | 260    |
| 10.1.2. Хэш-соединение (HASH JOIN)                                                                   | 262    |
| 10.1.3. Соединение сортировки и слияния (SORT MERGE JOIN)                                            | 264    |
| 10.1.4. Перекрестное соединение (CORTASIAN JOIN)                                                     | 267    |
| <b>10.2. План выполнения SQL-запроса.</b>                                                            |        |
| Интерпретация основных операций                                                                      | 267    |
| План выполнения SQL-запроса                                                                          | 267    |
| <b>10.3. Анализ эффективности выполнения SQL-запросов.</b>                                           |        |
| Возможные подходы повышения производительности SQL-запросов                                          | 276    |
| Вопросы                                                                                              | 296    |
| <b>Литература</b>                                                                                    | 297    |
|  <b>Приложения</b> | 298    |
| <b>Приложение 1</b>                                                                                  |        |
| Описание схемы базы данных                                                                           | 298-1  |
| <b>Приложение 2</b>                                                                                  |        |
| Создание базы данных                                                                                 | 298-6  |
| П2.1. SQL-скрипт создания базы данных в MS SQL Server                                                | 298-6  |
| П2.2. SQL-скрипт создания базы данных в Oracle                                                       | 298-8  |
| <b>Приложение 3</b>                                                                                  |        |
| Некоторые встроенные функции SQL                                                                     | 298-13 |
| П3.1. Встроенные функции для MS SQL Server                                                           | 298-13 |
| П3.2. Некоторые встроенные функции в Oracle                                                          | 298-16 |
| <b>Приложение 4</b>                                                                                  |        |
| Содержимое таблиц базы данных                                                                        | 298-20 |

