Министерство науки и высшего образования
Российской Федерации
федеральное государственное бюджетное образовательное
учреждение высшего образования
РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ
ГИДРОМЕТЕОРОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

Т.М. Татарникова

ЗАЩИТА БАЗ ДАННЫХ

Конспект лекций

Санкт-Петербург
РГГМУ
2020

УДК 004.424.2(075.8)
ББК 32.973-018.2я73
Т12

Татарникова Т.М.
Защита баз данных. – СПб.: РГГМУ, 2020. – 164 с.

Рецензент: Цехановский В.В., канд. техн. наук, профессор, заведующий кафедрой информационных сиситем Сангкт-Петербургского государственного электротехнического университета «ЛЭТИ» им. В.И. Ульянова (Ленина).

Курс «Защита баз данных» направлен на изучение основ проектирования баз данных, включая анализ предметной области, моделирование данных и реализацию их физической модели. На примерах рассматриваются основные функции управления данными, такие как определение, манипулирование, поиск, защита данных. Дается характеристика базам данных NoSQL, ориентированных на хранение и обработку больших объемов данных. Приводится сравнительная характеристика существующих архитектурных решений управления доступом к базам данных и механизм транзакций как способ параллельной обработки запросов пользователей.

ISBN 978-5-86813-495-1

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Курс «Защита баз данных» включает несколько разделов, которые логически выстроены для лучшего усвоения материала.

Защита данных – это одна из многих функций управления данными. Управлять данными напрямую с физического носителя, где они хранятся невозможно. Поэтому физические данные отображаются в логическую модель – базу данных, управление которой берет на себя система управления базами данных (СУБД).

Большая часть лекционного материала посвящена процессу проектирования баз данных и управления данными с помощью СУБД. Описание процесса проектирования, включает анализ предметной области, моделирование данных и реализацию физической модели данных. Все этапы проектирования сопровождаются примерами.

Логические модели могут быть разными, но в первую очередь рассматриваются реляционные базы данных.

Также должное внимание уделено новой технологии NoSQL, ориентированной на хранение и обработку больших объемов данных. Основным свойством технологии NoSQL, является возможность функционирования баз данных на большом кластере. Работа на кластере повышает сложность базы данных и требует распределения данных внутри кластера – по серверам. В лекциях подробно рассмотрены такие методы распределения как репликация и фрагментация.

Одним из глобальных отличий реляционной базы данных от базы данных NoSQL – это то как обеспечивается согласованность данных. Согласованность данных в реляционной модели решается с использованием механизма транзакций, а в NoSQL является компромиссом трех свойств – согласованности, доступности и устойчивости к разделению данных. Эти вопросы также обсуждаются в курсе лекций.

Приводится сравнительная характеристика существующих архитектурных решений управления доступом к базам данных.

Обсуждаются механизмы обеспечения целостности, защиты от сбоев и конфиденциальности данных.

1. ВВЕДЕНИЕ В УПРАЛЕНИЕ ДАННЫМИ

В главе приведены основные понятия и определения, которыми оперируют специалисты в области управления данными, дана характеристика функциям СУБД и рассматриваются модели структурированных данных.

1.1 Основные понятия и определения

Управление данными – процессы, связанные с автоматизацией сбора, накоплением, хранением и использованием информации появляющейся
в результате деятельности человека.

Данные – это машинное представление информации – то что обрабатывает и хранит компьютер.

Употребление термина «**информационная система**» говорит об участии человека в процессах хранения, обработки, выдачи информации. Термин «**база данных**» говорит об определенным образом организованной совокупности данных, хранящихся во внешней памяти компьютера.

База данных (БД) – это именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области. БД также принято называть хранилищем данных информационной системы

Объект – человек, предмет, событие, место или понятие, о котором записаны данные (например, в банковском деле примерами объектов могут служить клиенты, банковские счета, ссуды по закладам и т.д.). Таким образом, речь идет не о физических объектах, а об информационных объектах (данных).

Предметная область – часть реального мира, отражаемая в БД, может относиться к любому типу организации (например, банк, университет, завод, больница).

Чтобы управлять данными они должны быть определенным образом представлены, например выбрана их структура, модель представления, методы обработки, решен вопрос организации использования этих данных в различных вычислительных средах.

В общем случае место процесса управления данными можно отразить следующей схемой, приведенной на рис. 1.1.

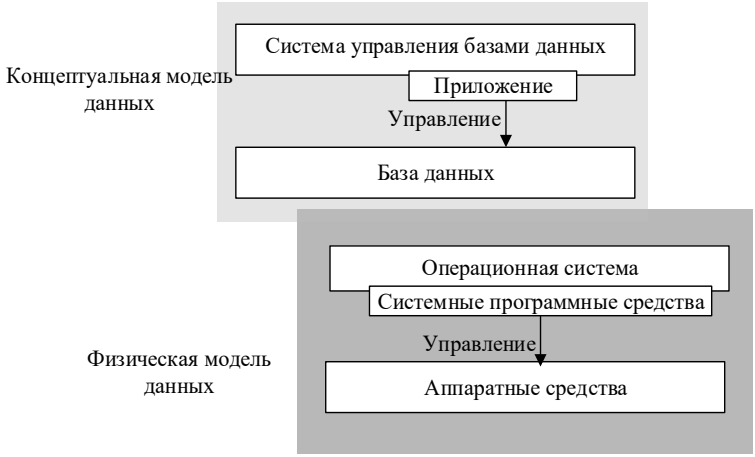


Рис. 1.1. Общее представление процесса управления данными

Данные в вычислительной среде (совокупность аппаратного обеспечения и операционной системы) все одинаковы, то есть имеют машинное представление. Для работы с данными, они представляются в виде какой-то модели хранения, то есть в виде базы данных (БД). Можно сказать, что БД – это концептуальная модель отображения физической модели данных. Управление этими данными реализует система управления базами данных (СУБД), равно как и создание самой базы данных.

Система управления базами данных (СУБД) – совокупность языковых и программных средств, предназначенных для создания, ведения и совместного использования БД многими пользователями.

Пользователи могут работать с БД средствами различных приложений, установленных на компьютере или смартфоне.

Приложения – специальные программы, с помощью которых пользователи работают с базой данных.

Последовательность управления данными рассмотрим на схеме, приведенной на рис.1.2.

1. Пользователь посылает СУБД запрос на получение данных (1).
2. Система проверяет легитимность пользователя (2), (3). Если она подтверждается, то СУБД получает информацию о запрошенной части концептуальной модели (3), (4) и запрашивает информацию о местоположении данных на физическом уровне – файлы и их

физические адреса (6). В СУБД возвращается информация о местоположении данных в терминах операционной системы (7).

3. СУБД просит операционную систему (ОС) предоставить необходимые данные (8). ОС осуществляет перекачку информации с устройств хранения и пересылает ее в системный буфер (9). По окончании пересылки ОС оповещает СУБД об этом (10).

4. СУБД выбирает из доставленной информации, находящейся в системном буфере, только то, что нужно пользователю (11), и пересылает эти данные в рабочую область пользователя (12) – облако или внешнюю память.

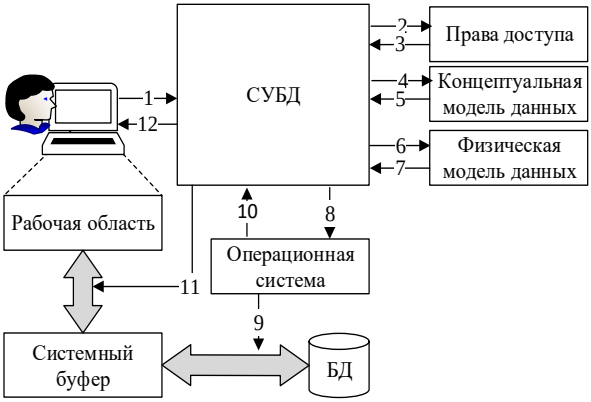


Рис. 1.2. Последовательность управления данными

1.2 Функции СУБД

К числу основных функций СУБД принято относить следующие:

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

1) Непосредственное управление данными во внешней памяти

Как известно, для сохранности данных и дальнейшей работы с ними, нужно иметь в наличии постоянные запоминающие устройства (носители информации), например HDD, SSD, Raid и другие. Собственно, носители информации – это и есть аппаратные средства физической модели данных, приведенной на рис. 1.1.

Носители информации нужны как для хранения данных, непосредственно входящих в БД, так и для служебной информации (адресов, индексов, метаданных), которая служит, например, для убыстрения доступа к данным.

2) Управление буферами оперативной памяти

Как правило, система управления базами данных использует собственный механизм работы с устройствами внешней памяти. СУБД самостоятельно взаимодействует с файловой системой устройства внешней памяти, что позволяет пользователю не задумываться над особенностями работы СУБД на нижнем уровне.

Поскольку при обращении к любому элементу данных производится обмен данными с внешней памятью, то, очевидно, что вся система будет работать со скоростью внешнего носителя. Единственным на практике способом ускорения работы с БД является буферизация данных в оперативной памяти.

В оперативной памяти создаются буферы (кеш) с теми данными, которые в этот момент используются. Таким образом, в буферах временно хранятся фрагменты БД, данные из которых предполагается использовать при обращении к СУБД или планируется записать в базу после обработки.

В современных СУБД поддерживается собственный набор буферов оперативной памяти с механизмами их обслуживания.

3) Управление транзакциями

Транзакция – это некоторая неделимая последовательность операций над данными, которая отслеживается СУБД от начала и до завершения.

Если по каким-либо причинам (сбои и отказы оборудования, ошибки в программном обеспечении) транзакция остается незавершенной, то она отменяется.

Механизм транзакций используется в СУБД для поддержания логической целостности данных в базе.

Пусть БД содержит два файла СОТРУДНИКИ и ОТДЕЛЫ. Тогда, единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотрудника является объединение элементарных операций над файлами СОТРУДНИКИ и ОТДЕЛЫ в одну транзакцию.

Поддержание механизма транзакций является обязательным условием СУБД.

Свойство целостности БД до начала выполнения транзакции и после ее завершения делает удобным использование понятия транзакции как единицы активности пользователя по отношению к БД.

При соответствующем управлении параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей ощущает себя единственным пользователем СУБД.

4) Журнализация

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти.

Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя.

Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризуемые потерей информации во внешней памяти.

Примерами программных сбоев могут быть: аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной.

Очевидно, что при любом сбое для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, поддержание надежности хранения данных в БД требует избыточности хранения данных, причем та часть данных, которая используется для восстановления, должна храниться особо надежно.

Наиболее распространенным методом поддержания избыточной информации является ведение журнала изменений БД.

Журнал – это особая часть БД, недоступная пользователям СУБД, которая поддерживается специальными средствами. Иногда поддерживаются две копии журнала, располагаемые на разных физических дисках, в которые поступают записи обо всех изменениях БД.

5) Поддержка языков БД

Любая СУБД должна поддерживать языки баз данных для работы с данными.

Для реляционных баз данных стандартным языком использования сталь язык SQL (Stuctured Query Language). Этот язык позволяет определять модель данных и манипулировать этими данными.

Для нереляционных БД пока стандарты отсутствует, имеются пионерские разработки, аналогичные SQL-подобным языкам, например GraphQL; UnQL (Unstructured Data Query Language) и другие.

1.3. Модели структурированных данных

Система управления базами данных основывается на использовании определенной модели данных, отражающих взаимосвязи объектов. Современная классификация СУБД предусматривает реализацию моделей двух групп. В первую группу входят иерархические, сетевые и реляционные модели данных, при чем последние де-факто стали стандартом баз данных. Основу моделей этой группы образуют структурированные данные.

Вторую группу образуют модели неструктурированных данных, на базе которых проектируются так называемые базы данных NoSQL.

1.3.1 Иерархическая модель данных

Иерархическая (древовидная) структура строится из узлов и ветвей. Узел представляет собой совокупность атрибутов данных, описывающих некоторый объект. Наивысший узел древовидной структуры называется корнем. Зависимые узлы располагаются на более низких уровнях дерева. Уровень, на котором находится текущий узел, определяется расстоянием от корневого узла (рис. 1.3).

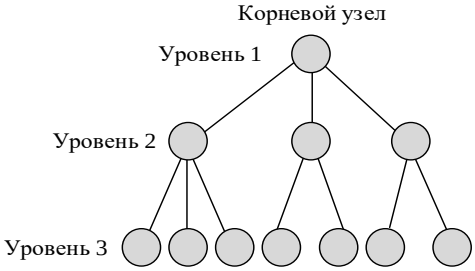


Рис. 1.3. Иерархическая древовидная структура

Иерархическая модель организует данные в виде древовидной структуры. Каждый экземпляр корневого узла образует начало записи логической базы данных. Таким образом, иерархическая база данных состоит из нескольких деревьев. В иерархической модели данных узлы, находящиеся на уровне 2, называются порожденными узла 1-го уровня. Узел на уровне 1 называется исходным для узлов на уровне 2. Узлы, находящиеся на уровне 3, считаются порожденными узла 2-го уровня, который для них является исходным, и т.д.

Иерархическая структура всегда удовлетворяет следующим условиям:

1. Иерархия неизменно начинается с корневого узла.

2. Каждый узел состоит из одного или нескольких атрибутов, которые описывают объект в данном узле.

3. На низших уровнях могут находиться зависимые узлы. Узел, находящийся на предшествующем уровне, является исходным для новых зависимых узлов. Зависимые узлы могут добавляться как в вертикальном, так и в горизонтальном направлении без всяких ограничений. Исключение составляет только первый уровень: на первом уровне может находиться только один узел, называемый корневым.

4. Каждый узел, находящийся на уровне 2, соединен с одним и только одним узлом на уровне 1. Каждый узел, находящийся на уровне 3, соединен с одним и только одним узлом, находящимся на уровне 2 и т.д. Поскольку между двумя узлами может существовать лишь одна дуга (соединение), то дуги не нуждаются в метках.

5. Исходный узел может иметь в качестве зависимых один или несколько узлов, что соответствует связи типа «один-ко-многим» или «1:M» (рис. 1.4). Если узел не имеет ни одного зависимого узла, он не является исходным.

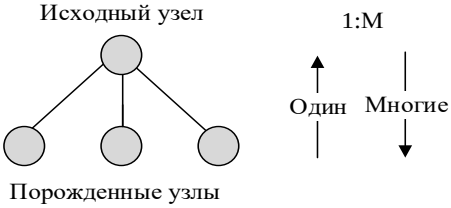


Рис. 1.4. Связь «один-ко-многим»

6. Доступ к каждому узлу, за исключением корневого, происходит через исходный узел. Выборка каждого узла, представленного в иерархии, осуществляется через его исходный узел. В связи с этим в иерархической модели данных пути доступа к каждому узлу являются уникальными. Например, на рис. 1.5 доступ к узлу I может осуществляться по пути A-G-H-I, а доступ к узлу D – по пути A-B-D. Поэтому иерархическая модель данных обеспечивает только линейные пути доступа к данным.

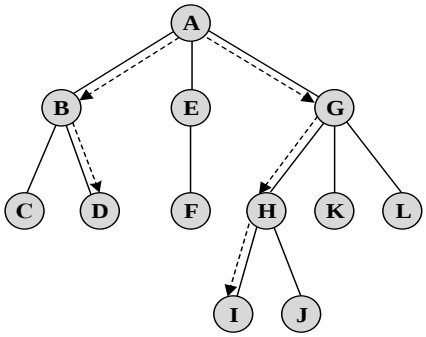


Рис. 1.5. Линейные пути доступа в иерархической модели данных

Пример иерархической БД, в которой хранятся сведения об успеваемости студентов университета приведен на рис. 1.6: в университете имеются несколько факультетов, на каждом факультете – несколько кафедр, на кафедре – несколько групп, в группе – определенное количество студентов, каждый студент имеет результаты сдачи сессии по нескольким дисциплинам. Везде связи между уровнями соответствуют «один-ко-многим» – 1:M.

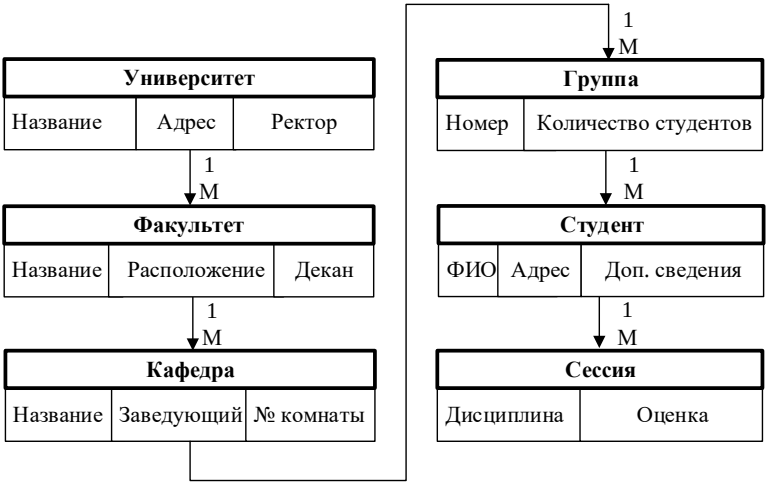


Рис. 1.6. Иерархическая модель БД «Сведения об успеваемости студентов университета

Поиск любой информации, например конкретного студента, сведений об его успеваемости, список групп, сведений о факультете или кафедре начнется с корневого узла «Университет».

Рассмотрим каким образом происходит включение (добавление) и удаление данных в иерархической модели.

Пусть дан фрагмент иерархической модели данных, представленный на рис. 1.7. Исходным узлом является ПАЦИЕНТ, порожденным узлом, в котором хранятся сведения о лечении пациента – ВРАЧ. Если у врача на лечении более одного пациента, то сведения о нем дублируются для каждого пациента.

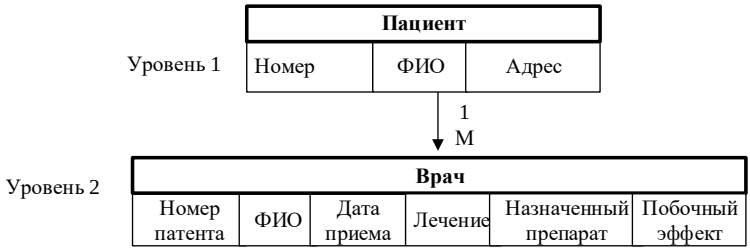


Рис. 1.7. Пример иерархической модели данных «ПАЦИЕНТ-ВРАЧ»

Включение данных. Экземпляр порожденного узла не может существовать в отсутствии экземпляра исходного узла. Если иерархическая модель данных подобна представленной на рис. 1.7, то в такую базу данных невозможно включить сведения о враче, под наблюдением которого еще нет ни одного пациента.

Удаление данных. При удалении экземпляра исходного узла также удалятся и все экземпляры порожденных узлов. Например, в иерархической модели данных, показанной на рис. 1.8, при удалении экземпляра узла ВРАЧ одновременно удаляются и все экземпляры узлов, содержащих сведения о пациентах, находящихся на лечении у данного врача. Это приводит к потере информации о пациентах.

Аномалии включения и удаления данных связаны с тем, что в иерархической модели данных взаимосвязи «многие-ко-многим» непосредственно не поддерживаются, а реализуется только взаимосвязь «один-ко-многим». Эти аномалии могут быть частично устранены за счет введения избыточности (повторения данных).

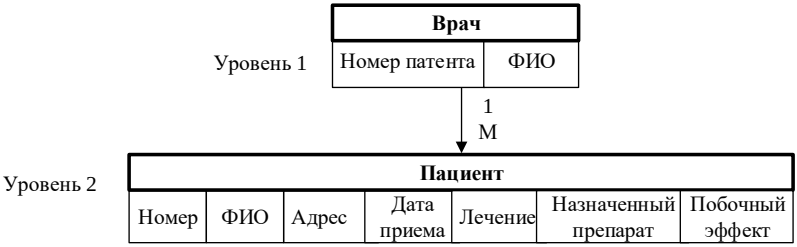


Рис. 1.8. Пример иерархической модели данных «ВРАЧ-ПАЦИЕНТ»

Таким образом, иерархическую модель данных можно охарактеризовать следующими **достоинствами и недостатками**:

– главное достоинство иерархической модели данных – это линейный доступ, что обеспечивает быстрый доступ к необходимым данным.

– главный недостаток иерархической модели данных связан с тем, что структура взаимосвязей между главным и подчиненными узлами только «один-ко-многим». Искусственная реализация взаимосвязей «многие-ко-многим» приводит к тому, что структура становится громоздкой и требуется хранение избыточных данных.

1.3.2. Сетевая модель данных

В сетевой модели данных объекты предметной области объединяются в «сеть». Графически сетевая модель представляется с помощью прямоугольников и стрелок. Эта система обозначений предложена Ч. Бахманом. Каждый тип записи может содержать нуль, или один или несколько атрибутов.

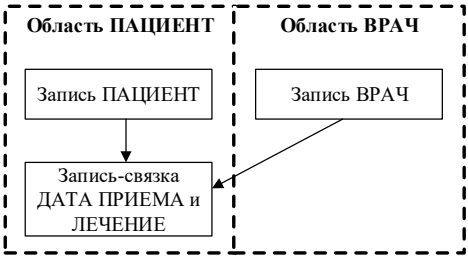


Рис. 1.9. Пример сетевой модели данных

На рис. 1.9 «Область» - это поименованная часть базы данных (участок памяти компьютера), в которой могут содержаться экземпляры записей и наборов или частей наборов. Каждая область может обладать собственными физическими характеристиками. Эти области могут обрабатываться как по отдельности, так и вместе с другими областями. Таким образом, в сетевой модели данных понятия главного и подчиненных объектов несколько расширены: для любого узла допускается несколько входных узлов наряду с возможностью наличия узлов без входов (взаимосвязь «многие-ко-многим»). Любой узел-объект может быть и главным, и подчиненным. Это означает, что каждый объект может участвовать в любом числе взаимосвязей. Графическое представление структуры связей узлов в моделях такого типа представляет собой сеть (рис. 1.10).

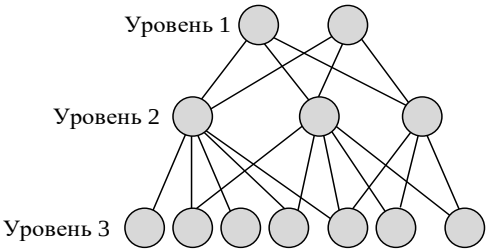


Рис. 1.10. Структура сетевой модели СУБД

Приведем пример фрагмента сетевой БД, хранящей информацию о расписании занятий на кафедре, приведен на рис. 1.11: есть несколько преподавателей, несколько групп, несколько параллельно идущих занятий, и все это связано в сетевую модель по принципу «многие-ко-многим» (M:M).

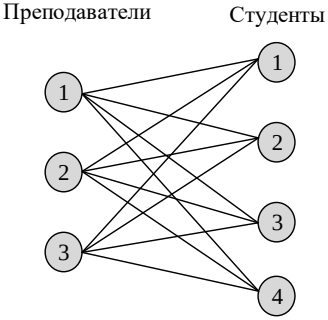


Рис. 1.11. Фрагмент сетевой модели данных

Рассмотрим особенности операций включения и удаления в сетевой модели данных.

Включение. Допускается добавление новой записи ВРАЧ в области ВРАЧ (рис. 1.9), даже если запись о лечении для него отсутствует. Кроме того, можно ввести сведения о враче, который еще не наблюдает ни одного пациента. Аналогичным образом можно вводить сведения о пациенте вне зависимости от сведений о враче.

Удаление. При удалении записи ВРАЧ удаляются все указатели, связывающие лечение, выполненные врачом. Информация же о пациентах, которых наблюдал данный врач, не уничтожается. Удаляется информация только о соответствующем враче. Аналогично при удалении записи ПАЦИЕНТ удаляются все указатели, связывающие перенесенные данным пациентом лечение. Информация же о врачах, наблюдавших данного пациента, остается неизменной.

Главным **достоинством** сетевой модели данных является простота реализации часто встречающихся в реальном мире взаимосвязей «многие-ко-многим».

Основной **недостаток** сетевой модели состоит в ее сложности. Программист БД должен детально знать логическую структуру БД. Недостатком является также возможная потеря независимости данных при реорганизации БД.

1.3.3. Реляционная модель СУБД

Классическая реляционная модель данных требует, чтобы данные хранились в так называемых плоских таблицах. В упрощенном виде плоская таблица – это таблица, каждая ячейка которой может быть однозначно идентифицирована указанием строки и столбца таблицы. Кроме того, в одном столбце все ячейки должны содержать данные одного простого типа.

Реляционная модель требует, чтобы типы данных, представленные в ней были простыми.

Для уточнения этого утверждения рассмотрим, какие вообще типы данных обычно рассматриваются в программировании. Как правило, типы данных делятся на три группы:

– Простые типы данных.

– Структурированные типы данных.

– Ссылочные типы данных.

Простые, или атомарные, типы данных не обладают внутренней структурой. Данные такого типа называют **скалярами**. К простым типам данных относятся следующие типы:

– логический;

– строковый;

– численный.

Различные языки программирования могут расширять и уточнять этот список, добавляя такие типы как:

– целый;

– вещественный;

– дата/время;

– денежный;

– перечислимый;

– и т.д.…

Работая с простыми типами данных, например с числовыми, манипулирование ими выполняется как с неделимыми целыми объектами.

Структурированные типы данных предназначены для задания сложных структур данных. Структурированные типы данных конструируются из составляющих элементов, называемых компонентами, которые, в свою очередь, могут обладать структурой. В качестве структурированных типов данных можно привести следующие типы данных:

– Массивы;

– Записи (Структуры).

При работе с массивами или записями, можно манипулировать ими и как с единым целым (создавать, удалять, копировать целые массивы или записи), так и поэлементно. Для структурированных типов данных есть специальные функции – конструкторы типов, позволяющие создавать массивы или записи из элементов более простых типов.

Ссылочный тип данных (указатели) предназначен для обеспечения возможности указания на другие данные. Указатели характерны для языков процедурного типа, в которых есть понятие области памяти для хранения данных. Ссылочный тип данных предназначен для обработки сложных изменяющихся структур, например деревьев, графов, рекурсивных структур.

Для реляционной модели данных тип используемых данных не важен. Требование, чтобы тип данных был **простым**, нужно понимать так, что в реляционных операциях не должна учитываться внутренняя структура данных. Конечно, должны быть описаны действия, которые

можно производить с данными как с единым целым, например, данные числового типа можно складывать, для строк возможна операция конкатенации и т.д.

С этой точки зрения, если рассматривать массив, например, как единое целое и не использовать поэлементных операций, то массив можно считать простым типом данных. Более того, можно создать свой, сколь угодно сложных тип данных, описать возможные действия с этим типом данных, и, если в операциях не требуется знание внутренней структуры данных, то такой тип данных также будет простым с точки зрения реляционной теории.

В реляционной модели данных с понятием тип данных тесно связано понятие домена, которое можно считать уточнением типа данных.

Домен – это семантическое понятие. Домен можно рассматривать как подмножество значений некоторого типа данных имеющих определенный смысл. Домен характеризуется следующими свойствами:

– Домен имеет уникальное имя (в пределах базы данных).

– Домен определен на некотором простом типе данных или на другом домене.

– Домен может иметь некоторое логическое условие, позволяющее описать подмножество данных, допустимых для данного домена.

– Домен несет определенную смысловую нагрузку.

Например, домен *D*, имеющий смысл "возраст сотрудника" можно описать как следующее подмножество множества натуральных чисел:

$$D = \{n \in N : n \geq 18 \text{ and } n \leq 60\}.$$

Отличие домена от понятия подмножества состоит именно в том, что домен отражает семантику, определенную предметной областью. Может быть несколько доменов, совпадающих как подмножества, но несущие различный смысл. Некорректно, с логической точки зрения, сравнивать значения из различных доменов, даже если они имеют одинаковый тип. Например, домены "Вес детали" и "Имеющееся количество" можно одинаково описать как множество неотрицательных целых чисел, но смысл этих доменов будет различным, и значит это будут разные домены.

Фундаментальным понятием реляционной модели данных является математическое понятие **отношения,** общая структура

которого представляется в виде плоской таблицы. Каждая строка значений соответствует логической записи, а заголовки столбцов являются характеристиками объектов, информацию о которых необходимо хранить в БД.

В реляционных БД существует своя (табличная) терминология (рис. 1.12):

Определение 1.1. **Атрибут отношения** *A* есть пара вида <Имя_атрибута: Имя_домена>.

Имена атрибутов A_1, A_2, …, A_n должны быть уникальны в пределах отношения. Часто имена атрибутов отношения совпадают с именами соответствующих доменов.

Определение 1.2. Отношение *R*, определенное на множестве доменов D_1, D_2, …, D_n (не обязательно различных), содержит две части: заголовок и тело.

Заголовок отношения содержит фиксированное количество атрибутов отношения:

$$\left(\langle A_1 : D_1\rangle, \langle A_2 : D_2\rangle, \ ..., \langle A_n : D_n\rangle\right)$$

Тело отношения содержит множество кортежей отношения. Каждый **кортеж отношения** представляет собой множество пар вида <Имя_атрибута : Значение_атрибута>:

$$(< A_1 : Val_1 >, < A_2 : Val_2 >, \ldots, < A_n : Val_n >)$$

таких, что значение Val_i атрибута A_i принадлежит домену D_i

Отношение обычно записывается в виде:

$$R\left(\langle A_1 : D_1\rangle, \langle A_2 : D_2\rangle, \ ..., \langle A_n : D_n\rangle\right),$$

или короче $R\left(A_1, A_2, \ ..., A_n\right)$, или просто *R*.

Число атрибутов в отношении называют **степенью** (или **-арностью**) отношения.

Мощность множества кортежей отношения называют **мощностью** или **кардинальным числом** отношения.

Первичный ключ – уникальный идентификатор отношения, то есть атрибут или такая комбинация атрибутов, что в любой момент времени не существует двух кортежей, содержащих одинаковое значение в этом атрибуте или комбинации атрибутов.

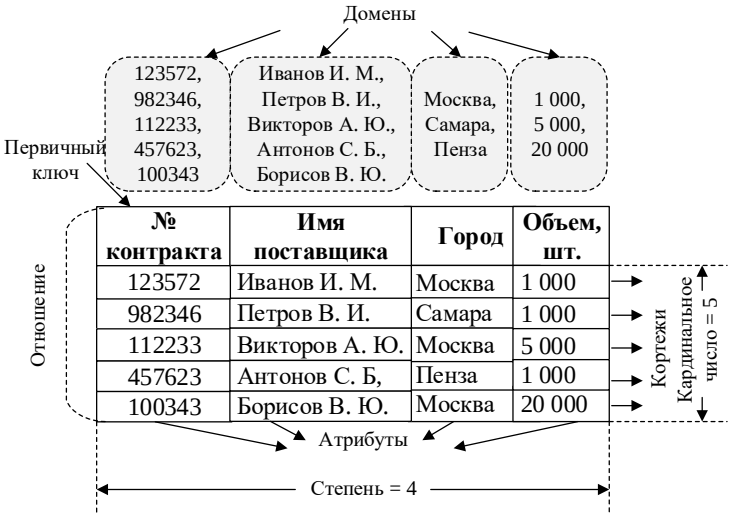


Рис. 1.12. Фрагмент реляционной модели данных

Возвращаясь к математическому понятию отношения можно сделать следующие выводы:

1. Заголовок отношения описывает домены, на которых задано отношение. Заголовок статичен, он не меняется во время работы с базой данных. Если в отношении изменены, добавлены или удалены атрибуты, то в результате получим уже другое отношение (пусть даже с прежним именем).

2. Тело отношения представляет собой набор кортежей. Таким образом, тело отношения собственно и является отношением в математическом смысле слова. Тело отношения может изменяться во время работы с базой данных – кортежи могут изменяться, добавляться и удаляться.

Определение 1.3. **Реляционной базой данных** называется набор отношений.

Определение 1.4. **Схемой реляционной базы** данных называется набор заголовков отношений, входящих в базу данных.

Термины, которыми оперирует реляционная модель данных, имеют соответствующие "табличные" синонимы, приведенные в таблице 1.1.

Таблица 1.1. Соответствие терминов

Реляционный термин	Соответствующий "табличный" термин
База данных	Набор таблиц
Схема базы данных	Набор заголовков таблиц
Отношение	Таблица
Заголовок отношения	Заголовок таблицы
Тело отношения	Тело таблицы
Атрибут отношения	Наименование столбца таблицы
Кортеж отношения	Строка таблицы
Степень (-арность) отношения	Количество столбцов таблицы
Мощность (кардинальное число) отношения	Количество строк таблицы
Домены и типы данных	Типы данных в ячейках таблицы

Хотя любое отношение и можно изобразить в виде таблицы, нужно понимать, что отношения не являются таблицами. Это близкие, но не совпадающие понятия.

Свойства отношений непосредственно следуют из приведенного выше определения отношения. В этих свойствах в основном и состоят различия между отношениями и таблицами:

1. В отношении нет одинаковых кортежей. Действительно, тело отношения есть множество кортежей и, как всякое множество, не может содержать неразличимые элементы. Таблицы в отличие от отношений могут содержать одинаковые строки.

2. Кортежи не упорядочены (сверху вниз). Причина та же - тело отношения есть множество, а множество не упорядочено. Это вторая причина, по которой нельзя отождествить отношения и таблицы - строки в таблицах упорядочены. Одно и то же отношение может быть изображено разными таблицами, в которых строки идут в различном порядке.

3. Атрибуты не упорядочены (слева направо). Т.к. каждый атрибут имеет уникальное имя в пределах отношения, то порядок атрибутов не имеет значения. Это также третья причина, по которой нельзя отождествить отношения и таблицы - столбцы в таблице упорядочены. Одно и то же отношение может быть изображено разными таблицами, в которых столбцы идут в различном порядке.

4. Все значения атрибутов атомарны. Это следует из того, что лежащие в их основе атрибуты имеют атомарные значения. Это

четвертое отличие отношений от таблиц - в ячейки таблиц можно поместить что угодно - массивы, структуры, и даже другие таблицы.

Одним из главных **достоинств реляционной модели** является простота и, соответственно, доступность для понимания конечным пользователем. Конечные пользователи не имеют дела с физической структурой памяти. Вместо этого они могут сосредоточиться на содержательной стороне проблемы.

Рассмотрим особенности операций включения и удаления в реляционной модели данных.

Включение. Допускается добавление новых кортежей, атрибутов и отношений. Включение должно основываться на понятии целостности базы данных.

Удаление. Допускается удаление кортежей, атрибутов, отношений и связей между отношениями. Удаление записи должно основываться на понятии целостности базы данных.

Проблема целостности – это обеспечение правильности (непротиворечивости) данных в БД в любой момент времени

Возникающие аномалии включения и удаления решаются декомпозицией (нормализацией) БД.

Американский математик Э.Ф. Кодд впервые сформулировал основные понятия и ограничения реляционной модели. Предложив реляционную модель данных, Э.Ф. Кодд создал и инструмент для удобной работы с таблицами (отношениями) – реляционную алгебру. Каждая операция этой алгебры использует одну или несколько таблиц (отношений) в качестве ее операндов и продуцирует в результате новую таблицу, то есть позволяет «разрезать» или «склеивать» таблицы.

Таким образом, главным **достоинством** реляционной модели данных является простота для понимания пользователем и наличие математического аппарата для работы с отношениями.

Основной **недостаток** реляционной модели состоит в том, что при большом объеме БД увеличивается время доступа и объем служебной информации.

Появление реляционной алгебры, простота и наглядность для пользователей-непрограммистов определили большую популярность реляционной модели на современном рынке СУБД. Поэтому далее при рассмотрении проектирования базы данных сосредоточимся только на БД, основанных на реляционной модели.

2. ПРОЕКТИРОВАНИЕ РЕЛЯЦИОННЫХ БД

Процесс проектирования БД представляет собой последовательность перехода от неформального словесного описания информационной структуры предметной области к формальному описанию объектов предметной области в терминах некоторой модели.

2.1. Этапы проектирования реляционных БД

Проект реляционной БД представляет собой набор взаимосвязанных отношений, в которых определены все атрибуты, первичные и внешние ключи отношений, заданы свойства отношений, обеспечивающие поддержку целостности данных. Фактически проект БД – это фундамент будущего программного комплекса, который будет использоваться достаточно долго и многими пользователями. В процессе жизненного цикла БД (рис. 2.1) можно добавлять объекты, переделывать интерфейсы, менять связи между отношениями не меняя проект БД. Если необходимы более серьезные изменения, например приложение пользователя, модель оптимального хранения данных, то БД проектируется заново, т.к. старый проект не удовлетворяет требованиям пользователей.

Рис. 2.1. Этапы жизненного цикла БД

В общем случае можно выделить следующие этапы проектирования:

– системный анализ и словесное описание информационных объектов предметной области;

– проектирование инфологической модели предметной области – частично формализованное описание объектов предметной области в терминах некоторой семантической модели, например, в ER-модели;

– логическое проектирование БД, то есть нормализация схемы отношений;

– физическое проектирование БД, то есть выбор эффективного размещения БД на внешних носителях для обеспечения наиболее эффективной работы приложения.

2.2. Системный анализ предметной области

С точки зрения проектирования БД в рамках системного анализа, необходимо осуществить первый этап – провести подробное словесное описание объектов предметной области и реальных связей, которые связывают описываемые объекты.

В общем случае существует два подхода к выбору состава и структуры предметной области.

Функциональный подход – реализует принцип движения «от задач» и применяется в случаях, когда заранее известны функции некоторой группы лиц и комплексов задач, для обслуживания информационных потребностей которых создается рассматриваемая БД. В этом случае можно четко определить минимально необходимый набор объектов предметной области, которые необходимо описать.

Предметный подход – когда информационные потребности будущих пользователей БД жестко не фиксируются. Они могут быть многоаспектными и весьма динамичными. Невозможно точно выделить минимальный набор объектов предметной области, которые необходимо описывать. В описание предметной области в этом случае включаются объекты и взаимосвязи наиболее характерные и существенные для нее. БД, конструируемая при этом, называется предметной, то есть она может быть использована при решении множества разнообразных, заранее не определенных задач. Конструирование предметной БД в некотором смысле кажется гораздо боле заманчивым, однако трудность всеобщего охвата предметной области с невозможностью конкретизации потребностей пользователей может привести к избыточно сложной схеме БД, которая для конкретных задач будет неэффективной.

Чаще всего на практике рекомендуется использовать некоторый компромиссный вариант, который, с одной стороны, ориентирован на конкретные задачи или функциональные потребности пользователей, а

с другой стороны, учитывает возможность наращивания новых приложений.

Системный анализ должен заканчиваться:

– подробным описанием информационных объектов предметной области, требующихся для решения конкретных задач и хранящихся в БД,

– формулировкой конкретных задач, которые будут решаться с помощью данной БД,

– кратким описанием алгоритмов решения задач,

– описанием выходных документов, которые должны генерироваться в системе,

– описанием входных документов, служащих основанием для заполнения данными БД.

Приведем пример описания предметной области.

Пусть требуется разработать информационную систему автоматизации учета получения и выдачи книг в библиотеке. Система должна предусматривать режимы ведения системного каталога, отражающего перечень областей знаний, по которым имеются книги в библиотеке. Внутри библиотеки области знаний в систематическом каталоге могут иметь уникальный внутренний номер и полное наименование. Каждая книга может содержать сведения из нескольких областей знаний.

Каждая книга в библиотеке может присутствовать в нескольких экземплярах. Каждая книга, хранящаяся в библиотеке, характеризуется следующими параметрами:

– уникальный шифр,

– название,

– фамилии авторов (могут отсутствовать),

– место издания (город),

– издательство,

– год издания,

– количество страниц,

– стоимость книги,

– количество экземпляров книги в библиотеке.

Книги могут иметь одинаковые названия, но они различаются по своему уникальному шифру (ISBN).

В библиотеке ведется картотека читателей.

На каждого читателя в картотеку заносятся следующие сведения: фамилия, имя, отчество, домашний адрес, телефон, дата рождения.

Каждому читателю присваивается уникальный номер читательского билета.

Каждый читатель может одновременно «держать на руках» не более 5 книг. Читатель не должен одновременно держать более одного экземпляра книги одного названия.

Любая книга в библиотеке может присутствовать в нескольких экземплярах, причем каждый экземпляр имеет следующие характеристики: уникальный инвентарный номер, шифр книги, совпадающий с уникальным шифром из описания книг, место размещения в библиотеке.

В случае выдачи экземпляра книги читателю в библиотеке хранится специальный вкладыш, в котором должны быть записаны следующие сведения: номер билета читателя, даты выдачи и возврата книги.

Предусмотреть следующие ограничения на информацию в системе:

1) книга может не иметь автора;

2) в библиотеку записывают читателей не моложе 17 лет.

3) в библиотеке присутствуют книги, изданные с 1960 по текущий год.

4) каждый читатель может брать не более 5 книг.

5) при регистрации каждый читатель должен сообщить телефон для связи (рабочий, домашний, мобильный).

6) каждая область знаний может содержать ссылки на множество книг, но каждая книга может относиться к различным областям знаний.

С данной информационной системой должны работать следующие группы пользователей: библиотекари, читатели, администрация библиотеки.

При работе с системой библиотекарь должен иметь возможность решать следующие задачи:

– принимать новые книги и регистрировать их в библиотеке;

– относить книги к одной или к нескольким областям знаний;

– проводить каталогизацию книг;

– проводить списание старых и не пользующихся спросом книг;

– вести учет выданных книг читателям;

– проводить списание утерянных книг по специальному акту списания или замены;

– проводить закрытие абонемента читателя.

Читатель должен иметь возможность решать следующие задачи:

просматривать системный каталог, то есть перечень всех областей знаний, книги по которым есть в библиотеке;

по выбранной области знаний получить полный перечень книг, имеющихся в библиотеке;

для выбранной книги получить инвентарный номер свободного экземпляра книги или получить сообщение о том, что свободных экземпляров книги нет;

получить полный список книг выбранного автора, которые имеются в библиотеке.

Администрация библиотеки должна иметь возможность получать сведения о должниках-читателях, которые не вернули вовремя взятые книги; сведения о книгах, которые не являются популярными, т.е. ни один экземпляр, которых не находится на руках у читателей; сведения о стоимости конкретной книги, для того чтобы установить возможности возмещения стоимости утерянной книги или замены ее другой книгой; сведения о наиболее популярных книгах, все экземпляры которых находятся у читателей.

Этот пример показывает, что перед началом разработки необходимо иметь точное представление о том, что же должно выполняться в нашей системе, какие пользователи в ней будут работать, какие задачи будет решать каждый пользователь.

2.3. Инфологическое проектирование

Инфологическое проектирование применяется на втором этапе проектирования БД, т.е. после словесного описания предметной области.

Инфологическое проектирование - это метод проектировании структуры базы данных, которое еще называется семантическим моделированием. Семантическое моделирование представляет собой моделирование структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты **диаграмм сущность-связь** (**ER - Entity-Relationship**).

Первый вариант модели сущность-связь был предложен в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. По сути, все варианты диаграмм сущность-связь исходят из одной идеи - рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое

изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.

Опишем работу с ER-диаграммами близко к нотации Баркера, как довольно легкой в понимании основных идей.

Основными понятиями ER-диаграммы является следующие.

Определение 2.1. **Сущность** – это класс однотипных объектов, информация о которых должна быть учтена в модели.

Каждая сущность должна иметь наименование, выраженное существительным в единственном числе.

Примерами сущностей могут быть такие классы объектов как "Поставщик", "Сотрудник", "Накладная".

Каждая сущность в модели изображается в виде прямоугольника с наименованием (рис. 2.2).

Рис. 2.2. Сущность «Сотрудник»

Определение 2.2. **Экземпляр сущности** – это конкретный представитель данной сущности.

Например, представителем сущности «Сотрудник» может быть «Сотрудник Иванов».

Экземпляры сущностей должны быть различимы, то есть сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

Определение 2.3. **Атрибут сущности** – это именованная характеристика, являющаяся некоторым свойством сущности.

Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с характеризующими прилагательными).

Примерами атрибутов сущности "Сотрудник" могут быть такие атрибуты как "Табельный номер", "Фамилия", "Имя", "Отчество", "Должность", "Зарплата" и т.п.

Атрибуты изображаются в пределах прямоугольника, определяющего сущность (рис. 2.3).

Рис. 2.3. Сущность «Сотрудник» с атрибутами

Определение 2.4. **Ключ сущности** – это неизбыточный набор атрибутов, значения которых в совокупности являются *уникальными* для каждого экземпляра сущности. Неизбыточность заключается в том, что удаление любого атрибута из ключа нарушается его уникальность.

Сущность может иметь несколько различных ключей. Ключевые атрибуты изображаются на диаграмме подчеркиванием (рис. 2.2).

Определение 2.5. **Связь** – это некоторая ассоциация между двумя сущностями. Одна сущность может быть связана с другой сущностью или сама с собою.

Связи позволяют по одной сущности находить другие сущности, связанные с нею.

Например, связи между сущностями могут выражаться следующими фразами – «СОТРУДНИК может участвовать в нескольких ПРОЕКТАХ» (рис. 2.3), «Каждый СОТРУДНИК может числиться только в одном ОТДЕЛЕ».

Графически связь изображается линией, соединяющей две сущности (рис. 2.4).

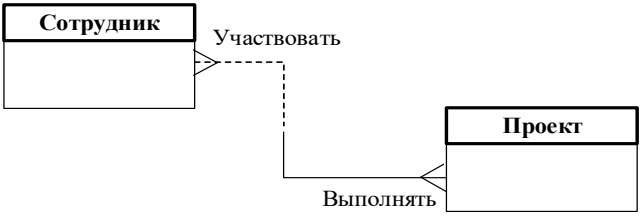


Рис. 2.4. Связь между сущностями

Каждая связь имеет два конца и одно или два наименования. Наименование обычно выражается в неопределенной глагольной форме: «иметь», «принадлежать» и т.п. Каждое из наименований

относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности. Каждая связь может иметь один из следующих **типов связи** (рис. 2.5).

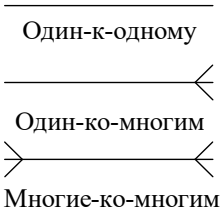


Рис. 2.5. Типы связей

Связь типа **один-к-одному** означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). Связь один-к-одному чаще всего свидетельствует о том, что на самом деле мы имеем всего одну сущность, неправильно разделенную на две.

Связь типа **один-ко-многим** означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой). Это наиболее часто используемый тип связи. Левая сущность (со стороны «один») называется **родительской**, правая (со стороны «много») - **дочерней**. Характерный пример такой связи приведен на рис. 2.5.

Связь типа **много-ко-многим** означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности. Тип связи много-ко-многим является *временным* типом связи, допустимым на ранних этапах разработки модели. В дальнейшем этот тип связи должен быть заменен двумя связями типа один-ко-многим путем создания промежуточной сущности.

Каждая связь может иметь одну из двух **модальностей связи** (рис. 2.6).

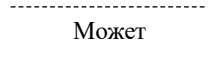


Рис. 2.6. Модальности связей

Модальность «**может**» означает, что экземпляр одной сущности может быть связан с одним или несколькими экземплярами другой сущности, а может быть и не связан ни с одним экземпляром.

Модальность «**должен**» означает, что экземпляр одной сущности обязан быть связан не менее чем с одним экземпляром другой сущности.

Связь может иметь разную модальность с разных концов, как на рис. 2.5.

Описанный графический синтаксис позволяет однозначно читать диаграммы, пользуясь следующей схемой построения фраз:

<Каждый экземпляр СУЩНОСТИ 1> <МОДАЛЬНОСТЬ СВЯЗИ> <НАИМЕНОВАНИЕ СВЯЗИ> <ТИП СВЯЗИ> <экземпляр СУЩНОСТИ 2>.

Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рис. 2.4 читается так:

Слева направо: «Каждый сотрудник может участвовать в нескольких проектах».

Справа налево: «Каждый проект обязан выполняться командой (несколькими) сотрудниками».

Рассмотрим пример разработки простой ER-модели. При разработке ER-моделей мы должны получить следующую информацию о предметной области:

1. Список сущностей предметной области.
2. Список атрибутов сущностей.
3. Описание взаимосвязей между сущностями.

ER-диаграммы удобны тем, что процесс выделения сущностей, атрибутов и связей является итерационным. Разработав первый приближенный вариант диаграмм, можно их уточнять, опрашивая экспертов предметной области. При этом документацией, в которой фиксируются результаты бесед, являются сами ER-диаграммы.

Предположим, что стоит задача разработать информационную систему по заказу некоторой оптовой торговой фирмы. В первую очередь необходимо изучить предметную область и процессы, происходящие в ней. Для этого опрашиваются сотрудники фирмы, изучается документация, изучаются формы заказов, накладных и т.п.

Например, в ходе беседы с менеджером по продажам, выяснилось, что он (менеджер) считает, что проектируемая система должна выполнять следующие действия:

– Хранить информацию о покупателях.

– Печатать накладные на отпущенные товары.

– Следить за наличием товаров на складе.

Выделим все существительные в этих предложениях – это будут потенциальные кандидаты на сущности и атрибуты, и проанализируем их (непонятные термины будем выделять знаком вопроса):

– *Покупатель* - явный кандидат на сущность.

– *Накладная* - явный кандидат на сущность.

– *Товар* - явный кандидат на сущность.

– (?) *Склад* - а вообще, сколько складов имеет фирма? Если несколько, то это будет кандидатом на новую сущность.

– (?) *Наличие товара* – это, скорее всего, атрибут, но атрибут какой сущности?

Сразу возникает очевидная связь между сущностями – «Покупатели могут покупать много товаров» и «Товары могут продаваться многим покупателям». Первый вариант диаграммы выглядит так как на рис. 2.7.

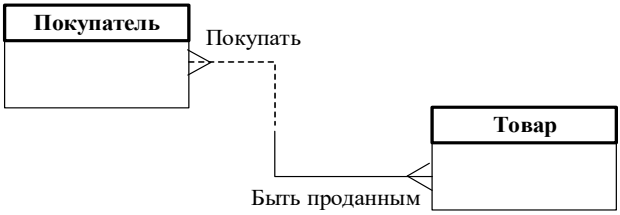


Рис. 2.7. Первый вариант диаграммы

Задав дополнительные вопросы менеджеру, выяснилось, что фирма имеет несколько складов. Причем, каждый товар может храниться на нескольких складах и быть проданным с любого склада. Куда поместить сущности «Накладная» и «Склад» и с чем их связать? Спросим себя, как связаны эти сущности между собой и с сущностями «Покупатель» и «Товар»? Покупатели покупают товары, получая при этом накладные, в которые внесены данные о количестве и цене купленного товара. Каждый покупатель может получить несколько накладных. Каждая накладная обязана выписываться на одного покупателя. Каждая накладная обязана содержать несколько товаров (не бывает пустых накладных). Каждый товар, в свою очередь, может быть продан нескольким покупателям через несколько накладных. Кроме того, каждая накладная должна быть выписана с определенного склада, и с любого склада может быть выписано много накладных.

Таким образом, после уточнения, диаграмма будет выглядеть следующим образом (рис. 2.8).

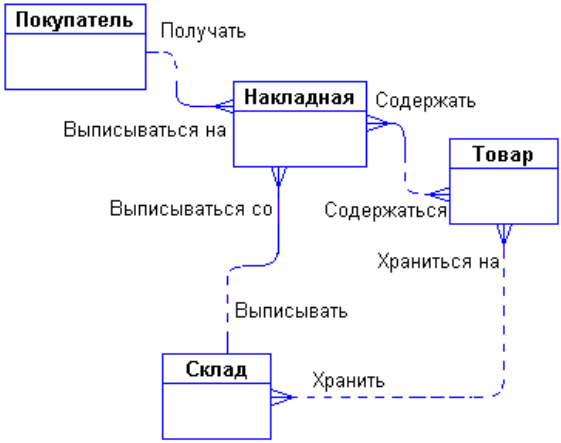


Рис. 2.8. Уточненная диаграмма

Теперь определим атрибуты сущностей. Беседуя с сотрудниками фирмы, выяснилось следующее:

– Каждый покупатель является юридическим лицом и имеет наименование, адрес, банковские реквизиты.

– Каждый товар имеет наименование, цену, а также характеризуется единицами измерения.

– Каждая накладная имеет уникальный номер, дату выписки, список товаров с количествами и ценами, а также общую сумму накладной. Накладная выписывается с определенного склада и на определенного покупателя.

– Каждый склад имеет свое наименование.

Снова выпишем все существительные, которые будут потенциальными атрибутами, и проанализируем их:

– *Юридическое лицо* – термин риторический, мы не работаем с физическими лицами. Не обращаем внимания.

– *Наименование покупателя* – явная характеристика покупателя.

– *Адрес* - явная характеристика покупателя.

– *Банковские реквизиты* – явная характеристика покупателя.

– *Наименование товара* – явная характеристика товара.

– (?) *Цена товара* – похоже, что это характеристика товара. Отличается ли эта характеристика от цены в накладной?

– *Единица измерения* – явная характеристика товара.

– *Номер накладной* – явная уникальная характеристика накладной.

– *Дата накладной* – явная характеристика накладной.

– (?) *Список товаров в накладной* – список не может быть атрибутом. Вероятно, нужно выделить этот список в отдельную сущность.

– (?) *Количество товара в накладной* – это явная характеристика, но характеристика чего? Это характеристика не просто «товара», а «товара в накладной».

– (?) *Цена товара в накладной* – опять же это должна быть не просто характеристика товара, а характеристика товара в накладной. Но цена товара уже встречалась выше – это одно и то же?

– *Сумма накладной* – явная характеристика накладной. Эта характеристика не является независимой. Сумма накладной равна сумме стоимостей всех товаров, входящих в накладную.

– *Наименование склада* – явная характеристика склада.

В ходе дополнительной беседы с менеджером удалось прояснить различные понятия цен. Оказалось, что каждый товар имеет некоторую текущую цену. Эта цена, по которой товар продается в данный момент. Естественно, что эта цена может меняться со временем. Цена одного и того же товара в разных накладных, выписанных в разное время, может быть различной. Таким образом, имеется *две цены* – цена товара в накладной и текущая цена товара.

С возникающим понятием «Список товаров в накладной» все довольно ясно. Сущности «Накладная» и «Товар» связаны друг с другом отношением типа много-ко-многим. Такая связь, как отмечалось ранее, должна быть расщеплена на две связи типа один-ко-многим. Для этого требуется дополнительная сущность. Этой сущностью и будет сущность «Список товаров в накладной». Связь ее с сущностями «Накладная» и «Товар» характеризуется следующими фразами – «Каждая накладная обязана иметь несколько записей из списка товаров в накладной», «Каждая запись из списка товаров в накладной обязана включаться ровно в одну накладную», «Каждый товар может включаться в несколько записей из списка товаров в накладной», «Каждая запись из списка товаров в накладной обязана быть связана ровно с одним товаром». Атрибуты «Количество товара в накладной» и «Цена товара в накладной» являются атрибутами сущности «Список товаров в накладной».

Точно также поступим со связью, соединяющей сущности «Склад» и «Товар». Введем дополнительную сущность «Товар на складе». Атрибутом этой сущности будет «Количество товара на

складе». Таким образом, товар будет числиться на любом складе и количество его на каждом складе будет свое.

Теперь можно внести все это в диаграмму (рис. 2.9).

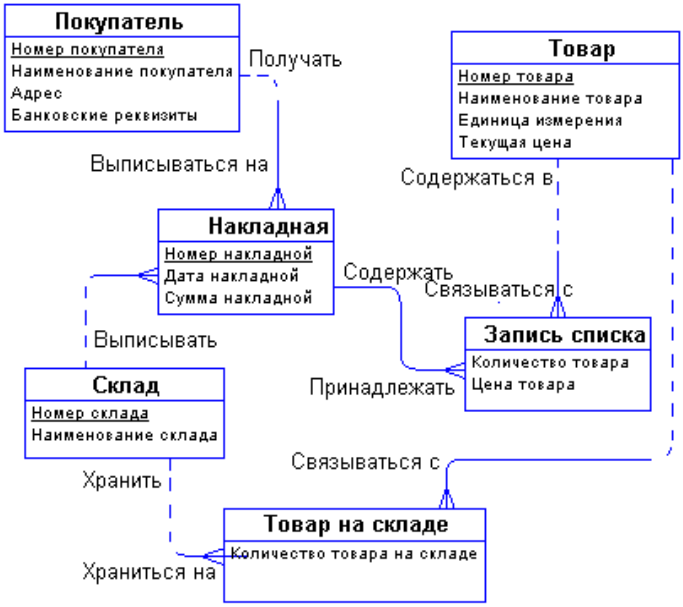


Рис. 2.9. Концептуальная ER-диаграмма

Разработанный выше пример ER-диаграммы является примером **концептуальной диаграммы**. Это означает, что диаграмма не учитывает особенности конкретной СУБД. По данной концептуальной диаграмме можно построить **физическую диаграмму**, которая уже будут учитываться такие особенности СУБД, как допустимые типы и наименования полей и таблиц, ограничения целостности и т.п. Физический вариант диаграммы, приведенной на рис. 3.8 может выглядеть следующим образом (рис. 2.10).

На данной диаграмме каждая сущность представляет собой таблицу базы данных, каждый атрибут становится колонкой соответствующей таблицы. Обращаем внимание на то, что во многих таблицах, например, CUST_DETAIL и PROD_IN_SKLAD, соответствующих сущностям «Запись списка накладной» и «Товар на складе», появились новые атрибуты, которых не было в концептуальной модели - это ключевые атрибуты родительских

таблиц, мигрировавших в дочерние таблицы для того, чтобы обеспечить связь между таблицами посредством внешних ключей.

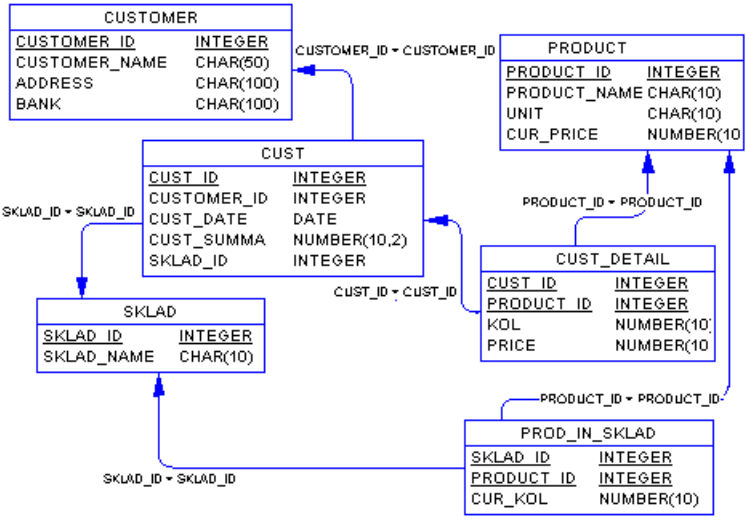


Рис. 2.10. Физический вариант диаграммы

2.4. Логическое проектирование БД

В реляционных БД логическое проектирование приводит к разработке схемы БД, то есть совокупности схем отношений, которые адекватно моделируют абстрактные объекты предметной области и семантические связи между ними.

В общем случае в результате выполнения этого этапа должны быть получены следующие результирующие документы:

– описание концептуальной схемы БД в терминах выбранной СУБД;

– описание внешних моделей в терминах выбранной СУБД;

– описание декларативных правил поддержки целостности БД;

– разработка процедур поддержки семантической целостности БД.

Разработанная на этом этапе схема БД должна быть корректной, поэтому назовем логическое проектирование процессом разработки корректной схемы реляционной БД. Корректной назовем схему БД, в

которой отсутствуют нежелательные зависимости между атрибутами отношений.

Проектирование схемы БД может быть выполнено двумя путями:

– путем декомпозиции (разбиения), когда исходное множество отношений, входящих в схему БД, заменяется другим множеством отношений (число их при этом возрастает), являющихся проекциями исходных отношений;

– путем синтеза, то есть путем компоновки из заданных исходных элементарных зависимостей между объектами предметной области схемы БД.

Классическая технология проектирования реляционных БД связана с теорией нормализации, основанной на анализе функциональных зависимостей между атрибутами отношений.

Процесс проектирования с использованием декомпозиции представляет собой процесс последовательной нормализации схем отношений, при этом каждая последующая итерация соответствует нормальной форме более высокого уровня и обладает лучшими свойствами по сравнению с предыдущей.

Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений.

В теории реляционных БД обычно выделяется следующая последовательность нормальных форм:

1-я (1NF);

2-я (2 NF);

3-я (3 NF);

Бойса-Кодда (BC-NF);

4-я (4 NF);

5-я (5 NF) или форма проекции-соединения.

Основные свойства нормальных форм:

– каждая следующая нормальная форма в некотором смысле улучшает свойства предыдущей;

– при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются

Отношение находится в 1-й нормальной форме тогда и только тогда, когда на пересечении каждого столбца, и каждой строки находятся только элементарные значения атрибутов.

Отношения, находящиеся в 1-й нормальной форме, часто называют просто нормализованными отношениями, поэтому ненормализованные отношения могут интерпретироваться как

таблицы с неравномерным заполнением, например, «Расписание» (табл. 2.1).

Табл. 2.1. Расписание

Преподаватель	**День недели**	**№ пары**	**Дисциплина**	**Тип занятий**	**Группа**
Иванов М.О.	Понедельник Пятница	2 2	Моделирование систем Моделирование систем	Лекция Лекция	И-37 И-34, 35
Петрова И.А.	Среда Четверг	1 2	Информатика Информатика	Лекция Лаб. раб.	Т-37 Т-34, 35
Смирнова Т.М.	Вторник Четверг Пятница	1 1 3	Базы данных Базы данных Базы данных	Лекция Лаб. раб. Лаб. раб.	С-37 Т-34 Т-35

Здесь на пересечении одной строки и одного столбца находится целый набор элементарных значений, соответствующих набору дней, перечню пар, набору дисциплин, по которым проводит занятия один преподаватель. Для приведения отношения «Расписание» к 1-й нормальной форме необходимо дополнить каждую строку фамилией преподавателя (табл. 2.2).

Табл. 2.2. Нормализованное расписание

Преподаватель	**День недели**	**№ пары**	**Дисциплина**	**Тип занятий**	**Группа**
Иванов М.О. Иванов М.О.	Понедельник Пятница	2 2	Моделирование систем Моделирование систем	Лекция Лекция	И-37 И-34,35
Петрова И.А. Петрова И.А.	Среда Четверг	1 2	Информатика Информатика	Лекция Лаб. раб.	Т-37 Т-34,35
Смирнова Т.М. Смирнова Т.М. Смирнова Т.М.	Вторник Четверг Пятница	1 1 3	Базы данных Базы данных Базы данных	Лекция Лаб. раб. Лаб. раб.	С-37 Т-34 Т-35

Отношение находится во 2-й нормальной форме тогда и только тогда, когда оно находится в 1-й нормальной форме и не содержит неполных функциональных зависимостей непервичных атрибутов первичного ключа.

Рассмотрим отношение, моделирующее сдачу студентами текущей сессии. Структура этого отношения определяется следующим

набором атрибутов: «ФИО», «№ зачетной книжки», «группа», «дисциплина», «оценка».

Каждый студент сдает целый набор дисциплин в процессе сессии, поэтому первичным ключом отношения может быть «№ зачетной книжки», «Дисциплина», который однозначно определяет каждую строку отношения. С другой стороны, атрибуты «ФИО» и «группа» зависят только от части первичного ключа – от значения атрибута «№ зачетной книжки», поэтому необходимо констатировать наличие неполных функциональных зависимостей в данном отношении. Для приведения данного отношения ко второй нормальной форме следует разбить его на проекции, при этом должно быть соблюдено условие восстановления исходного отношения без потерь.

Такими проекциями могут быть два отношения:

(ФИО, № зач. книжки, группа),

(№ зачетной книжки, дисциплина, оценка).

Этот набор отношений не содержит неполных функциональных зависимостей, поэтому эти отношения находятся во 2-й нормальной форме.

А почему надо приводить отношения ко 2-й нормальной форме? Иначе говоря, какие аномалии могут возникнуть, если оставить исходное отношение и не разбивать его на два? Предположим, студент переведен из одной группы в другую. Тогда в первом случае (если не разбивать исходное отношение на два) надо найти все записи с данным студентом и в них изменить значение атрибута «Группа» на новое. Во втором же случае меняется только один кортеж в первом отношении. И конечно, опасность нарушения корректности БД в первом случае выше. Может получиться так, что часть кортежей поменяет значения атрибута «группа», а часть по причине сбоя в работе аппаратуры останется в старом состоянии. Тогда наша БД будет содержать записи, которые относят одного студента одновременно к разным группам. Чтобы этого не произошло, необходимо принимать дополнительные непростые меры, например, организовать процесс согласованного изменения с использованием сложного механизма транзакций. Если же БД приведена ко 2-й нормальной форме, то достаточно изменить только один кортеж. Кроме того, если есть студенты, которые еще не сдавали экзамены, то в исходном отношении вообще нет возможности хранить о них информацию, а во второй схеме информация о студентах и их принадлежности к конкретной группе хранится отдельно от информации, которая связана со сдачей экзаменов, и поэтому можно отдельно работать со студентами и отдельно хранить и

обрабатывать информацию об успеваемости и сдаче экзаменов, что в действительности и происходит.

Отношение находится в 3-й нормальной форме тогда и только тогда, когда оно находится во 2-й нормальной форме и не содержит транзитивных зависимостей.

Рассмотрим отношение, связывающее студентов с группами, факультетами и специальностями, в которых он учится: (ФИО, № зачетной книжки, группа, факультет, специальность, выпускающая кафедра).

Первичным ключом отношения является «№ зачетной книжки», при этом рассмотрим остальные функциональные зависимости. Группа, в которой учится студент, однозначно определяет факультет, на котором он учится, а также специальность и выпускающую кафедру. Кроме того, выпускающая кафедра однозначно определяет факультет, на котором обучаются студенты, выпускаемые данной кафедрой. Но если предположим, что одну специальность могут выпускать несколько кафедр, то специальность не определяет выпускающую кафедру.

В этом случае имеются следующие функциональные зависимости:

№ зачетной книжки→ФИО,

№ зачетной книжки→группа,

№ зачетной книжки→факультет,

№ зачетной книжки→специальность,

№ зачетной книжки→выпускающая кафедра,

группа →факультет.

группа →специальность,

группа →выпускающая кафедра,

выпускающая кафедра→факультет.

Указанные зависимости образуют транзитивные группы, поэтому во избежание этого, можно предложить следующий набор отношений:

(№ зачетной книжки, ФИО, специальность, группа),

(группа, выпускающая кафедра),

(выпускающая кафедра, факультет).

Первичные ключи отношений выделены. Теперь необходимо удостовериться, что при естественном соединении не теряется ни одна строка и не получается лишних кортежей. Данное упражнение предлагается выполнить самостоятельно. Полученный набор отношений находится в 3-й нормальной форме.

Отношение находится в нормальной форме Бойса-Кодда, если оно находится в 3-й нормальной форме и каждый детерминант отношения является возможным ключом отношения.

Рассмотрим отношение, моделирующее сдачу студентом текущих экзаменов. Предположим, что студент может сдавать экзамен по одной дисциплине несколько раз, если он получил неудовлетворительную оценку. Отношение имеет следующую структуру: № зачетной книжки, идентификатор студента, дисциплина, дата, оценка). Возможными ключами отношения являются № зачетной книжки, дисциплина, дата и идентификатор студента, дисциплина, дата.

Определим имеющиеся функциональные зависимости:

№ зачетной книжки, дисциплина, дата→оценка;

идентификатор студента, дисциплина, дата→оценка;

№ зачетной книжки→идентификатор студента;

идентификатор студента→№ зачетной книжки.

Это отношение находится в 3-й нормальной форме, потому что неполных функциональных зависимостей непервичных атрибутов от атрибутов возможного ключа здесь нет, а также нет транзитивных зависимостей. Но под 4-ю нормальную форму данное отношение не подходит, так как есть два атрибута – «№ зачетной книжки» и «идентификатор студента», которые не являются возможными ключами отношения.

Для приведения отношения к нормальной форме Бойса-Кодда надо разделить отношение, например, на два со следующими схемами:

идентификатор студента, дисциплина, дата, оценка);

№ зачетной книжки, идентификатор студента или наоборот;

№ зачетной книжки, дисциплина, дата, оценка;

№ зачетной книжки, идентификатор студента.

В большинстве случаев достижение 3-й нормальной формы или даже формы Бойса-Кодда считается достаточным для реальных проектов БД, но в теории нормализации существуют нормальные формы высших порядков, которые уже связаны не с функциональными зависимостями атрибутов отношений, а отражают более тонкие вопросы семантики предметной области и связаны с другими видами зависимостей.

Перед тем, как рассмотреть нормальные формы высших порядков, остановимся на некоторых понятиях, которые необходимы для определения этих форм.

В отношении R (A, B, C) существует многозначная зависимость $R(A) -> > R(B)$ *в том и только в том случае, если множество значений*

B, соответствующее паре значений A и C, зависит только от A и не зависит от C.

Пусть дано отношение, которое моделирует предстоящую сдачу экзаменов на сессии. Допустим, оно имеет вид: № зачетной книжки, группа, дисциплина.

Перечень дисциплин, которые должен сдавать студент, однозначно определяется не его фамилией, а номером группы (то есть специальностью, на которой он учится).

В данном отношении существуют следующие две многозначные зависимости:

группа–>>дисциплина;

группа –>>№ зачетной книжки.

Это означает, что каждой группе однозначно соответствует перечень дисциплин по учебному плану и № группы определяет список студентов, которые в этой группе учатся. Если работать с исходным отношением, то невозможно хранить информацию о новой группе и ее учебном плане – перечне дисциплин, которые должна пройти группа до тех пор, пока в нее не будут зачислены студенты. С одной стороны, при изменении перечня дисциплин по учебному плану, например, при добавлении новой дисциплины, внести эти изменения в отношение для всех студентов, занимающихся в данной группе, весьма затруднительно. С другой стороны, если добавлять студентов в уже существующую группу, то необходимо добавить множество кортежей, соответствующих перечню дисциплин для данной группы. Эти аномалии модификации отношения как раз и связаны с наличием двух многозначных зависимостей.

В теории реляционных БД доказывается, что в общем случае в отношении R (A, B, C) существует многозначная зависимость $R(A)$ –>>$R(B)$ в том и только в том случае, когда существует многозначная зависимость $R(A)$ –>>$R(C)$.

Дальнейшая нормализация отношений основывается на теореме Фейджина.

Теорема Фейджина. Отношение $R(A, B, C)$ можно спроецировать без потерь в отношение $R1(A, B)$ и $R2(A, C)$ в том и только в том случае, когда существует многозначная зависимость A–>>B и A–>>C.

Под проецированием без потерь понимается такой способ декомпозиции отношения путем применения операции проекции, при котором исходное отношение полностью и без избыточности восстанавливается путем естественного соединения полученных отношений. Практически теорема доказывает наличие эквивалентной

схемы для отношения, в котором существует несколько многозначных зависимостей.

Отношение R находится в 4-й нормальной форме (4NF) в том и только в том случае, если в случае существования многозначной зависимости R (A–>>B) все остальные атрибуты R функционально зависят от A.

В рассматриваемом примере можно произвести декомпозицию исходного отношения в два отношения:

№ зачетной книжки, группа,

группа, дисциплина,

которые находятся в 4-й нормальной форме и свободны от отмеченных аномалий. Действительно, обе операции модификации теперь упрощаются: добавление нового студента связано с добавлением всего одного кортежа в первое отношение, а добавление новой дисциплины – с добавлением одного кортежа во второе отношение; кроме того, во втором отношении можно хранить любое количество групп с определенным перечнем дисциплин, в которые пока еще не зачислены студенты.

Анализ нового вида зависимостей (зависимостей “проекции-соединения”) – это 5-ф нормальная форма. Этот вид зависимостей является в некотором роде обобщением многозначных зависимостей.

Отношение R (X, Y, ..., Z) удовлетворяет зависимости соединения (X, Y, ..., Z) в том и только в том случае, когда R восстанавливается без потерь путем соединения своих проекций на X, Y, ..., Z, где X, Y, ..., Z – наборы атрибутов отношения R.

Наличие зависимости «проекция-соединение» в отношении делает его в некотором роде избыточным и затрудняет операции модификации.

Отношение R находится в 5-й нормальной форме (нормальной форме «проекции-соединения») в том и только в том случае, когда любая зависимость соединения в R следует из существования некоторого возможного ключа в R.

Рассмотрим отношение *R*1(Преподаватель, Кафедра, Дисциплина).

Предположим, что каждый преподаватель может работать на нескольких кафедрах и на каждой кафедре может вести несколько дисциплин. В этом случае ключом отношения является полный набор из трех атрибутов. В отношении отсутствуют многозначные зависимости, и поэтому отношение находится в 4NF.

Введем следующие обозначения наборов атрибутов:

ПК (преподаватель, кафедра),

ПД (преподаватель, дисциплина),

КД (кафедра, дисциплина).

Допустим, что отношение *R*1 удовлетворяет зависимости проекции соединения (ПК, ПД, КД). Тогда отношение *R*1 не находится в нормальной форме проекции соединения, потому что единственным ключом его является полный набор атрибутов, а наличие зависимости проекции соединения связано с набором атрибутов, которые не составляют возможные ключи отношения R1. Для того чтобы привести это отношение к нормальной форме проекции соединения, его надо представить в виде трех отношений:

*R*2 (преподаватель, кафедра);

*R*3 (преподаватель, дисциплина);

*R*4 (кафедра, дисциплина).

Пятая нормальная форма редко используется на практике. В большей степени она является теоретическим исследованием.

2.5. Физическое проектирование БД

Проектирование физической модели данных представляет отображение логической модели данных в среду выбранной СУБД. Исходными данными для проектирования физической модели данных являются: логическая модель данных с комплектом сопроводительной документации, функциональные и технические характеристики выбранной СУБД, требования к производительности системы.

Основными задачами, решаемыми на этапе проектирования физической модели данных, являются:

– принятие решения о целесообразности денормализации данных;

– создание таблиц, определение свойств ключей и атрибутов;

– связывание таблиц, определение требований целостности;

– реализация ограничений предметной области;

– конструирование карты транзакций;

– обоснование целесообразности использования дополнительных индексов;

– выбор способа физической организации данных;

– определение способов доступа к данным пользователей, проектирование представлений;

– построение системы защиты данных;

– создание требуемых хранимых процедур, функций и триггеров.

Большинство этапов физического проектирования реализуется с учетом возможностей выбранной СУБД.

Первой задачей физического проектирования баз данных является **принятие решения о целесообразности денормализации таблиц**.

Денормализация – это процесс целенаправленного введения избыточности в базы данных, позволяющий улучшить производительность системы. Увеличение избыточности данных приводит к уменьшению количества операций соединения таблиц в процессе выполнения SQL-запросов и, как следствие, улучшает реактивность информационной системы. Однако, при этом увеличиваются затраты на обновление данных, уменьшается гибкость системы и усложняется ее реализация. Поэтому денормализацию целесообразно производить только для таблиц, к которым осуществляются очень частые обращения на выборку данных, и редкие – на изменение. Основными видами денормализации являются: нисходящая, восходящая и внутритабличная.

Нисходящая денормализация предполагает включение атрибута родительской таблицы в дочернюю.

Восходящая денормализация базируется на создании в родительской таблице дополнительного столбца, содержащего агрегатные данные, которые вычисляются на основе информации, представленной в соответствующих кортежах дочерней таблицы. Целесообразность введения такого столбца определяется интенсивностью запросов на получение итоговых данных, количеством строк в дочерней таблице, по которым определяются итоговые показатели, соотношением между интенсивностью запросов на чтение и запросов на обновление данных. Наличие такого столбца позволяет создать для него вторичный индекс и существенно ускорить доступ к итоговым показателям. Однако его введение приводит к возрастанию затрат по поддерживанию целостности данных при выполнении таких операций, как ввод и удаление строк из дочерней таблицы, обновление значений атрибутов, по которым производится вычисление интегральных показателей.

Внутритабличная денормализация предполагает введение в структуру таблицы дополнительного атрибута, значение которого является производным от других атрибутов, входящих в эту же таблицу. Целесообразность внутритабличной денормализации определяется наличием интенсивных запросов к значениям дополнительного атрибута, содержащего интегрированную информацию. Организация вторичных индексов для таких атрибутов позволяет существенно ускорить время выполнения запросов.

В процессе решения задач «**создание таблиц, определение свойств ключей и атрибутов**» и «**связывание таблиц, определение требований целостности**» необходимо определить тип и размер данных для атрибутов таблицы, предложить процедуры реализации ограничений первичных и внешних ключей, обязательности значений

атрибутов. В качестве исходных данных используется логическая модель данных, в которой атрибутам назначены определенные домены. Основная проблема описания атрибутов при создании таблиц состоит в корректном сопоставлении доменов типам и размерам данных, поддерживаемых используемой СУБД, а также наличием в ней тех или иных средств поддержки ограничений целостности.

При **реализации ограничений предметной области** необходимо предусмотреть средства для проверки значений в атрибутах таблиц, имеющих ограниченное множество значений: пол человека может быть только «мужской» или «женский», оценка – 2, 3, 4 или 5 и т.д. В качестве таких средств могут быть использованы правила или специально созданные функции, триггеры, хранимые процедуры. Если СУБД не поддерживает те или иные из указанных средств, то такие ограничения приходится реализовать на уровне приложения, обслуживающего базу данных.

Для того чтобы разрабатываемый физический проект БД обладал требуемым уровнем эффективности, необходимо получить максимум сведений о тех транзакциях и запросах, которые будут выполняться в базе данных. Потребуются как качественные, так и количественные характеристики. Для успешного планирования каждой транзакции необходимо знать следующее:

– транзакции, выполняемые наиболее часто и оказывающие существенное влияние на производительность;

– транзакции, наиболее важные для работы организации;

– периоды времени на протяжении суток/недель, в которые нагрузка БД возрастает до максимума (называемые периодами пиковой нагрузки).

Эта информация используется для определения компонентов базы данных, которые могут вызвать проблемы производительности. Кроме того, необходимо определить такие характеристики транзакций высокого уровня, как атрибуты, модифицируемые в транзакциях обновления, или критерии, которые служат для ограничения количества строк, возвращаемых по запросу. Эта информация используется для определения наиболее подходящей файловой организации и создания индексов.

Во многих случаях проанализировать все ожидаемые транзакции просто невозможно, поэтому необходимо тем или иным образом выбрать наиболее важные. Существует эмпирическое правило, согласно которому выполнение около 20% наиболее активных запросов пользователей создает примерно 80% нагрузки на базу данных (правило Парето). Для определения того, какие из транзакций

подлежат детальному анализу, строится таблица соответствия, состоящая из отношений, доступ к которым происходит при выполнении каждой транзакции.

Например, в табл. 2.3 приведено соответствие количества транзакций в единицу времени и отношений для типичных операций ввода, обновления, удаления и выборки. Таблица показывает, что операции с отношениями «Студенты» и «Занятия» редки и независимы друг от друга. Доступ к отношению «Прогулы» выполняется очень часто, и при этом обязательно используются другие отношения. Поэтому, задача обеспечения эффективного доступа к этому отношению может оказаться очень важной с точки зрения обеспечения высокой производительности системы.

Табл. 2.3. Соответствие количества транзакций в единицу времени и отношений для типичных операций ввода, обновления, удаления и выборки

Отношение	Транзакция											
	Операции «со студентами»				Операции с «занятиями»				Операции с «прогулами»			
	Чтение	Добавление	Удаление	Изменение	Чтение	Добавление	Удаление	Изменение	Чтение	Добавление	Удаление	Изменение
Студенты	2	1	0	0	0	0	0	0	50	25	0	0
Занятия	0	0	0	0	2	5	1	1	50	25	0	0
Прогулы	0	0	0	0	0	0	1	1	50	25	2	2

Этап **определение способов доступа к данным пользователей, проектирование представлений** подразумевает создание представлений БД для различных категорий пользователей. В многопользовательских БД представления обеспечивают, с одной стороны, удобство работы пользователя с таблицей-представлением, являющимся объединением данных различных таблиц, с другой, являются мощным средством разграничения доступа и организации защиты данных.

Если для некоторых транзакций требуется частый доступ к определенным отношениям, то приобретает особую важность задача **изучения** характера их выполнения для **обоснования целесообразности использования дополнительных индексов** и **выбора способа физической организации данных**.

3. МАНИПУЛИРОВАНИЕ ДАННЫМИ

Манипулирование – это одна из многих других функций управления данными.

Теоретической основой современных языков манипулирования структурированными данными является **реляционная алгебра**. Рассмотреть операции реляционной алгебры и разберем на примерах их назначение.

3.1. Реляционная алгебра

Реляционная алгебра – набор математических операций, позволяющих манипулировать табличными данными.

Реляционная алгебра, определенная Э. Ф. Коддом состоит из 8 операций, состоящих из 2 групп по 4 операции в каждой:

– традиционные (объединение, пересечение, вычитание, декартово произведение);

– специальные (выборка, проекция, соединение, деление).

Рассмотрим примеры выполнения этих операций, которые в теории множеств имеют обозначения, представленные на рис. 3.1.

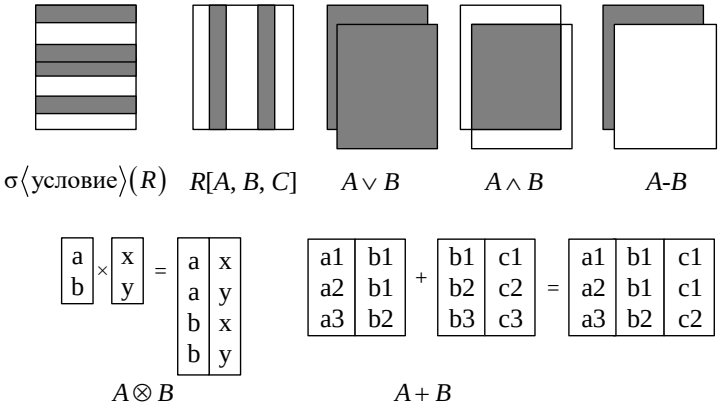


Рис. 3.1. Операции реляционной алгебры

Выборка. Результатом выполнения этой операции являются все кортежи из определенного отношения, которые удовлетворяют определенным условиям.

Обозначение: σ <условие>(*R*), где σ – обозначение операции; *R* – имя отношения; ⟨условие⟩ – операции сравнения (>, <, =, >=, <=, <>)

Рассмотрим пример. Пусть дана табл. 3.1 (отношение) «Кадры» некоторого фрагмента БД.

Табл. 3.1 «Кадры»

№	ФИО	Возраст	Город
1	Иванов И.И.	40	Москва
2	Петров В.Д.	36	СПб
3	Сидоров Р.О.	21	СПб
4	Смирнов К.А.	20	Москва

Необходимо выполнить выборку лиц в возрасте до 36 лет из таблицы «Кадры» – σ⟨Возраст < 36⟩(Кадры).

Результат выполнения операции выборки «σ⟨Возраст < 36⟩(Кадры)»

№	ФИО	Возраст	Город
3	Сидоров Р.О.	21	СПб
4	Смирнов К.А.	20	Москва

Проекция. Результатом выполнения этой операции являются все кортежи определенного отношения после исключения из него некоторых повторяющихся атрибутов.

Обозначение: *R*[*A, B, C*], где *R* – имя отношения; *A, B, C* – имена атрибутов.

Рассмотрим пример. Пусть дана табл. 3.2 (отношение) «Соревнования» некоторого фрагмента БД.

Табл. 3.2 «Соревнования»

ФИО спортсмена	Вид спорта	Город
Иванов И.Д.	Плавание	Москва
Петров С.М.	Легкая атлетика	СПб
Сидоров А.А.	Тяжелая атлетика	Красноярск
Смирнов С.А.	Фигурное катание	СПб
Архипов Л.И.	Плавание	Москва
Алексеев А.С.	Тяжелая атлетика	Томск
Викторов М.В	Фигурное катание	СПб

Необходимо представить список городов, которые представляют спортсмены из табл. 3.2. Таким образом, необходимо применить операцию проекции по атрибуту «Город» или формально: Соревнования[Город]

Результат выполнения операции «Соревнования[Город]»:

Город
Москва
СПб
Красноярск
Томск

Или другой пример. Необходимо определить, какие виды спорта из табл. 3.2 представлены определенными городами. Таким образом, необходимо применить операцию проекции по атрибутам «Вид спорта» и «Город» или формально: Соревнования[Вид спорта, Город]

Результат выполнения операции «Соревнования[Вид спорта, Город]»:

Вид спорта	Город
Плавание	Москва
Легкая атлетика	СПб
Тяжелая атлетика	Красноярск
Фигурное катание	СПб
Тяжелая атлетика	Томск

Произведение. Результат операции – всевозможные картежи, которые являются сочетанием 2 картежей, принадлежащих соответственно двум определенным отношениям.

Операция произведение выполняется над двумя отношениями *A* и *B*. которые характеризуются произвольными значениями степени отношения n_A и n_B *и* мощности отношений m_A и m_B. Результирующее отношение *C* имеет степень $nC = nA \times nB$ и мощность $mC = mA + mB$, а его кортежи формируются путем выполнения процедуры конкатенации каждого кортежа отношения *A* с каждым кортежем отношения *B*. Результирующее отношение слева содержит перечень атрибутов отношения *A*, справа – перечень атрибутов отношения *B*.

Обозначение: $A \otimes B$, где *A, B* – имена отношений

Рассмотрим пример. Пусть даны два отношения *A* и *B*.

A

Код поставки
a
b
c

B

Код изделия
x
y

Результат операции $A \otimes B$:

Код поставки	Код изделия
a	x
a	y
b	x
b	y
c	a
c	y

Пересечение. Результатом является отношение, которое содержит все кортежи, одновременно принадлежащие двум отношениям.

Обозначение: $A \wedge B$, где *A*, *B* – имена отношений.

Пусть даны два отношения *A* (табл. 3.3) и *B* (табл. 3.4).

Табл. 3.3. «Отношение *A*»

№	ФИО	Возраст	Город
1	Иванов И.О.	20	Москва
2	Петров С.Т.	20	Москва

Табл. 3.4. «Отношение *B*»

№	ФИО	Возраст	Город
1	Иванов И.О.	20	Москва
2	Сидоров В.Т.	16	СПб

Результат $A \wedge B$:

№	ФИО	Возраст	Город
1	Иванов И.О.	20	Москва

Объединение – результатом будет отношение, содержащее все кортежи, которые принадлежат одному из двух определенных отношений, или обоими.

Обозначение: $A \vee B$, где *A*, *B* – имена отношений.

Рассмотрим пример. Даны отношения *A* (табл. 3.5) и *B* (табл. 3.6).

Табл. 3.5. «Отношение *A*»

ФИО	Возраст	Город
Иванов И.О.	20	Москва
Петров С.Т.	20	Москва

Табл. 3.6. «Отношение *B*»

ФИО	Возраст	Город
Иванов И.О.	20	Москва
Сидоров В.Т.	16	СПб

Результат $A \vee B$:

ФИО	Возраст	Город
Иванов И.О.	20	Москва
Петров С.Т.	20	Москва
Сидоров В.Т.	16	СПб

Вычитание – результатом будет отношение, содержащее все кортежи, принадлежащие только первому из двух отношений.

A – *B*, где *A*, *B* – имена отношений

Рассмотрим пример. Даны отношения *A* (табл. 3.7) и *B* (табл. 3.8).

Табл. 3.7. «Отношение *A*»

№	ФИО	Возраст	Город
1	Иванов И.О.	20	Москва
2	Петров С.Т.	20	Москва

Табл. 3.8. «Отношение *B*»

№	ФИО	Возраст	Город
1	Иванов И.О.	20	Москва
3	Сидоров В.Т.	16	СПб

Результат $A - B$:

№	ФИО	Возраст	Город
2	Петров С.Т.	20	Москва

Соединение – результирующее отношение, которое содержит кортежи 1-го и 2-го отношения, имеющих общее значение 1-го или нескольких полей. И такие общие значения в результирующем кортеже появляются только один раз.

Обозначение: (*A* + *B*)общие поля, где *A*, *B* – имена отношений

Рассмотрим пример. Даны отношения *A* (табл. 3.9) и *B* (табл. 3.10). В отношении *A* представлен список преподавателей – руководителей выпускных квалификационных работ. В отношении *B* – список студентов, которые выбрали себе руководителей выпускных работ. Необходимо построить список студентов и их руководителей. Очевидно, что необходимо применить операцию соединения отношений *A* и *B*.

Табл. 3.9. «Отношение *A*»

ID преподавателя	ФИО преподавателя
S1	Иванов И.О.
S2	Петрова С.Т.
S3	Сидоров В.Т.

Табл. 3.10. Отношение B

№ зач_книжки	ФИО студента	ID преподавателя
P1	Терентьев К.Д.	S1
P2	Елисеева В.А.	S1
P3	Антонов В.А.	S3
P4	Владимирова С.Т.	S1
P5	Мишин В.Ф.	S2
P6	Дадонов П.Д.	S2

Результат соединения операции «(*A* + *B*)ID преподавателя»:

ФИО студента	ФИО преподавателя
Терентьев К.Д.	Иванов И.О.
Елисеева В.А.	Иванов И.О.
Владимирова С.Т.	Иванов И.О.
Мишин В.Ф.	Петров С.Т.
Дадонов П.Д.	Петров С.Т.
Антонов В.А.	Сидоров В.Т.

Деление. Деление выполнятся для двух отношений: в первом задействованы два поля, а во втором – одно. Результатом будет отношение, содержащее все значения одного атрибута отношения с двумя полями, которые соответствуют всем значениям в отношении с одним полем.

Обозначение: *A* / *B*, где *A*, *B* – имена отношений

Рассмотрим пример. Дано отношение *A* (табл. 3.11). В отношении *A* представлен список деталей и их поставщиков. Необходимо из этого общего списка построить список из поставщиков, которые поставляют только определенные детали, например, только детали P1 и P3.

Очевидно необходимо применить операцию деления отношения *A* на отношение *B* (табл. 3.12).

Табл. 3.11. «Отношение *A*»

Код детали	Код поставщика
P1	S1
P3	S1
P1	S2
P2	S3
P1	S3
P3	S3

Табл. 3.12. «Отношение *B*»

Код детали
P1
P3

Результат применения операции *A*/*B*:

Код поставщика
S1
S3

Реляционная алгебра базируется на теории множеств и является основой логики работы базы данных. Доступ к реляционным данным осуществляется при помощи реляционной алгебры. В реализациях конкретных реляционных СУБД реляционная алгебра сейчас не используется в чистом виде. Фактическим стандартом доступа к реляционным данным стал язык SQL (Structured Query Language), который представляет собой смесь операторов реляционной алгебры, использующий синтаксис, близкий к фразам английского языка и расширенный дополнительными возможностями, отсутствующими в реляционной алгебре.

3.2. Типы данных SQL

Прежде чем приступить к изучению непосредственно SQL-инструкций, рассмотрим типы данных, которыми оперирует язык SQL

В языке SQL имеются средства, позволяющие для каждого атрибута указывать тип данных. Сразу же сделаем замечание о том, что в разных СУБД в части реализации языка SQL определение типов данных не полностью согласуются с требованиями официального стандарта SQL. Это объясняется, в частности, желанием обеспечить совместимость SQL с другими языками программирования. Тем не менее базовыми типами определения данных являются:

– строковый,

– числовой,

– дата время,

– неопределенный (пропущенные) данные.

Строковый тип позволяет хранить любые данные, представленные в виде символов. Это могут быть специальные символы, цифры и буквы, которые в своей совокупности будут обрабатываться как строки в любом запросе SQL. В табл. 3.13 приведены варианты строкового типа данных.

Табл. 3.13. Варианты строкового типа данных

Тип данных	Назначение
CHAR (size)	Используется для хранения строк. Параметр в скобках позволяет фиксировать длину хранимой строки. Максимальный размер – 255 байт.
VARCHAR (size)	Аналогично предыдущему типу. Отличие от CHAR – для хранения значения данного типа выделяется требуемое количество памяти.
TINY TEXT	Используется для хранения строковых данных размером до 255 символов.
TEXT	Используется для хранения текстовой информации, размер которой не превышает 65 535 букв.
BLOB	Аналогичен типу TEXT, размер может достигать 65 535 знаков. На практике – для хранения звуковых данных, рисунков, электронной документации и пр.
MEDIUM TEXT	Аналогичен типу TEXT, размер может достигать 16 777 215 букв или символов.
MEDIUM BLOB	Используется для сохранения в базе электронных документов, размер которых не превышает отметку в 16 777 215 знаков.
LONG TEXT	Функционально аналогичен предыдущим типам, но с увеличенным объемом памяти до 4 гигабайт
LONG BLOB	Позволяет помещать в базу данные больших объемов (4 294 967 295 символа).
ENUM (a, b, c, etc.)	Специальный тип данных, использующийся для задания списка возможных значений. Позволяет указать 65535 значений. Строки

	рассматриваемого типа могут принимать единственное значение из указанных в множестве. В случае, когда будет происходить добавление значений, которые не присутствуют в заданном списке, в таблицу будут записаны пустые значения.
SET ()	Задает множество допустимых значений. В отличие от предыдущего типа, используется для содержания 64 параметров, которые могут быть проинициализированы любым или несколькими элементами из заданных аргументов.

Официальный стандарт SQL поддерживает только один тип представления текста CHAR (size). Параметр в скобках позволяет фиксировать длину хранимой строки. Максимальный размер – 255 байт. Если во вводимой в поле текстовой константе фактическое число символов меньше числа, определенного параметром длина, то эта константа автоматически дополняется справа пробелами до заданного числа символов, а если превышает, до усекает. Например, CHAR (8) – хранит строки из 8 символов и занимает 8 байтов памяти. Любое из следующих значений: '', 'Иван', 'Ирина', 'Сергей' будет занимать по 8 байтов памяти. А при попытке ввести значение 'Александра', оно будет усечено до 'Александ', то есть до 8 символов

Некоторые реализации языка SQL поддерживают в качестве типа данных строки переменной длины. Этот тип может обозначаться ключевым словом VARCHAR (size) описывает текстовую строку, которая может иметь произвольную длину до определенного конкретной реализацией SQL максимума (в Oracle – до 2000 символов). В отличие от типа CHAR в этом случае при вводе текстовой константы, фактическая длина которой меньше заданной, не производится ее дополнение пробелами до заданного максимального значения. Константы, имеющие тип VARCHAR, в выражениях SQL заключаются в одиночные кавычки, например, 'текст'.

Если длина строки не указана явно, она полагается равной одному символу во всех случаях. По сравнению с типом CHAR тип данных VARCHAR позволяет более экономно использовать память, выделяемую для хранения текстовых значений, и оказывается более удобным при выполнении операций, связанных со сравнением текстовых констант.

Остальные строковые типы TINY TEXT, TEXT, BLOB, MEDIUM TEXT, MEDIUM BLOB, LONG TEXT и LONG BLOB, приведенные в

табл. 3.24 идентичны, разница только в выделяемом размере памяти для хранения строк.

Обратим внимание на тип ENUM и SET. Их встроенная поддержка имеется только у MySQL. Объявление этого типа данных позволяет создавать список значений.

Тип ENUM – строки этого типа могут принимать только одно из значений указанного множества. Например, ENUM ('да', 'нет') – в столбце с таким типом может храниться только одно из имеющихся значений. Удобно использовать, если предусмотрено, что в столбце должен храниться ответ на вопрос.

SET – строки этого типа могут принимать любой или все элементы из значений указанного множества. Например, SET ('первый', 'второй') – в столбце с таким типом может храниться одно из перечисленных значений, оба сразу или значение может отсутствовать вовсе.

Числовой тип данных условно можно представить двумя группами:

– Целочисленный – для хранения целых чисел

– Дробный – для хранения чисел с плавающей точкой

Группа целочисленного типа данных основывается на использовании базового типа INTEGER с некоторым расширением его свойств (табл. 3.14).

Табл. 3.14. Варианты числового типа данных

Тип данных	Назначение
INT (size)	Хранение целочисленных значений, образующих диапазон $\left[-2^{31};\ 2^{31}-1\right]$
TINYINT (size)	Хранение целочисленных значений в диапазоне [-128; 127]
SMALLINT (size)	Хранение целочисленных значений в диапазоне [-32 768; 32 767]
MEDIUMINT (size)	Хранение целочисленных значений в диапазоне $\left[-2^{23};\ 2^{23}-1\right]$
BIGINT (size)	Хранение целочисленных значений в диапазоне $\left[-2^{63};\ 2^{63}-1\right]$

Тип INT (size) хранит любое целое число в диапазоне от -2^{31} до 2^{31}–1. Например, INT (4) – предполагается, что значения будут четырехзначные.

Число может быть объявлено положительным с помощью ключевого слова UNSIGNED. Тогда элементам столбца нельзя будет присвоить отрицательное значение. Например, INT UNSIGNED хранит любое число в диапазоне от 0 до 2^{31}–1.

Size – это необязательный параметр, указывающий количество отводимых под число символов.

Необязательный атрибут ZEROFILL позволяет свободные позиции по умолчанию заполнить нулями. Например, задав INT (5) ZEROFILL означает, что свободные позиции слева будут заполнены нулями. Например, значение «2» будет отображаться, как «00002».

Остальные целочисленные типы, приведенные в табл. 3.14 аналогичны типу INT, разница заключается только в диапазонах выделяемой для хранения чисел памяти.

Группа дробного тип данных основывается на использовании базового типа FLOAT с некоторым расширением его свойств (табл. 3.15).

Табл. 3.15. Варианты дробного типа данных

Тип данных	Назначение
FLOAT (size, d)	Хранение дробных числа указываемой точности d.
DOUBLE (size, d)	Хранения дробных чисел с двойной точностью
DECIMAL(size, d)	Хранение дробных значений в виде строк.

Тип FLOAT (size, d) используется для хранения вещественных чисел (с плавающей точкой), где size – количество отводимых под число символов, d – количество символов дробной части. Число может быть объявлено положительным с помощью ключевого слова UNSIGNED, но диапазон значений от этого не изменится. FLOAT (5,2) – будет хранить числа из 5 символов, 2 из которых будут идти после запятой, например: 46,58.

Остальные дробные типы, приведенные в табл. 3.15 аналогичны типу FLOAT, разница состоит только в диапазонах выделяемой для хранения чисел памяти и точности – количества знаков после запятой.

Необходимо понимать, чем больше диапазон значений у типа данных, тем больше памяти он занимает. Поэтому, если предполагается, что значения в столбце не будут превышать значения 100, то следует использовать тип TINYINT. Если при этом все значения будут положительными, то следует использовать атрибут UNSIGNED. Правильный выбор типа данных позволяет сэкономить место для хранения этих данных.

Тип данных даты и времени является нестандартным и предназначен для хранения даты или времени в определенном формате (табл. 3.16). Наличие этого типа позволяет поддерживать специальную арифметику дат и времен.

Табл. 3.16. Варианты типа данных даты и времени

Тип данных	Назначение
DATE	Хранение даты в формате ГОД-МЕСЯЦ-ДЕНЬ (“ГГГГ-ММ-ДД” или "yyyy-mm-dd"). Обычно значения разделены через «-», однако в качестве разделителя может быть задействован любой символ, кроме цифр.
TIME	Позволяет заносить в ячейку таблицы значения времени. Все значения задаются форматом «hh:mm:ss»
DATETIME	Объединяет функции предыдущих двух типов. Формат хранения представлен следующим образом: «yyyy-mm-dd hh:mm:ss».
TIMESTAMP	Сохраняет дату и время, исчисляемое количеством секунд, прошедших начиная с полуночи 1.01.1970 года и до заданного значения.
YEAR (M)	Используется для хранения значений года в двух- или четырехзначном формате.

Тип данных даты и времени предоставляет дополнительные преимущества при разработке информационных систем, работа которых зависит от показателей времени, например интернет-магазины, платежные системы.

Тип данных NULL применяется для обозначения отсутствующих, пропущенных или неизвестных значений атрибута.

NULL лишь условно можно назвать типом данных. По сути это указатель возможности отсутствия значения. Например, при регистрации на каком-либо сайте, как правило, предлагается заполнить форму, в которой присутствуют, как обязательные, так и необязательные поля. Естественно регистрация пользователя невозможна без указания логина и пароля, а вот дату рождения и пол можно указать по желанию. Для того, чтобы хранить такую информацию в БД используют два значения: NOT NULL (значение не может отсутствовать), NULL (значение может отсутствовать).

По умолчанию всем столбцам присваивается тип NOT NULL, поэтому его можно явно не указывать.

3.3. Определение данных

К операторам **определения данных** языка SQL относятся команды создания базы данных (таблиц, индексов и т.д.) и редактирования ее схемы (табл. 3.17).

Табл. 3.17. Операторы определения данных языка SQL

№	Оператор	Назначение
1	CREATE DATABASE	Создание схемы БД
2	DROP DATABASE	Удаление схемы БД
3	CREATE TABLE	Создание таблицы
4	DROP TABLE	Удаление таблицы
5	ALTER TABLE	Изменение таблицы
6	CREATE INDEX	Создание индекса
7	DROP INDEX	Удаление индекса
8	CREATE DOMAIN	Создание домена
9	ALTER DOMAIN	Изменение домена
10	DROP DOMAIN	Удаление домена

Создание базы данных реализуется оператором CREATE DATABASE. Он является универсальным и предназначен для многих СУБД, но в СУБД Oracle аналогом этой команды является команда CREATE SCHEMA.

Полный синтаксис создания БД выглядит так:

```
CREATE DATABASE <имя базы данных>;
```

где «;» – знак, означающий завершение инструкции SQL.

Пример создания новой БД с именем «Кадры»:

```
CREATE DATABASE Кадры;
```

Имя создаваемой БД должно быть уникальным, в противном случае возникает ошибка выполнения команды. Для того, чтобы она не возникала, можно использовать ключевые слова IF NOT EXISTS, например:

```
CREATE DATABASE IF NOT EXISTS Кадры;
```

Удаления БД реализуется командой DROP:

```
DROP DATABASE <имя базы данных>;
```

Например, так выглядит команда удаления БД с именем «Кадры»:

```
DROP DATABASE Кадры;
```

Применение этой команды удаляет все объекты БД «Кадры».
Создание таблиц БД выполняется командой CREATE TABLE.
Формат команды:

```
CREATE TABLE <имя новой таблицы>
(<Имя столбца1><тип данных> [(<размер>)],
<Имя столбца2><тип данных> [(<размер>)],…,
<Имя столбца n><тип данных> [(<размер>)]);
```

Создадим таблицу с именем «Сотрудники» и полями: Табельный номер, ФИО, Должность, Дата приема, Оклад, Отдел. Каждому полю определим соответствующий тип данных, а поле Табельный номер (Таб_ номер) зададим, как первичный ключ. Кроме указания типа данных для поля Таб_ номер необходимо записать следующие ограничения для него:

– Ограничение NOT NULL, которое гарантирует, что столбец не содержит пустых ячеек;

– Ограничение PRIMARY KEY, которое создает первичный ключ для таблицы.

Пример кода SQL-инструкции для создания таблицы «Сотрудники» выглядит следующим образом:

```
CREATE TABLE Сотрудники (
Таб_номер INT (6) NOT NULL PRIMARY KEY,
Фамилия CHAR (15),
Должность CHAR (15),
Дата_приема DATE,
Оклад FLOAT (6,2),
Отдел INT (3)
);
```

Или другой вариант, при котором ограничения для поля, содержащего первичный ключ записаны отдельно друг от друга:

```
CREATE TABLE Сотрудники (
Таб.номер INT (6) NOT NULL,
Фамилия CHAR (15),
Должность CHAR (15),
Дата_приема DATE,
Оклад FLOAT (6,2),
Отдел INT (3));
PRIMARY KEY (Таб.номер)
);
```

Замечания:

1. Спецификация максимальных длин не означает обязательного соответствующего расходования физической памяти БД.
2. Порядок столбцов определяет порядок столбцов в будущей таблице.
3. Первичный ключ должен быть определен для каждой таблицы БД. Если этого не сделать, то будет невозможно однозначно идентифицировать каждую запись таблицы.

Изменение макета существующей таблицы реализуется командой ALTER TABLE.

Макет таблицы изменяется, если добавляется новое поле или наоборот, удаляется существующее поле. В зависимости от этого во второй строке записываем:

– ADD COLUMN – добавление нового поля в таблицу

– DROP COLUMN – удалить поле из таблицы

Формат команды на добавление нового поля в таблицу:

```
ALTER TABLE <имя таблицы>
ADD COLUMN <имя поля><тип данных>[(<размер>)];
```

Формат команды на удаление поля из таблицы:

```
ALTER TABLE <имя таблицы>
DROP COLUMN <имя поля>;
```

Пусть дана таблица «Покупатели» с полями: Номер накладной, Покупатель, Продавец, Контакты. Рассмотрим пример добавления в таблицу «Покупатели» нового поля «Реквизиты»:

```
ALTER TABLE Покупатели
ADD COLUMN Реквизиты Text (150);
```

и удалим поле «Контакты»:

```
ALTER TABLE Покупатели
DROP COLUMN Контакты;
```

Добавление первичного колюча происходит по-другому, т.к. необходимо указывать ограничения. Это еще один вариант определения первичного ключа – то есть можно сначала создать таблицу с простыми атрибутами, а потом добавить поле первичного ключа:

```
CREATE TABLE Сотрудники (
Фамилия CHAR (15),
Должность CHAR (15),
Дата_приема DATE,
Оклад FLOAT (6,2),
Отдел INT (3));
);
ALTER TABLE Сотрудники
ADD(Таб.номер) INT (6) NOT NULL PRIMARY KEY;
```

Создание индекса реализуется командой CREATE INDEX.

Индексы, как было определено ранее, представляют собой структуры, предназначенные для повышения производительности работы с данными.

Индекс содержит отсортированные значения одного или нескольких столбцов таблицы со ссылкой на соответствующую им строки исходной таблицы. Поэтому поиск нужной строки по индексу производится многократно быстрее, чем последовательный построчный перебор значений. Таким образом, если стоит задача

ускорить работу с таблицами, то можно создавать индексы. Пользователи индексов не видят.

Формат команды создания индекса:

```
CREATE [UNIQUE] INDEX <имя_индекса> ON <имя_таблицы>
(<имя_столбца>,...)
```

Инструкция CREATE INDEX используется для создания простого индекса в таблице, то есть допускаются повторения значения поля.

Инструкция CREATE UNIQUE INDEX для создания уникального индекса. При этом СУБД будет автоматически проверять каждое его значение на уникальность.

Естественно возникает вопрос: какие поля надо индексировать и надо ли вообще индексировать? Ответ на этот вопрос следующий:

1. Обязательно надо строить индексы для первичных ключей, поскольку по их значениям осуществляется доступ к данным при операциях соединения двух и более таблиц.

2. Необходимо анализировать наиболее частые запросы к БД. Для минимизации времени этих запросов необходимо создавать индексы.

Первоначальное определение структуры индексов производится разработчиком на стадии создания БД. В дальнейшем она уточняется администратором системы по результатам анализа ее работы, учета наиболее часто выполняющихся запросов и т.д.

Еще один вариант определения первичного ключа через создание уникального индекса выглядит следующим образом:

```
CREATE TABLE Сотрудники (
Таб.номер INT (6),
Фамилия CHAR (15),
Должность CHAR (15),
Дата_приема DATE,
Оклад FLOAT (6,2),
Отдел INT (3)
);
CREATE UNIQUE INDEX Key1
ON Сотрудники (Таб.номер)
WITH PRIMARY;
```

Удаление индекса реализуется командой DROP INDEX. Формат команды:

```
DROP INDEX <имя_индекса>
```

Содержимое таблицы при удалении индекса не изменится, изменяются свойства поля, которое было индексом. Поэтому внешний вид таблицы при создании индекса и при его удалении одинаков.

Создание домена реализуется командой CREATE DOMAIN.

Домен, как было определено ранее, это объект БД, на котором определяются атрибуты. Домены также позволяют задать ограничения на множество значений поля. Эти особенности учтены в формате инструкции для создания домена.

Формат команды:

```
    CREATE DOMAIN <имя домена> [AS] <тип данных>
[значение по умолчанию] [список ограничений
целостности]
```

где <имя домена> – название создаваемого домена;
<тип данных> – поддерживаемый SQL тип данных, который является базовым для создаваемого домена;
[AS] – необязательное ключевое слово, разделяющее названия домена и базового типа данных;
[значение по умолчанию] – необязательный параметр, определяющий значение по умолчанию, которое применяется к элементам каждого столбца, определенного на данном домене и не имеющего собственного определения значения по умолчанию;
[список ограничений целостности] – список ограничений целостности, применяемых к каждому столбцу, основанному на домене.

В качестве примера опишем домены, на которых определяется таблица «Студенты» с полями номер, ФИО, группа.

Для домена поля «Номер» создадим ограничение на значение. Пусть в поле «Номер» можно вводить только положительное число:

```
    CREATE DOMAIN Д_Номер INTEGER CONSTRAINT О_Номер
CHECK (VALUE >0);
```

Для домена второго поля «ФИО» создадим ограничение на тип данных. Пусть в поле «ФИО» можно вводить строки переменной длины до 50 символов со значением по умолчанию "???":

```
CREATE DOMAIN Д_ФИО VARCHAR(50) DEFAULT '???';
```

Для домена третьего поля «Номер группы» создадим ограничение на размерность, например не более трех десятичных символов:

```
CREATE DOMAIN Д_Группа INT(3);
```

Изменить существующий домен можно с помощью оператора ALTER.

Формат команды:

```
ALTER DOMAIN <имя домена> <действие на изменение>;
```

В существующем домене можно произвести следующие изменения:

– определить новое значение по умолчанию;

– удалить существующее;

– ввести новое ограничение целостности.

Например, удалим ограничение на значение домена Д_Номер и добавим значение по умолчанию для домена Д_Группа:

```
ALTER DOMAIN Д_Номер DROP CONSTRAINT О_Номер;
ALTER DOMAIN Д_Группа SET DEFAULT 382;
```

Удалить существующий домен можно оператором DROP DOMAIN. Формат команды:

```
DROP DOMAIN <имя домена> {RESTRICT | CASCADE};
```

где опция RESTRICT означает, что домен не будет удален, если он использован в определении какого-либо столбца или ограничения целостности;

опция CASCADE означает, что домен будет удален. При этом, столбцы, которые были определены на этом домене, автоматически наследуют свойства удаленного домена.

Например, удалим созданный ранее домен Д_Номер:

```
DROP DOMAIN Д_Номер RESTRICT;
```

Столбцы, определенные на удаленном домене, автоматически переопределяются следующим образом:

– каждый столбец теперь относится к типу данных, на основе которого определялся удаляемый домен;

– если у столбца не было определено собственное значение по умолчанию, то теперь у него значение по умолчанию, которое было у удаляемого домена;

– каждый столбец наследует все ограничения удаляемого домена.

Надо отметить, что в Microsoft SQL Server оператор CREATE DOMAIN не поддерживается. В данной СУБД подобную задачу можно частично решить с помощью оператора CREATE TYPE.

3.4. Манипулирование данными

Одна из основных функций управления данными – манипулирование. К **манипулированию данными** относятся четыре основных команды языка SQL:

INSERT – вставка новых сведений в БД.

UPDATE – обновление сведений**,** хранимых в базе данных

DELETE – удаление сведений из базы данных

SELECT – извлечение сведений, содержащих в базе данных

Современные СУБД поддерживают операции манипулирования данными, с помощью которых пользователи могут создавать выражения для выполнения перечисленных операций.

Вставка значений. Ввод новой строки в таблицу реализуется командой INSERT INTO

Существует два способа использования инструкции:

Первый способ – ввод новой записи как целой строки, формат команды:

```
INSERT INTO <имя таблицы>
VALUES (<значения ячеек строки>);
```

При использовании этого варианта необходимо в строке values записывать значения всех ячеек строки.

Второй способ – ввод значений в отдельные столбцы, формат команды:

```
INSERT INTO <имя таблицы> (<имена столбцов>)
VALUES (<значения ячеек отдельных столбцов строки>);
```

Поскольку в таком формате можно вносить данные в отдельные столбцы, то в строке insert перечисляются имена столбцов, куда будут вносится значения и в строке values перечисляются эти значения в строгом соответствии с указанными столбцами.

Рассмотрим пример на оба способа. Пусть дана табл. 3.18 «Командировки» с хранящимися в ней данными о командированных сотрудниках.

Табл. 3.18. Командировки

Код	Фамилия	Город	Билет
1001	Смирнов	Москва	Нет
1002	Алексеев	СПб	Да
1003	Белов	СПб	Да

Первый способ – ввод новой записи как целой строки. В строке values записываем значения всех ячеек строки. В рассматриваемом примере табл. 3.18 четыре ячейки, соответственно необходимо записать значения во все четыре ячейки:

```
INSERT INTO Командировки
VALUES (1004, «Антонов», «Москва», «NULL»);
```

После ввода новой записи появится новая строка в табл. 3.18:

Код	Фамилия	Город	Билет
1001	Смирнов	Москва	Нет
1002	Алексеев	СПб	Да
1003	Белов	СПб	Да
1004	Антонов	Москва	

Второй способ – ввод значений (данных) в отдельные столбцы. Например, внесем данные только в столбцы Код, Фамилия, Город табл. 3.18.

```
INSERT INTO Продавцы (Код, Продавец, Город)
VALUES (1004, «Антонов», «Москва»);
```

Тогда в остальных столбцах по умолчанию появятся нулевые значения:

Код	Фамилия	Город	Билет
1001	Смирнов	Москва	Нет
1002	Алексеев	СПб	Да
1003	Белов	СПб	Да

1004	Антонов	Москва	0

Команда INSERT также используется для переноса данных из одной таблицы в другую. В этом случае команда VALUES заменяется на команду SELECT.

Пусть из существующей таблицы 3.29 «Командировки» необходимо выбрать все записи с сотрудниками, командированными в город Санкт-Петербург (СПб) и перенести эти записи во вновь создаваемую таблицу «Командированные в СПб»:

```
INSERT INTO [Командированные в СПб]
SELECT*
FROM Командировки
WHERE Город=СПб;
```

Удаление строк из таблицы реализуется командой DELETE. Команда DELETE не может удалять отдельные поля, а только выбранные записи целиком. Формат команды:

```
DELETE FROM <имя таблицы>;
```

Но есть и такой вариант:

```
DELETE*
FROM <имя таблицы>;
```

Если нужно удалить только строки в соответствии с условием, то это условие надо указать в строке where. Формат команды:

```
DELETE FROM <имя таблицы>
WHERE <условие>;
```

Например, удалим из таблицы «Командировки» записи, где командированный еще не купил билет, то есть поле «Билет» имеет значение «Нет»:

```
DELETE FROM Командировки
WHERE Билет=Нет or Билет=NULL;
```

Если есть несколько связных таблиц, то оператор DELETE удаляет данные из этих таблиц, используя «Каскадное удаление».

Обновление (модификация) данных, содержащихся в БД реализуется командой UPDATE. Команда позволяет изменить значение отдельного столбца. Формат команды:

```
UPDATE <имя таблицы>
SET <новое значение>, <новое значение>,…
WHERE <условие>;
```

Рассмотрим на примере той же таблицы «Командировки». Пусть сотрудника Смирнова руководство его организации решили командировать не в Москву, а в СПб и забронировали билет. Тогда SQL-запрос обновления данных по сотруднику с номером 1001 будет выглядеть так

```
UPDATE Командировки
SET Город=СПб, Билет=Да
WHERE Код=1001;
```

Если обновляемые данные должны быть предварительно вычислены, то в команде UPDATE есть возможность внести вычисляемое выражение в строке SET, где собственно записывается обновляемое значение. Формат команды:

```
UPDATE <имя таблицы>
SET <новое значение=выражение>
WHERE <условие>;
```

Пусть, стоимость билета, приобретенного в последней декаде текущего месяца, дорожает на 20%. Если билеты кем-то из командированных приобретены после 20 марта, то это необходимо обновить в БД:

```
UPDATE Командировки
SET Цена=Цена*1.2
WHERE Дата >=#20.03.2019#
```

3.5. Поисковые запросы

Еще одна важная функция управления данными – это поиск по определенным параметрам. СУБД в части языка SQL поддерживает группу команд (инструкций), позволяющих пользователю создавать выражения для выполнения поисковых запросов.

Простейшая инструкция поиска информации имеет вид:

```
SELECT <список полей>
FROM (имена таблиц);
```

где SELECT – ключевое слово, оно сообщающее БД, что инструкция SQL является запросом-выборкой;
<список полей> – перечисление полей через «,» которые надо вывести в новой таблице. Символ «*» в этом месте вместо <список полей> обозначает, что все поля будут выведены в новой таблице.

Для устранения дублирующих строк после SELECT необходимо исполнить предикат DISTINCT.

Рассмотрим пример. Пусть в табл. 3.19 с именем «Заказы» хранятся сведения о заказах и надо выяснить, были ли сделаны заказы у конкретных продавцов.

Табл. 3.19. Заказы

№	Дата продажи	Стоимость заказ	Покупатель	Продавец
301	10.03.18	15 000	201	105
302	10.03.18	23 000	205	106
303	10.03.18	10 000	204	102
304	10.03.18	52 000	206	105
305	10.03.18	12 000	209	102

Код SQL-запроса, соответствующего заданию:

```
SELECT Продавец
FROM Заказы;
```

Выполнение данного запроса приведет к следующему результату:

Продавец
105
106
102
105
102

Код SQL-запроса, устраняющего дублирование продавцов в результате запроса:

```
SELECT DISTINCT Продавец
FROM Заказы;
```

Выполнение данной инструкции запроса приведет к следующему результату:

Продавец
105
106
102

Если необходимо выполнить **выборку в соответствии с условием,** то в код инструкции добавляется ключевое слово WHERE. Формат команды:

```
SELECT *
FROM Покупатели
WHERE <условие>;
```

Таким образом, запрос на выборку извлекает данные из одной или нескольких таблиц на основе заданных условий отбора записей.

Операторы, которые могут быть использованы для задания условий, приведены в табл. 3.20.

Табл. 3.20. Операторы задания условий в запросах на выборку

Оператор	Назначение	Примечание
Like(“Подстановочный знак”)	–Известна только часть значения –Требуется найти значения, начинающиеся с конкретной буквы или соответствующие определенному шаблону	* – соответствует любой цифре или любому символу. Может использоваться в качестве первого или последнего символа текстовой строки. Например, wh* – поиск слов what, white и why. ? – соответствует любому текстовому символу. Например, B?ll – поиск слов ball, bell и bill. [ ] – соответствует любому одному символу из заключенных в скобки. Например, B[ae]ll – поиск

		слов ball и bell, но не bill. ! – соответствует любому одному символу, кроме заключенных в скобки. Например, b[!ae]ll – поиск слов bill и bull, но не bell или ball. - – соответствует любому символу из диапазона. Необходимо указывать этот диапазон по возрастанию (от A до Z, но не от Z до A). Например, b[a-c]d – поиск слов bad, bbd и bcd. # – соответствует любой цифре. Например, 1#3 – поиск значений 103, 113, 123.
between >, <, >=, <= <>	Выбор записей, значения которых находятся с заданными значениями в определенном отношении	>234 – числа, превышающие 234 Between #02.02.18# And #01.12.19# – даты в диапазоне от 02.02.18 до 01.12.19 <1200.45 – числа, меньшие чем 1200,45 >="Иванов" – все фамилии, начиная с «Иванов» и до конца алфавита
Not Символ*	Выбор записей, содержащих значения, не совпадающие с заданным	В строку «Условие» для соответствующего поля вводится оператор Not.
Not Null (Is Not Null)	Выбор записей с непустыми значениями	В строку «Условие» для соответствующего поля вводится оператор Not Null или Is Not Null.
Is Null.	Извлечение записей с пустыми значениями	В строку «Условие» для соответствующего поля вводится оператор Is Null.

Date()	Выбор записей, содержащих значение текущей даты	В строку «Условие» для соответствующего поля вводится оператор Date() (без пробела между скобками).
In(Список значений)	Выбор записей, содержащихся в списке значений	В строку «Условие» для соответствующего поля вводится оператор In. Значения в списке заключаются в «» и разделяются ;

Как видно из табл. 3.20 разнообразие операторов позволяет осуществлять поиск по шаблону (оператор Like), поиск на основе сравнений (операторы «больше», «меньше», Between, not и другие), поиск записей, содержащих пустые и непустые поля (операторы Null, Not Null), поиск данных по текущей дате (оператор Date) и поиск данных по списку (оператор In).

Рассмотрим несколько примеров поисковых запросов с применением условных операторов, которые представлены в табл. 3.20.

Пусть в таблице «Поставщики», фрагмент которой представлен в табл. 3.21 хранятся сведения о работниках, осуществляющих поставки в разные города. Считаем, что записанных в ней данных достаточно много, чтобы обрабатывать вручную. Поэтому в следующих примерах будем писать поисковые запросы, имея ввиду, что источником данных является таблица «Поставщики».

Табл. 3.21. Поставщики

Номер	Дата поставки	Имя	Город	Рейтинг	Поставщик
201	01.03.19	Сидоренко П.Д.	Москва	200	305
202	02.03.19	Власов М.Ю.	СПб	300	306
203	03.03.19	Щукин И.Д.	Москва	500	407
204	10.03.19	Романов В.А.	Омск	100	408
205	15.03.19	Курдюков Л.М.	Томск	300	502

Первое задание – получить список поставщиков из табл. 3.21 с рейтингом выше 200 баллов.

Приведем код SQL-запроса, решающего это задание:

```
SELECT *
```

```
FROM Поставщики
WHERE Рейтинг > 200;
```

Результат выполнения запроса:

Номер	Дата поставки	Имя	Город	Рейтинг	Поставщик
202	02.03.19	Власов М.Ю.	СПб	300	306
203	03.03.19	Щукин И.Д.	Москва	500	407
205	15.03.19	Курдюков Л.М.	Томск	300	502

Второе задание – получить список поставщиков из табл. 3.21, выполняющие поставки товаров в города Москва и СПб.

Код SQL-запроса, выполняющего это задание:

```
SELECT *
FROM Поставщики
WHERE Город IN ('Москва','СПб');
```

Результат выполнения запроса:

Номер	Дата поставки	Имя	Город	Рейтинг	Поставщик
201	01.03.19	Сидоренко П.Д.	Москва	200	305
202	02.03.19	Власов М.Ю.	СПб	300	306
203	03.03.19	Щукин И.Д.	Москва	500	407

Третье задание – получить список заказов (только их номера) из табл. 3.21 первой недели марта 2019.

Код SQL-запроса, выполняющего это задание:

```
SELECT Номер
FROM Поставщики
WHERE Дата_поставки between #01.03.2019# and #07.03.2019#;
```

Результат выполнения запроса:

Номер
201
202

203

Четвертое задание – найти список заказов и их покупателей из табл. 3.21 на текущую дату.

Код SQL-запроса, выполняющего это задание:

```
SELECT *
FROM Поставщики
WHERE Date();
```

Поиск данных в БД может осуществляться сразу по нескольким условиям. С этой целью в предложении WHERE можно использовать булевы операторы AND, OR, NOT. Таблица истинности булевых операторов:

Оператор	Условие 1	Условие 2	Результат
AND	**TRUE**	**TRUE**	**TRUE**
	TRUE	FALSE	FALSE
	FALSE	TRUE	FALSE
	FALSE	FALSE	FALSE
OR	**TRUE**	**FALSE**	**TRUE**
	TRUE	**FALSE**	**TRUE**
	FALSE	**TRUE**	**TRUE**
	FALSE	FALSE	FALSE
NOT	**TRUE**		**FALSE**
	FALSE		**TRUE**

<условие 1> AND <условие 2> означает, что результат операции будет истинным, если оба условия будут иметь истинное значение.

<условие 1> OR <условие 2> означает, что результат операции будет истинным, если одно из условий или оба условия будут иметь истинное значение.

NOT <условие> означает, что условие принимает противоположное значение.

Пятое задание – найти работников, которые успешно выполняют свою работу (с высоким рейтингом) в Москве. Пусть успех у нас определяется рейтингом выше 100 баллов.

Код SQL-запроса, выполняющего это задание:

```
SELECT *
FROM Поставщики
WHERE Город = «Москва» AND рейтинг > 100”;
```

Результат поискового запроса может быть отсортирован (упорядочен) по возрастанию или убыванию. Это удобно, когда поиск содержит большое число записей.

Если использовать в запросе параметр ORDER BY, то это будет указанием на необходимость сортировки**.** По умолчанию осуществляется сортировка по возрастанию, которая может задаваться ключевым словом ASC. Для выполнения сортировки по убыванию – DESC.

Шестое задание – упорядочить информацию по продавцам табл. 3.19 в порядке убывания стоимости заказа.

Код SQL-запроса, выполняющего это задание:

```
SELECT *
FROM Заказы
ORDER BY Продавец, [Стоимость заказа] DESC;
```

Результат выполнения запроса:

№	Дата продажи	Стоимость заказ	Покупатель	Продавец
305	10.03.18	12 000	209	102
303	10.03.18	10 000	204	102
304	10.03.18	52 000	206	105
301	10.03.18	15 000	201	105
302	10.03.18	23 000	205	106

Итак, любой поисковый запрос начинается с инструкции SELECT, а далее добавляются условия поиска, которые могут задавать поиск по шаблону, поиск на основе сравнений. Также поиск можно выполнять сразу по нескольким условиям, а также сортировать данные, полученные в результате поиска.

3.6. Итоговые функции

Среди операций манипулирования данными выделяется группа операций, которая получила название итоговых функций. Они используются для получения итоговых данных по таблицам, например, когда надо просуммировать какие-либо данные без их выборки.

Итоговым запросом назовем такой запрос, который группирует данные по некоторому признаку и находит итоговое значение для каждой группы.

Формат итогового запроса:

```
SELECT <список полей>, функция агрегирования (поле агрегирования) AS [Имя нового поля, для размещения итога]
FROM (имена таблиц)
GROUP BY (имя поля по которому группируются записи);
```

где оператор GROUP BY позволяет найти в БД подмножество значений отдельного поля в терминах другого поля и применить функцию агрегирования не ко всему множеству, а лишь к некоторому подмножеству записей.

Итоговые функции еще называют статистическими, агрегатными или суммирующими. Эти функции обрабатывают набор строк для подсчета и возвращения одного значения. Таких функций несколько, они приведены в табл. 3.22.

Табл. 3.22. Функции агрегирования

Функция	Назначение
SUM	Сумма всех значений заданного поля в любой группе. Операцию можно использовать только для числовых или денежных полей
AVG	Вычисление среднего арифметического всех данных поля в любой группе. Можно использовать только для числовых или денежных полей
MIN	Возвращает наименьшее значение найденное в этом поле внутри любой группы. Для числовых полей возвращается наименьшее число, для текстовых – наименьшее из символьных значений. Пустые поля – игнорируются
MAX	Возвращает наибольшее значение найденное в этом поле внутри любой группы. Для числовых полей возвращается наибольшее число, для текстовых – наибольшее из символьных значений. Пустые поля – игнорируются
COUNT	Возвращает количество записей
FIRST	Возвращает первое значение этого поля в группе

LAST	Возвращает последнее значение этого поля в группе
StDev	Среднеквадратичное отклонение от среднего значения поля
VAR	Дисперсия значений поля

Познакомимся с функциями агрегирования на примере написания итоговых запросов к табл. 3.19 «Заказы».

Предположим, необходимо узнать минимальную, максимальную и среднюю стоимость заказа. Таким образом, из таблицы Заказы надо взять минимальное, максимальное и среднее значения по столбцу «Стоимость заказа». Это простой итоговый запрос без группировки данных, код SQL-запроса которого выглядит следующим образом:

```
SELECT  MIN([Стоимость заказа]), MAX([Стоимость
заказа]), AVG([Стоимость заказа]) FROM Заказы;
```

Результат выполнения запроса:

MIN(Стоимость заказа)	MAX(Стоимость заказа)	AVG(Стоимость заказа)
10 000	52 000	22 400

Рассмотрим еще один пример с группировкой данных. Пусть необходимо определить максимальную стоимость заказа из табл. 3.19, оформленного каждым продавцом.

У каждого продавца – несколько заказов, следовательно, максимальную стоимость надо найти для каждого продавца, значит поле «Продавец» группируется. Внутри группы применим итоговую функцию MAX к полю «Стоимость заказа».

Код SQL-запроса:

```
SELECT Продавец, MAX[Стоимость заказа] AS [Наибольший заказ]
FROM Заказы
GROUP BY Продавец;
```

К результату итогового запроса также можно применять параметр сортировки. Для упорядочивания значений полей используется параметр ORDER BY. По умолчанию осуществляется сортировка по возрастанию, которая может задаваться ключевым словом ASC. Для выполнения сортировки по убыванию – DESC.

Определим суммарную стоимость заказов из табл. 3.19, оформленных каждым продавцом и выведем их в порядке убывания этой суммы:

```
SELECT Продавец, SUM[Стоимость заказа] AS [Сумма заказов]
FROM Заказы
GROUP BY Продавец
ORDER BY SUM[Стоимость заказа]  DESC;
```

Результат выполнения запроса:

Продавец	Сумма заказов
105	67 000
106	23 000
102	22 000

Если имя поля состоит из двух слов, например «Стоимость заказа» или «Сумма заказов», как в рассматриваемом примере, то нужно использовать квадратные скобки для идентификации такого поля.

Условие в итоговом запросе задается с помощью оператора HAVING.

Оператор HAVING работает только с групповыми запросами, и если WHERE определяет записи, которые должны быть выбраны, то HAVING устанавливает, какие записи, сгруппированные с помощью параметра GROUP BY, должны отображаться на экране.

Рассмотрим пример. Определим суммарную стоимость заказов свыше 20 000 руб. из табл. 3.19, оформленных продавцами и вывести их в порядке убывания этой суммы:

```
SELECT Продавец, SUM[Стоимость заказа] AS [Сумма
заказов]
FROM Заказы
GROUP BY Продавец
HAVING MAX(Сумма) > 20 000
ORDER BY SUM[Стоимость заказа];
```

Результат выполнения запроса:

Продавец	Наибольший заказ
105	67 000
106	23 000

Таким образом, рассмотренные функции агрегирования позволяют строить запросы на подведение итогов по отдельным полям и группам, которые могут быть назначены по определенному признаку.

3.7 Вложение запросов

Как правило, база данных включает несколько таблиц, которые соединяются по идентификаторам или индексам. Поэтому, когда есть необходимость по идентификатору из одной таблицы получить полную информацию об объекте, которая хранится в другой таблице то применяют вложенные запросы.

Формат команды:

```
SELECT <поля таблиц>
FROM <имена таблиц>
WHERE условие = (SELECT <поле>
          FROM <имя таблицы>
          WHERE условие = [запрашиваемое значение поля?]);
```

Чтобы понять назначение вложенных запросов рассмотрим пример.

Имеются две таблицы: в первой «Заказы» (табл. 3.23) хранится статистика по Заказам, во второй «Продавцы» (табл. 3.24) – список всех сотрудников, оформляющих заказы. Очевидно, что связь между этими таблицами 1:M – один сотрудник может оформлять множество заказов.

Табл. 3.23. Заказы

№	Дата продажи	Стоимость заказа	Покупатель	ID_Продавца
301	10.03.18	15 000	201	105
302	10.03.18	23 000	205	106
303	10.03.18	10 000	204	102
304	10.03.18	52 000	206	105
305	10.03.18	12 000	209	102

Табл. 3.24. Продавцы

Код	Продавец	Город
102	Смирнов Е.П.	Москва
105	Алексеев В.Н.	СПб
106	Белов А.А.	СПб

Когда есть необходимость узнать, кто оформлял тот или иной заказ, стоимость конкретного заказа и кто его оформил, то придется фактически строить два запроса:

1) к таблице «Заказы», откуда узнаем ID продавца и

2) к таблице «Продавцы», чтобы узнать его имя.

В SQL предусмотрена возможность объединять такие запросы в один путем превращения одного из них в подзапрос (вложенный запрос).

Код SQL-запроса, который по имени продавца выводит информацию о его заказах:

```
SELECT Заказы *
FROM Заказы
WHERE Заказы.ID_Продавца = (SELECT Код
                        FROM Продавцы
                        WHERE Продавец = [Имя продавца?]);
```

Разберем, как это работает. Сначала SQL выполнит внутренний запрос: появится сообщение «Имя продавца?», в ответ, на которое можно ввести имя одного из продавцов, например, Смирнов Е.П. Из табл. «Продавцы» (строка From) будет найден код 102 и передан во внешний запрос. После определения кода во внешнем запросе из табл. «Заказы» будут отображены записи, удовлетворяющие условию: Заказы.ID_Продавца = 102.

Результат выполнения запроса:

303	10.03.18	10 000	204	102
305	10.03.18	12 000	209	102

Замечание:

1. В этом варианте запроса подзапрос (команда SELECT) должен вернуть только одно значение поля, иначе будет «отказ».

2. Если в результате подзапроса не было одной строки, то предикат WHERE примет значение UNKNOWN (неизвестно) вместо обычных true и false, следовательно, ни одна из записей возвращена на будет.

Для оценки события (Да – есть и Нет – отсутствует) применяется оператор EXISTS.

Как правило, этот оператор используется для индикации того, что какая-либо строка удовлетворяет условию. То есть фактически оператор EXISTS не возвращает строки, а лишь указывает, что в базе данных есть как минимум одна строка, которые соответствует данному запросу. Поскольку возвращения набора строк не происходит, то подзапросы с подобным оператором выполняются довольно быстро.

Рассмотрим пример. Выведем содержимое таблицы 3.23 «Заказы», если в ней есть хотя бы один заказ от 10 марта 2018:

```
SELECT *
FROM Заказы
```

```
WHERE EXISTS (SELECT *
              From Заказы
              Where [Дата продажи]= #10.03.18#);
```

Поскольку заказ с такой датой есть и не один (оценка события – Да), то подзапрос возвращает две строки, оператор EXISTS принимает значение true, и все столбцы из таблицы «Заказы» переписываются в выходной набор.

Замечание:

1. EXISTS – это булево выражение, следовательно, его можно комбинировать с любыми другими булевыми выражениями с помощью логических операций AND, NOT, OR.

2. В отличие от предыдущих запросов, EXISTS работает не с одним полем, а со всей строкой, поэтому, в команде SELECT (в подзапросе), как правило указвается «*»

3. В рассмотренном выше примере оператор EXISTS считается один раз для первой строки внешнего запроса.

Для оценки сравнения значений некоторого поля и заданного условия могут применяться специальные операторы ANY и ALL.

Операторы ANY и ALL используются с предложением WHERE или HAVING. Оператор ANY возвращает true, если какое-либо из значений подзапроса соответствует условию. Оператор ALL возвращает true, если все значения подзапроса удовлетворяют условию.

Формат команды:

```
SELECT *
FROM < Имя внешней таблицы >
WHERE Поле = ANY (ALL)
                Select Поле
                    From <Имя внутренней таблицы>
                    Where <условие>);
```

Рассмотрим пример. Пусть работа по оформлению заказов (табл. 3.23 и табл. 3.24) происходит по сменно. В какие-то дни работает одна группа сотрудников, в другие дни – другие смены. За каждой сменой числится свой список оформленных заказов.

Построим SQL-запрос, который позволяет узнать, какая смена (кто конкретно) работал с 09 марта по 11 марта 2018. Для получения запрошенной информации достаточно найти совпадение хотя бы одного ID-продавца из таблицы «Заказы» с кодом из таблицы «Продавцы». И если произойдет первое совпадение, то выведем на экран полный список смены пофамильно. Код этого SQL-запроса:

```
SELECT *
FROM Продавцы
WHERE Код = ANY
                Select ID_Продавца
              From Заказы
              Where Продавцы.Код = Заказы.Продавец);
```

Замечание:

1. Любой запрос с ANY можно сделать при помощи оператора EXISTS (обратное утверждение неверно).

2. В запросе ANY можно использовать отношения: <, >, >=, <> и т.д.

Оператор ALL возвращает true, если все значения подзапроса удовлетворяют условию. Рассмотрим пример. Пусть требуется выбрать все заказы из табл. 3.23 «Заказы», превосходящие величину любого из них, сделанного 10.03.18.

```
SELECT *
FROM Заказы
WHERE Сумма > ALL
         Select [Стоимость заказа]
         From Заказы
         Where Дата = #10.03.18#);
```

Результат выполнения запроса:

№	Дата	Стоимость	Покупатель	Продавец
304	11.03.18	52 000	209	102

Замечание: как правило, оператор ALL используется с неравенствами, так как значение *x*= ALL(‘*x* равняется всем’) получает истину в одном единственно случае, когда в результате выполнения подзапроса найдены одно значение или несколько значений, каждое из которых равно *x*.

3.8. Соединение таблиц

Одной из важнейших черт запросов SQL является возможность определять связь между таблицами и работать с ними, как единым целым.

В SQL существует два способа соединения таблиц:

– Основанное на определении условия в предложении WHERE (эквисоединение без установления связи);

– Основанное на операции JOIN с установлением связи.

Рассмотрим пример на создание эквисоединения между таблицами

Даны табл.3.23 «Заказы» и табл. 3.24 «Продавцы». Необходимо найти список пар «Покупатель – ФИО продавца», чей заказ превышает 13 000.

```
SELECT Покупатель, [ФИО продавца], [Стоимость заказа]
FROM Заказы, Продавцы
WHERE Заказы.ID_Продавца = Продавцы.Код AND
[Стоимость заказа] > 13 000;
```

Здесь связь таблиц устанавливается с помощью запятой при их перечислении после параметра FROM.

Перебор записей осуществляется ($n \times m$) раз, где n – количество записей первой таблицы, а m – количество записей второй таблицы. Воздается множество всех возможных комбинаций строк и для любой проверяется условие: Заказы.ID_Продавца = Продавцы.Код из предложения WHERE.

Соединение таблиц, основанное на условиях в предложении WHERE, называются «эквисоединением таблиц». Две таблицы соединены только в рамках запроса.

Второй способ соединения – применение операции JOIN, когда устанавливается не эквисоединение, а настоящее соединение между таблицами, которое является основой схемы БД. Определение связей позволяет быстрее выполнять запросы, включающие несколько таблиц.

Связь, как мы определяли в предыдущем модуле – отношение между двумя однотипными полями соединяемых таблиц. Графически она отображается линией, соединяющей два поля.

Поскольку после соединения таблиц надо будем работать с данными, которые могут принадлежать полям, по которым выполнено соединение, то при построении запросов надо использовать полный синтаксис написания полей: <Имя таблицы>. <Имя поля>.

Существует два варианта соединения таблиц:

– симметричное, когда соединяемые таблицы равнозначны;

– внешнее, когда одна таблица главная, другая подчиненная.

Формат команды симметричного соединения таблиц

```
FROM таблица 1 INNER JOIN таблица 2 ON таблица
1.поле 1 = таблица 2.поле 2;
```

где таблица 1, таблица 2 – имена таблиц, записи которых подлежат объединению;

поле 1, поле 2 – имена полей, которые должны быть объединены. Если эти поля не являются числовыми, то должны иметь одинаковый тип данных и содержать данные одного рода, но могут иметь разные имена;
предложение ON описывает условие объединения таблиц.

Пусть по-прежнему источниками данных являются табл. 3.23 «Заказы» и табл. 3.24 «Продавцы» с соответствующими полями. Очевидно, что соединение этих таблиц должно выполняться по полям «Код» и «ID_Продавца».

Установим симметричное соединение этих таблиц – это значит, что согласно установленной связи в общий пул симметричного соединения войдут только пересекающиеся по соединяемому полю записи, остальные отсечены – например продавца с кодом 106 нет в заказах, поэтому нет смысла рассматривать эту запись в таблице «Продавцы».

Рассмотрим пример на симметричное соединение. Пусть, как и ранее требуется найти список пар «Покупатель – ФИО продавца», чей заказ превышает 13 000.

В строке from указываем симметричное соединение, в сроке where – записываем условие [Стоимость заказа] > 13 000.

Надо отметить, что результат тот же, что и в предыдущем примере, но время выполнения меньше, поскольку для любой строки таблицы «Заказы» просматриваются только те строки таблицы «Продавцы», для которых выполняется условие симметричного соединения.

Формат команды внешнего соединения таблиц:

```
FROM Таблица 1 LEFT JOIN Таблица 2 ON таблица 1.поле 1 = таблица 2. поле 2.
```

где операция LEFT(RIGHT) указывает, из какой таблицы брать все записи: если LEFT, то нужно брать все записи таблицы, расположенной слева от JOIN, если RIGHT – из таблицы справа.

Рассмотрим тот же набор исходных данных – табл. «Заказы» и табл. «Продавцы», но построим внешнее соединение, т.к. решим другую задачу.

Итак, есть полный список продавцов, работающих в компании – это справочная информация. Каждый раз работу выполняют некоторые продавцы из этого списка, а не все сразу. Поэтому таблица «Продавцы» – главная, а «Заказы» подчиненная.

Рассмотрим ситуацию, когда именно внешнее соединение между таблицами необходимо для получения запрашиваемых данных.

Например, требуется найти итоги работы на дату 10.03.18, а конкретно – количество заказов и их общую стоимость для каждого продавца компании.

Код SQL-запроса для решения этой задачи:

```
    SELECT [ФИО продавца], Count(Заказы.Продавец) AS
[Количество заказов], Sum(Заказы.[Стоимость заказа]) AS [Всего
на сумму]
    FROM Продавцы LEFT JOIN Заказы ON Продавцы.Код =
Заказы.Продавец
    GROUP BY Продавцы.Код
    WHERE [Дата продажи]=#10.03.18#;
```

Поскольку требуется найти итого работы для каждого продавца фирмы, то надо назначить справочную таблиц «Продавцы» главной, сделать подсчеты и, если кто-то из продавцов в этот день не работал, то его итоги будут представлены пустыми ячейками.

Итак, после выполнения второй строки – будет установлено внешнее соединение. Заказы продавцов, работающих 10.03.18 будут объединены и найдена сумма, оформленных ими заказов в этот день.

Результат запроса следующий:

ФИО продавца	Количество заказов	Всего на сумму
Смирнов Е.П.	2	22 000
Алексеев В.Н.	3	90 000
Белов А.А.		

В тех случаях, если надо исключить дублирование значений, при построении запроса к одной таблице используется предикат DISTINCT. Когда же в запросе фигурируют две и более таблиц, то есть имеется их соединение и нужно исключить дублирование значений отдельного поля, выводимого на экран, то используется аргумент DISTINGCROW команды SELECT.

Формат такого запроса:

```
SELECT DISTINGCTROW <поля >
FROM Таблица 1 INNER JOIN Таблица 2 ON
Таблица1.Поле1 = Таблица2.Поле2;
```

Здесь сразу же после команды SELECT указывается, что надо исключить дублирование поля, а затем устанавливается связь между таблицами.

Рассмотрим пример на устранение дублирования. Выведем список продавцов (по фамилиям), которые смогли 10.03.18 оформить заказы (не важно сколько, важно, что они есть).

Результат запроса:

Продавец	ФИО продавца
105	Алексеев В.Н.
102	Смирнов Е.П.

Таким образом, есть несколько способов и вариантов соединения таблиц. Способы – это эквисоединение, которое создается только на конкретный запрос и соединение с применением команды JOIN, которая устанавливает настоящее соединения по пересекающимся полям. Вариантами являются симметричное и внешнее соединение. Оба варианта используются в зависимости от поставленной задачи поиска данных.

4. БАЗЫ ДАННЫХ NOSQL

Базы данных NoSQL появились в ответ на необходимость обрабатывать «большие» данные на крупных аппаратных платформах, состоящих из вычислительных кластеров. Таким образом, в настоящее время реляционные БД уже не являются единственной моделью отображения данных. У разработчиков информационных систем появился выбор модели хранения и управления данными.

4.1. Нереляционная модель данных

За то время, пока реляционная модель была практически единственной при отображении физической модели хранения данных, накопилось ряд проблем, которые реляционные БД решают неэффективно или даже неестественно.

Вот некоторые из них.

1 Для реляционных баз данных характерна потеря соответствия. Проявляется она в том, что реляционные базы данных не позволяют хранить агрегаты. Агрегат – термин, пришедший из предметно-ориентированного проектирования, где агрегатом называют коллекцию связанных объектов, которая интерпретируется как единое целое, что-то наподобие коробки, которую мы подписываем и складываем в нее предметы, которые должны находиться вместе.

Например, чтобы отобразить всю информацию о покупателе и всех его покупках необходимо собрать данные из многих таблиц: покупатель, заказ, пункт заказа, цена и др. и хранить их в виде соответствия. При значительном росте числа таблиц формирование агрегата покупателя существенно осложняется.

2. С увеличением объема хранимых данных возникает задача фрагментации таблиц базы данных по разным серверам, объединенных в кластер. При выполнении сложных запросов производительность системы существенно уменьшается в результате межмашинного обмена данными между серверами кластера. Возникает проблема обеспечения надежности кластера.

3. Схема базы данных состоит из подсхем, каждая из которых отражает предметную область какого-либо подразделения организации. Модификация подсхемы одного подразделения и реорганизация соответствующих таблиц приводит к вынужденной приостановке работы остальных подразделений. Поддержка целостности данных и оперативности такого взаимодействия является сложной задачей.

Как попытка решить накопившиеся проблемы реляционных баз данных появились альтернативные средства хранения и обработки данных, получившие название «базы данных NoSQL». Пионерами в этой области выступили две компании: Google и Amazon,

Идея нереляционных баз данных очень проста: данные хранятся в виде записей <ключ, значение>, а схема базы данных отсутствует. При этом, в поле «значение» может храниться агрегат, например, вся информация о покупателе и его покупках. Так решается первая проблема, присущая реляционным БД.

Данные в виде агрегатов автоматически равномерно распределяются и реплицируются по серверам кластера. Таким образом решается вторая проблема, присущая реляционным БД.

Отсутствие схемы базы данных позволяет включать или удалять атрибуты на уровне отдельной записи, не затрагивая работу остальной части системы, что решает третью проблему, свойственную реляционным БД.

Термин "NoSQL" в настоящее время не имеет строгого определения. И не существует авторитетного органа, который бы предложил такое определение, поэтому можно говорить о некоторых общих свойствах БД, относящихся к категории NoSQL.

Первое свойство – очевидный факт: базы данных NoSQL не используют язык SQL. Некоторые из них имеют свой язык запросов, многие из которых похожи на SQL чтобы их легче было изучить. Однако до сих пор не реализован ни один язык, который достиг той же степени гибкости, что и стандартный язык SQL.

Другое важное свойство этих баз данных заключается в том, что они представляют собой проекты с открытым исходным кодом. Несмотря на то что термин "NoSQL" часто применяется к системам с закрытым исходным кодом, существует мнение, что NoSQL – это феномен с открытым исходным кодом.

Большинство баз данных NoSQL создавались в ответ на необходимость работать на кластерах. Для обеспечения согласованности в реляционных БД используют транзакции. Это изначально противоречит кластерной среде, поэтому базы данных NoSQL предлагают спектр вариантов для обеспечения согласованности и распределения данных.

Базы данных NoSQL учитывают емкость веб-сайтов начала XXI века, поэтому обычно только системы, разработанные примерно в это время, называются NoSQL, тем самым исключая базы данных, созданные в прошлом веке.

Базы данных NoSQL работают без схемы, позволяя свободно добавлять поля в базу данных без предварительного изменения структуры. Это очень важно для БД с неоднородными данными

Все свойства, указанные выше, являются общими для баз данных NoSQL. Ни одно из них нельзя считать определяющим.

Таким образом сформулируем: базы данных NoSQL – это распределенные нереляционные базы данных с открытым исходным кодом. Из известных баз данных NoSQL можно назвать Cassandra, Riak, DynamoDB, Hypertable, MongoDB и другие.

Все БД NoSQL являются неструктурированными. Когда данные хранятся в реляционной базе, то сначала определяется схема БД. В базах данных NoSQL хранение данных происходит по-другому.

Каждая БД, реализованная по технологии NoSQL использует свою собственную модель. Эти модели разделяются на четыре категории:

– ключ-значение;

– документ;

– семейство столбцов;

– граф.

БД типа "ключ-значение" позволяет хранить данные по ключу.

Документная база данных по существу делает то же самое, поскольку не накладывает ограничений на структуру хранящихся документов.

Семейство столбцов позволяет хранить любые данные в любом столбце.

Графовые базы данных отображают объекты и связи между ними в виде графа. Путем добавления, удаления, изменения ребер и узлов графа происходит управление данными.

Первые три модели объединяет свойство агрегатной ориентацией. Рассмотрим, что понимают под агрегатами и как они влияют на модели данных.

Реляционная модель хранимую информацию разделяет на кортежи (строки). Кортеж – это ограниченная структура данных. Он хранит набор значений, поэтому не может содержать запись, список значений или другой кортеж. Эта простота образует основу реляционной модели и позволяет интерпретировать все операции как операции над кортежами и возвращение кортежей.

Агрегатная ориентация придерживается другого подхода. Она учитывает необходимость оперировать данными, имеющими более сложную структуру, чем набор кортежей. Агрегат можно сравнить со сложной записью, которая может содержать списки и другие

структуры записей (рис. 4.1). Агрегат не имеет строгого шаблона и в зависимости от задачи может иметь структуру разной сложности. Таким образом, агрегат представляет собой единицу для манипулирования данными и управления их согласованностью.

Рис. 4.1. Структура агрегата

Модификация агрегатов происходит с помощью атомарных операций. Взаимодействие с БД, как хранилищем данных выполняется посредством агрегатов.

Агрегаты облегчают работу баз данных на кластерах, поскольку представляет собой естественную единицу репликации и фрагментации. Кроме того, агрегаты упрощают разработку прикладных программ, которые часто манипулируют данными с помощью агрегированных структур.

Продемонстрируем сказанное на примере. Предположим, что разрабатывается веб-сайт для электронной торговли и планируется продавать товары непосредственно клиентами через web, что требует хранения информации о пользователях, каталогах товаров, заказы, адреса поставки и даты платежей. Этот сценарий используем для моделирования данных с помощью реляционной модели, а также с помощью технологии NoSQL, что позволит проанализировать их преимущества и недостатки.

Разработку реляционной базы данных можно начать с модели данных, представленной на рис. 4.2. В ней выполнены все правила реляционной модели и проведена нормализация отношений.

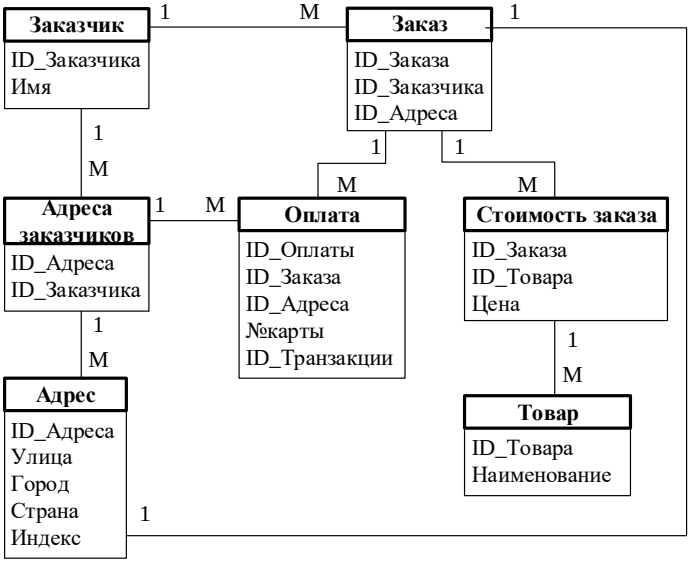


Рис. 4.2. Модель данных

Реляционная БД, соответствующая модели на рис. 4.2 включает 7 отношений. На рис. 4.3 приведены фрагменты некоторых из них.

Заказчик

ID_Заказчика	Имя
1	Яковлев С.А.
...	...

Заказ

ID_Заказа	ID_Заказчика	ID_Адреса
99	1	77
...	...	...

Адреса заказчиков

ID_Адреса	ID_Заказчика
77	1
...	...

...

Товар

ID_Товара	Наименование
27	«NoSQL»
...	...

Рис. 4.3. Фрагменты таблиц, являющихся частью реляционной БД

Посмотрим, как будет выглядеть эта модель, если применить агрегатно-ориентированный подход. Модель изобразим средствами

UML-диаграммы (рис. 4.4). В UML-диаграмме ромб обозначает агрегацию.

Данные будем представлять в формате JSON[1], который является основным способом представления данных в технологии NoSQL. В модели есть два основных агрегата: З**аказчик** и **Заказ**.

Агрегат **«Заказчик»** содержит список адресов заказчиков.

Агрегат «**Заказ»** содержит список заказанных товаров, адреса поставки и данные о платежах. Запись о платеже сама содержит адрес заказчика, выполняющего данный платеж.

Отдельная логическая запись, содержащая адрес, в этом примере появляется три раза, но вместо использования идентификатора она интерпретируется как значение и каждый раз копируется. При использовании агрегатов можно копировать всю адресную структуру в агрегат.

Название товара показано в качестве части заказа, – этот вид денормализации напоминает компромисс, принятый в реляционных базах данных, но по отношению к агрегатам он носит более общий характер, потому что есть необходимость минимизировать количество агрегатов, к которым будет осуществляться доступ при работе с данными

Связь между заказчиком и заказом не хранится в агрегатах. Аналогично связь, идущая от заказа, может идти к отдельной агрегированной структуре для товаров, но она не хранится в этой структуре.

[1] JSON – текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми

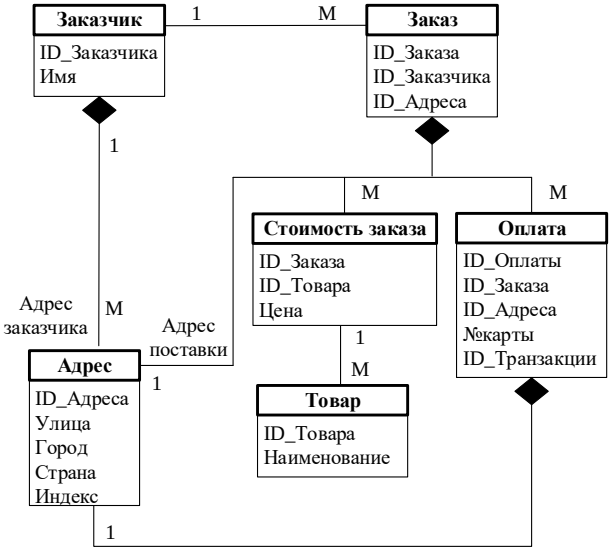


Рис. 4.4. Агрегатная модель данных

```
// Заказчик
{ «ID_Заказчика":1,
  "Имя":"Яковлев С.А.",
  "Адрес":
        [
        {"ID_Адрес":" 55",
         "Улица":" Б. Морская",
         "Город": "Санкт-Петербург",
         "Страна":"Россия",
         "Индекс": "190000"
        }
        ],
}
//Заказ
{"ID_Заказа":99,
 " ID_Заказчика":1,
 "Стоимость заказа": [
        { "ID_Товара":27,
          "Цена": 320,
          "Наименование": "NoSQL"
         }
         ] ,
 "ID_Адрес": [ { "ID_Адреса": "55"}],
```

```
 "Оплата": [
         { "№карты":"1000-1000-1000-1000",
          "ID_транзакции":"abelif879rft",
          "ID_Адреса": {"ID_Адреса": "55"}
         }
         ],
}
```

В данном примере важен не столько конкретный способ изображения границы агрегата, сколько тот факт, что необходимо думать о доступе к данным при разработке модели данных для приложения.

Действительно, можно иначе изобразить границы агрегатов, поместив все заказы отдельного заказчика в агрегат «Заказчик», как на рис. 4.5.

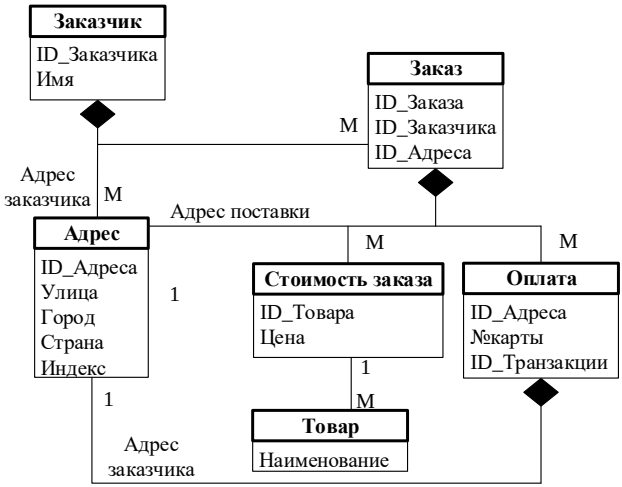


Рис. 4.5. Агрегатная модель, в которой все объекты объединены в один агрегат «Заказчик»

```
//Заказчик
{«ID_Заказчика":1,
  "Имя":"Яковлев С.А.",
  "Адрес": [
```

```
        {"ID_Адрес":" 55“,
          "Улица":" Б. Морская",
          "Страна":"Россия",
          "Индекс": "190000"
        } ],
  "Заказ": [
        {"ID_Заказа":" 99",
         "Стоимость заказа":[
                {“ID_Товара":27,
                  "Цена": "300 “
                }
                ]
        }
        ],
         "Оплата":[
                 {"№карты":"1000-1000-1000-1000",
                   "ID_транзакции":"abelif879rft",
                   "ID_Адреса": {"ID_Адреса": "55"}
                 }  ]
}
```

Универсального способа для изображения границ агрегатов не существует. Это целиком зависит от целей манипулирования данными. Например, чтобы отобразить всю информацию о заказчике и всех его заказах, программист должен собрать в оперативной памяти данные из многих таблиц: заказчик, заказ, адрес заказа, цена и др. При значительном росте числа таблиц разработчик часто просто забывает назначение той или иной таблицы, осложняется связывание таблиц при выполнении запроса, то есть существенно осложняется формирование агрегата. Поэтому, если необходимо получать доступ к записи о заказчике и ко всем его заказам одновременно, то, вероятно, предпочтительнее один агрегат. Однако, если необходимо в каждый момент времени получать доступ к отдельному заказу, то лучше предусмотреть отдельный агрегат для каждого заказа. Естественно, это сильно зависит от данных; даже в рамках одной системы разные приложения могут иметь разные предпочтения.

Большинство баз данных NoSQL не поддерживают механизм транзакций. Известно, что транзакции – это механизм, который обеспечивают согласованность данных. Но в БД NoSQL поддерживаются другие механизмы манипуляции с отдельными агрегатами по очереди из кода приложения.

Итак, обобщая информацию об агрегатной (нереляционной) модели данных можно сделать следующие выводы:

1. В агрегатной (нереляционной) модели данных существует неявная схема, подразумеваемая программистом БД.

2. Границы агрегатов выбираются программистом БД и во многом зависят от целей манипулирования данными.

3. Агрегаты объединяют в одно целое данные, доступ к которым осуществляется одновременно.

4. Агрегаты указывают, какие части данных должны храниться на одном и том же узле.

5. Если реляционные базы данных позволяют манипулировать любой комбинацией строк из любой таблицы в рамках одной транзакции, то в нереляционных БД аналогичная задача решается разделением данных по агрегатам.

4.2. Распределение и согласованность

Основным свойством технологии NoSQL является возможность функционирования баз данных на большом кластере, то есть размещение базы данных на кластере серверов. Этот процесс также называется горизонтальным масштабированием базы данных.

Агрегатно-ориентированный подход хорошо согласуется с горизонтальным масштабированием, поскольку агрегат является естественной единицей распределения. В зависимости от выбранной модели распределения можно создать хранилище данных, предоставляющее возможность обрабатывать больший объем данных, обрабатывать более интенсивный трафик чтения или записи, а также избегать перегрузки и торможения компьютерной сети.

С другой стороны, работа на кластерах повышает сложность базы, поэтому при выборе модели распределения взвешиваются все аргументы за и против.

Существуют два способа распределения данных: репликация и фрагментация (рис. 4.6).

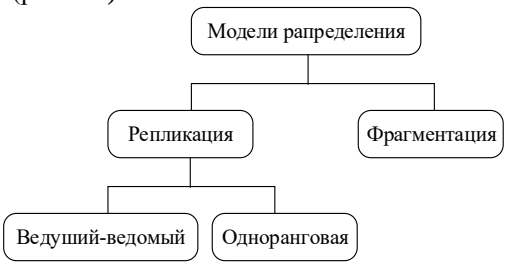


Рис. 4.6. Классификация способов распределения данных

Репликация подразумевает копирование одних и тех же данных на нескольких узлах (реплик). Количество реплик называется коэффициентом репликации. Репликация бывает двух видов: ведущий-ведомый и одноранговая.

Фрагментация подразумевает, что разные части БД размещаются на множество серверов. Если механизм фрагментации не применять, то пользователи будут обращаться к одному большому серверу. При наличии механизма фрагментации пользователи будут обращаться к разным узлам и благодаря этому получать быстрые ответы. Например, если БД распределена в кластере из 10 серверов, то каждый из них будет загружен только на 10%. Разумеется, идеальный случай крайне редок. Для того чтобы приблизиться к идеалу, необходимо, чтобы данные, которые запрашиваются одновременно, размещались вместе на одном и том же узле для ускорения доступа к ним. Поэтому актуальность вопроса заключается в том, как сгруппировать данные так, чтобы один пользователь в основном получал данные с одного сервера. Это тот случай, когда помогает агрегатная ориентация, т.к. главное свойство агрегата заключается в том, что он объединяет данные, которые, как правило, запрашиваются одновременно. Таким образом, агрегаты являются естественной единицей распределения.

Репликация и фрагментация являются ортогональными методами: можно использовать любую из двух или обе вместе.

Рассмотрим эти методы подробнее.

При **распределении по схеме "ведущий-ведомый"** происходит репликация данных по многим узлам. Один узел назначается ведущим (master), или главным. Этот ведущий узел является доверенным источником данных и обычно несет ответственность за выполнение всех модификаций этих данных. Остальные узлы являются ведомыми (slaves), или вторичными. Процесс репликации синхронизирует ведомые узлы с ведущим.

Репликация "ведущий-ведомый" – это решение для тех баз данных, к которым интенсивно выполняется операция чтения. То есть, чтобы выполнить больше запросов на чтение, необходимо добавить больше ведомых узлов и направлять на них все запросы на чтение. Однако если обновлять данные на ведущем узле, есть опасность, что чем больше ведомых узлов, тем выше вероятность, что пользователи будут получать несогласованные данные. То есть это неудачное решение для баз данных с интенсивным трафиком записи.

Второе преимущество репликации "ведущий-ведомый" – это отказоустойчивость чтения, то есть, если на ведущем узле произойдет отказ, ведомые узлы смогут по-прежнему обрабатывать запросы на

чтение. Сбой ведущего узла сделает запись невозможной, пока его работа не будет восстановлена или не будет подключен новый ведущий узел. Как раз наличие реплик ведущего узла на ведомых узлах ускоряет процесс его восстановления после сбоя.

Таким образом, репликация имеет не только привлекательные свойства, но и неизбежный недостаток – несогласованность. Существует опасность, что разные клиенты, читающие данные с ведомых узлов, получат разные значения из-за того, что обновления не успеют распространиться по всем ведомым узлам.

По существу, ведущий узел остается узким местом и единственной точкой отказа в модели «ведущий-ведомый».

Одноранговая репликация решает эту проблему, устраняя ведущий узел. Все реплики имеют одинаковый вес, все могут выполнять операции записи, и потеря любой из них не приводит к потере доступа к хранилищу данных.

Самая большая сложность, как и раньше, – согласованность. Когда выполняется запись в два разных места, есть риск, что два человека попробуют обновить одну и ту же запись в один и тот же момент времени. Таким образом возникает конфликт "запись-запись". Несогласованность чтения тоже приводит к проблемам, но они являются преодолимыми. А вот, несогласованность записи имеет необратимый характер.

Одним из глобальных отличий реляционной базы данных от базы данных NoSQL – это то как обеспечивается согласованность данных.

Согласованность – это обеспечение внутренней непротиворечивости данных. Различают строгую и итоговую согласованность. Строгая гарантирует, что данные вернутся полностью достоверными и не устареют. Итоговая согласованность не может гарантировать, что данные вернуться полностью достоверными, но в итоге данные обновятся на всех репликах.

Согласованность проявляется в разных формах. Рассмотрим примеры.

Представим процедуру **обновления данных**, при котором случайно два пользователя, назовем их User1 и User2 внесли обновления. Эта проблема называется конфликтом "запись-запись". Она возникает, когда два пользователя БД обновляют одни и те же данные в один и тот же момент времени *tw* (рис. 4.7). Когда записи достигают сервера, тот их сериализирует – обрабатывает одну, а потом другую, например в алфавитном порядке, и реализует сначала обновление первого пользователя User1, а затем обновление User2. Если контроля согласованности нет, то обновление первого

пользователя будет выполнено, а затем немедленно перекрыто обновлением второго пользователя. В этом случае обновление первого пользователя называется потерянным. Это явление можно считать нарушением **согласованности обновления**, потому что обновление второго пользователя использовало состояние до обновления первого, но применялось после него.

Репликация повышает вероятность конфликтов "запись-запись". При обновлении реплик на разных узлах независимо друг от друга необходимы специальные меры обеспечения согласованности данных. Решение, которое способствует снижению вероятности возникновения конфликта "запись-запись" заключается в том, чтобы все записи определенных данных хранить на одном узле. Это решение использовалось во всех распределенных моделях, рассмотренных выше, кроме одноранговой репликации.

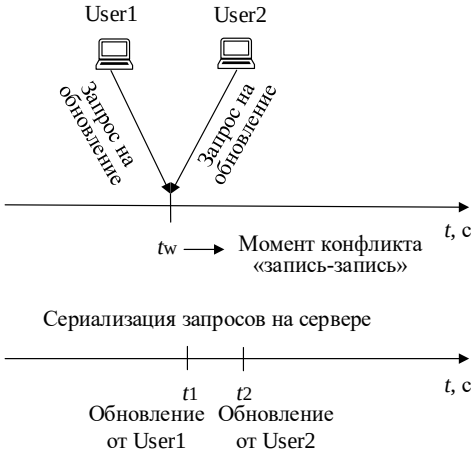


Рис. 4.7. Конфликт обновления данных

Представим процедуру чтения данных. Пусть оформляется заказ на определенные товары с определенными расходами на доставку. Расходы на доставку рассчитываются на основе товаров, указанных в заказе. Если добавляется новая позиция, то вычисления необходимо выполнить заново и обновить запись о расходах на поставку.

Опасность несогласованности заключается в том, что первый пользователь сначала добавляет позицию в свой заказ, а затем второй пользователь считывает эти позиции и расходы на доставку, а после

этого первый пользователь обновляет запись о расходах на доставку. Этот вид ошибки называется **несогласованным чтением** или конфликтом «чтение-запись».

На рис. 4.8 показана ситуация, в которой второй пользователь выполнил чтение в середине процедуры записи, выполняемой первым пользователем. Для исключения такой ситуации используется метод логической согласованности. Она гарантирует, что разные элементы данных будут изменяться вместе. Например, для того чтобы избежать логической несогласованности при конфликте "чтение-запись", реляционные базы данных используют понятие транзакций. Обе записи первого пользователя упаковываются в одну транзакцию, и тем самым система гарантирует, что второй пользователь будет читать оба элемента данных либо до, либо после обновления.

Агрегатная модель базы данных NoSQL позволяет избежать несогласованности сделав заказ, стоимость доставки и товарные позиции заказа частями одного и того же агрегата.

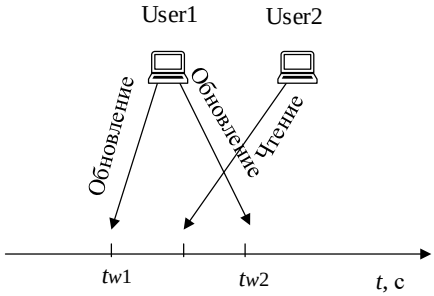


Рис. 4.8. Конфликт чтения данных

Разумеется, не все данные можно записать в один и тот же агрегат, поэтому любые обновления, влияющие на несколько агрегатов, оставляют интервал времени, в течение которого клиенты могут выполнить несогласованное чтение. Продолжительность этого интервала называется окном несогласованности. Система NoSQL может иметь довольно узкое окно несогласованности. Например, в документации компании Amazon сказано, что окно несогласованности обычно не превышает секунды.

С появлением репликации появился и новый вид несогласованности. На рис. 4.9 приведено пояснение этого вида несогласованности.

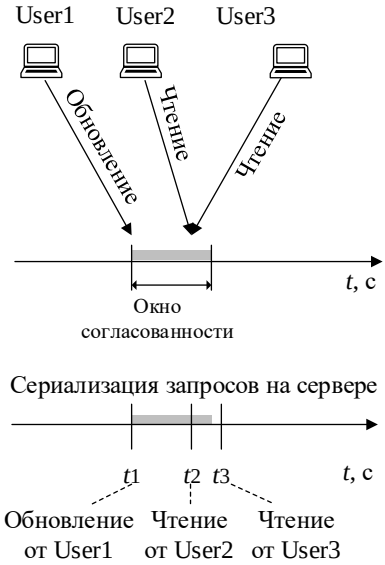


Рис. 4.9. Нарушение согласованности репликаций

Пользователь User1 вносит обновления, но наличие окна несогласованности означает, что разные пользователи БД увидят разные данные в одно и то же время. Поэтому User2 получит недостоверные данные (еще не обновленные), а пользователь User3 – уже обновленные.

Таким образом, согласованность репликаций гарантирует, что при чтении с разных реплик один и тот же элемент данных имеет одно и то же значение.

Разумеется, в конце концов, обновления будут полностью распределены по узлам, и User2 увидит обновленные данные. По этой причине такая ситуация называется итоговой согласованностью или согласованностью «в конечном счете». Это значит, что в любой момент времени узлы могут быть несогласованными, но, если нет новых обновлений, в конце концов все узлы будут обновлены и получат одно и то же значение.

Согласованность – это важное свойство БД, но, к сожалению, иногда ею приходится жертвовать. Почти всегда можно разработать систему, предотвращающую несогласованность, но практически невозможно сделать это без ущерба для остальных характеристик

системы. В результате часто приходится жертвовать согласованностью в пользу чего-то другого.

В NoSQL этот компромисс формулируется теоремой CAP. CAP – это акроним от трех свойств:

– **Согласованность** (**C**onsistency) – непротиворечивость данных.

– **Доступность** (**A**vailability) – предельное время отклика, которое допустимо.

– **Устойчивость к разделению** (**P**artition tolerance) означает, что кластер может восстанавливать обмен данными после обрыва связей в кластере, который разделен на многочисленные фрагменты, не способные взаимодействовать друг с другом

Основное утверждение этой теоремы гласит, что из трех свойств – согласованности данных, доступности и устойчивости к разделению – можно обеспечить не больше двух. На практике CAP реализует следующее утверждение: в системе, которая подвержена разделению, следует искать компромисс между согласованностью и доступностью. Полученная в результате система не будет ни хорошо согласованной, ни идеально доступной, но она будет представлять собой разумное сочетание этих свойств.

Чем больше узлов задействовано в запросе, тем выше вероятность возникновения конфликта «запись-запись» и ниже вероятность конфликта «чтение-запись». Отсюда естественен вопрос: «Сколько узлов (реплик) должно быть вовлечено в запрос, чтобы обеспечить строгую согласованность данных?»

Введем следующие обозначения:

R – коэффициент репликации;

W – количество узлов, участвующих в записи;

N – количество узлов, участвующих в чтении.

Согласно теореме CAP, не нужно, чтобы все узлы подтверждали запись для обеспечения строгой согласованности; нужно, чтобы это сделали большинство. Это явление называется кворумом записи и выражается неравенством

$$W>N/2 \qquad (4.1)$$

Аналогично существует кворум чтения, но это более сложное понятие. Кворум чтения зависит от того, сколько узлов должны подтвердить запись. Пусть R=3. Если все записи должны подтвердить два узла (W=2), то необходимо установить контакт по крайней мере с двумя узлами, чтобы гарантировать получение последних данных. Если же записи подтверждаются только одним узлом (W=1), надо связаться со всеми тремя узлами, чтобы гарантировать получение

последних обновлений. В последнем случае нет кворума записи, поэтому возникает конфликт обновлений, но, контактируя с достаточно большим количеством читателей, обнаружение конфликта гарантированно. Таким образом, можно получить строго согласованные результаты чтения, даже если нет строгой согласованности записей. Неравенство получения строгой согласованности:

$$R+W>N \tag{4.2}$$

Неравенства (4.1) и (4.2) выведены для одноранговой модели распределения.

В случае распределения "ведущий-ведомый", чтобы избежать конфликтов "запись-запись", достаточно записать данные на ведущий узел. Чтобы избежать конфликтов "чтение-запись", достаточно выполнять чтение только с ведущего узла. Уточним, что количество узлов в кластере и коэффициент репликации – это разные числа. Например, при фрагментации баз данных кластер можете иметь 100 узлов при коэффициенте репликации, равном 3.

Обобщим информацию о правилах распределения и согласованности данных в следующих выводах:

1. Конфликты «запись-запись» возникают, когда два клиента пытаются записать одни и те же данные в одно и то же время.

2. Конфликты «чтения-записи» в распределенных системах возникают, когда некоторые узлы получают обновленные данные, а другие нет.

3. Итоговая согласованность означает, что определенная часть системы станет согласованной, как только все записи будут распространены по всем узлам.

4. Чтобы обеспечить хорошую согласованность данных в распределенных системах возникает необходимость компромисса между согласованностью и временем реакции (теорема CAP).

4.3. Базы данных «ключ – значение»

Базы данных «ключ – значение» состоит из множества агрегатов, каждый из которых имеет ключ или идентификатор, который используется для доступа к данным. Агрегаты в такой базе данных являются непроницаемыми, то есть просматривать содержимое агрегата можно только с помощью его ключа.

Внешне БД «ключ – значение» похожа на реляционную БД с двумя столбцами, например ID и Имя (табл. 4.1), где столбец ID – первичный ключ, а столбец Имя содержит значение. При этом значение – это двоичный объект данных, который записан в хранилище без детализации его внутренней структуры в виде хэш-кода. Это позволяет с одной стороны избавиться от метаданных, то есть хранить прямое значение, а с другой стороны обеспечить целостность данных. Что именно хранится в этом объекте может определить только приложение.

Табл. 4.1. БД «ключ – значение»

ID	Имя
AAAAA	1001010100010001…
AABAB	0100010110011101…
DFA766	0000111101101101…
FABCC4	1100110010101110…

Из-за хранения данных в поле значение в виде хэш-кода говорят, что БД "ключ-значение" – это хэш-таблица. Выбор функции хэширования должен обеспечить равномерное распределение хэшированных ключей по хранилищу данных.

С точки зрения интерфейса прикладного программирования хранилище типа "ключ-значение" – самое простое из БД NoSQL. Управление данными в плане манипулирования ими сводится к тому, что клиент может либо получить значение по ключу, либо записать значение по ключу, либо удалить ключ из хранилища данных. Поскольку хранилища типа "ключ-значение" всегда используют доступ по первичному ключу, они обычно имеют высокую производительность и легко масштабируются.

Приложение вычисляет ID и значение и сохраняет эту пару. Если ключ ID уже существует, то текущее значение замещается, в противном случае создается новая запись.

Если есть необходимость хранить данные сеанса пользователя, информацию о его корзине товаров и предпочтениях, то можно записать их в один агрегат с одним ключом для всех перечисленных объектов (рис. 4.10).

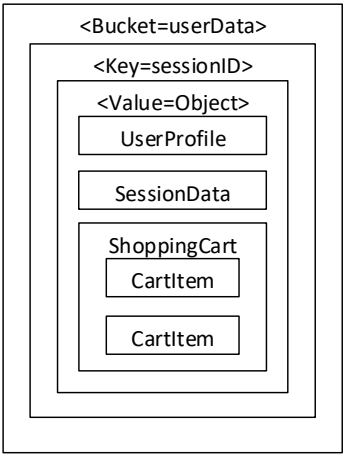


Рис. 4.10. Пример агрегата БД «ключ-значение»

Недостатком хранения всех объектов в одном агрегате является тот факт, что агрегаты могут иметь разные типы, которые могут вызвать конфликты ключей.

В качестве альтернативы к ключу можно было бы добавить имя объекта, чтобы при необходимости можно было извлечь отдельный объект по этому имени (вложение ключ-значение), как на рис. 4.11.

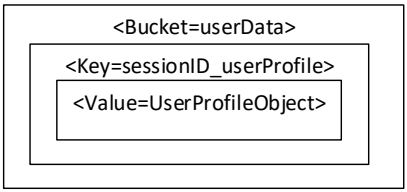


Рис. 4.11. Альтернативный вариант агрегата

Соответствие терминологии, принятой в реляционной модели БД и модели «ключ-значение» приведено в табл. 4.2.

Табл. 4.2. Соответствие терминов

Реляционная модель БД	Модель «ключ-значение»
База данных	Кластер
Таблица	Сегмент

Строка	Ключ-значение
Идентификатор строки	Значение

При использовании хранилищ типа "ключ-значение" большое внимание уделяется выбору структуры ключа – алгоритмам генерации ключа, использованию меток времени, индивидуальных данных пользователя в этих алгоритмах.

Благодаря своим характеристикам базы данных типа "ключ-значение" часто используют для хранения данных о пользовательских сессиях (при этом в качестве ключа используется идентификатор сессии), корзин покупателей, профилей пользователей и т.п.

Рассмотрим примеры, в которых хранилища типа "ключ-значение" на практике зарекомендовали себя с лучшей стороны:

Хранение информации о сессии.

Каждая веб-сессия является уникальной и имеет уникальный идентификатор SessionId. Приложение записывает идентификатор SessionID в БД и всю информацию о сессии одним запросом. Операции выполняются очень быстро, поскольку вся информация о сессии хранится в одном объекте.

Профили пользователей, предпочтения, товары.

Почти каждый пользователь имеет уникальный атрибут UserID, UserName или какой-то другой идентификатор, а также предпочтения, например, язык, цвет, часовой пояс, выбранные товары и т.д. Все это можно поместить в один объект и получать предпочтения пользователя с помощью одной операции. Аналогично можно хранить профили товаров.

Корзины заказа.

Коммерческие веб-сайты используют корзины заказа, связанные с пользователем. Если требуется, чтобы корзина заказа была доступна постоянно, независимо от браузеров, компьютеров и сессий, всю информацию о покупках можно поместить в объект Value с ключом UserId.

Существуют ситуации, в которых хранилища типа "ключ-значение" не являются оптимальным выбором. Приведем примеры таких ситуаций:

Запрос по данным.

В базах данных типа «ключ-значение» отсутствие структурированности данных не позволяет задать условия к конкретному внутреннему агрегату.

Отношения между данными.

Если между разными наборами данных необходимо установить отношения или поддерживать корреляцию между разными наборами ключей, то базы данных типа "ключ-значение" не имеют такой возможности.

Операции с множествами.

Поскольку операции в каждый момент времени ограничены одним ключом, невозможно работать с несколькими ключами одновременно. Если требуется обработать несколько ключей сразу, то это придется делать на клиентской стороне.

Транзакции, состоящие из многих операций.

Если при сохранении нескольких ключей при записи одного из них произошел сбой нет возможности вернуться в исходное положение или выполнить откат остальных операций.

Изучение особенностей баз данных типа «ключ-значение» позволяет сделать следующие выводы:

1. База данных «ключ-значение» – это хеш-таблица.
2. Доступ к данным БД реализуется только по ключу.
3. Для извлечения информации об отдельном объекте необходимо создать его уникальный ключ.
4. Объекты можно объединять в агрегаты.
5. База данных типа «ключ-значение» может быть распределенной (кластеры).
6. Базы данных типа «ключ-значение» имеют узкую область применения.
7. Базы данных типа «ключ-значение» не годятся для задач (запросов), типичных для реляционной модели БД.

4.4. Документные БД

Документные базы данных концептуально похожи на БД типа "ключ-значение", только в качестве значений хранятся документы. Таким образом, основной единицей манипулирования является документ, основная часть которого является неструктурированным текстом.

Документы хранятся примерно одинаково, но сами они не обязательно должны быть одинаковыми, то есть это тексты, графические, звуковые, мультимедийные документы.

Документные БД предназначены для создания, хранения и выдачи по запросам документов, содержащих требуемую информацию. В ответе на запрос к такой БД пользователь получает

список документов, в определенной мере содержащих нужную ему информацию.

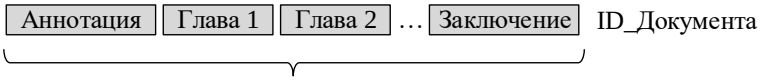
Мера соответствия полученного результата построенному запросу оценивается релевантностью. Релевантность – это характеристика, которая выражает полезность информации, относительно запроса, отправляемого в информационную систему.

Поисковые запросы к документным БД – это поиск смысловой (семантической) информации, например: выдать статьи, посвященные документным БД, то есть содержащие термин "документные БД".

Таким образом, поиск в документной БД сводится к сравнению смыслового содержания запроса со смысловым содержанием хранящихся в БД документов. Семантическая природа документов определяет необходимость учитывать синонимию, полисемию, омонимию, контекстную обусловленность смысла отдельного слова и возможность выразить один смысл многими способами.

База данных хранит и извлекает документы в форматах XML, JSON, jpeg, avi, wav и т.д.

Записью документальной базы данных является документ, который задается как набор необязательных полей, для каждого из которых определены имя и тип (рис. 4.12).



Р

ис. 4.12. Вид записи документальной БД

Для документных БД допустимы стандартные типы, задающие числовые, символьные и другие величины, но основной тип текстовый. Текстовые поля обладают переменной длиной и композиционной структурой, что не имеет прямых аналогов среди стандартных типов языков программирования. Текстовое поле состоит из параграфов, параграф – из предложений, предложение – из слов.

С точки зрения хранения, в документных БД идентифицируемым элементом данных будет поле, а с точки зрения поиска – слово.

Система управления документными базами данных присваивает каждому документу уникальный номер, а каждому ключевому слову документа ставится в соответствие указатель на списки экземпляров, являющихся перечнем документов, в которых встречается данное слово (то есть создается индекс). Каждый список экземпляров

содержит заголовок, из которого можно узнать число экземпляров слова во всем файле документов, а также число документов, в которых это слово встречается.

Документальная БД включает в себя как минимум три области хранения данных, представляемые из-за своего большого размера, как правило, в виде файлов операционной системы (в действительности их всегда больше):

– файл словаря, устанавливающий соответствие между словом, встречающимся в БД, и его кодом;

– инверсный (инвертированный, обратный) список, содержащий для каждого слова БД список документов, его содержащих, используется при текстовом поиске;

– текстовый файл, содержащий собственно документы, используется при выдаче (просмотре) документов.

На рис. 4.12 приведена принципиальная схема организации поиска документов, характерная для большинства современных документальных БД.

Рис. 4.12. Принципиальная схема организации поиска документов

Рассмотрим пример упрощенной реализации документальной БД в среде реляционной СУБД. С логической точки зрения она имеет «стандартную» структуру и включает две компоненты: регистрационные карты (РК) и полные тексты (ПТ).

Регистрационные карты представляют собой форматированные записи, содержащие относительно стандартный набор библиографических данных, а также ссылку на соответствующий полный текст (рис. справа).

Полные тексты документов состоят из страниц двух типов:

– логических, то есть структурных единиц текста – пунктов, параграфов, статьей;

– физических – фрагментов одинаковой длины, принудительно разбивающих длинный неструктурированный текст.

Организация физической структура документальной БД предполагает наличие следующих элементов:

Таблица ПТ – одна или несколько таблиц, в которых содержатся полные тексты документов. На логическом уровне образует представленную на рис.2 иерархическую структуру: БД, документ, страница.

Словарь ПТ – таблица представляет собой список ключевых слов и стандартных словосочетаний, извлеченных из текста, сопровождаемых частотами появления.

Инверсная таблица ПТ (или инверсный список ПТ) – таблица, содержащая список ключевых слов и словосочетаний, сопровождаемых номерами страниц.

Словарь и инверсная таблицы используются для сквозного полнотекстового поиска.

Таблица РК – таблица регистрационных карт, каждая запись которой содержит заглавие, дату регистрации, номер, вид документа, ссылки на страницы полного текста (ПТ) и другие поля.

Словарь РК – это таблица, содержащая значения полей регистрационных карт совместно с частотой появления и ссылками на записи таблицы РК.

Инверсная таблица РК (или инверсный список РК) содержит слова и словосочетания и ссылки на записи таблицы РК.

Словарная и инверсная таблицы используются для поиска записей РК, с последующим доступом к страницам полного текста.

Наряду со словарем РК иногда может использоваться словарь синонимов, служащий для обеспечения двуязычного поиска в словарных таблицах.

Благодаря такой организации физической структуры документной БД можно выполнять поиск двух видов:

– поиск по регистрационным картам и

– поиск по полным текстам.

Первый вид поиска соответствует случаю, когда пользователь что-то знает о документе, например, название, автора, дату выпуска и т.д. Самый простой случай, когда пользователь знает все. Тогда просто анализируется таблица РК, из нее отбирается нужная регистрационная

карта, из которой отбирается указатели на страницы полного текста документа. Далее эти страницы выбираются из таблицы ПТ.

Несколько сложнее поиск в случае, когда пользователь знает только часть атрибутов регистрационной карты, например, только одно название или только словосочетание из названия. В этом случае предварительно анализируется словарь и инверсная таблица РК, после чего отыскивается сама РК.

Полнотекстовый поиск соответствует ситуации, когда пользователь ничего не знает о документе и может указать только ключевые слова для него. В этом случае прежде всего используется инверсная таблица ПТ, из которой отыскивается список страниц, содержащих эти слова. Если такой список оказывается очень велик, может быть использован словарь ПТ, позволяющий сократить его в соответствии с частотой появления слов.

Соответствие терминологии, принятой в реляционной модели БД и документной модели БД приведено в табл. 4.3.

Табл. 4.3. Соответствие терминов

Реляционная модель БД	Документная модель БД
Схема данных	База данных
Таблица	Коллекция документов
Строка	Документ
Идентификатор строки	ID документа

Документные базы данных обеспечивают разнообразные функциональные возможности запросов.

Одним из преимуществ документных баз данных по сравнению с хранилищами типа "ключ-значение" является то, что можно послать запрос к содержанию документа, не извлекая весь документ по его ключу, чтобы просмотреть его. Это свойство делает такие базы данных похожими на модель запроса реляционных БД.

Рассмотрим некоторые запросы к базе данных MongoDB, в которой можно хранить документы.

Допустим, мы хотим вернуть все документы из коллекции заказов (все строки из таблицы заказов). На языке SQL этот запрос выглядит так:

```
SELECT *
FROM Заказы;
```

Эквивалентный запрос в оболочке базы Mongo выглядит следующим образом:

```
db. Заказы. find ()
```

Выбор заказов для конкретного идентификатора, равного 883c2c5Ь4e5Ь, можно записать так:

```
SELECT *
FROM Заказы
WHERE ID_покупателя = "883c2c5Ь4e5Ь";
```

Эквивалентный запрос в базе Mongo на получение всех заказов для конкретного идентификатора выглядит следующим образом:

```
db.order.find({" ID_покупателя":"883c2c5b4e5b"})
```

Аналогично выбрать записи orderld и orderDate для заданного клиента на языке SQL можно записать так:

```
SELECT orderId,orderDate
FROM order
WHERE customerId = "883c2c5Ь4e5Ь"
```

Эквивалент этого запроса в базе Mongo можно записать следующим образом:

```
db.Заказы.find({ID_покупателя:"883c2c5b4e5b"},
{ID_заказа:1, Дата_заказа:1})
```

Аналогично можно формировать запросы для подсчета, суммирования и выполнения других операций.

Преимущества документных БД в сравнении реляционными БД:

1. В сравнении с реляционными базами данных лучшая производительность при индексировании больших объемов данных и большим количестве запросов на чтение.
2. Легче масштабируются в сравнении с SQL решениями.
3. Децентрализированы – работа на скластере.
4. Легко менять "схему" данных: не нужно выполнять никаких операций обновления для добавления новых полей.
5. Хранение неструктурированных данных.

6. Единое место хранения всей информации об объекте, что требует меньше операций вида "join".

7. Простой интерфейс общения с БД (ключ → значение, нет SQL).

Недостатки документных БД:

1. Отсутствие транзакционной логики и контроля целостности в большинстве реализаций: необходимо реализовывать ее в логике приложения.

2. Для обработки данных необходимо использование дополнительного языка программирования.

Рассмотрим примеры, где документные БД подходят лучше всего:

Регистрация событий.

Приложения по-разному регистрируют события; на предприятиях существует множество разных приложений, желающих регистрировать события. Документные базы данных могут хранить все эти типы событий и действовать как центральное хранилище событий. Это особенно важно в ситуациях, когда тип данных, собираемых событиями, постоянно изменяется.

Системы управления информационным наполнением, блог-платформы.

Поскольку документные базы данных не имеют предопределенной схемы и обычно понимают JSON-документы, они хорошо работают в системах управления информационным наполнением или в приложениях по публикации веб-сайтов, управляющих комментариями пользователей, их регистрацией, профилями, а также представлением документов в веб.

Веб-аналитика и аналитика в реальном времени.

Документные базы данных могут хранить данные, необходимые для анализа в реальном времени; поскольку части документов можно обновлять, можно очень легко хранить представления страниц или информацию об отдельных посетителях, а также добавлять новые показатели эффективности без изменения схемы.

Приложения для электронной коммерции.

Приложения для электронной коммерции часто должны иметь гибкую схему товаров и заказов, а также возможность изменять свои модели данных без дорогостоящего перепроектирования кода базы данных или обновления структуры базы.

Документные БД не рекомендуется использовать в следующих задачах.

Сложные транзакции, охватывающие разные операции.

Если есть необходимость выполнять атомарные операции с несколькими документами, то документные базы данных для этого, как правило, не подходят.

Запросы к изменяющейся агрегатной структуре.

Гибкая схема означает, что база данных не накладывает на схему никаких ограничений. Данные сохраняются в виде сущностей приложения. Если возникает необходимость в специальном запросе к этим сущностям, то запросы можно изменять. Поскольку данные сохраняются как агрегат, то при постоянном изменении структуры агрегата необходимо сохранять его с наименьшим уровнем детализации, т.е. фактически нормализовать данные. В таком сценарии документная база данных может оказаться неработоспособной.

Изучение особенностей документных баз данных позволяет сделать следующие выводы:

1. Единицей хранения в документных БД является документ, основная часть которого – неструктурированный текст.
2. Документальная БД включает в себя как минимум три области хранения данных: файл словаря, инверсный список, текстовый файл.
3. Поиск в документных БД может быть реализован по метаданным (регистрационным картам) или ключевым словам непосредственно в контенте документа.
4. Схема данных отсутствует. Не нужно выполнять никаких операций обновления для добавления новых полей.
5. В системе управления документными БД отсутствует транзакционная логика и контроль целостности.
6. Для обработки данных необходимо использование дополнительного языка программирования.

4.5. Базы данных «семейство столбцов»

Семейство столбцов (столбчатая БД) – это коллекция строк, содержащая множество столбцов, ассоциированных с ключом строки. Семейства столбцов группируют взаимосвязанные данные, доступ к которым обеспечивается как к единому целому.

Основной единицей хранения в базе данных является столбец, состоящий из пары "имя-значение", в которой имя играет роль ключа. Каждая из пар "ключ-значение" может хранится с меткой времени, которая используется для того, чтобы задавать срок действия данных, разрешать конфликты записи, обрабатывать устаревшие данные и выполнять другие функции.

Модель семейства столбцов можно представить как двухуровневую агрегатную структуру. Как и в хранилищах типа "ключ-значение", главный ключ – это идентификатор строки, отмечая нужный в данный момент агрегат.

Отличительной особенностью "семейства столбцов" является то, что строка-агрегат сама состоит из ассоциативного массива более детализированных значений. Эти значения второго уровня называются столбцами (рис. 4.13).

Помимо доступа к строкам как к единому целому, операции также допускают извлечение конкретного столбца, так что, для того чтобы получить имя клиента в БД на рис. 4.13 можно написать команду наподобие get ('1234', 'name').

Столбцы организуются в семейства. Каждый столбец является частью одного семейства столбцов и единица доступа. Данные в конкретном семействе столбцов обычно доступны одновременно.

Итак, данные в столбчатой базе данных структурированы следующим образом:

– Ориентация по строкам: каждая строка – это агрегат (например, клиент с идентификатором 1234), а семейства столбцов содержат фрагменты данных (профиль, история заказов) в этом агрегате.

– Ориентация по столбцам: каждое семейство столбцов определяет тип записи (например, профили клиентов), причем каждой записи соответствуют строки. В таком случае строку можно интерпретировать как объединение записей из всех семейств столбцов.

Последний аспект отражает "столбцовую" природу баз данных типа "семейство столбцов". Базы данных этого типа имеют двумерный характер.

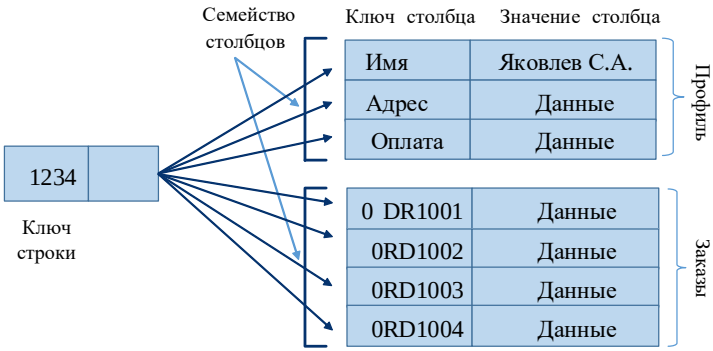


Рис. 4.13. Столбчатая БД

Представлять семейства столбцов в виде таблиц неправильно, т.к. в этой БД можно добавлять любой столбец в любую строку, а строки могут иметь самые разные ключи.

Новые столбцы добавляются в строки при обычном доступе к базе данных.

Определение нового семейства столбцов происходит намного реже и может вызвать остановку работы базы данных.

Поскольку столбцы можно добавлять свободно, список элементов можно легко моделировать, сделав каждый элемент отдельным столбцом.

Строка семейства столбцов – это агрегат. Строки бывают широкими и "худыми".

"Худые"· строки содержат несколько столбцов, причем одни и те же столбцы используются в разных строках. В данном случае семейство столбцов определяет тип записи, каждая строка является записью, а каждый столбец – полем.

Широкая строка содержит много разных столбцов (возможно, тысячи). Широкое семейство столбцов моделирует список, в котором каждый столбец представляет собой элемент в этом списке. Широкие семейства столбцов могут определять определенный порядок следования своих столбцов.

Каждая из пар "ключ-значение" всегда хранится с меткой времени, которая используется для того, чтобы задавать срок действия данных, разрешать конфликты записи, обрабатывать устаревшие данные и выполнять другие функции. Если данные столбца больше не

используются, то это место можно восстановить позднее на этапе уплотнения. Рассмотрим пример:

```
{
Имя: "Полное имя",
Значение: "Яковлев Сергей",
Метка: 12345667890
}
```

Здесь столбец «Имя» содержит ключ строки «Полное имя» и значение «Яковлев Сергей», а также связанную с ними метку времени «12345667890».

Строка – это коллекция столбцов, ассоциированных с ключом; коллекция таких строк образует семейство столбцов. Если семейство столбцов состоит из обычных столбцов, оно называется стандартным.

Каждое семейство столбцов можно сравнить с контейнером строк в таблице реляционной БД, в которой ключ идентифицирует строку, а строка состоит из множества столбцов. Отличие заключается в том, что разные строки не обязаны содержать одинаковые столбцы, причем столбцы могут добавляться в любую строку в любое время и добавлять их в остальные строки не обязательно, например:

```
//Семейство столбцов
{//row
  "Яковлев Сергей": {
       Имя: "Сергей",
       Фамилия: "Яковлев",
       Последний визит: "2019/05/12"
  }
//Строка
 "Татарникова Татьяна": {
      Имя: "Татьяна",
      Фамилия: "Татарникова",
      Город: "Санкт-Петербург"
  }
}
```

В этом примере строки «Яковлев Сергей» и «Татарникова Татьяна» имеют разные столбцы; обе эти строки являются частью семейства столбцов.

Соответствие терминологии, принятой в реляционной модели БД и документной модели БД приведено в табл. 4.4.

Табл. 4.4. Соответствие терминов

Реляционная модель БД	Столбчатая БД
Атрибут	Столбец
Строка	Коллекция столбцов
Идентификатор строки	Ключ строки
Таблица	Агрегат (семейство столбцов)

Рассмотрим примеры, где БД типа «семейство столбцов» подходят лучше всего:

Регистрация событий.

Семейства столбцов, способные хранить любые структуры данных, приспособлены для хранения информации о событиях, например, состояние приложения или ошибки, обнаруженные приложением.

Системы управления информационным: наполнением, блог-платформы.

С помощью семейств столбцов можно хранить записи блогов с ключевыми словами, категориями, ссылками и обратными ссылками в разных столбцах. Комментарии можно хранить либо в той же строке, либо переместить в другое пространство ключей; аналогично пользователей блога и актуальные блоги можно поместить в разные семейства столбцов.

Счетчики.

В веб-приложениях часто возникает необходимость подсчета и классификации посетителей, чтобы вычислить аналитические показатели веб-страницы. После создания семейства столбцов каждому пользователю веб-приложения можно выделить произвольное количество столбцов для посещенных им веб-страниц.

Срок действия.

Иногда возникает необходимость создать демоверсии для пользователей или в определенное время размещать на веб-сайте рекламные объявления. Для этого можно использовать столбцы с ограниченным сроком действия: то есть создавать столбцы, которые автоматически удаляются по истечении определенного периода времени. Это время называется TTL (Time To Live – время существования) и задается в секундах. По истечении периода TTL столбец удаляется; если столбец больше не существует, запрос можно отменить, а рекламное объявление снять.

Существуют проблемы, которые семейство столбцов решает неэффективно. Например, это относится к системам, использующим для выполнения операций чтения и записи транзакции. Если

необходимо, чтобы база данных агрегировала данные с помощью запросов (например, SUM или AVG), это следует делать на клиентской стороне с помощью данных, извлеченных из всех строк.

Столбчатые БД не стоит использовать для создания ранних прототипов или первичных промышленных систем: на ранних стадиях еще не известно, как изменится шаблон запросов, а изменяя шаблоны запросов, мы вынуждены изменять проект семейства столбцов.

Изучение особенностей баз данных типа «семейство столбцов» позволяет сделать следующие выводы:

1. Коллекция столбцов – это строка (агрегат).
2. Коллекция строк образует семейство столбцов.
3. Единицей хранения является столбец.
4. Столбцы могут добавляться в любую строку в любое время.
5. БД типа «семейство столбцов» в основном используются для регистрации событий и счета.
6. В столбчатых БД отсутствуют средства реализации итоговых функций и транзакций.

4.6. Графовые БД

Основу графовой модели БД образуют маленькие записи со сложными связями. В такой БД граф – это не диаграмма, а структура данных с узлами, соединенными ребрами.

На рис. 4.14 показана веб-информация с очень маленькими узлами и многочисленными связями между ними. Работая с этой структурой, мы можем задавать вопросы вроде "найти книгу в категории "Базы данных", написанную кем-то, чей друг мне нравится.

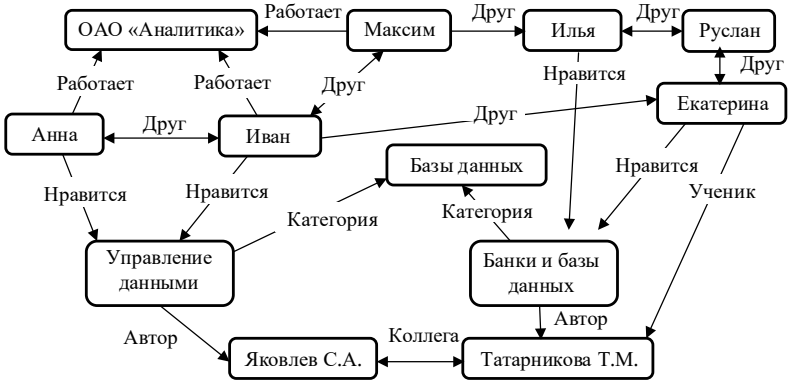


Рис. 4.14. Пример графовой БД

Графовые базы данных специально предназначены для хранения такой информации, но в более крупном масштабе, чем можно показать на диаграмме. Они идеально подходят для хранения любых данных, связанных со сложными отношениями, например, социальных сетей, товарных предпочтений или правил приема на работу.

Фундаментальная модель данных графовых баз очень простая: узлы – это сущности, соединенные ребрами (связями). Узлы имеют свойства. Ребра могут иметь свойства и направление.

Организация графа позволяет один раз записать данные, а затем интерпретировать их разными способами в соответствии со связями.

Как только построен граф узлов и ребер, база данных позволит послать к ней запрос. В этом проявляется важное различие между графовыми и реляционными базами данных.

Несмотря на то что реляционные базы данных могут реализовывать связи с помощью внешних ключей, операции соединения требуют навигации, которая может оказаться затратной. Следовательно, в моделях данных с большим количеством связей производительность упадет.

В графических базах данных обход узлов требует очень небольших затрат. В основном это объясняется тем, что графовые базы данных переносят большую часть работы, связанной с навигацией по связям с момента запроса на момент вставки. Это естественно оправдывает себя в ситуациях, когда производительность запроса важнее скорости вставки. Большую часть времени вы ищете данные, перемещаясь по ребрам сети с запросами вроде таких:

– «найти людей с именем Анна»;

– «найти книгу в категории «Базы данных»

– «найти автора(ов) книги «Управление данными»»;

– «найти, чем интересуется Иван» и т.д.

Однако для организации поиска необходима отправная точка, поэтому некоторые узлы могут быть индексированы атрибутом, например идентификатором. Таким образом, можно начать с поиска идентификатора (например, найти людей с именами Анна и Барбара), а затем начать перемещение по ребрам.

Как следует из рассмотренных свойств, графовые базы предназначены для задач, в которых большую часть времени выполняется поиск – происходит перемещение по связям.

Акцент на связях сильно отличает графовые базы данных от агрегатно- ориентированных. Это отличие проявляется в том, что, во-

первых, графовые базы данных чаще работают на одном сервере, а не распределены по кластерам, а во-вторых, в графовых БД реализован механизм транзакций, обеспечивающий согласованность данных.

Единственное, что связывает их с агрегатно-ориентированными базами данных – это отрицание реляционной модели.

Отношения (связи) между узлами создаются в двух направлениях. Рассмотрим графовую БД, изображенную на рис. 4.15. Например, узел Анна работает в ОАО “Аналитика”, а узел Екатерина нет.

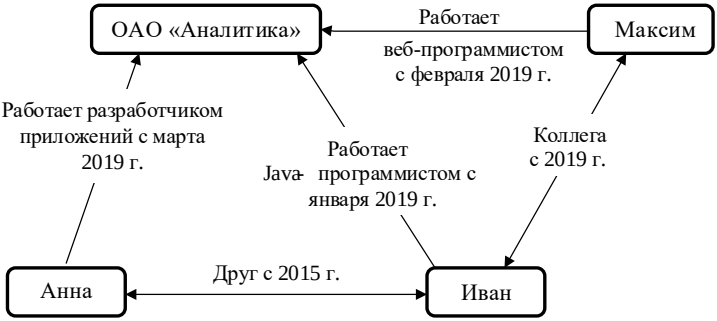


Рис. 4.14. Графовая БД

Направленность отношения позволяет проектировать сложные предметно-ориентированные модели. Известные входящие и исходящие отношения можно обходить в обоих направлениях.

Отношения являются полноправными элементами графовых баз данных. Собственно, ценность графовых баз данных в основном обусловлена отношениями. Отношения имеют не только тип, начальный и конечный узел, но и свои собственные свойства. Используя эти свойства, в отношение можно внести информацию, например такую как: когда узлы стали "друзьями", каково расстояние между узлами и что между ними общего. Эти свойства отношений можно использовать при создании запроса к графу.

Так как мощь графовых баз данных в основном обеспечивается отношениями и их свойствами, требуется серьезная аналитическая и конструкторская работа по моделированию отношений между объектами предметной области. Добавление новых типов узлов в графовую БД выполняется просто. Изменение существующих узлов и связей между ними эквивалентно осуществлению миграции данных, потому что эти изменения необходимо вносить в каждом узле и в каждом отношении между существующими данными.

Графовые БД поддерживают транзакции. Прежде чем изменить какой-нибудь узел или добавить какое-то отношение к существующим узлам, необходимо начать транзакцию, в которую упаковываются операции изменения. Исключение составляет только операция чтения. Чтение можно выполнять без создания транзакций.

Ниже приведен код транзакции, которая применяется к БД, граф которой приведен на рис. 4.14, и создает в ней узел и задает его свойства. Здесь транзакция начата функцией success и закончена функцией finish:

```
Transaction.success = database.beginTx();
try {
      Node друг = GraphDB.createNode();
      друг.setProperty("name", "Илья");
      Максим.createRelationshipTo(Илья, друг);
      Иван.createRelationshipTo(Илья, друг);
      transaction.success();
} finally {
      transaction.finish();
```

На рис. 4.15 приведен граф БД после выполнения транзакции на добавление нового узла и его свойств.

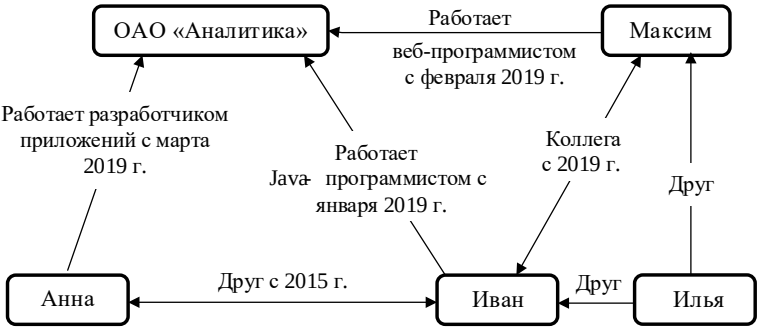


Рис. 4.15. Графовая БД после добавления нового узла

Рассмотрим возможности запросов, которые применяются в графовых БД.

Поскольку свойства узла индексируются и свойства отношений также индексируются, то узел или ребро можно найти по значению

индекса. Индексы узлов задаются либо в момент их добавления в базу данных, либо при их последующем обходе.

Если ищем узел Анна в БД на рис. 4.15, то можем запросить индекс для свойства name, имеющего значение Анна. Имея этот узел, можем выяснить все его отношения, в том числе входящие и исходящие.

При запросе свойств связей можно применять фильтры направлений.

Настоящая мощь графовых БД проявляется в ситуациях, когда необходимо обойти граф на любой глубине и указать начальную точку обхода. Это особенно полезно при попытках найти узлы, связанные с начальной точкой, на более чем одном уровне глубины.

Одно из преимуществ графовых БД – это разнообразие возможностей поиска путей между двумя узлами: можно определить, есть ли несколько путей, найти все пути или кратчайший путь. Эта функциональная возможность используется в социальных сетях для демонстрации отношений между двумя узлами.

Соответствие терминологии, принятой в реляционной модели БД и графовой модели БД приведено в табл. 4.5.

Табл. 4.5. Соответствие терминов

Реляционная модель БД	Столбчатая БД
Атрибут	Узел
Строка	Подграф
Идентификатор строки	Индекс узла
Таблица	Граф

Рассмотрим примеры, где БД типа «граф» подходят лучше всего:

Связанные данные.

Графовые данные можно развернуть и очень эффективно использовать в социальных сетях. Вообще любая предметная область с богатыми взаимными связями подходит для описания с помощью графа. Если в одной и той же базе данных существуют отношения между сущностями из разных предметных областей (например, социальные, географические и коммерческие связи), можно повысить их информативность, предусмотрев возможность обхода графа с пересечением границ предметных областей.

Маршрутизация, диспетчеризация и геолокационные сервисы.

Каждое место назначения или адрес можно представить в виде узла, а все узлы, в которые необходимо выполнить доставку, можно моделировать с помощью графа. Отношения между узлами могут

иметь отношение, описывающее расстояние. Это позволяет обеспечить эффективную доставку товаров. Свойства расстояния и адреса можно использовать в графах, описывающих предпочтения пользователей, так что приложение может выдавать рекомендации о хороших ресторанах или местах для развлечений, расположенных поблизости. Можно также создать узлы для понравившихся точек, например ресторана, и уведомить пользователей, что они находятся поблизости, используя геолокационную службу.

Справочные базы данных.

После того как в системе созданы узлы и отношения, их можно использовать для выдачи рекомендаций типа "ваши друзья также купили этот товар" или "при заказе этого товара обычно также заказывают следующие товары". У таких справочных баз данных есть один интересный побочный эффект – при увеличении объема баз данных количество узлов и отношений, доступных для выдачи рекомендаций, быстро увеличивается.

В некоторых ситуациях графовые базы данных могут оказаться неприемлемыми:

– при необходимости обновлять все или часть сущностей, например, при выработке аналитического решения, в котором свойства всех сущностей могут быть изменены. Изменение свойства во всех узлах является сложной труднореализуемой операцией;

– при выполнении глобальных операций над графом, то есть тех, что затрагивают весь граф.

Изучение особенностей баз данных типа «граф» позволяет сделать следующие выводы:

1. Графовая модель БД – это информационные объекты и отношения между ними, представленные в виде графа.
2. Графовая модель БД не является агрегатно-ориентированной.
3. Единицей хранения в графовых БД является узел (объект) с приписанными ему свойствами
4. Основная операция, выполняемая в графовых БД – поиск информации.
5. Для реализации поиска в графовых БД необходимо выполнить индексацию узлов и отношений между ними.
6. Изменения в структуре графовой модели БД выполняются с помощью транзакций.
7. Возможность поиска путей между двумя узлами.

5. УПРАВЛЕНИЕ ДОСТУПОМ К БАЗАМ ДАННЫХ

База данных – это возможность решения многих задач несколькими пользователями.

Поэтому основная характеристика современных СУБД – наличие многопользовательской технологии работы.

5.1. Архитектурные решения

Рассмотрим основные архитектурные решения, которые реализуют многопользовательский доступ. Известны три архитектуры СУБД, которые приходили на смену друг другу с развитием технологий баз данных. Это централизованная, файл-серверная и клиент-серверная архитектура.

Исторически первой архитектурой считается централизованная, в которой база данных, СУБД и прикладная программа (приложение) располагаются на одном компьютере. Сегодня такая архитектура изжила себя, но с точки зрения того, что все на одном компьютере можно сказать, что на новом витке развития ИТ эта архитектура переродилась во встраиваемую: СУБД тесно связана с приложением и работает на том же компьютере, но не требуя профессионального администрирования и не требуется доступ с многих компьютеров.

На «рабочем столе» практически каждого персонального компьютера или смартфона есть программы, в которых может быть встраиваемая СУБД: почтовые клиенты и мессенджеры, телефонные справочники, заметки и т.п., которые хранят данные владельца гаджета. Централизованная или встраиваемая архитектура приведена на рис. 5.1.

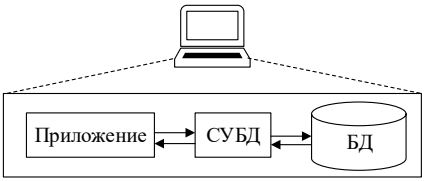


Рис. 5.1. Централизованная (встраиваемая) архитектура

Работа централизованной (встраиваемой) архитектуры построена следующим образом:

– БД в виде набора файлов находится во внешней памяти компьютера. На том же компьютере установлены СУБД и приложение для работы с БД.

– Пользователь запускает приложение, инициируя обращение к БД на выборку/обновление информации.

– Все обращения к БД идут через СУБД, которая инкапсулирует внутри себя все сведения о физической структуре БД.

– СУБД инициирует обращения к данным, обеспечивая выполнение запросов пользователя.

– Результат СУБД возвращает в приложение.

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Как становится очевидно из принципа работы централизованной архитектуры, многопользовательская технология работы обеспечивается:

а) режимом мультипрограммирования, то есть одновременно могут работать процессор и внешние устройства,

б) режимом разделения времени, то есть пользователям по очереди выделялись кванты времени на выполнении их программ.

Таким образом, основной недостаток этой модели – резкое снижение производительности при увеличении числа пользователей.

"Файл-сервер" – это архитектура при которой БД находится на сервере, СУБД на клиентском компьютере, доступ СУБД к данным осуществляется через локальную сеть, то есть это архитектура с сетевым доступом.

Файл-сервер – это машина, которая отвечает на запрос. Как правило это один из компьютеров сети – выделенный сервер (файлы базы данных)

Рабочая станция – это запрашивающая машина (обычно персональный компьютер).

Синхронизация чтений и обновлений осуществляется посредством файловых блокировок.

При запросе файлы с сервера передаются на рабочие станции пользователей, где осуществляется основная часть обработки данных. Сервер выполняет роль хранилища файлов, который в обработке данных не участвует.

Архитектура «файл-сервер» приведена на рис. 5.2.

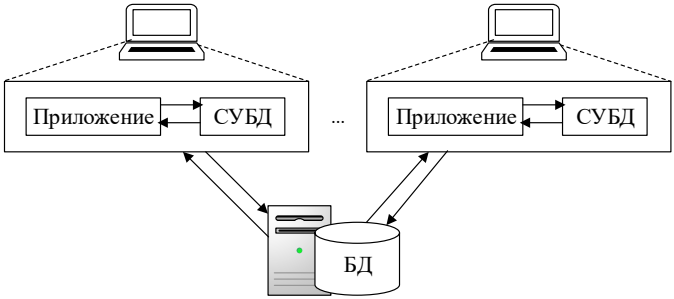


Рис. 5.2. Архитектура «файл-сервер»

Работа архитектуры «файл-сервер» построена следующим образом:

– БД в виде набора файлов находится на файловом сервере.
– Локальная сеть состоит из рабочих станций, с установленными СУБД и приложением для работы с БД.
– На каждой из рабочих станций инициируются обращения к БД.
– Все обращения к БД идут через СУБД.
– Часть файлов БД копируется на рабочую станцию и обрабатывается.
– Результат обработки СУБД возвращает в приложение.
– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Преимущество архитектуры «Файл-сервер»:
– низкая нагрузка на процессор файлового сервера.

Недостатки архитектуры «Файл-сервер»:
– потенциально высокая загрузка локальной сети;
– невозможность централизованного управления;
– обеспечения высокой надежности, доступности и безопасности.

СУБД с архитектурой «Файл-сервер» применялись в локальных приложениях с функциями управления БД.

Технология считается устаревшей, а ее использование в крупных информационных системах недостатком.

Клиент-сервер – это сетевая архитектура, в которой узлы являются либо клиентами, либо серверами.

Клиент – это запрашивающая машина (обычно персональный компьютер).

Сервер – это машина, которая отвечает на запрос.

Клиент и сервер – это физические устройства/программное обеспечение (аппаратный или программный компонент вычислительной системы).

Взаимодействие клиента и сервера осуществляется по определенному протоколу.

Программа-клиент может запрашивать с сервера какие-либо данные, манипулировать данными непосредственно на сервере, запускать на сервере новые процессы и т. п.

Архитектура "клиент-сервер" разделяет функции приложения пользователя, называемого клиентом и сервера. Схема архитектуры «клиент-сервер» приведена на рис. 5.3.

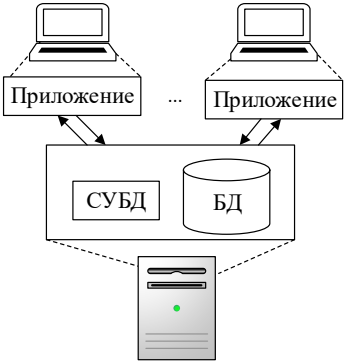


Рис. 5.3. Архитектура «клиент-сервер»

Работа построена следующим образом:

– База данных в виде набора файлов находится на жестком диске специально выделенного компьютера (сервера сети).

– СУБД располагается также на сервере сети.

– Существует локальная сеть, состоящая из клиентских компьютеров, на каждом из которых установлено клиентское приложение для работы с БД.

– На каждом из клиентских компьютеров пользователи имеют возможность запустить приложение (инициирует запрос к БД). По сети от клиента к серверу передается лишь текст запроса SQL.

– СУБД инкапсулирует внутри себя все сведения о физической структуре БД, расположенной на сервере.

– СУБД инициирует обращения к данным, находящимся на сервере, в результате которых на сервере осуществляется вся обработка данных и лишь результат выполнения запроса копируется на клиентский компьютер. Таким образом СУБД возвращает результат в приложение.

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

В функции приложения-клиента входит:

– Посылка запросов серверу.

– Интерпретация результатов запросов, полученных от сервера.

– Представление результатов пользователю в некоторой форме (интерфейс пользователя).

В функции серверной части входит:

– Прием запросов от приложений-клиентов.

– Интерпретация запросов.

– Оптимизация и выполнение запросов к БД.

– Отправка результатов приложению-клиенту.

– Обеспечение системы безопасности и разграничение доступа.

– Управление целостностью БД.

– Реализация стабильности многопользовательского режима работы.

Архитектура «клиент-сервер» обладает следующими достоинствами:

– Существенно уменьшается сетевой трафик.

– Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть), а, следовательно, снижаются требования к аппаратным мощностям клиентских компьютеров.

– Наличие специального программного средства – SQL-сервера – приводит к тому, что существенная часть проектных и программистских задач становится уже решенной.

– Существенно повышается целостность и безопасность БД.

Архитектура «клиент-сервер» обладает следующими недостатками:

– более высокие финансовые затраты на аппаратное и программное обеспечение,

– трудности со своевременным обновлением клиентских приложений на всех компьютерах-клиентах.

Схема многозвенной (трехзвенной) архитектуры «Клиент-сервер» приведена на рис. 5.4.

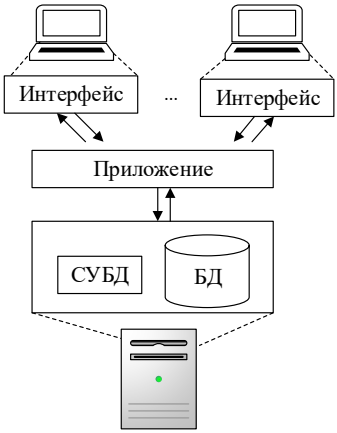


Рис. 5.4. Трехзвенная архитектура

Работа трехзвенной архитектуры построена следующим образом:

– База данных в виде набора файлов находится на сервере.

– СУБД располагается также на сервере сети.

– Существует специально выделенный сервер приложений, на котором располагается программное обеспечение (ПО) делового анализа (бизнес-логика).

– Существует множество клиентских компьютеров, на каждом из которых установлен так называемый "тонкий клиент" – клиентское приложение, реализующее интерфейс пользователя.

– На каждом из клиентских компьютеров может быть запущено приложение – тонкий клиент. Оно инициирует обращение к ПО делового анализа, расположенному на сервере приложений.

– Сервер приложений анализирует требования пользователя и формирует запросы к БД. По сети от сервера приложений к серверу БД передается лишь текст запроса на языке SQL.

– СУБД инкапсулирует внутри себя все сведения о физической структуре БД, расположенной на сервере.

– СУБД инициирует обращения к данным, находящимся на сервере, в результате которых результат выполнения запроса копируется на сервер приложений.

– Сервер приложений возвращает результат в клиентское приложение (пользователю).

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Достоинства трехзвенной архитектуры заключаются в следующем:

– Уменьшается объем передаваемого сетевого трафика между «Клиентом» и «Сервером».

– Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть).

– Наличие SQL-сервера приводит к тому, что существенная часть задач уже решена.

– Существенно повышается целостность и безопасность БД.

– При изменении бизнес-логики более нет необходимости изменять клиентские приложения и обновлять их у всех пользователей.

5.2. Технология «клиент-сервер»

Основным достоинством архитектуры «клиент-сервер» является возможность распределения БД по нескольким серверам и параллельной обработки запросов благодаря возможности обращения к разным серверам.

БД физически распределяется по узлам данных на основе фрагментации и репликации данных.

При наличии схемы реляционной базы данных каждое отношение может фрагментироваться на горизонтальные или вертикальные разделы. **Горизонтальная фрагментация** реализуется при помощи операции селекции, которая направляет каждый кортеж отношения в один из разделов, руководствуясь предикатом фрагментации. При **вертикальной фрагментации** отношение делится на разделы при помощи операции проекции. За счет фрагментации данные приближаются к месту их наиболее интенсивного использования, что потенциально снижает затраты на пересылки; уменьшаются также размеры отношений, участвующих в пользовательских запросах.

При отсутствии схемы реляционной базы данных, то есть БД NoSQL – фрагментация выполняется по агрегатам – горизонтальная фрагментация и по принципу «ведущий-ведомый» для вертикальной фрагментации.

Высокая производительность – одна из важнейших целей, на достижение которой направлены технологии параллельных СУБД

Возможность параллельной обработки запросов реализуется СУБД.

Известны три вида параллелизма, присущие обработке данных.

Межзапросный параллелизм предполагает одновременное выполнение множества запросов, относящихся к разным транзакциям.

Под **внутризапросным параллелизмом** понимается одновременное выполнение сразу нескольких операций, относящихся к одному и тому же запросу.

И внутризапросный, и межзапросный параллелизм реализуется на основе разделения данных, аналогичного горизонтальному фрагментированию.

Наконец, понятие **внутриоперационного параллелизма** означает параллельное выполнение одной операции в виде набора субопераций с применением, в дополнение к фрагментации данных, также и фрагментации функций.

На рис. 5.5 приведены все перечисленные виды параллелизма.

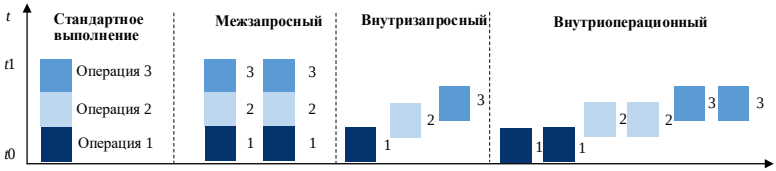
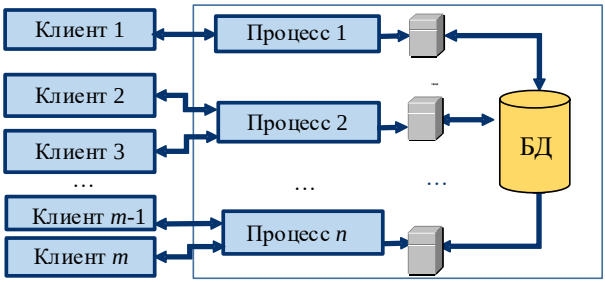


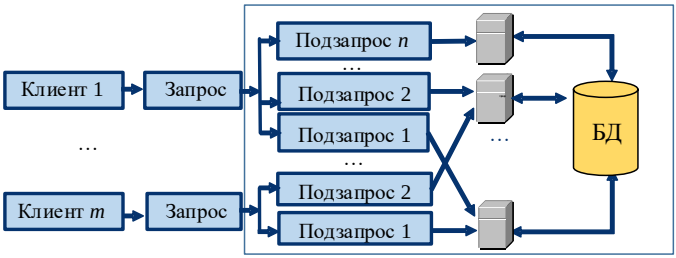
Рис. 5.5. Виды параллелизма обработки данных

Распределенность и параллельная работа на серверах повышает сложность базы данных, поэтому при выборе модели параллелизма взвешиваются все аргументы.

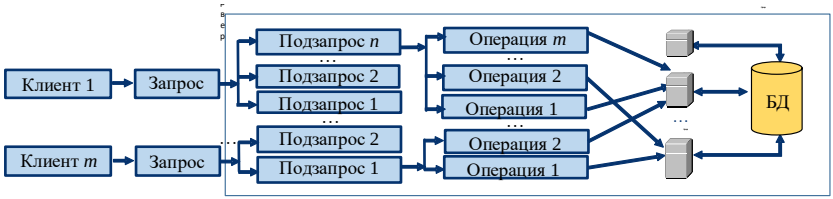
На рис. 5.6 приведены структурные схемы, которые объясняют разницу между видами организации параллельной работы СУБД.



а – межзапросный параллелизм



б – внутризапросный параллелизм



в – внутриоперационный параллелизм

Рис. 5.6. Структурные схемы параллельной работы СУБД

В распределенных базах данных с поддержкой репликации каждому логическому элементу данных соответствует несколько физических копий. В такого рода системах возникает проблема поддержкой согласованности копий. Наиболее известным критерием согласованности является критерий **полной эквивалентности копий**, который требует, чтобы по завершении окна несогласованности все копии логического элемента данных были идентичны.

Протокол управления репликами отвечает за отображение операций над x в операции над физическими копиями x (x_1, x_2 ,..., x_n).

Известный (типичный) протокол управления репликами, следующий критерию полной эквивалентности копий, известен под названием ROWA (**Read-Once/Write-All** – чтение из одной копии, запись во все копии).

Каждая операция записи в логический элемент данных x отображается на множество операций записи во все физические копии x.

Протокол ROWA прост и прямолинеен, но он требует доступности всех копий элемента данных, чтобы завершить

транзакцию. Сбой на одном из узлов приведет к блокированию транзакции, что снижает доступность базы данных.

Поэтому распространение получили протоколы, основанные на механизме голосования на основе кворума, которые рассматривались в БД NoSQL.

Распределенная обработка – это архитектура клиент-сервер, где БД, СУБД, приложения размещены на нескольких серверах. Под технологией «клиент-сервер» также подразумевают облачные вычисления (cloud computing), когда хотят подчеркнуть прозрачность технологии.

Облачные вычисления – это модель обеспечения удобного сетевого доступа по требованию к некоторому общему фонду конфигурируемых вычислительных ресурсов, например к сетям передачи данных, серверам, системам хранения данных, приложениям и сервисам – как вместе, так и по отдельности.

«Облако» строится на основе центров обработки данных (ЦОД). Структура ЦОД состоит из серверов и систем хранения данных, объединенных локальной сетью (рис. 5.7).

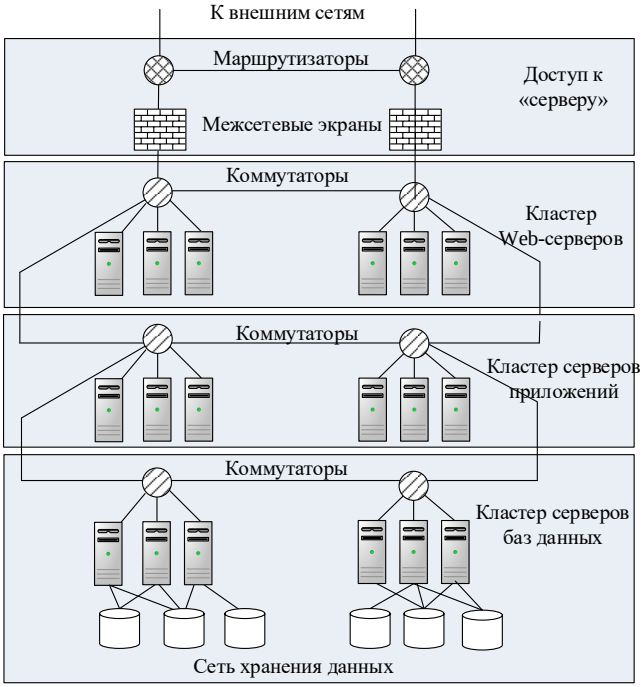


Рис. 5.7. Структура ЦОД

В БД NoSQL распределенность и параллельность решается другими механизмами.

БД NoSQL – это молодое направление, которое возникло в ответ на Большие данные, с которыми традиционные реляционные СУБД плохо справляются.

Эффективная обработка огромных объемов является нетривиальной задачей, для решения которой в 2004 году компания Google разработала модель распределенных вычислений под названием MapReduce (дословно – отображение-свертка).

MapReduce – это модель (шаблон) параллельной обработки данных на кластерах: включает две параллельные операции Map и Reduce (рис. 5.8).

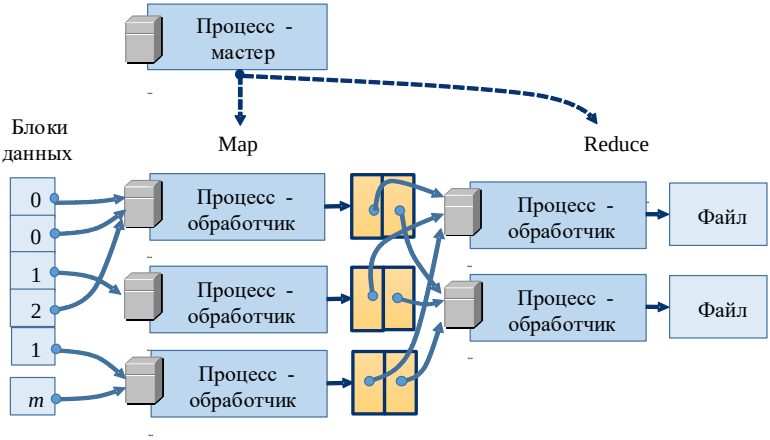


Рис. 5.8. Модель MapReduce

Процедура Map: Главный узел (master) делит задачу на части, распределяя их по остальным машинам.

Reduce: Узел Master получает предварительные результаты, и формирует из них конечный результат.

1) Входные файлы разбиваются на m блоков. Данные хранятся в распределенной файловой системе. Типичный размер блока составляет 16-64МБ.

2) На всех узлах кластера системы запускается n экземпляров программы. Один процесс назначается координатором - мастером, а остальные обработчиками.

3) Мастер-процесс распределяет между процессами m задач типа map и r задач типа reduce: $m+ r = n –1$.

4) Процесс типа map выполняет чтение его блока данных и обрабатывает информацию пользовательской функцией map, которая формирует в памяти узла множество пар (ключ-значение)

5) Периодически данные из памяти узла записываются на локальный диск, который разбит на r областей. Записанные на локальный диск данные передаются мастер-процессу, который передает эту информацию процессам типа reduce. Номер области $R = 0, 1, \ldots, r-1$ выбирается по формуле: R= hash(key) mod r

6) Получив информацию от мастер-процесса, reduce-процесс выполняет удаленный вызов процедур для чтения данных с локального диска. Прочитав все данные reduce-процесс выполняет их сортировку по ключу, чтобы пары с одинаковыми ключами располагались последовательно.

7) Reduce-процесс передает каждый уникальный ключ и список соответствующих ему значений в пользовательскую функцию reduce, которая порождает на основе этих данных новые данные (ключ-значение). Результаты функции reduce дописывается в выходной файл его reduce-области.

Таким образом, технологии распределенных и параллельных БД направлены на:

– Повышение производительности обработки данных (параллельная работа).

– Повышение надежности (благодаря репликации данных, исключаются одиночные точки отказа).

– Решение вопросов, связанных с возрастанием объема баз данных (масштабирование).

5.3. Транзакции

Данные в БД являются разделяемым ресурсом. Многопользовательский доступ к данным подразумевает одновременное выполнение двух и более запросов к одним и тем же объектам данных (таблицам, блокам и т.п.). Для организации одновременного доступа не обязательно наличие многопроцессорной системы. На однопроцессорной ЭВМ запросы выполняются не одновременно, а параллельно. Для каждого запроса выделяется некоторое количество процессорного времени (квант времени), по истечении которого выполнение запроса приостанавливается, он ставится в очередь запросов, а на выполнение запускается следующий по очереди запрос. Таким образом, процессорное время делится между

запросами, и создаётся иллюзия, что запросы выполняются одновременно.

При параллельном доступе к данным запросы на чтение не мешают друг другу. Наоборот, если один запрос считал данные в оперативную память (в буфер данных), то другой запрос не будет тратить время на обращение к диску за этими данными, а получит их из буфера данных. Проблемы возникают в том случае, если доступ подразумевает внесение изменений. Для того чтобы исключить нарушения логической целостности данных при многопользовательском доступе, используется механизм транзакций.

Транзакция – это упорядоченная последовательность операторов обработки данных, которая переводит базу данных из одного согласованного состояния в другое.

Все команды работы с данными выполняются в рамках транзакций. Для каждого сеанса связи с БД в каждый момент времени может существовать единственная транзакция или не быть ни одной транзакции.

Транзакция обладает следующими свойствами:

1. Логическая неделимость (**атомарность**, Atomicity) означает, что выполняются либо все операции (команды), входящие в транзакцию, либо ни одной. Система гарантирует невозможность запоминания части изменений, произведённых транзакцией. До тех пор, пока транзакция не завершена, её можно "откатить", т.е. отменить все сделанные командами транзакции изменения. Успешное выполнение транзакции (фиксация) означает, что все команды транзакции проанализированы, интерпретированы как правильные и безошибочно исполнены.

2. **Согласованность** (Consistency): транзакция начинается на согласованном множестве данных и после её завершения множество данных согласовано. Состояние БД является согласованным, если данные удовлетворяют всем установленным ограничениям целостности и относятся к одному моменту в состоянии предметной области.

3. **Изолированность** (Isolation), т.е. отсутствие влияния транзакций друг на друга.

4. **Устойчивость** (Durability): результаты завершённой транзакции не могут быть потеряны. Возврат БД в предыдущее состояние может быть достигнут только путём запуска компенсирующей транзакции.

Транзакции, удовлетворяющие этим свойствам, называют ACID-транзакциями (по первым буквам названий свойств).

Для управления транзакциями в системах, поддерживающих механизм транзакций и язык SQL, используются следующие операторы:

– **фиксация** транзакции (запоминание изменений): COMMIT [WORK];

– **откат** транзакции (отмена изменений): ROLLBACK [WORK];

– создание точки сохранения: SAVEPOINT <имя_точки_сохранения>;

(Ключевое слово WORK необязательно). Для фиксации или отката транзакции система создаёт неявные точки фиксации и отката (рис. 5.9).

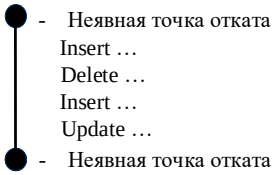


Рис. 5.9. Неявные точки фиксации и отката транзакции

По команде rollback система откатит транзакцию на начало (на неявную точку отката), а по команде commit – зафиксирует всё до неявной точки фиксации, которая соответствует последней завершённой команде в транзак-ции. Если в транзакции из нескольких команд во время выполнения очередной команды возникнет ошибка, то система откатит только эту ошибочную команду, т.е. отменит её результаты и сохранит прежнюю неявную точку фиксации.

Для обеспечения целостности транзакции СУБД может откладывать запись изменений в БД до момента успешного выполнения всех операций, входящих в транзакцию, и получения команды подтверждения транзакции (commit). Но чаще используется другой подход: система записывает изменения в БД, не дожидаясь завершения транзакции, а старые значения данных сохраняет на время выполнения транзакции в сегментах отката.

Сегмент отката (rollback segment, RBS) – это специальная область памяти на диске, в которую записывается информация обо всех текущих (незавершённых) изменениях. Обычно записывается "старое" и "новое" содержимое изменённых записей, чтобы можно было восстановить прежнее состояние БД при откате транзакции (по команде rollback) или при откате текущей операции (в случае возникновения ошибки). Данные в RBS хранятся до тех пор, пока

транзакция, изменяющая эти данные, не будет завершена. Потом они могут быть перезаписаны данными более поздних транзакций.

Команда savepoint запоминает промежуточную "текущую копию" состояния базы данных для того, чтобы при необходимости можно было вернуться к состоянию БД в точке сохранения: откатить работу от текущего момента до точки сохранения (rollback to <имя_точки>) или зафиксировать работу от начала транзакции до точки сохранения (commit to <имя_точки>). На одну транзакцию может быть несколько точек сохранения (ограничение на их количество зависит от СУБД).

Для сохранения сведений о транзакциях СУБД ведёт журнал транзакций. **Журнал транзакций** – это часть БД, в которую поступают данные обо всех изменениях всех объектов БД. Журнал недоступен пользователям СУБД и поддерживается особо тщательно (иногда ведутся две копии журнала, хранимые на разных физических носителях). Форма записи в журнал изменений зависит от СУБД. Но обычно там фиксируется следующее:

– номер транзакции (номера присваиваются автоматически по возрастанию);

– состояние транзакции (завершена фиксацией или откатом, не завершена, находится в состоянии ожидания);

– точки сохранения (явные и неявные);

– команды, составляющие транзакцию, и проч.

Начало транзакции соответствует появлению первого исполняемого SQL-оператора. При этом в журнале появляется запись об этой транзакции.

По стандарту ANSI/ISO транзакция завершается при наступлении одного из следующих событий:

– Поступила команда **commit** (результаты транзакции фиксируются).

– Поступила команда **rollback** (результаты транзакции откатываются).

– Успешно завершена программа (**exit, quit**), в рамках которой выполнялась транзакция. В этом случае транзакция фиксируется автоматически.

– Программа, выполняющая транзакцию, завершена аварийно (**abort**). При этом транзакция автоматически откатывается.

Примечания:

1. Возможна работа в режиме **AUTOCOMMIT**, когда каждая команда воспринимается системой как транзакция. В этом режиме пользователи меньше задерживают друг друга, требуется меньше

памяти для сегмента отката, зато результаты ошибочно выполненной операции нельзя отменить командой **rollback**.

2. В некоторых СУБД реализованы расширенные модели транзакций, в которых существуют дополнительные ситуации фиксации транзакций. Например, в СУБД Oracle команды DDL выполняются в режиме **AUTOCOMMIT**, т.е. не могут быть откачены.

Все изменения данных выполняются в оперативной памяти в буфере данных, затем фиксируются в журнале транзакций и в сегменте отката, и периодически (при выполнении контрольной точки) переписываются на диск. Процесс формирования **контрольной точки** (КТ) заключается в синхронизации данных, находящихся на диске (т.е. во вторичной памяти) с теми данными, которые находятся в ОП: все модифицированные данные из ОП переписываются во вторичную память. В разных системах процесс формирования контрольной точки запускается по-разному. Например, в СУБД Oracle КТ формируется:

- при поступлении команды commit,
- при переполнении буфера данных,
- в момент заполнения очередного файла журнала транзакций,
- через три секунды со времени последней записи на диск.

Внесение изменений в журнал транзакций всегда носит опережающий характер по отношению к записи изменений в основную часть БД (протокол WAL – Write Ahead Log). Эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала транзакций раньше, чем изменённый объект попадёт во внешнюю память основной части БД. Если СУБД корректно соблюдает протокол WAL, то с помощью журнала транзакций можно решить все проблемы восстановления БД после сбоя, если сбой препятствуют дальнейшему функционированию системы (например, после сбоя приложения или фонового процесса СУБД).

Таким образом, при использовании протокола WAL измененные данные почти сразу попадают в базу данных, ещё по поступления команды commit. Поэтому фиксация транзакции чаще всего заключается в следующем:

1. Изменения, внесённые транзакцией, помечаются как постоянные.
2. Уничтожаются все точки сохранения для данной транзакции.
3. Если выполнение транзакций осуществляется с помощью блокировок, то освобождаются объекты, заблокированные транзакцией (см. раздел 5.5).

4. В журнале транзакций транзакция помечается как завершенная, уничтожаются системные записи о транзакции в оперативной памяти.

А при откате транзакции вместо п.1 обычно выполняется считывание из сегмента отката прежних значений данных и переписывание их обратно в БД (остальные пункты сохранятся без изменений). Поэтому откат транзакции практически всегда занимает больше времени, чем фиксация.

5.4. Взаимовлияние транзакций

Транзакции в многопользовательской БД должны быть изолированы друг от друга, т.е. в идеале каждая из них должна выполняться так, как будто выполняется только она одна. В реальности транзакции выполняются одновременно и могут влиять на результаты друг друга, если они обращаются к одному и тому же набору данных и хотя бы одна из транзакций изменяет данные.

В общем случае взаимовлияние транзакций может проявляться в виде:

– потери изменений;
– чернового чтения;
– неповторяемого чтения;
– фантомов

Потеря изменений могла бы произойти при одновременном обновлении двумя и более транзакциями одного и того же набора данных. Транзакция, закончившаяся последней, перезаписала бы результаты изменений, внесённых предыдущими транзакциями, и они были бы потеряны.

Представим, что одновременно начали выполняться две транзакции:

```
транзакция 1 – UPDATE СОТРУДНИКИ
SET Оклад = 39200
WHERE    Номер = 1123;
транзакция 2 – UPDATE СОТРУДНИКИ
SET Должность = "старший экономист"
WHERE    Номер = 1123;
```

Обе транзакции считали одну и ту же запись (1123, "Рудин В.П.", "экономист", 28300) и внесли каждая свои изменения: в бухгалтерии

изменили оклад (транзакция 1), в отделе кадров – должность (транзакция 2). Результаты транзакции 1 будут потеряны (рис. 5.10).

Отношение «Сотрудник»

Номер	ФИО	Должность	Оклад	
1 123	Рудин В.П.	экономист	28300	
1 123	Рудин В.П.	экономист	**39200**	Транзакция 1
1 123	Рудин В.П.	**старший экономист**	28300	Транзакция 2

Рис. 5.10. Недопустимое взаимовлияние транзакций: потеря изменений

СУБД не допускает такого взаимовлияния транзакций, при котором возможна потеря изменений!

Ситуация **чернового чтения** возникает, когда транзакция считывает изменения, вносимые другой (незавершенной) транзакцией. Если эта вторая транзакция не будет зафиксирована, то данные, полученные в результате чернового чтения, будут некорректными. Транзакции, осуществляющие черновое чтение, могут использоваться только при невысоких требованиях к согласованности данных: например, если транзакция считает статистические показатели, когда отклонения отдельных значений данных слабо влияют на общий результат.

При повторяемом чтении один и тот же запрос, повторно выполняемый одной транзакцией, возвращает один и тот же набор данных (т.е. игнорирует изменения, вносимые другими завершёнными и незавершёнными транзакциями). **Неповторяемое чтение** является противоположностью повторяемого, т.е. транзакция "видит" изменения, внесённые другими (завершёнными!) транзакциями. Следствием этого может быть несогласован-ность результатов запроса, когда часть данных запроса соответствует состоянию БД до внесения изменений, а часть – состоянию БД после внесения и фиксации изменений.

Фантомы – это особый тип неповторяемого чтения. Возникновение фантомов может происходить в ситуации, когда одна и та же транзакция сначала производит обновление набора данных, а затем считывание этого же набора. Если считывание данных начинается раньше, чем закончится их обновление, то в результате чтения можно получить несогласованный (не обновлённый или частично обновлённый) набор данных. При последующих запросах это явление пропадает, т.к. на самом деле запрошенные данные после завершения обновления будут согласованными в соответствие со свойствами транзакции.

Для разграничения двух пишущих транзакций и предотвращения потери изменений СУБД используют механизмы блокировок или временных отметок, а для разграничения пишущей и читающих транзакций – специальные правила поведения транзакций, которые называются уровнями изоляции транзакций.

Уровни изоляции транзакций

Стандарт ANSI/ISO для SQL устанавливает различные уровни изоляции для операций, выполняемых над БД, которые работают в многопользовательском режиме. **Уровень изоляции** определяет, может ли читающая транзакция *считывать* ("видеть") результаты работы других одновременно выполняемых завершённых и/или незавершённых пишущих транзакций (табл. 5.1). Использование уровней изоляции обеспечивает предсказуемость работы приложений.

Таблица 5.1. Уровни изоляции по стандарту ANSI / ISO

Уровень изоляции	Черновое чтение	Неповторяемое чтение	Фантомы
Read Uncommited – чтение незавершённых транзакций	да	да	да
Read Commited – чтение завершённых транзакций	нет	да	да
Repeatable Read – повторяемое чтение	нет	нет	да
Serializable – последовательное чтение	нет	нет	нет

По умолчанию в СУБД обычно установлен уровень Read Commited.

Уровень изоляции позволяет транзакциям в большей или меньшей степени влиять друг на друга: при повышении уровня изоляции повышается согласованность данных, но снижается степень параллельности работы и, следовательно, производительность системы.

5.4. Блокировки транзакций

Блокировка – это временное ограничение доступа к данным, участвующим в транзакции, со стороны других транзакций.

Блокировка относится к пессимистическим алгоритмам, т.к. предполагается, что существует высокая вероятность одновременного

обращения нескольких пишущих транзакций к одним и тем же данным. Различают следующие типы блокировок:

– по степени доступности данных: разделяемые и исключающие;

– по множеству блокируемых данных: строчные, страничные, табличные;

– по способу установки: автоматические и явные.

Строчные, страничные и табличные блокировки накладываются соответственно на строку таблицы, страницу (блок) памяти и на всю таблицу целиком. Табличная блокировка приводит к неоправданным задержкам исполнения запросов и сводит на нет параллельность работы. Другие виды блокировки увеличивают параллелизм работы, но требуют накладных расходов на поддержание блокировок: наложение и снятие блокировок требует времени, а для хранения информации о наложенной блокировке нужна дополнительная память (для каждой записи или блока данных).

Разделяемая блокировка, установленная на определённый ресурс, предоставляет транзакциям право коллективного доступа к этому ресурсу. Обычно этот вид блокировок используется для того, чтобы запретить другим транзакциям производить необратимые изменения. Например, если на таблицу целиком наложена разделяемая блокировка, то ни одна транзакция не сможет удалить эту таблицу или изменить её структуру до тех пор, пока эта блокировка не будет снята. (При выполнении запросов на чтение обычно накладывается разделяемая блокировка на таблицу.)

Исключающая блокировка предоставляет право на монопольный доступ к ресурсу. Исключающая (монопольная) блокировка таблицы накладывается, например, в случае выполнения операции ALTER TABLE, т.е. изменения структуры таблицы. На отдельные записи (блоки) монопольная блокировка накладывается тогда, когда эти записи (блоки) подвергаются модификации.

Блокировка может быть **автоматической** и **явной**. Если запускается новая транзакция, СУБД сначала проверяет, не заблокирована ли другой транзакцией строка, требуемая этой транзакции: если нет, то строка автоматически блокируется и выполняется операция над данными; если строка заблокирована, транзакция ожидает снятия блокировки. Явная блокировка, накладываемая командой LOCK TABLE языка SQL, обычно используется тогда, когда транзакция затрагивает существенную часть отношения. Это позволяет не тратить время на построчную блокировку таблицы. Кроме того, при большом количестве построчных блокировок транзакция может не завершиться (из-за

возникновения взаимных блокировок, например), и тогда все сделанные изменения придётся откатить, что снизит производительность системы.

Явную блокировку также можно наложить с помощью ключевых слов *for update*, например:

```
SELECT *
FROM <имя_таблицы>
WHERE <условие>
for update;
```

При этом блокировка будет накладываться на те записи, которые удовлетворяют <условию>.

И явные, и неявные блокировки снимаются при завершении транзакции.

Блокировки могут стать причиной бесконечного ожидания и тупиковых ситуаций. **Бесконечное ожидание** возможно в том случае, если не соблюдается очерёдность обслуживания транзакций и транзакция, поступившая раньше других, всё время отодвигается в конец очереди. Решение этой проблемы основывается на выполнении правила FIFO (first input – first output): "первый пришел – первый ушел".

Тупиковые ситуации (deadlocks) возникают при взаимных блокировках транзакций, которые выполняются на пересекающихся множествах данных (рис. 5.11). Здесь приведён пример взаимной блокировки трех транзакций T_i на отношениях R_j. Транзакция T_1 заблокировала данные B_1 в отношении R_1 и ждёт освобождения данных B_2 в отношении R_2, которые заблокированы транзакцией T_2, ожидающей освобождения данных B_3 в отношении R_3, заблокированных транзакцией T_3, которая не может продолжить выполнение из-за транзакции T_1. Если не предпринимать никаких дополнительных действий, то эти транзакции никогда не завершатся, т.к. они вечно будут ждать друг друга.

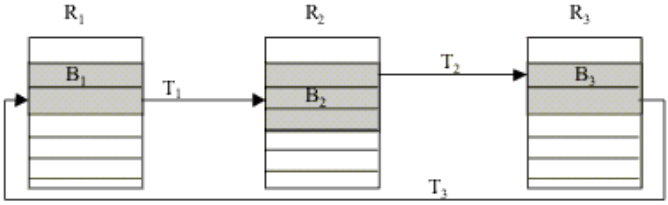


Рис. 5.11. Взаимная блокировка трех транзакций

Существует много стратегий разрешения проблемы взаимной блокировки, в частности:

1. Транзакция запрашивает сразу все требуемые блокировки. Такой метод снижает степень параллелизма в работе системы. Также он не может применяться в тех случаях, когда заранее неизвестно, какие данные потребуются, например, если выборка данных из одной таблицы осуществляется на основании данных из другой таблицы, которые выбираются в том же запросе.

2. СУБД отслеживает возникающие тупики и отменяет одну из транзакций с последующим рестартом через случайный промежуток времени. Этот метод требует дополнительных накладных расходов.

3.Вводится **таймаут** (time-out) – максимальное время, в течение которого транзакция может находиться в состоянии ожидания. Если транзакция находится в состоянии ожидания дольше таймаута, считается, что она находится в состоянии тупика, и СУБД инициирует её откат с последующим рестартом через случайный промежуток времени.

Использование временных отметок относится к оптимистическим алгоритмам разграничения транзакций. Для их эффективного функционирования необходимо, чтобы вероятность одновременного обращения нескольких пишущих транзакций к одним и тем же данным была невелика.

Временная отметка – это уникальный идентификатор, который СУБД создаёт для обозначения относительного момента запуска транзакции. Временная отметка может быть создана с помощью системных часов или путём присвоения каждой следующей транзакции очередного номера (SCN – system change number). Каждая транзакция T_i имеет временную отметку t_i, и каждый элемент данных в БД (запись или блок) имеет две отметки: $t_{read}(x)$ – временная отметка транзакции, которая последней считала элемент x, и $t_{write}(x)$ – временная отметка транзакции, которая последней записала элемент x.

При выполнении транзакции T_i система сравнивает отметку t_i и отметки $t_{read}(x)$ и $t_{write}(x)$ элемента $\boldsymbol{x}$ для обнаружения конфликтов:

1. Для читающей транзакции T_i : если $t_i < t_{write}(x)$, то элемент данных $\boldsymbol{x}$ перезаписан более поздней транзакцией, и его значение может оказаться несогласованным с теми данными, которые эта транзакция уже успела прочитать.

2. Для пишущей транзакции:

– если $t_i < t_{read}(x)$, то элемент данных x считан более поздней транзакцией. Если транзакция T изменит значение элемента x, то в другой транзакции может возникнуть ошибка.

– если $t_i < t_{write}(x)$, то элемент x перезаписан более поздней транзакцией, и транзакция T пытается поместить в БД устаревшее значение элемента $\boldsymbol{x}$.

Во всех случаях обнаружения конфликта система перезапускает текущую транзакцию T_i с более поздней временной отметкой. Если конфликта нет, то транзакция выполняется. Очевидно следующее: если разные транзакции часто обращаются к одним и тем же данным одновременно, то транзакции часто будут перезапускаться, и эффективность такого механизма будет невелика.

Для увеличения эффективности выполнения запросов некоторые СУБД используют алгоритм многовариантности. Этот алгоритм позволяет обеспечивать согласованность данных при чтении, не блокируя эти данные.

Согласованность данных для операции чтения заключается в том, что все значения данных должны относиться к тому моменту, когда начиналась эта операция. Для этого можно предварительно запретить другим транзакциям изменять эти данные до окончания операции чтения, но это снижает степень параллельности работы системы.

При использовании алгоритма многовариантности каждый блок данных хранит номер последней транзакции, которая модифицировала данные, хранящиеся в этом блоке (SCN – system change number). И каждая транзакция имеет свой SCN. При чтении данных СУБД сравнивает номер транзакции и номер считываемого блока данных:

– если блок данных не модифицировался с момента начала чтения, то данные считываются из этого блока;

– если данные успели измениться, то система обратится к сегменту отката и считает оттуда значения данных, относящиеся к моменту начала чтения.

Недостатком этого метода является возможность возникновения ошибки при чтении данных, если старые значения данных в сегменте отката будут перезаписаны. При этом будет выдано сообщение об ошибке и операцию чтения придётся перезапускать вручную. Для устранения подобных проблем можно увеличить размер сегмента отката или разбить одну большую операцию чтения на несколько (но при этом согласованность данных обеспечиваться не будет).

Использование блокировок гарантирует сериальность планов выполнения смеси транзакций за счет общего замедления работы - конфликтующие транзакции ожидают, когда транзакция, первой заблокировавшая некоторый объект, не освободит его. Без блокировок не обойтись, если все транзакции *изменяют* данные. Но если в смеси транзакций присутствуют как транзакции, изменяющие данные, так и

только читающие данные, можно применить альтернативный механизм обеспечения сериальности, свободный от недостатков метода блокировок. Этот метод состоит в том, что транзакциям, читающим данные, предоставляется как бы "своя" версия данных, имевшаяся в момент начала читающей транзакции. При этом транзакция не накладывает блокировок на читаемые данные, и, поэтому, не блокирует другие транзакции, изменяющие данные. Такой механизм называется ***механизм выделения версий*** и заключается в использовании журнала транзакций для генерации разных версий данных.

Журнал транзакций предназначен для выполнения операции отката при неуспешном выполнении транзакции или для восстановления данных после сбоя системы. Журнал транзакций содержит старые копии данных, измененных транзакциями.

Кратко суть метода состоит в следующем:

– Для каждой транзакции (или запроса) запоминается текущий системный номер SCN. Чем позже начата транзакция, тем больше ее SCN.

– При записи страниц данных на диск фиксируется SCN транзакции, производящей эту запись. Этот SCN становится текущим системным номером страницы данных.

– Транзакции, только читающие данные не блокируют ничего в базе данных.

– Если транзакция А читает страницу данных, то SCN транзакции А сравнивается с SCN читаемой страницы данных.

– Если SCN страницы данных меньше или равен SCN транзакции А, то транзакция А читает эту страницу.

– Если SCN страницы данных больше SCN транзакции А, то это означает, что некоторая транзакция В, начавшаяся позже транзакции А, успела изменить или сейчас изменяет данные страницы. В этом случае транзакция А просматривает журнал транзакция назад в поиске первой записи об изменении нужной страницы данных с SCN меньшим, чем SCN транзакции А. Найдя такую запись, транзакция А использует старый вариант данных страницы.

6. ЗАЩИТА БАЗ ДАННЫХ

Защита данных - это организационные, программные и технические методы и средства, направленные на удовлетворение ограничений, установленных для типов данных и экземпляров типов данных в БД.

Защита данных включает предупреждение случайного или несанкционированного доступа к данным, их изменения или разрушения со стороны пользователей или при сбоях аппаратуры. Реализация защиты включает:

– контроль достоверности данных с помощью ограничений целостности;

– обеспечение безопасности данных (физической целостности данных);

– обеспечение секретности данных.

6.1. Обеспечение целостности данных

Обеспечение целостности данных касается защиты от внесения непреднамеренных ошибок и предотвращения последних. Оно достигается за счёт проверки ограничений целостности – условий, которым должны удовлетворять значения данных.

Рассмотрим различные типы ограничений целостности в языке SQL:

1. Уникальность значения первичного ключа (PRIMARY KEY).

2. Уникальность ключевого поля или комбинации значений ключевых полей:

UNIQUE(A),

где A – один или несколько атрибутов, указанных через запятую.
(1,2 – явные структурные ограничения целостности).

3. Обязательность/необязательность значения (NOT NULL/NULL).

4. Задание диапазона значений атрибута Field:

CHECK(field BETWEEN min_value AND max_value)

5. Задание взаимоотношений между значениями атрибутов Field1 и Field2:

CHECK (field1 @ field2),

где @ – оператор отношения (например, знак ">").

6. Задание списка возможных значений (констант) для атрибута Field:

CHECK (field IN (value1, value2,…, valueN)).

7. Определение формата атрибута Field (даты, числа и др.).
Например:
CHECK (field LIKE '_ _ _-_ _-_ _') -- формат телефонного номера

8. Определение домена атрибута на основе значений другого атрибута: Определение формата атрибута;
Field (даты, числа и др.).

Например: множество значений некоторого атрибута отношения является подмножеством значений другого атрибута этого или другого отношения (внешний ключ, FOREIGN KEY)./li>
(3.-8. – явные ограничения целостности на значения данных.)

9. Ограничения на обновление данных (например, каждое следующее значение атрибута должно быть больше предыдущего). В SQL напрямую не реализуется, требует использования специальных возможностей СУБД (триггеров).

10. Ограничения на параллельное выполнение операций (механизм транзакций) и проверка ограничений целостности после окончания внесения взаимосвязанных изменений.

Реализация ограничений целостности возлагается на СУБД или выполняется с помощью специальных программных модулей.

СУБД проводит проверку выполнения ограничений целостности для команд DDL до выполнения команды, а для команд DML либо сразу после выполнения команды, либо после выполнения всей транзакции. (По стандарту ISO этим можно управлять; по умолчанию проверка проводится после каждой операции DML).

6.2. Защита от сбоев БД

Под функцией безопасности (или физической защиты) данных подразумевается предотвращение разрушения или искажения данных в результате программного или аппаратного сбоя. Обеспечение безопасности является внутренней задачей СУБД, поскольку связано с её нормальным функционированием, и решается на уровне СУБД. Цель **восстановления базы данных** после сбоя – обеспечить, чтобы результаты всех подтверждённых транзакций были отражены в восстановленной БД, и вернуться к нормальному продолжению работы как можно быстрее, изолируя пользователей от проблем, вызванных сбоем.

Рассмотрим виды сбоев.

В СУБД предусмотрены специальные механизмы, призванные нивелировать последствия сбоев в работе базы данных. Рассмотрим наиболее типичные сбои и способы защиты от них:

Сбой предложения.

Сбой происходит при логической ошибке предложения во время его обработки (например, предложение нарушает ограничение целостности таблицы). Когда возникает сбой предложения, СУБД автоматически откатывает результаты этого предложения, генерирует сообщение об ошибке и возвращает управление пользователю (приложению пользователя).

Сбой пользовательского процесса.

Это ошибка в процессе (приложении), работающем с БД, например, аварийное разъединение или прекращение процесса. Сбившийся процесс пользователя не может продолжать работу, тогда как СУБД и процессы других пользователей могут. Система автоматически откатывает неподтверждённые транзакции сбившегося пользовательского процесса и освобождает все ресурсы, занятые этим процессом.

Сбой процесса сервера.

Такой сбой вызван проблемой, препятствующей продолжению работы сервера. Это может быть аппаратная проблема, такая как отказ питания, или программная проблема, такая как сбой операционной системы. Восстановление после сбоя процесса сервера может потребовать перезагрузки БД, при этом автоматически происходит откат всех незавершённых транзакций.

Сбой носителя (*диска*).

Эта ошибка может возникнуть при попытке записи или чтения файла, необходимого для работы базы данных (файла БД, файла журнала транзакций и проч.). Типичным примером является отказ дисковой головки, который приводит к потере всех файлов на данном устройстве. В этой ситуации сервер БД не может продолжать работу, и для восстановления базы данных требуется участие человека (обычно, администратора базы данных, АБД).

Ошибка пользователя.

Например, пользователь может случайно удалить нужные записи или таблицы. Ошибки пользователей могут потребовать участия человека (АБД) для восстановления базы данных в состояние на момент возникновения ошибки.

Таким образом, после некоторых сбоев система может восстановить БД автоматически, а ошибка пользователя или сбой диска требуют участия в восстановлении человека.

Средства физической защиты данных.

В качестве средств физической защиты данных чаще всего применяются резервное копирование и журналы транзакций.

Резервное копирование означает периодическое сохранение файлов БД на внешнем запоминающем устройстве. Оно выполняется тогда, когда состояние файлов БД является непротиворечивым. Резервная копия (РК) не должна создаваться на том же диске, на котором находится сама БД, т.к. при аварии диска базу невозможно будет восстановить. В случае сбоя (или аварии диска) БД восстанавливается на основе последней копии.

Полная резервная копия включает всю базу данных (все файлы БД, в том числе вспомогательные, состав которых зависит от СУБД). *Частичная резервная копия* включает часть БД, определённую пользователем. Резервная копия может быть *инкрементной*: она состоит только из тех блоков (страниц памяти), которые изменились со времени последнего резервного копирования. Создание инкрементной копии происходит быстрее, чем полной, но оно возможно только после создания полной резервной копии.

Создание частичной и инкрементной РК выполняется средствами СУБД, а создание полной РК – средствами СУБД или ОС (например, с помощью команды copy). В резервную копию, созданную средствами СУБД, обычно включаются только те блоки памяти, которые реально содержат данные.

Периодичность резервного копирования определяется администратором системы и зависит от многих факторов: объём БД, интенсивность запросов к БД, интенсивность обновления данных и др. Как правило, технология проведения резервного копирования такова:

раз в неделю (день, месяц) осуществляется полное копирование;

раз в день (час, неделю) – частичное или инкрементное копирование.

Все изменения, произведённые в данных после последнего резервного копирования, утрачиваются; но при наличии *архива журнала транзакций* их можно выполнить ещё раз, обеспечив полное восстановление БД на момент возникновения сбоя. Дело в том, что журнал транзакций содержит сведения только о текущих транзакциях. После завершения транзакции информация о ней может быть перезаписана. Для того чтобы в случае сбоя обеспечить возможность полного восстановления БД, необходимо вести архив журнала

транзакций, т.е. сохранять копии файлов журнала транзакций вместе с резервной копией базы данных.

Восстановление базы данных.

В том случае, если нельзя восстановить БД после сбоя автоматически, восстановление БД выполняется в два этапа:

перенос на рабочий диск резервной копии базы данных (или той её части, которая была повреждена);

перезапуск сервера БД с повторным проведением всех транзакций, зафиксированных после создания резервной копии и до момента возникновения сбоя.

Если в системе есть архив транзакций, то повторное проведение транзакций может проходить автоматически или под управлением пользователя.

Если произошёл сбой процесса сервера, то требуется перезагрузка сервера для восстановления БД. При перезагрузке СУБД может по содержимому системных файлов узнать, что произошёл сбой, и выполнить восстановление автоматически (если это возможно). Восстановление БД в этой ситуации означает приведение всех данных в БД в согласованное состояние, т.е. откат незавершённых транзакций и проверку того, что все изменения, внесённые завершёнными транзакциями, попали на диск.

Для оптимизации регистрации изменений некоторые СУБД могут записывать в журнал информацию о незавершённых транзакциях, предвидя их завершение. Более того, не дожидаясь подтверждения транзакции, СУБД переписывает на диск модифицированные блоки (при формировании контрольной точки). Поэтому в каждый момент времени в журнале транзакций и в БД может находиться небольшое число записей, модифицированных незавершёнными транзакциями. Эти записи помечаются соответствующим образом. С другой стороны, т.к. изменения сначала попадают в журнал транзакций и только потом в файл базы данных, в любой момент времени БД может не содержать блоков данных, модифицированных подтверждёнными транзакциями. Поэтому в результате сбоя могут возникнуть две потенциальные ситуации:

Блоки, содержащие подтверждённые модификации, не были записаны в файлы данных, так что эти изменения отражены лишь в журнале тран-закций. Следовательно, журнал транзакций содержит подтверждённые данные, которые должны быть переписаны в файлы данных.

Журнал транзакций и блоки данных содержат изменения, которые не были подтверждены. Изменения, внесенные

неподтверждёнными транзакциями, во время восстановления БД должны быть удалены из файлов данных.

Для того чтобы разрешить эти ситуации, СУБД автоматически выполняет два этапа при восстановлении после сбоев: прокрутку вперед и прокрутку назад.

Прокрутка вперед заключается в применении к файлам данных всех изменений, зарегистрированных в журнале транзакций. После прокрутки вперед файлы данных содержат все как подтверждённые, так и неподтверждённые изменения, которые были зарегистрированы в журнале транзакций.

Прокрутка назад заключается в отмене всех изменений, которые не были подтверждены. Для этого используются журнал транзакций и сегменты отката, информация из которых позволяет определить и отменить те транзакции, которые не были подтверждены, хотя и попали на диск в файлы БД.

После выполнения этих этапов восстановления БД находится в согласованном состоянии и с ней можно работать.

6.3. Конфиденциальность данных

Функция конфиденциальности – это защита от несанкционированного доступа к данным легальных пользователей или посторонних лиц.

Для этого вся информация делится на общедоступные данные и конфиденци-альные, доступ к которым разрешен только для отдельных групп лиц. Решение этого вопроса относится к компетенции юридических органов или администрации предприятия, для которого создаётся БД, и является внешней функцией по отношению к БД.

Рассмотрим техническую сторону обеспечения защиты данных в БД от несанкционированного доступа. Общий принцип управления доступом к базе данных такой: СУБД не должна разрешать пользователю выполнение какой-либо операции над данными, если он не получил на это права. Санкционирование доступа к данным осуществляется администратором БД. В обязанности администратора БД входит:

назначение отдельным группам пользователей прав доступа (привилегий) к отдельным группам данных в соответствии с правилами ПО;

организация системы контроля доступа к данным;

тестирование вновь создаваемых средств защиты данных;

периодическое проведение проверок правильности работы системы защиты, исследование и предотвращение сбоев в её работе.

Примечание: администратор БД обычно назначает права доступа в соответствии с проектом БД, который должен включать перечень групп пользователей и их привилегии.

Для каждого пользователя система поддерживает паспорт пользователя, содержащий его идентификатор, имя процедуры подтверждения подлинности и перечень разрешённых операций. Подтверждение подлинности заключается в доказательстве того, что пользователь является именно тем человеком, за которого себя выдаёт. Чаще всего подтверждение подлинности выполняется путём парольной идентификации. Перечень операций обычно определяется той группой, к которой принадлежит пользователь. В реальных системах иногда предусматривается возможность очень ограниченного доступа к данным постороннего человека, не требующая идентификации (доступ типа "гость").

Парольная идентификация заключается в присвоении каждому пользователю двух параметров: имени (login) и пароля (password). При входе в систему она запрашивает у пользователя его имя, а для подтверждения того, что это имя ввёл его владелец, система запрашивает пароль. Имя выдаётся пользователю при регистрации администратором, пароль пользователь устанавливает сам.

При задании пароля желательно соблюдать следующие требования:

длина пароля должна быть не менее 6-и символов;

пароль должен содержать комбинацию букв и цифр или специальных знаков, пароль не может содержать пробелы;

пароли должны часто меняться.

Для контроля выполнения этих требований обычно применяются специальные программы.

Управление доступом к данным осуществляется через СУБД, которая и обеспечивает защиту данных. Но такие данные вне СУБД становятся общедоступны. Если известен формат БД, можно осуществить к ней доступ с помощью другой программы (СУБД), и никакие ограничения при этом не помешают. Для таких случаев предусмотрено кодирование данных. Используются различные методы кодирования: перекомпоновка символов в кортеже, замена одних символов (групп символов) другими символами (группами символов) и т.д. Кодирование может быть применено не ко всему кортежу, а только к ключевым полям. Декодирование производится непосредственно в процессе обработки, что, естественно, увеличивает

время доступа к данным. Поэтому к кодированию прибегают только в случае высоких требований к конфиденциальности данных.

Не зависимо от подхода, все решения относительно допуска к выполнению тех или иных операций принимаются администратором БД. Все решения администратора БД посылаются в виде файла к СУБД. В последствии СУБД следит за порядком в системе баз данных.

Общий порядок допуска к БД можно представить в виде следующего алгоритма.

1. Решения администратора БД по распределению полномочий должны быть известны системе. Решения в виде матрицы полномочий (файла) передаются в СУБД.

2. Пользователь, начинающий работать с БД должен зарегистрироваться в ней (задать свое имя и пароль). Этот процесс носит название **идентификация**).

3. Пользователь запрашивает нужный файл и указывает тип операции(й), которую (ые) собирается произвести над этими данными.

4. СУБД сравнивает зарегистрированное имя и пароль с приписанными им полномочиями. Этот процесс носит название - **аутентификация**. Если приписанные этому имени полномочия совпадают с теми, которые запрашивает пользователь, то доступ получен, иначе – отказ.

Имя (login)	Пароль	Ресурсы			Реакция системы на нарушение правил
		Файл 1	…	Файл *N*	
Tm-tat	12345	Обновление Удаление	Удаление	Добавление	Звуковой сигнал
…	…	…	…	…	
SISok	SadIn	Чтение	Чтение	Чтение	Сообщение об отказе выполнить запрашиваемые действия

Рис. 6.1. Общий вид матрицы полномочий

Идентификация – задание уникального идентификатора пользователя. Идентификация выполняет следующие защитные функции:

– установление подлинности и определение полномочий пользователя при его допуске;

– контроль установленных полномочий и регистрация заданных действий пользователя в процессе его сеанса работы;

– учет обращений к информационным ресурсам.

Идентификатор - последовательность любых символов.

Идентификатор должен быть заранее зарегистрирован администратором службы безопасности.

В процессе регистрации для каждого пользователя заносятся следующие элементы данных:

– фамилия, имя, отчество

– уникальный идентификатор пользователя;

– имя процедуры установления подлинности;

– пароль;

– ограничения на использование пароля, например, минимальное и максимальное время, в течение которого указанный пароль будет считаться действительным;

– полномочия пользователя по доступу к информационным ресурсам.

Аутентификация – процесс установления подлинности - заключается в проверке, является ли пользователь, пытающийся осуществить доступ к ресурсам, тем, за кого себя выдает.

Общая схема идентификации и установления подлинности пользователя при доступе к ресурсам приведена на рис. 6.2.

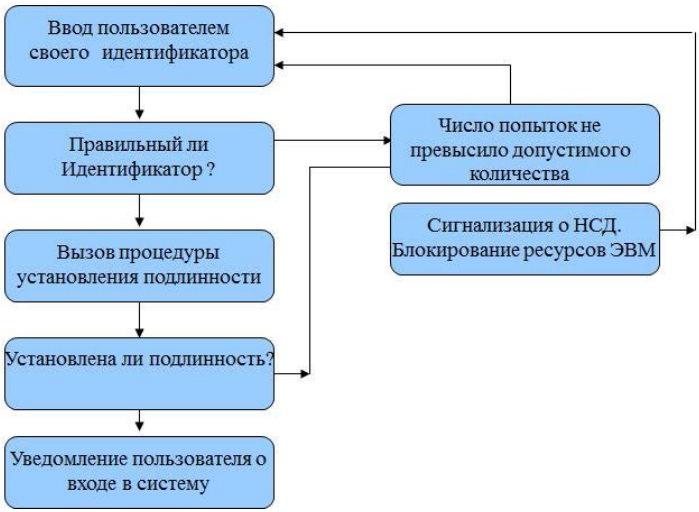


Рис. 6.2. Общая схема идентификации и аутентификации при доступе к БД

Существует два основных подхода к реализации безопасности СУБД: избирательный и обязательный.

При избирательном подходе каждый пользователь обладает различными правами (привилегиями или полномочиями) при работе с разными данными. Соответственно разные пользователи обладают разными правами доступа к одним и тем же данным.

В целом, избирательный подход характеризуется гибкостью.

При обязательном подходе каждой группе данных (файлу) присваивается некоторый классификационный уровень. Каждому пользователю присваивается уровень допуска. Доступом к определенным данным обладают только пользователи с соответствующим уровнем допуска.

В целом, обязательный подход характеризуется жесткостью, т.е. применяется к БД, в которых данные имеют неподвижную (жесткую структуру).

Основная идея обязательного подхода состоит в следующем:

– Каждому файлу присваивается уровень классификации («Секретно», «Совершенно секретно», «Для служебного пользования»)

– Каждому пользователю присваивается уровень допуска при регистрации с такими же градациями («Пользователь секретного», «Пользователь совершенно секретного», «Служебный пользователь»).

Таким образом, можно сформулировать следующие правила безопасности обязательного подхода:

1. Пользователь *i* имеет доступ к файлу *j* только если его уровень допуска >= уровню классификации объекта *j*.

2. Пользователь *i* может модифицировать файл *j*, только если его уровень допуска = уровню классификации файла *j*.

Предоставление прав доступа (привилегий) в системах, поддерживающих язык SQL, осуществляется с помощью двух команд:

GRANT – предоставление одной или нескольких привилегий пользователю (или группе пользователей):

```
GRANT {<список привилегий> | ALL PRIVILEGES}
ON <имя объекта>
TO {<список пользователей> | PUBLIC}
[WITH GRANT OPTION];
```

где <список привилегий> – набор прав, которые необходимо предоставить, или ALL PRIVILEGES – все права на данный объект;
<имя объекта> – имя объекта БД, к которому предоставляется доступ;

<список пользователей> – перечень пользователей (или ролей, см. дальше), которым будут предоставлены указанные права;

PUBLIC – предопределённый пользователь, привилегии которого доступны всем пользователям БД;

WITH GRANT OPTION – ключевые слова, дающие возможность пользователям из списка пользователей предоставлять назначенные права другим пользователям (т.е. передавать эти права).

Права, подразумеваемые под словами ALL PRIVILEGES, зависят от типа объекта. Примерный перечень прав в зависимости от типа объекта БД приведён в табл. 6.1.

Таблица 6.1. Использование объектных привилегий

Привилегия	Операции	Таблицы	Представления	Процедурные объекты
ALTER	изменение определения объекта	+	+	+
DELETE	удаление данных	+	+	...
EXECUTE	выполнение объекта	...	...	+
INSERT	добавление данных	+	+	...
SELECT	чтение данных	+	+	...
UPDATE	изменение данных	+	+	...

Примечание: процедурные объекты – это хранимые процедуры и функции.

REVOKE – отмена привилегий:

```
REVOKE [GRANT OPTION FOR]
{<список привилегий> | ALL PRIVILEGES}
ON <имя объекта>
FROM {<список пользователей> | PUBLIC}
   {RESTRICT|CASCADE};
```

где [GRANT OPTION FOR] – отмена права передачи привилегий;

CASCADE – при отмене привилегий у пользователя отменяются все привилегии, которые он передавал другим пользователям;
RESTRICT – если при отмене привилегий у пользователя необходимо отменить переданные другим пользователям привилегии, то операция завершается с ошибкой.

Другие ключевые слова имеют то же значение, что и в команде GRANT.

Для того чтобы упростить процесс управления доступом, многие СУБД предоставляют возможность объединять пользователей в группы или определять роли.

Роль – это совокупность привилегий, предоставляемых пользователю и/или другим ролям. Такой подход позволяет предоставить конкретному пользователю определённую роль или отнести его к определённой группе пользователей, обладающей набором прав в соответствии с задачами, которые на неё возложены.

Кроме привилегий на доступ к объектам СУБД еще может поддерживать так называемые *системные привилегии*: это права пользователя на создание/изменение/удаление (create/alter/drop) объектов различных типов. В некоторых системах такими привилегиями обладают только пользователи, включенные в группу администратора БД. Другие СУБД предоставляют возможность назначения дифференцированных системных привилегий любому пользователю в случае такой необходимости.

ЗАКЛЮЧЕНИЕ

Теоретический материал, приведенный в учебном издании является подробным конспектом лекций по курсу «Защита баз данных».

Содержание лекционного материала соответствует рабочей программы дисциплины.

Процесс изучения дисциплины направлен на формирование следующих компетенций

ОПК-4 способность понимать значение информации в развитии современного общества, применять достижения информационных технологий для поиска и обработки информации

ОПК-5 способность применять программные средства системного и прикладного назначения, языки, методы и инструментальные средства программирования для решения профессиональных задач

В результате изучения теоретического материала обучающийся будет способен:

– знать технологии проектирования баз данных, функции управления данными, языки баз данных;

– уметь обосновать модель данных, выбрать систему управления базами данных, планировать запросы и составлять выражения на языке баз данных.

ЛИТЕРАТУРА

1. Гордеев, С. И. Организация баз данных в 2 ч. Часть 1: учебник для вузов / С. И. Гордеев, В. Н. Волошина. – 2-е изд., испр. и доп. – М.: Издательство Юрайт, 2018. – 311 с.

2. Гордеев, С. И. Организация баз данных в 2 ч. Часть 2: учебник для вузов / С. И. Гордеев, В. Н. Волошина. – 2-е изд., испр. и доп. – М.: Издательство Юрайт, 2018. – 501 с.

3. Маркин, А. В. Программирование на sql в 2 ч. Часть 1: учебник и практикум для бакалавриата и магистратуры / А. В. Маркин. – М.: Издательство Юрайт, 2018. – 362 с.

4. Маркин, А. В. Программирование на sql в 2 ч. Часть 2: учебник и практикум для бакалавриата и магистратуры / А. В. Маркин. – М.: Издательство Юрайт, 2018. – 292 с.

5. Култыгин, О. П. Администрирование баз данных. СУБД MS SQL Server: учеб. пособие / О. П. Култыгин. – М.: МФПА, 2012. – 232 с.

6. Парфенов, Ю. П. Постреляционные хранилища данных: учебное пособие для вузов / Ю. П. Парфенов; под науч. ред. Н. В. Папуловской. – М.: Издательство Юрайт, 2018. – 121 с.

7. Советов, Б. Я. Базы данных: учебник для прикладного бакалавриата / Б. Я. Советов, В. В. Цехановский, В. Д. Чертовской. – 2-е изд. – М. : Издательство Юрайт, 2018. – 463 с.

8. Стасышин, В. М. Базы данных: технологии доступа: учебное пособие для академического бакалавриата / В. М. Стасышин, Т. Л. Стасышина. – 2-е изд., испр. и доп. – М. : Издательство Юрайт, 2018. – 178 с.

9. Нестеров, С. А. Базы данных: учебник и практикум для академического бакалавриата / С. А. Нестеров. – М.: Издательство Юрайт, 2018. – 230 с.

Учебное издание

Татарникова Татьяна Михайловна, доктор технических наук, доцент

ЗАЩИТА БАЗ ДАННЫХ

Печатается в авторской редакции.

Подписано в печать 21.05.2020. Формат 60×90 1/16. Гарнитура Times New Roman.
Печать цифровая. Усл. печ. л. 10,25. Тираж 30 экз. Заказ № 921.
РГГМУ, 192007, Санкт-Петербург, Воронежская, 79