

ВВЕДЕНИЕ

Базы данных являются основным компонентом современных информационных систем и играют ключевую роль в организации, хранении и управлении данными.

Настоящее учебное пособие посвящено основам технологий баз данных и составлено с учетом требований студентов и начинающих специалистов, которые хотят разобраться в основных принципах работы баз данных и научиться применять их в практических задачах.

В пособии последовательно рассматриваются основные модели баз данных и их особенности, а также вопросы проектирования баз данных. Затем подробно рассматривается реляционная модель данных, которая является основой большинства систем управления базами данных, а также изучается структурированный язык запросов (SQL), который широко используется для работы с реляционными базами данных. Третья часть учебного пособия охватывает вопросы создания представлений и индексов, использования процедурных решений языка SQL, а также управление транзакциями. В заключительной части пособия представлены основные нереляционные решения для работы с данными, а также основы обработки больших данных.

Материал пособия выстроен таким образом, чтобы обучающиеся имели возможность познакомиться с основными концепциями работы с реляционными базами данных с использованием структурированного языка запросов SQL на примере MySQL и PostgreSQL, а также нереляционной документоориентированной системой управления базами данных MongoDB.

1 ВВЕДЕНИЕ В БАЗЫ ДАННЫХ

1.1 Понятие базы данных

Базы данных (БД) используются повсеместно, практически ни одно приложение не обходится без базы данных. Сайты, финансовые и медицинские учреждения, учебные заведения, различные организации так или иначе используют системы хранения и обработки данных. В качестве одного из примера базы данных, часто используемой студентами, можно привести базу данных с информацией о расписании, оценках, посещаемости и т.д., которая может используется для отображения данных через личный кабинет на сайте университета. Другим примером может являться база данных склада с информацией о наличии товара. В любом случае, базы данных могут использоваться как для хранения данных совсем небольшого объема, так и для обработки терабайт информации.

Что же такое база данных? Существуют различные определения этого термина, отражающих скорее субъективное мнение тех или иных авторов, однако общепризнанная единая формулировка отсутствует.

Так, согласно ГОСТу (34.321–96), устанавливающему эталонную модель управления данными, под базой данных понимается «совокупность взаимосвязанных данных, организованных в соответствии со схемой базы данных таким образом, чтобы с ними мог работать пользователь» [1].

Гражданский кодекс РФ определяет базу данных как представленную в объективной форме совокупность самостоятельных материалов, систематизированных таким образом, чтобы эти материалы могли быть найдены и обработаны с помощью электронной вычислительной машины.

Кроме того, под базой данных часто понимают совокупность данных, характеризующую актуальное состояние некоторой предметной области и используемую для удовлетворения информационных потребностей организации, а не только отдельных пользователей [2].

Указанные выше определения имеют различные акценты, но в совокупности позволяют сформулировать некоторые характерные признаки базы данных:

- база данных хранится и обрабатывается в вычислительной системе. Таким образом, любые внекомпьютерные хранилища информации (такие как архивы, библиотеки, картотеки и тому подобное) базами данных не являются;
- данные в базах данных логически структурированы и систематизированы с целью обеспечения возможности их эффективного поиска и обработки в вычислительной системе. Структурированность подразумевает явное выделение составных частей и связей между ними, а также типизацию элементов и связей, при которой с типом элемента соотносится определённая семантика и допустимые операции;
- в базе данных хранятся не только рабочие данные, описывающие какую-то предметную область, но и метаданные, описывающие логическую структуру самих данных в формальном виде. Такую важную способность баз данных часто называют *самодокументированием*;
- схема базы данных включает в себя описания содержания, структуры и ограничений целостности, используемых для создания и поддержки базы данных;
- потребителем данных может выступать не только человек, но и программно-аппаратные комплексы.

1.2 Развитие баз данных

Можно выделить несколько этапов в развитии баз данных, схематично показанных на рисунке 1 [2].

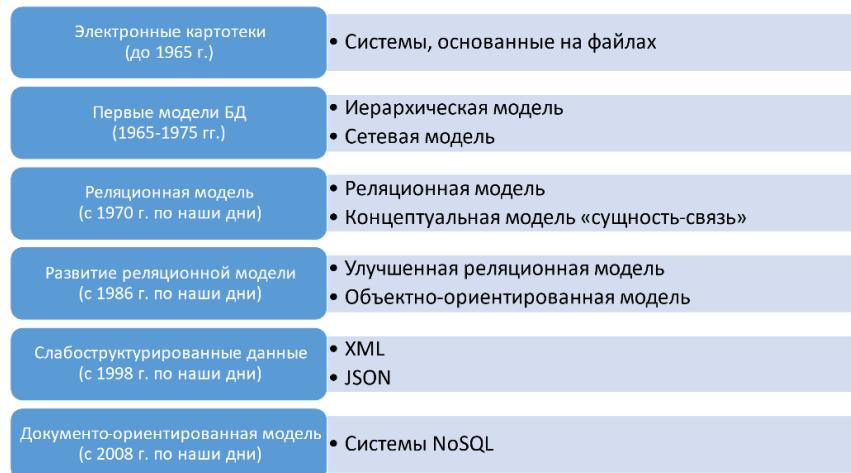


Рисунок 1 – Этапы развития баз данных

История возникновения баз данных берет свое начало с конца 50-х годов прошлого века, когда появилось программируемое оборудование для обработки записей. Разработанное программное обеспечение поддерживало обработку записей на основе файлов. Для хранения данных использовались перфокарты.

Замещение файловых систем первыми моделями баз данных (так называемой сетевой и иерархической модели) произошло в период с середины 60-х по первую половину 70-х годов XX века.

Следующий важный этап связан с появлением в начале 1970-х реляционной модели данных. В 1970 году доктор Эдгар Кодд из исследовательской лаборатории IBM опубликовал статью под названием «Реляционная модель данных для больших общих банков данных», в которой впервые прозвучал термин «реляционная модель».

Реляционная модель данных основывается на теории множеств и математическом понятии отношение («relation»). В качестве неформального синонима термину «отношение» часто встречается слово таблица. Необходимо помнить, что «таблица» есть понятие нестрогое и неформальное и часто означает не «отношение» как абстрактное понятие, а визуальное представление отношения на бумаге или экране. Для уменьшения избыточности данных в реляционной модели данных используется нормализация.

В середине 1980-х годов на рынке программных продуктов стали активно появляться программные средства, построенные на основе объектно-ориентированной парадигмы. В рамках этих моделей данные моделируются в виде объектов, их атрибутов, методов и классов, что позволяет работать с объектами баз данных так же, как с объектами в программировании в объектно-ориентированных языках программирования.

В тех ситуациях, когда на первое место выступает не столько решение задачи хранения записей, сколько простота организации и универсальность обмена данными, стоит обратить внимание на идею работы со слабоструктуризованными данными. Для обслуживания слабоструктуризованных данных в 1998 году был разработан отдельный стандарт – расширяемый язык разметки (или XML). Следует отметить, что слабоструктурированные данные в формате XML не нуждаются в отдельных системах управления базами данных, а для обработки XML разработан специальный прикладной интерфейс программирования.

Наконец, в начале двухтысячных годов появились новые подходы к хранению данных, среди которых можно выделить документо-ориентированную модель. Катализатором развития документо-ориентированных баз данных стало появление распределенных систем и технологий обработки больших данных. Опыт эксплуатации реляционных баз данных показал, что нормализованные

данные недостаточно эффективны в распределенной среде. Ключевым преимуществом документо-ориентированного подхода как раз и стала возможность использования неструктурированных данных в распределенных системах, в которых доминирующими требованиями выступают скорость обработки данных и масштабируемость.

Рассмотрим представленные модели подробнее.

1.2.1 Иерархическая модель базы данных

Иерархическая модель – это модель данных, в которой используется представление данных в виде древовидной (или иерархической) структуры, состоящей из объектов различных уровней. Каждый из узлов, в свою очередь, мог являться родительским по отношению к одному или нескольким нижерасположенным узлам, но у дочернего узла не может быть более одного родителя. Таким образом, в иерархической модели реализована одна из наиболее часто встречающихся в реальном мире связь между сущностями – связь «один ко многим» (рисунок 2).

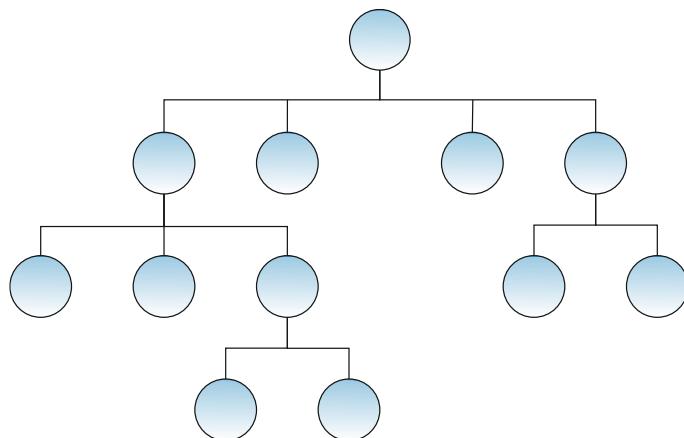


Рисунок 2 – Иерархическая модель базы данных

Рассмотрим пример базы данных. В городе находится много компаний, в компании образовано много отделов, в отделе работает много сотрудников. Проект базы данных «отдел – сотрудник», реализованный средствами иерархической модели, мог бы выглядеть так, как представлено на рисунке 3. Отдел определяется своим названием, для сотрудников хранится информации об их фамилии, имени, отчестве и даты рождения. Узлы сотрудников выступают в качестве подчиненных по отношению к узлам отделов и не могут существовать без них. В представленном примере в отделе Дирекция работает один сотрудник, в бухгалтерии и отделе кадров работают по два сотрудника, т.е. моделируется связь один ко многим.

При желании схему можно и развить, например, подчинив сотрудникам узлы с заказами, описанием работ, номерами счетов и т.п. Подобное развитие не требует кардинального изменения уже существующих узлов дерева, необходимо только добавить очередные узлы на новом уровне. Однако, если один сотрудник будет работать в нескольких отделах, всю информацию о нем придется дублировать, т.к. связь многие ко многим не поддерживается.

Модель



Пример

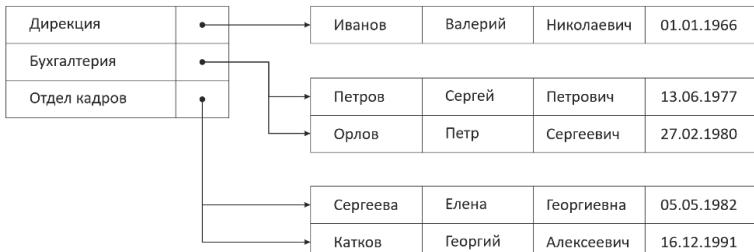


Рисунок 3 – Пример иерархической базы данных

По сравнению с системами файлов иерархическая модель обладает рядом преимуществ. Следует отметить следующие:

1. Простота понимания структуры данных. Иерархическое построение данных интуитивно понятно, что существенно упрощает проектирование баз данных.
2. Целостность данных. Иерархические базы данных представляли собой неразрозненные приложения и файлы. Все данные находились под контролем одной системы управления базами данных, которая не допускала некорректные действия с записями (например, удаление родительского узла, у которого оставались «осиротевшими» дочерние элементы).
3. Независимость данных. Данные больше не принадлежат одному приложению. Наличие системного каталога позволяет работать с базой данных приложениям, которые умеют читать метаданные.
4. Безопасность данных. Контролируемый доступ к данным осуществляется с помощью системы управления базами данных, в которую закладывается предпочтительная политика безопасности.
5. Высокая эффективность алгоритмов поиска, т.к. стоимость процедуры «спуска по дереву» пропорциональна глубине дерева.

Однако, поборов недостатки систем, основанных на файлах, иерархическая модель приобрела новые. Вот только некоторые из них:

1. Ограничения в организации отношений между сущностями. Иерархическая модель позволяет организовать последовательную связь «один ко многим» между данными только от родителя к потомку, но не в состоянии реализовать отношения «многие ко многим».

2. Структурная зависимость. Иерархическая структура предполагала, что физически данные также станут храниться в виде дерева. Серьезное изменение структуры (например, переподчинение узлов) могло привести к тому, что прикладные приложения теряли возможность навигации по данным.
3. Высокая стоимость операций модификации за счет применения алгоритма расщепления и слияния вершин дерева при вставке и удалении записей.
4. Отсутствие стандартизации. Как следствие, всегда существовала проблема переносимости.

1.2.2 Сетевая модель базы данных

Сетевая модель базы данных является расширением иерархической модели данных. В данном случае за основу была взята не древовидная модель данных, а структура, построенная на основе графа. Пример сетевой модели в виде графа показан на рисунке 4. Разница состоит в том, что в иерархической модели запись-потомок должна иметь в точности одного предка, а в сетевой структуре данных у потомка может быть любое число предков.

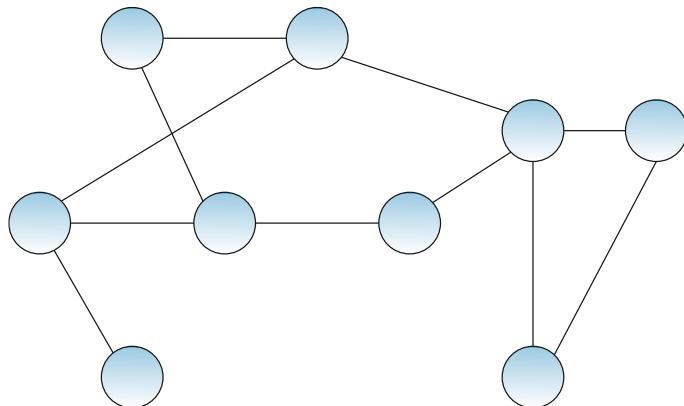


Рисунок 4 – Сетевая модель базы данных

По сравнению с иерархической, сетевая модель имеет одно серьезное преимущество: она позволяет назначать произвольное количество связей между узлами графа. Поэтому она в состоянии создавать базы данных, более точно отражающие связи реального мира, в частности в сетевых базах данных без особого труда можно было формировать отношения «многие ко многим» или замыкать связь узла на себя самого.

При удалении записи в сетевой базе данных уничтожается только этот соответствующий ей элемент и связь с ним, все остальное остается на месте. Это разительное отличие от иерархически организованной базы данных, ведь в ней при удалении одного узла удалению подлежала вся нижерасположенная ветвь дочерних элементов. Либо же требовалось проводить относительно сложную процедуру переподчинения дочерних узлов.

Говоря о недостатках модели, следует отметить, что большое количество произвольных связей повышает сложность схемы базы данных и, как следствие, вызывает дополнительные трудности при обеспечении целостности данных. Кроме того, возрастила сложность разработки прикладного программного обеспечения.

1.2.3 Реляционная модель базы данных

Довольно скоро на смену иерархической и сетевой моделям пришла принципиально новая модель данных – реляционная.

Реляционная модель данных – логическая модель, основанная на множестве взаимосвязанных именованных отношений. В свою очередь, реляционная база данных – база данных, основанная на реляционной модели данных. Термин «реляционный» означает, что теория основана на математическом понятии «отношение».

Отношение – это информационная модель реального объекта предметной области, формально представленная множеством однотипных кортежей. **Кортеж** отношения представляет собой экзем-

пляр моделируемого объекта, свойства которого определяются значениями соответствующих атрибутов («полей») кортежа. Связи между кортежами отношений (при их наличии) реализуются через простой механизм **«внешних ключей»**, являющихся, по существу, ссылками на атрибуты связываемых кортежей отношений.

Такое упрощение позволяет ассоциировать отношения реляционной базы данных с прямоугольными плоскими таблицами, а кортежи отношений – со строками таких таблиц, что упрощает представление о структуре базы. Это упрощение не совсем верно, но достаточно сильно распространено. Ключевой особенностью, почему формально нельзя отождествлять отношение и таблицу заключается в том, что отношение не имеет внутри себя порядка. Когда говорят о таблице, подразумевают, что внутри таблицы есть какой-то порядок, т.е. есть первая запись и есть последняя запись. Внутри отношения такого порядка нет, это неупорядоченное множество. Когда делается запрос к базе данных, то результат возвращается обычно в каком-то порядке. Если порядок явно не задать, то его можно считать случайным. Внутренняя структура системы управления базой данных обычно такова, что этот случайный порядок будет совпадать от запроса к запросу, т.е. может сложиться ложное впечатление, что все данные всегда будут упорядочены по какому-либо ключевому полю или в порядке добавления – это не так. В любой момент порядок может поменяться.

В качестве примера на рисунке 5 показана часть реляционной базы данных фильмов, которая хранит информацию о фильмах и снимавшихся в них актерах. Слева представлена полная схема таблиц, справа – часть данных (часть строк и столбцов), хранящихся в этих таблицах.

Учитывая введенные формальные определения, отношениями являются сущности «фильм» и «актер», которые представлены соответствующими таблицами `«film»` и `«actor»` в базе данных. Кортеж-

жем, описывающим отдельный экземпляр фильмы или актера, будет являться строка в соответствующей таблице. Например, кортежем является строка с данными об актрисе «Penelope Guinness». Атрибутами кортежа являются столбцы таблицы. Примером атрибута является фамилия в таблице актеров.

Для связи таблиц используются внешние ключи. В данном случае, показана связь «многие ко многим», т.е. в одном фильме снималось много актеров, один актер снимался во многих фильмах. Для описания этих связей используется дополнительная таблица «film_actor». Используя ссылки, заданные внешними ключами, можно получить список актеров, играющих в определенном фильме.

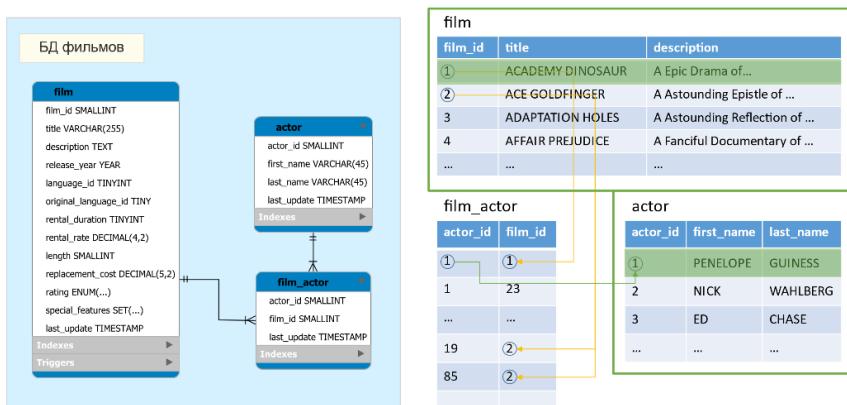


Рисунок 5 – Пример реляционной базы данных

Таким образом, реляционная модель данных включает в себя несколько аспектов. Первый аспект структурный – отношения, кортежи, атрибуты.

Второй аспект – аспект целостности, т.е. поддержание целостности и полноты данных на уровне системы управления базой данных (СУБД). Может показаться, что контроль целостности на уровне СУБД лишний. Однако многие вещи на уровне приложения

контролировать очень сложно. Т.е. можно легко проверить полноту данные на уровне приложения и запретить добавлять, скажем, фильм без названия. Но проверить, например, уникальность идентификатора уже сложнее.

И еще один компонент реляционной модели данных – это манипулирование данными, т.е. изменение данных в базе или изменение структуры отношений с использованием операторов манипулирования отношениями.

Достоинством реляционной модели стала простота представления структуры базы данных и высокая гибкость управления данными. Таблицы реляционной модели жестко структурированы, что упрощает их обслуживание. Кроме того, следует отметить высокую эффективность выполнения модифицирующих операций. Наконец, появление структурированного языка запросов SQL, ставшего стандартом, также способствовало развитию реляционных баз данных.

Конечно, у реляционной модели есть и свои недостатки. По существу, это компромиссное решение между потребностью отражать сущности реального мира и связи между ними и ограниченными возможностями математической теории множеств, переложенной на программный код.

Так, из-за борьбы с избыточностью данных в реляционной базе данных в процессе нормализации данные распределяются по нескольким отношениям (таблицам). Как следствие, для получения сводного отношения приходится собирать данные по разным таблицам и осуществлять множество соединений. Чем больше соединений следует провести, тем больше времени и ресурсов будет затрачено. Другая проблема реляционной модели заключается в особенностях организации связи между отношениями. Для создания отношения «многие ко многим» приходится вводить дополнительные ассоциативные таблицы и применять два типа связей «один ко многим», как было показано в примере с базой данных фильмов.

1.2.4 Объектно-ориентированная модель базы данных

В середине 1980-х годов на рынке программных продуктов стали активно появляться программные средства, построенные на основе объектно-ориентированной парадигмы. Объектные и объектно-реляционные модели основаны на хранении в базе данных объектов, определяемых в рамках объектной модели языка программирования, их атрибутов, методов и классов.

С точки зрения точности моделирования реального мира объектно-ориентированная модель данных по всем статьям сможет превзойти реляционную, ведь в идеале она должна быть способной адекватно выражать информационные структуры любой сложности. Самым главным достоинством СУБД очередного поколения мог стать тот факт, что проведение логических операций над данными многократно усилится за счет возможностей объектно-ориентированного программирования (ООП). Этот подход позволил бы добиться очень высокой эффективности доступа в режиме навигации. Ожидалось, что объектные базы данных вытеснят все остальные классы моделей данных, однако по ряду причин этого не произошло: база данных может использоваться только клиентами, написанными на одном языке программирования, не удается создать высокоуровневый декларативный язык запросов, не удается обеспечить высокую производительность при массовой выборке данных.

В итоге системы, основанные на чисто объектных моделях данных, заняли относительно небольшой сегмент в многообразии применений СУБД, но некоторые объектные средства стали составной частью систем управления базами данных общего назначения, таких как PostgreSQL. В настоящее время многие высокопроизводительные системы реализуют объектные расширения, и поэтому их принято называть объектно-реляционными.

Наиболее важными видами объектных расширений можно считать возможность определения пользовательских типов данных, в том числе структурных, а также использование коллекций объектов.

1.2.5 Слабоструктурированные данные

Слабоструктурированные данные – это форма структурированных данных, не соответствующая строгой структуре таблиц и отношений в моделях реляционных баз данных, тем не менее эта форма данных содержит теги и другие маркеры для отделения семантических элементов и для обеспечения иерархической структуры записей и полей в наборе данных.

В некоторых классах приложений отделение описания структуры данных (т.е., при передаче документов по сети целесообразно определение структуры документа пересылать вместе с его содержимым. Поскольку в подобных случаях значительная часть данных представляет собой текст на естественном языке, такие данные принято называть слабоструктуризованными. Наиболее широко используемым форматом для представления слабоструктурированных данных является XML, довольно часто употребляется также JSON. Хотя ни XML, ни JSON не предполагалось использовать для хранения данных, оба этих формата можно рассматривать как модели данных.

В объектно-реляционных базах слабоструктурированные данные могут храниться как значения атрибутов отношений (таблиц), также имеются встроенные функции, позволяющие формировать значения слабоструктурированных типов из табличных данных и, наоборот, извлекать элементы слабоструктурированных значений в виде коллекций, что дает возможность сочетать реляционные и слабоструктурированные языки запросов (в частности, XPath и XQuery, а также SQL/JSON path, введенный в стандарте SQL 2016).

1.2.6 Модель ключ-значение

Опыт эксплуатации реляционных баз данных показал, что нормализованные данные недостаточно эффективны в распределенной среде. Если необходимо создать очень большую базу данных, которую нельзя использовать на одном сервере – необходимо выполнить кластеризацию базы. Но при использовании кластеров возникают проблемы обеспечения целостности данных из-за появление лишних задержек при доступе к данным на разных серверах. Иногда осознанно жертвуют целостностью, возможностью использовать сложные SQL-запросы, но получают более простое масштабирование.

Простым способом повышения производительности хранилищ данных стало упрощение модели данных. Вместо реляционных таблиц можно использовать ассоциативный массив или словарь, позволяющий работать с данными по ключу. Подобные базы данных получили название хранилищ ключ-значение. Данные здесь хранятся как совокупность пар ключ-значение, в которых ключ служит уникальным идентификатором. При этом в качестве ключей и значений может использоваться что угодно: от простых типов данных до сложных объектов.

В моделях подобного типа предполагается, что поиск данных возможен только по первичному ключу, а интерпретация значения выполняется в приложении. Привлекательной стороной таких систем является простота начального запуска, однако практически все функции, обычно выполняемые системами управления базами данных, в том числе взаимосвязи между объектами, ограничения целостности, высокоуровневые декларативные запросы, поиск по значениям неключевых атрибутов должны реализовываться в коде приложения, что существенно усложняет разработку приложений.

Поскольку от хранилища ключ значение требуется так мало, то базы данных этого типа могут демонстрировать невероятно высокую производительность, но в общем случае бесполезны, когда требуются сложные запросы и агрегирование. Подобные системы нашли свое применение в первую очередь при организации быстрого кеширования. Хранилища ключ-значений в первую очередь востребованы для хранения очень больших объемов данных, поэтому их разработка ведется многими крупными интернет-компаниями (Amazon, Apache, Google). Многие системы, первоначально позиционировавшиеся как системы этого типа, эволюционируют в направлении включения более сложных возможностей, приближающих их к более развитым СУБД, вплоть до реализации полноценных декларативных языков запросов.

1.2.7 Документо-ориентированная модель БД

Документо-ориентированная модель данных – явление сравнительно новое, появившееся в начале 2000-х годов. Данная модель выступает едва ли не антиподом реляционной теории и опирается на идею хранения иерархических структур данных, в которой каждый элемент представляет собой целостный, а не «размазанный» по двумерным отношениям, как в реляционной модели, документ. Документы хранятся не в таблицах, а в специальных хранилищах документов. Основным инструментом по доступу и управлению данными выступает декларативный язык NoSQL (Not only SQL).

В сравнении с реляционной моделью документо-ориентированная модель выглядит весьма необычно. Во-первых, здесь не используются структурные единицы реляционных баз данных, таких как атрибут, кортеж, отношение. Базовая структурная единица документо-ориентированной модели – это документ, кроме того, еще существует понятие коллекции – контейнера для схожих документов. С некоторой степенью допущения можно считать коллекцию

аналогом таблицы, а документ – строкой, но в реляционной таблице все строки обладают идентичной структурой, а в документо-ориентированной базе данных все документы могут быть произвольными. В качестве примера на рисунке 6 показан один документ, хранящий информацию о фильме. Документ представлен в формате JSON, который может содержать вложенные объекты, например, информацию о рейтинге фильма из базы данных IMDB или о наградах фильма, и списки значений, например, список актеров или жанров. В реляционной базе данных фильмы и актеры хранились в разных таблицах, а для связей между ними использовались внешние ключи.

```
{  
    "_id": {  
        "$oid": "573a139bf29313caabcf4dd0"  
    },  
    "fullplot": "A ticking-time-bomb insomniac and a slippery soap salesman...",  
    "imdb": {  
        "rating": 8.9,  
        "votes": 1191784,  
        "id": 137523  
    },  
    "year": 1999,  
    "plot": "An insomniac office worker, looking for a way to change his life...",  
    "genres": [ "Drama" ],  
    "rated": "R",  
    "metacritic": 66,  
    "title": "Fight Club",  
    "lastupdated": "2015-09-02 00:16:15.833000000",  
    "languages": [ "English" ],  
    "writers": [ "Chuck Palahniuk (novel)", "Jim Uhls (screenplay)" ],  
    "type": "movie",  
    "awards": {  
        "wins": 11,  
        "nominations": 22,  
        "text": "Nominated for 1 Oscar. Another 10 wins & 22 nominations."  
    },  
    "countries": [ "USA", "Germany" ],  
    "cast": [ "Edward Norton", "Brad Pitt", "Helena Bonham Carter", "Meat Loaf" ],  
    "directors": [ "David Fincher" ]  
}
```

Рисунок 6 – Пример записи из БД MongoDB с информацией о фильме

Таким образом, документо-ориентированные базы свободны от ключевого недостатка их реляционных коллег – существенных затрат на сборку данных из нескольких таблиц в единое целое для

формирования сводного результата запроса, т.к. единицей хранения выступает неделимый документ. Используя пример, показанный на рисунке 6, получение всей информации о фильме будет выполняться очень быстро, т.к. по факту будет делаться запрос на один документ по его ключу. Как правило, такие базы данных ориентированы на работу с большими массивами данных, находящимися в распределенных системах

Обратной стороной медали выступает ограниченность синтаксических конструкций по управлению данными языка NoSQL. Т.е. чтобы получить полную информацию об актере, необходимо получить отдельный документ, описывающий актера. Получение сводного отношения, например, всех фильмов, в которых снимался указанный актер, также усложняется. Кроме того, в документо-ориентированных СУБД отсутствует транзакционность в терминах реляционных баз данных.

1.2.8 Распределенная обработка MapReduce

Кратко рассмотрим еще один подход к распределенной обработке больших данных – MapReduce.

Идея MapReduce зародилась в стенах компании Google как результат поиска наиболее адекватных решений по распределенной обработке больших объемов данных на компьютерных кластерах. Суть парадигмы MapReduce заключается в том, что большая задача разделяется на ряд небольших заданий, каждое из которых может быть выполнено на любом из узлов кластера.

Допустим, в нашем исходном наборе данных имеются записи, содержащие разнотипные документы, и нам необходимо разбить их на однотипные группы. Решение такой задачи в MapReduce будет осуществлено в три этапа (рисунок 7):

1. Первая фаза Мар представляет собой предварительную параллельную обработку входных данных в соответствии с

правилами, заданными пользователем. Реализованные в коде функции Map() правила применяются к каждой поступающей на вход записи, в результате получится последовательность пар ключ-значение.

2. Пары ключ-значение (k, v) , возвращаемые Map, обрабатываются и сортируются по ключу. Ключи разделяются между всеми функциями Reduce(), поэтому все пары ключ-значение с одним и тем же ключом k отправляются на вход одной и той же функции Reduce().
3. Фаза Reduce осуществляет обработку ключа и всех ассоциированных с ним значений. Функция Reduce() обрабатывает входные данные в соответствии с правилами, запрограммированными пользователем, и возвращает последовательность из нуля, одной или более пар ключ-значение.

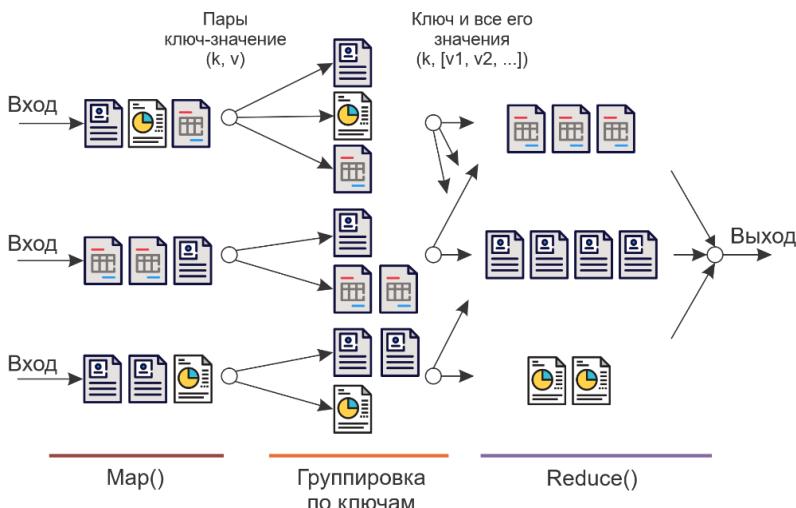


Рисунок 7 – Схема распределенной обработки MapReduce

Существенной особенностью модели MapReduce является то, что все запуски функций Map() и Reduce() могут осуществляться

независимо на разных компьютерах, входящих в распределенную систему или кластер. То же самое можно сказать и о промежуточной операции группировки по ключам. Все это в целом обеспечивает выполнение важного в работе с большими данными в распределенных системах принципа горизонтальной масштабируемости.

1.3 Системы управления базами данных

С появлением более совершенных моделей данных для управления данными стали разрабатывать специальное программное обеспечение, получившее название системы управления базами данных (СУБД).

1.3.1 Понятие СУБД

Система управления базами данных – это комплекс программ, позволяющих создать базу данных (БД) и манипулировать данными (вставлять, обновлять, удалять и выбирать). Система обеспечивает безопасность, надёжность хранения и целостность данных, а также предоставляет средства администрирования доступа к схемам данных и данным [3].

Зачем вообще нужны СУБД? Предположим, мы хотим создать свое приложение, содержащее информацию о коллекции музыкальных альбомов и их исполнителях. И решили пойти самым простым путем – записывать информацию в файл. Можно даже писать все данные в один файл формата JSON. И скорее всего этот подход будет хорошо работать большую часть времени – открыли файл, записали новую информацию, закрыли файл. Проблемы начнутся при различных внештатных ситуациях, например, при отключении электричества. При работе с файловой системой результат таких ситуаций может быть совершенно разный – можно потерять всю информацию, некорректно записать часть данных и нарушить всю

структуру JSON-файла и тому подобное – это что касается необходимости восстановления данных после сбоев. Отдельным вопросом является многопользовательский доступ к данным: что будет, если несколько человек захотят одновременно записать новые данные или обновить одни и те же существующие данные?

Чтобы самостоятельно не решать эти вопросы при разработке каждого нового приложения – т.е. не думать, как обеспечить производительность доступа к данным, защитить их от несанкционированного доступа, обеспечить многопользовательский доступ, резервное копирование и так далее – используются системы управления базами данных.

Таким образом, на СУБД возлагаются обязанности не просто по управлению базами данных, но и по предотвращению несанкционированного доступа к данным: контролю за многопользовательским (параллельным) доступом; поддержке целостности данных; восстановлению данных.

1.3.2 Функции СУБД

В настоящее время можно говорить о нескольких десятках функций, служб и сервисов, которыми обязана обладать современная СУБД. Рассмотрим основополагающие функции СУБД.

1. Доступность данных.

Под доступностью данных понимается предоставление пользователю возможности вставлять, редактировать, удалять и извлекать данные из базы данных, а также управление данными во внешней и в оперативной памяти. Управление данными во внешней памяти включает обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в базу, так и для служебных целей (например, для хранения индексов). В некоторых реализациях СУБД используются возможности существующих файловых систем, в других работа производится вплоть до уровня

устройств внешней памяти. СУБД обычно работают с данными значительных размеров; зачастую превышающими размер оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. В различных СУБД реализованы свои механизмы буферизации данных для управления данными в оперативной памяти. В любом случае, при осуществлении любой из операций работы с данными пользователь не должен вникать в особенности физической реализации системы, т.е. все операции должны быть прозрачны.

2. Метаописание данных.

СУБД должна предоставлять системный каталог, в котором содержится: описание хранимых в базе данных; описание связей между данными; ограничения целостности данных; регистрационные данные пользователей и другая служебная информация. Благодаря метаданным база данных становится доступной внешним приложениям, упрощается понимание смысла данных, усиливаются меры безопасности, может выполняться аудит информации.

3. Управление параллельностью.

Под параллельностью будем понимать реализацию механизма одновременного многопользовательского доступа к обрабатываемым данным с гарантией корректного обновления этих данных. Умение предоставить нескольким пользователям совместный доступ к разделяемым ресурсам – это едва ли не самая сложная задача, решаемая СУБД. СУБД должна суметь избежать конфликта совместного доступа двух или большего числа пользователей к одним и тем же строкам таблицы, или, по крайней мере, исключить какие-либо нежелательные последствия при возникновении конфликта.

4. Обработка данных в рамках транзакции.

СУБД гарантирует, что база данных будет всегда находиться в непротиворечивом состоянии вне зависимости от любых сбоев при проведении операций обновления данных. Для этого операции с данными (в первую очередь вставки, редактирования и удаления) объединяются в единый блок, называемый транзакцией.

Транзакция – это последовательность операций над базой данных, рассматриваемых СУБД как единое целое. Либо вся транзакция успешно выполняется, и СУБД фиксирует изменения в базе данных, произведенные этой транзакцией, во внешней памяти, либо вся транзакция не выполняется и ни одно из действий, входящих в ее состав никак не отражается на состоянии базы, в таком случае говорят об откате транзакции. При откате транзакции состояние базы данных будет восстановлено на момент времени, предшествующий вызову транзакции.

Транзакции необходимы для поддержания логической целостности БД, а поддержка механизма транзакций является обязательным условием даже для однопользовательских баз данных.

5. Обеспечение целостности данных.

Все содержащиеся в базе данные должны быть корректны и не-противоречивы. Это означает, что данные в таблицах могут модифицироваться только в соответствии с утвержденными правилами. В самом общем случае можно говорить о существовании трех правил поддержания целостности данных: целостность доменов или типов данных, целостность отношений, целостность связей между отношениями. Кроме того, разработчик имеет возможность описывать свои собственные бизнес-правила, которые называются корпоративными ограничениями.

6. Журнализация изменений, резервное копирование и восстановление базы данных после сбоев.

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 базы данных после любого аппаратного или программного сбоя. Для восстановления базы данных нужно располагать некоторой дополнительной информацией. Другими словами, поддержание надежности хранения данных в базе требует избыточности хранения данных, причем та часть данных, которая используется для восстановления, должна храниться особо надежно. Для восстановления базы после сбоя используют подход, основанный на периодическом полном резервном копировании базы данных и использовании журнала транзакций, который хранит все изменения, происходившие в базе с момента создания последней резервной копии. При ведении журнала придерживаются стратегии «упреждающей» записи в журнал. Эта стратегия заключается в том, что запись об изменении любого объекта базы данных должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части базы данных. Использование таких механизмов позволяет восстановить базу вплоть до незавершенных транзакций и продолжить работу.

7. Обмен данными.

СУБД обязана поддерживать современные сетевые технологии и предоставлять доступ к базе данных удаленным персональным компьютерам.

8. Контроль за доступом к данным.

Доступ к данным могут осуществлять только зарегистрированные в СУБД пользователи в соответствии с назначенными им администратором СУБД правами.

9. Поддержка языков базы данных.

Для работы с базой используются специальные языки, в целом называемые языками баз данных. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка: язык определения данных (DDL – Data

Definition Language) и язык манипулирования данными (DML – Data Manipulation Language).

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с базой данных, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс работы с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (Structured Query Language). Кроме упомянутых языков управления и манипулирования данными, в состав SQL еще входит и язык контроля данных (DCL – Data Control Language), операторы которого позволяют управлять правами доступа к схеме данных и данным.

Еще раз следует отметить, что список обязанностей СУБД на этом далеко не заканчивается. Современные системы предоставляют средства мониторинга, сервисы статистического анализа, утилиты экспорта/импорта данных, развитые средства проектирования баз данных и прикладного программного обеспечения, инструменты реорганизации файлов данных и индексных данных, службы аудита и многое другое. Однако все дополнительные сервисы и службы выполняют вспомогательную роль, а основы жизнедеятельности СУБД определяются перечисленными выше функциями.

1.3.3 Организация современных СУБД

На современном рынке программного обеспечения конкурирует около трех десятков коммерческих СУБД. Несмотря на то, что все эти программные продукты предназначены для решения практически одинаковых задач, входящие в перечисленные СУБД программные компоненты и взаимосвязи между ними далеко не идентичны. Поэтому построить структурную схему типовой СУБД довольно сложно. Рассмотрим основные компоненты СУБД, показанные на рисунке 8.

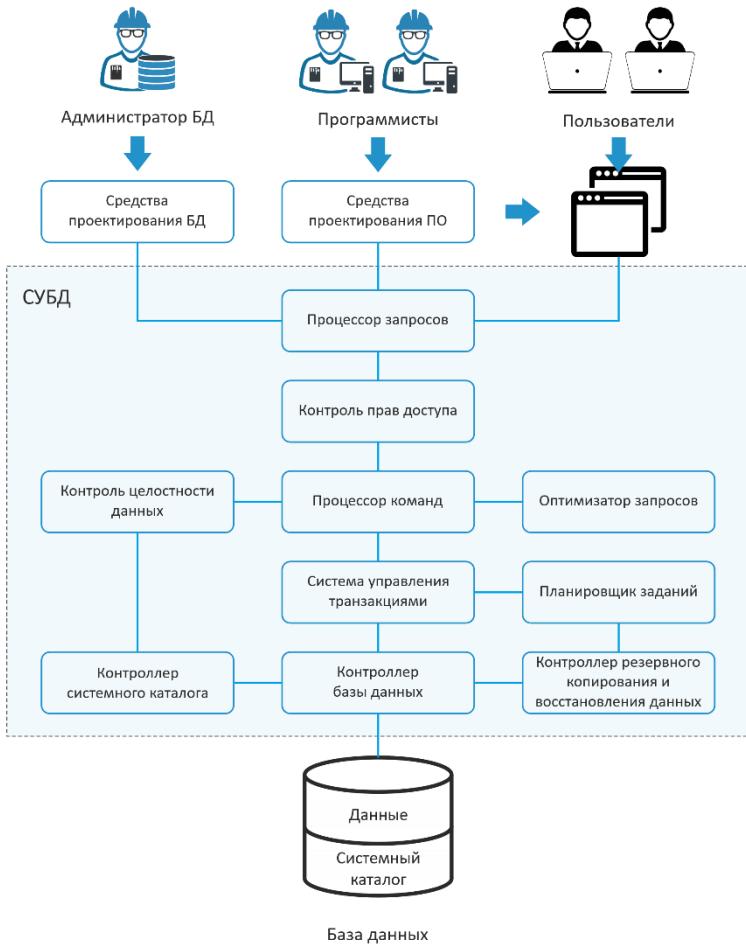


Рисунок 8 – Основные компоненты СУБД

На верхнем уровне системы расположены потребители услуг СУБД: администраторы базы данных, программисты и пользователи приложений. Администратор базы данных проектирует схему данных, назначает права доступа, настраивает резервное копирование данных с использованием различных средств проектирования, как входящих в состав СУБД, так и от сторонних производителей.

Программисты используют средства СУБД для разработки клиентских приложений и отчетов. Пользователи являются основным потребителем услуг СУБД.

Для взаимодействия с базой данных используется структурированный язык запросов SQL. Средства проектирования базы данных, программного обеспечения и клиентские приложения отправляют в адрес СУБД инструкции на языке SQL. Эти команды поступают на процессор запросов, который преобразует их в набор низкоуровневых команд, понятных ядру СУБД.

Далее выполняется контроль прав доступа для пользователя, от имени которого выполняются команды. Разные пользователи могут иметь разные права для работы с базой данных. Кто-то может только читать данные, кто-то редактировать только часть данных, а некоторым пользователям доступ к базе данных может быть полностью закрыт.

После проверки прав, команда поступает на процессор команд. На этом уровне сначала происходит проверка контроля целостности данных с использованием метаданных, полученных от контроллера системного каталога. Далее команда поступает в оптимизатор запросов, чья цель – найти наиболее эффективный способ выполнения поступивших команд. Оптимизированная команда компилируется и передается в систему управления транзакциями. Система управления транзакциями, во-первых, отвечает за полное и корректное выполнение блока команд и, во-вторых, совместно с планировщиком заданий обеспечивает параллельную многопользовательскую обработку данных.

Наконец, блок команд передается в контроллер баз данных. Задача модуля заключается в организации взаимодействия СУБД с файлами базы данных и файлами системного каталога. При этом для осуществления стандартных операций ввода-вывода задействуются возможности операционной системы.

1.3.4 Классификация СУБД

СУБД можно классифицировать по различным критериям: по модели данных, степени распределенности и по способу доступа к базе данных. Различные модели данных были рассмотрены ранее – это реляционная, объектно-ориентированная, документо-ориентированная и другие модели данных.

По степени распределенности различают локальные СУБД, когда все части СУБД размещаются на одном компьютере, и распределенные СУБД. В распределенных СУБД части могут размещаться на нескольких компьютерах.

По способу доступа к базам данных различают файл-серверные, клиент-серверные и встраиваемые СУБД.

В файл-серверных СУБД (рисунок 9) файлы данных располагаются централизованно на файл-сервере. Сами СУБД располагаются на каждой рабочей станции. Доступ СУБД к данным осуществляется посредством локальной сети.



Рисунок 9 – Файл-серверная СУБД

При обработке данных после получения запроса к данным происходит частичное копирование файлов базы данных на клиентские машины, где эта обработка и происходит. Если произошло изменение данных, то обработанные данные отправляются назад на

сервер с целью обновления базы данных. Синхронизация чтений и обновлений осуществляется посредством файловых блокировок.

Преимуществом этой архитектуры являются только простота реализации и невысокие требования к файловому серверу. В качестве недостатков можно перечислить необходимость устанавливать полную копию СУБД на каждой рабочей станции, потенциально высокая загрузка локальной сети, проседание производительности при одновременной работе многих клиентов с одними и теми же данными, высокие требования к клиентским рабочим местам, затруднённость или невозможность централизованного управления, низкий уровень обеспечения целостности и сохранности данных. На данный момент файл-серверная технология считается устаревшей, а её использование в крупных системах – недостатком.

Клиент-серверная СУБД (рисунок 10) располагается на отдельном производительном сервере вместе с базой данных и осуществляет доступ к базе данных непосредственно, в монопольном режиме. Все клиентские запросы на обработку данных обрабатываются клиент-серверной СУБД централизованно.

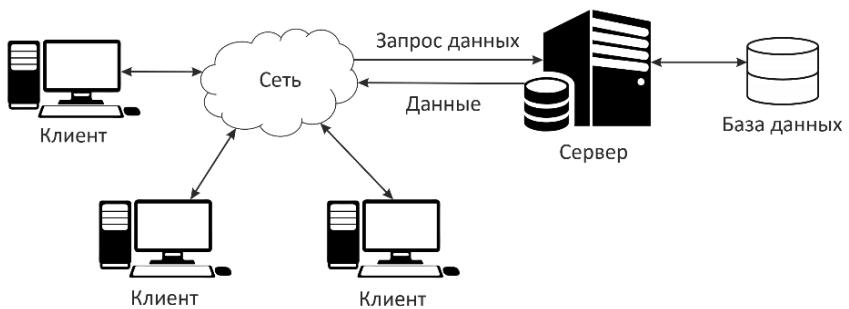


Рисунок 10 – Клиент-серверная СУБД

На клиентских станциях достаточно установить сравнительно нетребовательное к ресурсам пользовательское программное обеспечение и настроить сетевой доступ к серверу СУБД. Работа клиент-серверных систем принципиально отличается от работы в системах «файл-сервер». Теперь вместо перекачки файлов с базой данных клиентский компьютер отправляет серверу запрос, построенный на основе языка SQL. Получив и обработав инструкцию SQL, сервер возвращает клиентскому компьютеру результаты ее выполнения, например, выборку определенных данных.

Ключевой принцип технологии клиент-сервер в проектах баз данных заключается в разделении между участниками процесса функций по работе с данными. Обычно выделяют следующий перечень уровней функций:

1. Функции презентационной логики определяют особенности представления данных на экране и графический интерфейс пользователя в целом.
2. Функции бизнес-логики представляют собой алгоритмы решения задач приложения, указанная группа функций обычно реализуется на процедурном SQL и/или на языках программирования высокого уровня.
3. Функции логики БД обеспечивают решение задач выборки, вставки, удаления и редактирования данных. Перечисленные функции обработки данных обычно реализуются средствами SQL.
4. Функции СУБД обеспечивают управление информационными ресурсами.

В зависимости от того, как перечисленные группы функций распределены между клиентом и сервером, говорят о том или ином типе двухуровневого клиент-серверного проекта баз данных (рисунок 11).

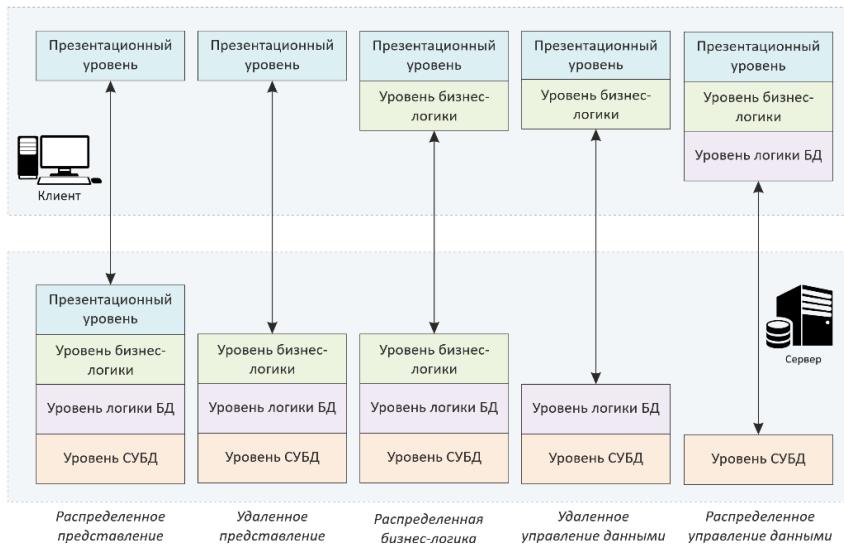


Рисунок 11 – Двухуровневая клиент-серверная СУБД

На ранних этапах развития клиент-серверной архитектуры доминирующим подходом считалось распределенное представление и удаленное представление данных. И в первом, и во втором случаях клиент частично или полностью выполнял лишь задачи, связанные с выводом данных на экран и вводом/редактированием этих данных пользователем. Все остальное делал сервер. В настоящее время, когда стоимость персональных компьютеров стала не столь высокой, а производительность существенно возросла, более популярными стали модели с распределенной бизнес-логикой и удаленным управлением данными. В подобных проектах клиентские станции частично или полностью забирают у сервера функции, реализующие бизнес-логику проекта. Тем самым с сервера существенно снижается нагрузка и, как следствие, упрощается масштабирование проекта.

Хотя двухуровневые модели распределения обязанностей между клиентом и сервером сегодня наиболее популярны, имеются

и еще более сложные решения. Среди них можно выделить трехуровневую модель, дополненную сервером приложений, являющимся посредником между клиентов и сервером баз данных.

Уже несколько десятилетий клиент-серверное построение базы данных является доминирующим. Причин тому несколько:

- значительно повышается доступность базы данных. Сервер представляет собой открытую систему, поэтому клиентские компьютеры могут функционировать под управлением различных операционных систем и с различным программным обеспечением;
- выделенный сервер СУБД в состоянии обеспечить параллельную многопользовательскую обработку данных;
- основные правила поддержки целостности и непротиворечивости данных описываются на одном сервере СУБД, клиентские приложения никаким образом не в состоянии обойти эти правила;
- экономно расходуется пропускная способность компьютерных сетей;
- благодаря централизованному хранению данных сравнительно просто поддерживать единые для всех правила безопасности базы данных;
- наличие стандарта на основной язык общения SQL обеспечивает широкие возможности доступа к серверу базы данных из программного обеспечения различных производителей;
- упрощены вопросы обслуживания и администрирования базы данных.

Недостаток клиент-серверных СУБД состоит в повышенных требованиях производительности к серверу.

Встраиваемая СУБД поставляется как составная часть некоторого программного продукта, не требуя процедуры самостоятельной установки.

У встраиваемых СУБД есть ряд особенностей. Такая СУБД предназначена для локального хранения данных своего приложения и не рассчитана на коллективное использование в сети. Встраиваемые СУБД могут использоваться, например, в почтовых клиентах или мессенджерах.

Физически встраиваемая СУБД чаще всего реализована в виде подключаемой библиотеки. Доступ к данным со стороны приложения может происходить посредством SQL либо через специальные программные интерфейсы, что позволяет обеспечить высокую скорость доступа к данным и малый расход памяти.

Область применения встраиваемых СУБД определяет и реализуемые в них функции. В таких СУБД, как правило, нет пользовательских прав и реализована простейшая изоляция транзакций с помощью стандартных механизмов операционной системы наподобие блокировки файлов. Также зачастую отсутствует архивация и репликация. Встраиваемая база данных надёжна настолько, насколько надёжна библиотека СУБД и файловая система, на которой база данных располагается. Тем не менее, есть множество способов потерять данные, так что такие решения по надёжности уступают серверным СУБД.

В качестве примеров встраиваемых СУБД можно привести SQLite, Oracle Berkeley DB или Microsoft SQL Server Compact.

1.3.5 Современные СУБД

В заключении кратко перечислим современные системы управления базами данных.

Из малых систем, рассчитанных на одного пользователя, сегодня наибольшей популярностью пользуются Microsoft Access и SQLite.

В перечень получивших широкое признание многопользовательских реляционных СУБД входят: Oracle, Microsoft SQL Server,

MySQL [4, 5], PostgreSQL [6, 7] и другие. Все СУБД так или иначе стараются поддерживать стандарт SQL для запроса и манипуляции данными.

Однако есть и различия. Например, практически в каждой из СУБД имеются специфичные для нее типы данных, не имеющие аналогов в стандарте. В качестве характерного примера можно привести PostgreSQL, в котором реализованы экзотические типы, предназначенные для хранения пространственных и геометрических данных (BOX, CIRCLE, LINE и т. д.). Не все СУБД поддерживают регулярные выражения. Кроме того, могут отличаться механизмы управления транзакциями, резервного копирования и способы реализации других функций СУБД.

Самыми известными программными решениями типа ключ-значение стали базы Amazon DynamoDB и Apache Cassandra.

В качестве примеров наиболее известных документо-ориентированных СУБД можно привести MongoDB [8, 9], CouchDB, Google Cloud Datastore и RavenDB.

2 РЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

2.1 Реляционная модель

В 1970 году Э. Кодд опубликовал статью, в которой впервые прозвучал термин «реляционная модель» [10]. Реляционная модель основывалась на математическом понятии «отношение» и опиралась на строгие математические выкладки, оставаясь интуитивно понятной.

Можно перечислить следующие достоинства реляционной модели:

- модель имела небольшой набор абстракций, которые позволяют сравнительно просто моделировать большую часть распространенных предметных областей и допускают точные формальные определения;
- модель имела простой и в то же время мощный математический аппарат, опирающийся главным образом на теорию множеств и математическую логику и обеспечивающий теоретический базис реляционного подхода к организации баз данных;
- модель предоставляла возможность манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

Основными понятиями реляционных баз данных являются тип данных, домен, атрибут, кортеж, первичный ключ и отношение.

Эти понятия проиллюстрированы на рисунке 12 на примере отношения «Сотрудники», содержащего информацию о сотрудниках некоторой организации. В примере отношение хранит следующую информацию о сотрудниках: идентификатор сотрудника, имя, фамилия, зарплата, дата рождения и отдел, в котором они работают.



Рисунок 12 – Отношение «Сотрудники»

2.2 Тип данных. Домен

Понятие **типа данных** в реляционной модели данных полностью адекватно соответствующему понятию типа данных в языках программирования. Обычно в современных реляционных базах данных допускается хранение символьных данных с фиксированной или переменной длиной, числовых данных, включая целые числа различного размера, вещественные числа с плавающей или фиксированной точностью, битовые данные, «temporalные» данных (такие как дата, время или временной интервал). Кроме того, многие СУБД поддерживают хранение пространственных и слабоструктурированных данных в формате XML или JSON. Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных. Например, в базе данных PostgreSQL поддерживается хранение данных специального типа «деньги».

Типизация хранимых значений не просто указывает на размерность в байтах, которую должна выделить система для размещения в памяти того или иного значения. Типизация также определяет, какие операции могут быть осуществлены с теми или иными данными.

Таким образом, понятие тип данных интегрирует в себе три компоненты:

- ограничение множества принадлежащих типу значений;
- определение применяемых к типу набора операций;
- определение способа отображения значений типа.

В примере с отношением «Сотрудники» (рисунок 12) представлены данные четырех типов: целые числа используются для хранения идентификатора сотрудника и идентификатора отдела, строки символов для хранения имени и фамилии, а также типы данных «дата» и «деньги».

Как показала практика, при обработке данных в современных информационных системах возможностей обычной типизации становится недостаточно. Фундаментальная для современных языков программирования идея типизации в реляционной модели получила дальнейшее развитие. Было введено новое понятие – **домен**.

Понятие домена более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат «истина», то элемент данных является элементом домена.

Домен не подменяет понятие типизации, а выступает в качестве дополнительного ограничителя обрабатываемой в базе данных информации.

Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа. Например, домен «Имена» в примере с сотрудниками определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака).

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В примере значения доменов «Номера пропусков» и «Номера отделов» относятся к типу целых чисел, но не являются сравнимыми. Однако стоит заметить, что в большинстве реляционных СУБД понятие домена не используется.

2.3 Отношение

2.3.1 Понятие отношения

Введем определение отношения в реляционной модели данных.

Отношение R состоит из *схемы* и *тела*.

Схема H отношения R – конечное множество упорядоченных пар вида (A_i, T_i) , где A_i – имя атрибута, а T_i – имя домена, $i = 1, \dots, n$. По определению требуется, чтобы все имена атрибутов в заголовке отношения были различными (уникальными).

Тело B отношения R – множество кортежей t .

Кортеж t , соответствующий схеме H – множество упорядоченных пар вида (A_i, v_i) , где v_i – допустимое значение домена T_i .

Количество кортежей называют кардинальным числом отношения (кардинальностью), или **мощностью** отношения. Количество атрибутов называют степенью, или «арностью» отношения.

Схема базы данных (в структурном смысле) – это набор именованных схем отношений. Реляционная база данных – это набор отношений, имена которых совпадают с именами схем отношений в схеме базы данных.

В классических реляционных базах данных после определения схемы изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных, т.е. определение новых и изменение существующих схем отношения. Это принято называть *эволюцией схемы базы данных*.

Упрощенным представлением отношения является таблица, заголовком которой является схема отношения, а строками – кортежи отношения-экземпляра; в этом случае имена атрибутов именуют столбцы этой таблицы. Поэтому иногда говорят «столбец таблицы», имея в виду «атрибут отношения».

Вернемся к рассмотренному примеру с отношением «Сотрудники». На рисунке 13 представлена таблица (или отношение) с 6 столбцами-атрибутами (идентификатор, фамилия, имя, зарплата, дата рождения и отдел). Кортеж определяется как множество пар вида (имя атрибута, значение домена). Кортеж содержит 6 атрибутов, т.е. арность отношения равна 6. Таблица содержит 5 строк, т.е. в данном случае мощность отношения равна 5.

Отношение (таблица)					
Имя столбца	Атрибут (столбец, поле)	last_name	first_name	salary	birthday
	employee_id	2934	Иванов	112 000	01.01.1966
		2935	Петров	144 000	13.06.1977
		2936	Орлов	92 000	27.02.1980
		2937	Сергеева	110 000	05.05.1982
		2938	Катков	112 000	16.12.1991
					310
					313
					310
					315

Рисунок 13 – Отношение «Сотрудники»

2.3.2 Фундаментальные свойства отношений

Рассмотрим фундаментальные свойства отношений.

1. Отсутствие кортежей-дубликатов.

Дублирование данных противоречит теории множеств, т.к. в классической теории по определению каждое множество состоит из различных элементов. Дубликаты данных значительно осложняют работу с программой, т.к. отсутствует возможность редактирования значений атрибутов только у одного кортежа. Обеспечение требования уникальности строки достигается путем ввода в таблицу дополнительных уникальных атрибутов.

2. Отсутствие упорядоченности кортежей.

Свойство отсутствия упорядоченности также является следствием определения отношения как множества кортежей. Отсутствие требования к поддержанию порядка на множестве кортежей отношения дает дополнительную гибкость СУБД при хранении баз данных во внешней памяти и при выполнении запросов к базе данных. Это не противоречит тому, что при формулировании запроса к базе данных, например, на языке SQL можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых столбцов. Такой результат, вообще говоря, не отношение, а некоторый упорядоченный список кортежей.

3. Отсутствие упорядоченности атрибутов.

Атрибуты отношений, как и строки, также не упорядочены, поскольку по определению схема отношения есть множество пар $\langle A, T \rangle$. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Это свойство теоретически позволяет, например, модифицировать схемы существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих. Как и при запросе к таблице строки можно упорядочить, так и порядок вывода столбцов точно так же назначается программистом средствами SQL.

4. Атомарность значений атрибутов.

Значения всех атрибутов являются атомарными. Это свойство следует из определения домена как некоторого подмножества значений простого типа данных, т.е. среди значений домена не могут содержаться множества значений или отношения.

Атомарность означает, что на пересечении строки и столбца должно находиться единственное значение. Примером нарушения атомарности может стать попытка записи в одну ячейку таблицы группы значений, например списка. Подобный подход также затрудняет процесс поиска и обработки информации.

Принято говорить, что в реляционных базах данных допускаются только нормализованные отношения или отношения, представленные в первой нормальной форме.

Потенциальным примером ненормализованного отношения является следующее отношение «Отделы», представленное на рисунке 14.

department_id	department				
	employee_id	last_name	first_name	salary	birthday
310	2934	Иванов	Валерий	112 000	01.01.1966
	2935	Петров	Сергей	144 000	13.06.1977
	2937	Сергеева	Елена	110 000	05.05.1982
313	2936	Орлов	Петр	92 000	27.02.1980
315	2938	Катков	Георгий	112 000	16.12.1991

Рисунок 14 – Отношение «Отделы»

Здесь каждый отдел хранит в себе несколько строк с информацией о работающих в нем сотрудниках. Как видно, в этом случае на пересечении строки и столбца находится множество значений (для

отдела с номером 310), т.е. значения атрибута не являются атомарными, а отношение в таком случае является ненормализованным.

2.4 Целостность данных

База данных должна содержать некоторый набор правил, необходимых для того, чтобы данные всегда находились в согласованном состоянии.

Целостность данных – соответствие значений всех данных в базе определенному непротиворечивому набору правил.

Можно выделить три базовых класса правил, призванных поддерживать целостность данных в реляционной базе данных:

- целостность доменов;
- целостность сущностей;
- ссылочная целостность.

2.4.1 Целостность доменов

Целостность доменов поддерживается за счет механизма доменных ограничений. Это как раз тот случай, когда описание домена включает в себя некоторые логические правила, запрещающие вносить некорректные значения в атрибуты таблицы. В простейшем случае допустимые значения могут быть просто перечислены или объявлен диапазон допустимых значений.

Особая роль в поддержании доменной целостности отводится особому маркеру NULL. При работе с базами данных может возникнуть ситуация, когда часть данных неизвестна или еще не определена. Для того чтобы обойти проблему неполных или неизвестных данных, в базах данных могут использоваться типы данных, дополненные так называемым NULL-значением. NULL-значение – это, собственно, не значение, а некий маркер, показывающий, что значение неизвестно.

Основной проблемой при работе с данными, которые могут содержать NULL-значения, является необходимость использования трехзначной логики. Трехзначная логика базируется на таблицах истинности, представленных на рисунке 15. Из таблиц можно определить, что, например, выражение «Истина» и NULL дают нам NULL, так же, как и выражение «NULL или Ложь».

Таблица истинности AND				Таблица истинности OR				Таблица истинности NOT	
AND	False	True	NULL	OR	False	True	NULL	Not	
False	False	False	False	False	False	True	NULL	False	True
True	False	True	NULL	True	True	True	True	True	False
NULL	False	NULL	NULL	NULL	NULL	True	NULL	NULL	NULL

Рисунок 15 – Таблицы истинности трехзначной логики

Имеется несколько следствий применения трехзначной логики:

- NULL-значение не равно самому себе. Действительно, выражение $NULL = NULL$ дает значение не ИСТИНА (*True*), а НЕИЗВЕСТНО (*NULL*). Соответственно выражение $x = x$ не обязательно ИСТИНА;
- неверно также, что NULL-значение не равно самому себе. Выражение $NULL \neq NULL$ также принимает значение не ИСТИНА, а НЕИЗВЕСТНО. Соответственно выражение $x \neq x$ не обязательно ЛОЖЬ;
- Выражение $x \text{ OR } \text{NOT}(x)$ не обязательно ИСТИНА. Соответственно в трехзначной логике не работает принцип исключенного третьего.

2.4.2 Целостность сущностей

Одним из фундаментальных свойств отношений является отсутствие кортежей-дубликатов. На самом деле, свойством уникальности в пределах отношения могут обладать отдельные атрибуты кортежей или группы атрибутов. Такие уникальные атрибуты удобно использовать для идентификации кортежей.

Потенциальный ключ – подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности. Уникальность означает, что не существует двух кортежей данного отношения, в которых значения этого подмножества атрибутов совпадают.

Минимальность означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, удовлетворяющее условию уникальности. Т.е. в набор атрибутов потенциального ключа не должны входить такие атрибуты, которые можно отбросить без ущерба для основного свойства – однозначно определять кортеж. Потенциальный ключ, состоящий из одного атрибута, называется *простым*. Потенциальный ключ, состоящий из нескольких атрибутов, называется *составным*.

В реляционной модели данных первичный ключ – это один из потенциальных ключей отношения, выбранный в качестве основного ключа. Если в отношении имеется единственный потенциальный ключ, он является и первичным ключом. Если потенциальных ключей несколько, один из них выбирается в качестве первичного, а другие называются «альтернативными».

Целостность сущностей направлена на обеспечение внутреннего единства отдельной сущности. Вне зависимости от того, что необходимо хранить в таблицах, следует соблюдать правило – каждая строка таблицы обязана быть уникальной. Для этого нужно контролировать корректность первичного ключа отношения. Суть требования к первичному ключу проста – *во входящих в его состав атрибутах не должно содержаться ни одного определителя NULL*. Если допускается, что во входящем в состав первичного ключа столбце может находиться неопределенное значение, то это означает, что поле перестает поддерживать механизм однозначной идентификации строки таблицы.

2.4.3 Связи между кортежами отношений

Реляционная модель призвана не только описать перечень типов сущностей, но и отразить особенности взаимодействия между ними. Помимо первичных ключей, в таблицах активно применяется еще одна разновидность ключевого поля – это **внешний ключ**. Внешние ключи предназначены для реализации связи между кортежами отношений. Внешние ключи являются по существу, ссылками на атрибуты связываемых кортежей. Различают три типа связи между сущностями: «один к одному», «один ко многим» и «многие ко многим».

При выделении связи выделяют главную или родительскую таблицу и зависимую, или дочернюю таблицу.

Связь «один к одному» встречает не часто. В этом случае объекту одной сущности можно сопоставить только один объект другой сущности. Например, пусть некоторым сотрудникам выданы служебные автомобили. Тогда внешний ключ «car_id» в таблице сотрудников будет ссылаться на одноименный первичный ключ в таблице автомобилей, как показано на рисунке 16.

Нередко этот тип связей предполагает разбиение одной большой таблицы на несколько маленьких. Основная родительская таблица в этом случае продолжает содержать часто используемые данные, а дочерняя зависимая таблица обычно хранит данные, которые используются реже.

Сотрудники				Служебные автомобили		
employee_id	last_name	first_name	car_id	car_id	state_number	model
2934	Иванов	Валерий	16	16	О 752 ТР 63 rus	BMW
2935	Петров	Сергей	17	17	К 785 ОУ 63 rus	Lada
2936	Орлов	Петр	NULL	18	В 200 ОУ 63rus	KIA
2937	Сергеева	Елена	NULL			
2938	Катков	Георгий	18			

Рисунок 16 – Связь «один к одному»

На практике наиболее распространена связь «один ко многим». В этом типе связей несколько строк из дочерней таблицы зависят от одной строки в родительской таблице.

Примером такой связи может являться связь «отделы-сотрудники»: в одном отделе работают много сотрудников, при этом сотрудник может работать только в одном отделе. На рисунке 17 в дочерней таблице «Сотрудники» существует внешний ключ «`department_id`», который устанавливает связи между таблицами сотрудников и отделов. Если необходимо, например, найти сотрудников отдела кадров, можно узнать ключ этого отдела в родительской таблице «Отделы». В нашем случае это 310. И далее найти сотрудников, для которых значения соответствующего атрибута «`department_id`» равны найденному значению ключа. Таким образом, в нашем примере в отделе кадров работают три сотрудника, т.е. используется связь «один ко многим».



Рисунок 17 – Связь «один ко многим»

При типе связей «многие ко многим» одна строка из таблицы *A* может быть связана с множеством строк из таблицы *B*. В свою очередь одна строка из таблицы *B* может быть связана с множеством строк из таблицы *A*.

Для иллюстрация этого типа связей воспользуемся примером, связь «фильмы-актеры»: в одном фильме снималось много актеров,

один актер снимался во многих фильмах (рисунок 18). Для описания этих связей используется дополнительная таблица «*film_actor*». Иногда данные из этой промежуточной таблицы также могут представлять собой отдельную сущность. Используя ссылки, заданные внешними ключами, можно получить список актеров, играющих в определенном фильме.

Технически, связь многие ко многим выполнена через две связи один-ко-многим.

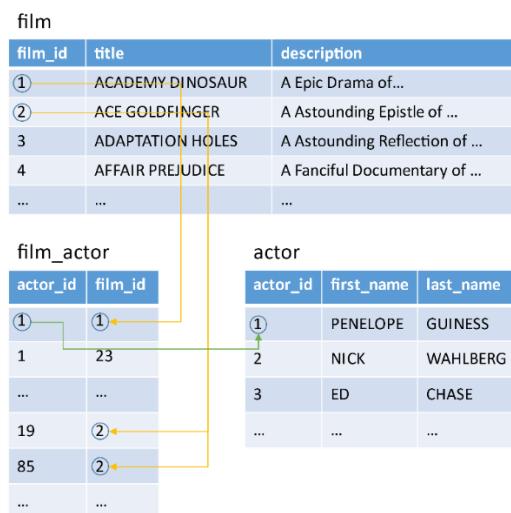


Рисунок 18 – Связь «многие ко многим»

Свойства внешних ключей:

- внешний ключ может быть простым и составным;
- внешний ключ должен быть определен на тех же доменах, что и соответствующий первичный ключ родительского отношения;
- внешний ключ, как правило, не обладает свойством уникальности. Так и должно быть, т.к. в дочернем отношении может быть несколько кортежей, ссылающихся на один и тот же

кортеж родительского отношения. Это, собственно, и дает тип отношения «один-ко-многим»;

- хотя каждое значение внешнего ключа обязано совпадать со значениями потенциального ключа в некотором кортеже родительского отношения, то обратное, вообще говоря, неверно;
- для внешнего ключа не требуется, чтобы он был компонентом некоторого потенциального ключа;
- NULL-значения для атрибутов внешнего ключа допустимы только в том случае, когда атрибуты внешнего ключа не входят в состав никакого потенциального ключа.

2.4.4 Ссылочная целостность

Т.к. внешние ключи фактически служат ссылками на кортежи в другом (или в том же самом) отношении, то эти ссылки не должны указывать на несуществующие объекты. Это определяет следующее правило целостности внешних ключей: *внешние ключи не должны быть несогласованными*, т.е. для каждого значения внешнего ключа должно существовать соответствующее значение первичного ключа в родительском отношении.

Ссылочная целостность может нарушиться в результате операций, изменяющих состояние базы данных. Таких операций три – вставка, обновление и удаление кортежей в отношениях. Т.к. в определении ссылочной целостности участвуют два отношения – родительское и дочернее, а в каждом из них возможны три операции – вставка, обновление, удаление, то нужно рассмотреть шесть различных вариантов.

Для родительского отношения:

- вставка кортежа в родительском отношении. При вставке кортежа в родительское отношение возникает новое значение потенциального ключа. Т.к. допустимо существование кортежей в родительском отношении, на которые нет ссылок из до-

черного отношения, то *вставка кортежей в родительское отношение не нарушает ссылочной целостности*;

– обновление кортежа в родительском отношении. При обновлении кортежа в родительском отношении может измениться значение потенциального ключа. Если есть кортежи в дочернем отношении, ссылающиеся на обновляемый кортеж, то значения их внешних ключей станут некорректными. *Обновление кортежа в родительском отношении может привести к нарушению ссылочной целостности*, если это обновление затрагивает значение потенциального ключа;

– удаление кортежа в родительском отношении. При удалении кортежа в родительском отношении удаляется значение потенциального ключа. Если есть кортежи в дочернем отношении, ссылающиеся на удаляемый кортеж, то значения их внешних ключей станут некорректными. *Удаление кортежей в родительском отношении может привести к нарушению ссылочной целостности*.

Для дочернего отношения:

– вставка кортежа в дочернее отношение. Нельзя вставить кортеж в дочернее отношение, если вставляемое значение внешнего ключа некорректно. *Вставка кортежа в дочернее отношение может привести к нарушению ссылочной целостности*;

– обновление кортежа в дочернем отношении. При обновлении кортежа в дочернем отношении можно попытаться некорректно изменить значение внешнего ключа. *Обновление кортежа в дочернем отношении может привести к нарушению ссылочной целостности*;

– удаление кортежа в дочернем отношении. *При удалении кортежа в дочернем отношении ссылочная целостность не нарушается*.

Существуют две основные стратегии поддержания ссылочной целостности:

- ограничить (RESTRICT);
- выполнить каскадно (CASCADE).

При использовании стратегии RESTRICT не разрешается выполнение операции, приводящей к нарушению ссылочной целостности. Это самая простая стратегия, требующая только проверки, имеются ли кортежи в дочернем отношении, связанные с некоторым кортежем в родительском отношении.

При использовании стратегии CASCADE разрешается выполнение требуемой операции, но при этом вносятся необходимые поправки в других отношениях так, чтобы не допустить нарушения ссылочной целостности и сохранить все имеющиеся связи. Изменение начинается в родительском отношении и каскадно выполняется в дочернем отношении. В реализации этой стратегии имеется одна тонкость, заключающаяся в том, что дочернее отношение само может быть родительским для некоторого третьего отношения. При этом может дополнительно потребоваться выполнение какой-либо стратегии и для этой связи и т.д. Если при этом какая-либо из каскадных операций (любого уровня) не может быть выполнена, то необходимо отказаться от первоначальной операции и вернуть базу данных в исходное состояние. Это самая сложная стратегия, но она хороша тем, что при этом не нарушается связь между кортежами родительского и дочернего отношений.

Эти стратегии являются стандартными и присутствуют во всех СУБД, в которых имеется поддержка ссылочной целостности.

Рассматривают и дополнительные стратегии поддержания ссылочной целостности:

- установить в *NULL* (SET NULL) – разрешить выполнение требуемой операции, но все возникающие некорректные значе-

ния внешних ключей изменять на NULL-значения. Эта стратегия имеет два недостатка. Во-первых, для нее требуется допустить использование NULL-значений. Во-вторых, кортежи дочернего отношения теряют всякую связь с кортежами родительского отношения. Установить, с каким кортежем родительского отношения были связаны измененные кортежи дочернего отношения, после выполнения операции уже нельзя;

- установить по умолчанию (SET DEFAULT) – разрешить выполнение требуемой операции, но все возникающие некорректные значения внешних ключей изменять на некоторое значение, принятое по умолчанию;
- игнорировать (IGNORE) – выполнять операции, не обращая внимания на нарушения ссылочной целостности. В этом случае в дочернем отношении могут появляться некорректные значения внешних ключей, и вся ответственность за целостность базы данных ложится на пользователя.

2.5 Реляционная алгебра

Реляционная модель данных основана на теории множеств и реляционной алгебре.

Реляционная алгебра представляет собой теоретический язык операций, которые на основе одного или нескольких отношений позволяют создавать другое отношение.

Операции реляционной алгебры:

- 1) выборка (selection) σ ;
- 2) проекция (projection) Π ;
- 3) декартово произведение (cartesian product) \times ;
- 4) объединение, сложение (union) U ;
- 5) вычитание, разность (set difference) $-$;
- 6) пересечение (intersection) \cap ;

- 7) деление (division) \;
- 8) соединение (join) ||.

Реляционная алгебра базируется на традиционных теоретико-множественных операциях (это пересечение, объединение, вычитание и декартово произведение), которые дополнены четырьмя операциями, специфичными для обработки реляционных данных (выборка, проекция, деление и соединение). Все эти операции обрабатывают отношения, которые (по определению) являются множествами кортежей. Кроме этих операций, в реляционную алгебру включают операцию переименования атрибутов (AS), позволяющую корректно формировать схему результирующего отношения, и операцию присваивания (=), позволяющую сохранять в базе данных результаты вычисления алгебраических выражений.

2.5.1 Выборка

При осуществлении выборки из исходного множества формируется результирующее подмножество, содержащее только удовлетворяющие определенному условию элементы.

На языке множеств выборку можно сформулировать следующим образом. Пусть существует предикат выборки F , аргументами которого выступают атрибуты реляционной таблицы, состоящей из множества кортежей R . Результатом выборки окажется подмножество кортежей R' , для которых предикат выборки F истинен:

$$R' = \sigma_F(R),$$

где F – предикат выборки, R – множество кортежей, R' – подмножество кортежей, для которых предикат выборки F истинен.

Рассмотрим пример – таблицу, хранящую данные о студентах (рисунок 19). С помощью операции выборки можно получить, например, список студентов, родившихся в 2000 году. Выборка из таблицы «Студенты» по этому условию составила три строки.

R – таблица «Студенты»				
Id	last_name	first_name	birthday	speciality
1	Орехов	Владимир	11.04.1999	Математика
2	Петровский	Александр	22.11.2000	Математика
3	Кульгина	Оксана	05.01.2000	Ин. язык
4	Самойлов	Евгений	15.07.2001	Информатика
5	Кузьмина	Ирина	02.03.2000	Физика
6	Яковенко	Константин	13.09.2001	Химия
7	Ищенко	Владимир	19.09.2002	Астрономия

R' – выборка из таблицы «Студенты»					
	2	Петровский	Александр	22.11.2000	Математика
3	Кульгина	Оксана	05.01.2000	Ин. язык	
5	Кузьмина	Ирина	02.03.2000	Физика	

Рисунок 19 – Операция «Выборка»

2.5.2 Проекция

В отличие от операции выборки, выделяющей горизонтальные элементы исходного множества, операция проекции направлена на получение вертикальной составляющей множества. Допустим, таблица R обладает n атрибутами, а нас интересует подмножество атрибутов с именами i_1, \dots, i_k , причем $k \leq n$. Тогда результатом проекции станет подмножество, которое включает в себя только определенные пользователем столбцы таблицы:

$$R' = \Pi_{i_1, i_2, \dots, i_k}(R),$$

где R – множество кортежей, n – количество атрибутов множества R , i_1, \dots, i_k – подмножество атрибутов для проекции $k \leq n$.

Для примера на рисунке 20 представлена проекция, включающая в себя только столбцы с фамилией и именем студентов.

R – таблица «Студенты»					Проекция	
Id	last_name	first_name	birthday	speciality	Орехов	Владимир
1	Орехов	Владимир	11.04.1999	Математика	Петровский	Александр
2	Петровский	Александр	22.11.2000	Математика	Кульгина	Оксана
3	Кульгина	Оксана	05.01.2000	Ин. язык	Самойлов	Евгений
4	Самойлов	Евгений	15.07.2001	Информатика	Кузьмина	Ирина
5	Кузьмина	Ирина	02.03.2000	Физика	Яковенко	Константин
6	Яковенко	Константин	13.09.2001	Химия	Ищенко	Владимир
7	Ищенко	Владимир	19.09.2002	Астрономия		

Рисунок 20 – Операция «Проекция»

2.5.3 Декартово произведение

Декартово произведение открывает набор операций, осуществляемых с двумя множествами кортежей R и S .

В декартовом произведении кортежи результирующего отношения формируются путем попарного соединения всех кортежей отношений-операндов R и S . Арность результирующего отношения будет равной сумме арностей всех перемножаемых отношений-операндов, а мощность – произведению их мощностей.

$$R' = R \times S.$$

Кортежи результирующего отношения R' формируются путем попарного соединения (конкатенации, или сцепления) всех кортежей отношений-операндов R и S . Т.е. в результате декартова произведения к каждой строке из одной таблицы R присоединяется одна строка из другой таблицы S . На практике подобная операция редко когда оказывается востребованной, ведь в результирующем множестве оказываются все возможные сочетания строк из двух таблиц.

В качестве примера можно привести следующий запрос. Допустим, что студенты хотят получить данные о том, какие учебные дисциплины им предстоит изучать (рисунок 21).



Рисунок 21 – Операция «Декартово произведение»

В примере каждая строка из таблицы студентов последовательно соединяется с каждой строкой из таблицы дисциплин для формирования итогового отношения. Арность итогового отношения, т.к. количество столбцов таблицы, равно 3 – сумме арностей отношений-операндов. Мощность результирующего отношения, т.е. количество строк в таблице, равна 12.

2.5.4 Объединение

Операция объединения проводится только с отношениями R и S , совместимыми по объединению, например с идентичной структурой. В результате объединения будет получено результирующее отношение, содержащую кортежи из двух исходных отношений.

$$R' = R \cup S.$$

Результирующее отношение R' включает все кортежи, входящие хотя бы в одно из отношений-операндов R или S .

Представленная на рисунке 22 операция объединения суммирует записи из таблиц «Студенты 1» и «Студенты 2» в результирующее отношение. Стоит заметить, что если исходные таблицы содержат одинаковые по содержанию строки, то в результате операции объединения из итогового множества должны быть удалены записи-дубликаты.



Рисунок 22 – Операция «Объединение»

2.5.5 Вычитание

При вычитании результирующее отношение включает все кортежи, входящие в отношение-операнд R , такие, что ни один из них не входит в отношение-операнд S .

$$R' = R \setminus S.$$

Операция вычитания позволяет выяснить, какие из строк первой таблицы отсутствуют во второй таблице. Как и в случае объединения, разность работает только с отношениями, совместимыми по объединению (т.е. с одинаковой структурой).

Рассмотрим операцию вычитания на примере. Предположим, что у нас есть две таблицы R и S , в первой находятся данные о студентах учебной группы, а во второй – данные о студентах, успешно сдавших экзаменационную сессию. Операция вычита позволит найти студентов, которым следует явиться на пересдачу (рисунок 23).

R – «Студенты 1»		S – «Студенты 2»		$R - S$
$last_name$	$first_name$	$last_name$	$first_name$	
Орехов	Владимир	Орехов	Владимир	
Петровский	Александр	Петровский	Александр	
Кульгина	Оксана	Кульгина	Оксана	
Самойлов	Евгений	Самойлов	Евгений	
Кузьмина	Ирина	Кузьмина	Ирина	
Яковенко	Константин	Ищенко	Владимир	
Ищенко	Владимир	Алфёров	Николай	
Алфёров	Николай	Яковенко	Константин	

Рисунок 23 – Операция «Вычитание»

2.5.6 Пересечение

При пересечении результирующее отношение R' включает в себя все кортежи, входящие в оба отношения-операнда R и S .

$$R' = R \cap S.$$

Операция пересечения может проводиться с таблицами, совместимыми по объединению. Допустим, что нам следует узнать, какие из студентов успешно сдали экзамен по высшей математике (таблица R) и экзамен по СУБД (таблица S). Пересечение двух отношений R и S предоставит нам запрашиваемый результат, т.е. будут найдены строки, содержащиеся в обеих таблицах (рисунок 24).

R – «Студенты 1»		S – «Студенты 2»		$R \cap S$	
<i>last_name</i>	<i>first_name</i>	<i>last_name</i>	<i>first_name</i>	<i>last_name</i>	<i>first_name</i>
Орехов	Владимир	Бобров	Семен	Петровский	Александр
Петровский	Александр	Петровский	Александр	Кульгина	Оксана
Кульгина	Оксана	Кульгина	Оксана	Самойлов	Евгений
Самойлов	Евгений	Самойлов	Евгений	Кузьмина	Ирина
Кузьмина	Ирина	Кузьмина	Ирина	Ищенко	Владимир
Яковенко	Константин	Ищенко	Владимир	Алфёров	Николай
Щукин	Александр	Алфёров	Николай		
Алфёров	Николай				

Рисунок 24 – Операция «Пересечение»

2.5.7 Деление

Результат деления исходной таблицы R на таблицу-делитель S будет содержать столбцы, отсутствующие в делителе. В качестве строк в результат войдут только те записи из делителя S , что при декартовом произведении результата на делитель содержатся в делитом R .

$$R' = R/S.$$

Рассмотрим операцию деления проще на примере (рисунок 25). Допустим, в таблице R содержатся данные о том, какие виды занятий определенных дисциплин ведут преподаватели университета. Нам требуется узнать, кто ведет лабораторные работы по СУБД и практические занятия по языкам программирования. Для этого

в таблицу S заносятся две строки с указанными данными и осуществляется деление.

Результат деления исходной таблицы R на таблицу-делитель S , во-первых, будет содержать столбцы, отсутствующие в делителе (в нашем случае это атрибут фамилии «`last_name`»). Во-вторых, в качестве строк в результат войдут только те записи, что при декартовом произведении результата на делитель содержатся в делимом R .

Таким образом, после деления будут получены две строки с фамилиями Орлов и Володина. Преподаватель Семенов не попал в результирующее отношение, так как он не ведет лабораторные по дисциплине СУБД.

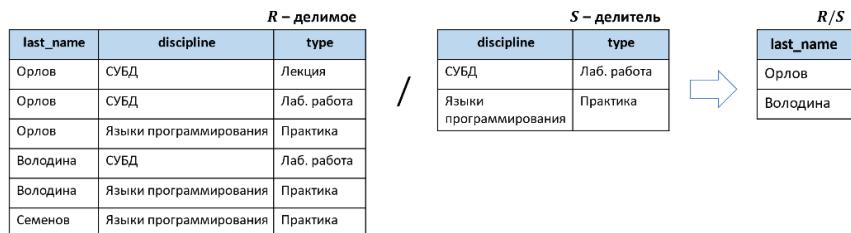


Рисунок 25 – Операция «Деление»

2.5.8 Соединение

Операция соединения – наиболее важная реляционная операция.

$$R' = R \bowtie S.$$

Кортежи результирующего отношения R' образуются путем соединения (конкатенации, или сцепления) кортежей отношений-операндов R и S , удовлетворяющих заданному условию (любому корректному логическому выражению). Выполнение операции соединения отношений можно рассматривать как операцию их расширенного декартова произведения с последующей фильтрацией множе-

ства кортежей полученного промежуточного отношения по заданному условию.

Допустим, в нашем распоряжении имеются две таблицы, в таблице S хранится информация о кафедрах вуза, а в таблице R – информация о преподавателях (рисунок 26). Продемонстрируем наиболее распространенную операцию естественного соединения. В этом случае таблицы соединяются по общему столбцу: в таблице кафедр S это первичный ключ CHAIR_ID, а в таблице преподавателей R это одноименный столбец внешнего ключа. Т.е. условием соединения в данном случае является совпадений значений атрибутов в столбцах chair_id. В примере преподаватели Орлов и Володина соединились по ключу со значением 1 со значением кафедры «Геоинформатика», Семенов – с кафедрой «Физика», Архипов и Андреев – с кафедрой «Высшая математика». В результирующей таблице не оказалось кафедры химии. Это особенность естественного соединения. Т.к. в таблице R нет ни одной записи с внешним ключом, ссылающимся на кафедру химии (т.е. нет записи с ключом 4), то кафедра химии не попала в итоговый результат.

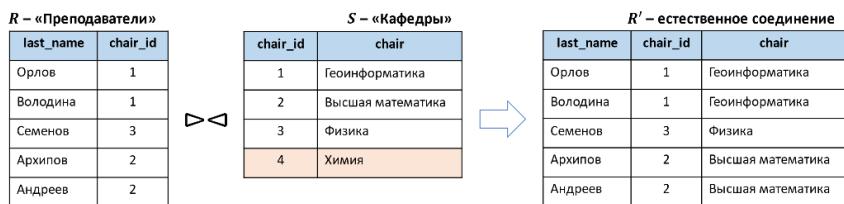


Рисунок 26 – Операция «Естественное соединение»

Если же нам требуется получить все кортежи из отношений, то вместо использования естественного соединения необходимо применить внешнее соединение. Различают правое и левое внешние соединения. На рисунке 27 представлено правое внешнее соединение. Правым внешним соединением называется соединение, при котором кортежи отношения кафедр S (таблица расположена справа), не

имеющие совпадающих значений в общих столбцах отношения преподавателей R (таблице слева), также включаются в результирующее отношение. Недостающие значения из левого отношения будут заменены пустым значением NULL. Для левого внешнего соединения, наоборот, в результирующее отношение включаются все кортежи из левого отношения, а недостающие значения из правого заменяются NULL'ом.

В примере на рисунке 27 для кафедры «Химия» нет записей о преподавателях, поэтому кортеж содержит значение NULL для атрибута last_name.



Рисунок 27 – Операция «Правое внешнее соединение»

В реляционной модели определено несколько разновидностей операции соединения:

- внутреннее соединение (`INNER JOIN`), при котором соединяются только те кортежи отношений-операндов, для которых выполняется заданное условие;
- левое (`LEFT JOIN`) и правое (`RIGHT JOIN`) внешние соединение, при котором результирующее отношение будет безусловно содержать все кортежи левого (или соответственно правого) отношения-операнда, в том числе и те, для которых нет «пары» в другом отношении-операнде, при этом «недостающие» атрибуты в таких кортежах результирующего отношения получат неопределенные `NULL`-значения;

- внешнее соединение (OUTER JOIN) – это одновременно и левое, и правое соединение;
- экви-соединение (EQUAL JOIN) – такое соединение, условие которого содержит оператор сравнения «равно»;
- естественное соединение (NATURAL JOIN) – это экви-соединение двух отношений, имеющих одинаковые атрибуты (как правило, это первичный и внешний ключи соединяемых отношений), равенство которых и является условием соединения кортежей (при этом совпадающий атрибут в схеме результирующего отношения не дублируется).

3 ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

3.1 Понятие информационной системы

По своей сути любая база данных представляет собой хранилище структурированных данных. Само по себе хранилище особого интереса не представляет до тех пор, пока оно не интегрируется в состав более сложной надсистемы, называемой информационной.

Информационная система (Information System) – это система, предназначенная для хранения, поиска и обработки информации.

Процесс создания информационной системы в общем случае включает этапы:

- 1) планирование системы;
- 2) анализ задач системы и требований к системе;
- 3) проектирование системы;
- 4) реализация и ввод системы в эксплуатацию;
- 5) сопровождение.

База данных – хотя и ключевая, но далеко не единственная составная часть информационной системы. Помимо базы данных, в типичную информационную систему входят: программное обеспечение поддержки базы данных, аппаратное обеспечение, прикладное программное обеспечение и эксплуатирующий систему персонал.

База данных не только функционирует внутри информационной системы, но и развивается вместе с этой системой, поэтому в начале 80-х годов был введен термин *жизненный цикл базы данных*. Жизненный цикл базы данных – это процесс проектирования, реализации и поддержки базы данных.

На рисунке 28 представлены основные этапы жизненного цикла БД.

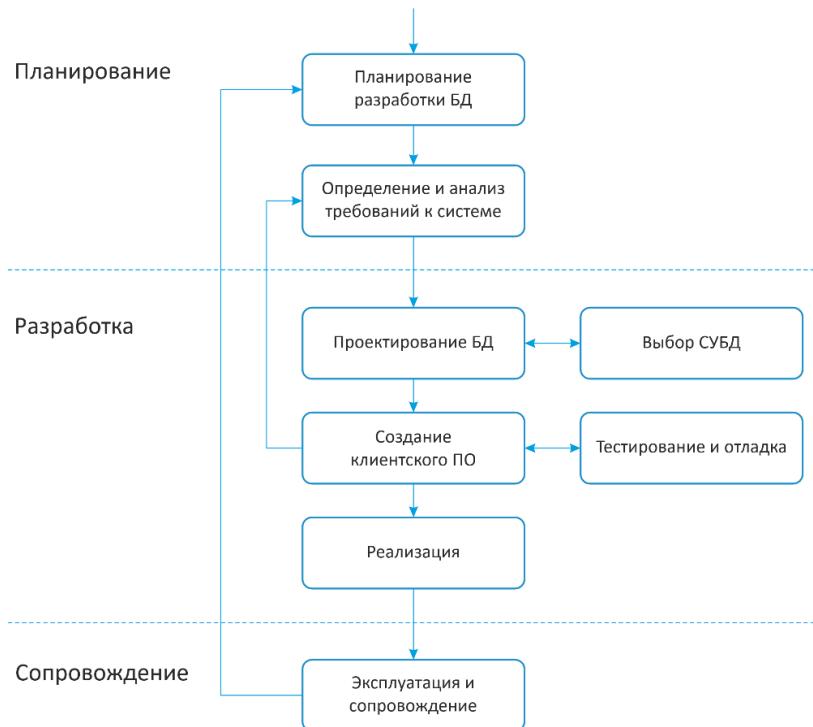


Рисунок 28 – Жизненный цикл БД

На этапе планирования разработки производится оценка объема работы, времени на разработку и реализацию проекта и стоимости работ.

Следующим этапом является определение и анализ требований к системе. На этом этапе должна быть собрана информация, необходимая для проектирования базы данных. Эта информация может включать, в т.ч., цели и задачи компаний, решаемые информационной системы, бизнес-процессы компании и т.д.

Далее выполняется проектирование, включающее в себя концептуальное, логическое и физическое проектирование базы данных.

Завершив работу над формированием физической структуры БД, проектировщик переходит к этапу разработки приложений, предназначенных для работы с БД.

На этапе реализации осуществляется выпуск окончательной версии программного продукта.

Эксплуатация и сопровождение – это заключительный этап жизненного цикла БД, в ходе которого осуществляется развертывание базы данных и прикладного ПО на серверах и клиентских станциях заказчика.

3.2 Проектирование базы данных

Этап проектирования базы данных в свою очередь обычно разделяется на три фазы (рисунок 29) [11]:

- 1) концептуальное проектирование;
- 2) логическое проектирование;
- 3) физическое проектирование.



Рисунок 29 – Этап проектирования БД

На этапе концептуального и логического проектирования нас не интересует конкретная СУБД. Особенности выбранной СУБД учитываются только при физическом проектировании базы данных.

3.2.1 Концептуальное проектирование

На этапе концептуального проектирования формируется вид будущей базы данных, но без особенностей ее физической реализации. Основным средством построения концептуальной модели БД выступает модель ER-модель «сущность-связь» или родственные ей модели.

Понятия «ER-модель» и «ER-диаграмма» часто не различают, хотя для визуализации ER-моделей могут быть использованы и другие графические нотации, либо визуализация может вообще не применяться (например, использоваться текстовое описание). Основные графические нотации включают в себя нотацию Чена, нотацию Crow's Foot, IDEF1X и другие.

Результатом фазы концептуального проектирования является ER-модель БД, включающая в себя описание:

- типов сущностей;
- связей между типами сущностей;
- атрибутов (желательно с предварительным описанием доменов и ограничений);
- первичных ключей.

3.2.2 Логическое проектирование

Фаза логического проектирования предназначена для преобразования обобщенной концептуальной модели в завершенную логическую.

На этом этапе уточняются требования, выявленные на концептуальной стадии проектирования, и выполняется упрощение модели (без снижения функциональных возможностей).

Для этого предварительная ER-модель проверяется с помощью правил нормализации. В результате будут получены не избыточные реляционные таблицы, свободные от присущих ненормализованным данным аномалий вставки, редактирования и удаления.

Помимо нормализации, на логическом этапе осуществляются следующие действия:

- уточняются ограничения на данные;
- определяются домены данных;
- вводятся бизнес-правила и корпоративные ограничения целостности;
- определяется местоположение будущих таблиц (в случае если речь идет о распределенной БД).

На этапах концептуального и логического проектирования обычно пользуются одной из двух стратегий проектирования БД: стратегией *восходящего проектирования* (bottom-up design) или стратегией *нисходящего проектирования* (top-down design).

Суть восходящего метода заключается в том, что сначала формируется полный список атрибутов (столбцов таблиц), подлежащих хранению. Позднее атрибуты группируются в типы сущностей, которые попадают в модель, т.е. выполняется процесс нормализации. Этот способ подходит обычно для сравнительно небольших проектов.

В нисходящей стратегии проектирование начинается с выявления основных типов сущностей, и затем для сущностей определяются атрибуты. Классический пример нисходящего метода – модель сущность-связь (ER-модель).

На практике моделирование структуры базы данных, особенно большой, при помощи алгоритма нормализации имеет серьезные недостатки. Первоначальное размещение всех атрибутов в одном отношении является очень неестественной операцией, т.к. интуи-

тивно сразу выделяется несколько отношений в соответствии с обнаруженными сущностями. Кроме того, часто невозможно сразу определить полный список атрибутов и выделить все зависимости, что необходимо для выполнения нормализации.

Поэтому на практике обычно выполняется проектирование с помощью ER-модели, к которой применяют нормализацию для уточнения отношений.

3.2.3 Физическое проектирование

Физическое проектирование – это уточнение решения с учетом имеющихся в наличии технологий, возможности реализации и требуемой производительности.

Целью физического проектирования становится перенос логической модели на платформу выбранной СУБД. Для этого необходимо:

- создать таблицы и связи между ними;
- создать вторичные индексы таблиц;
- разработать представления;
- реализовать бизнес-логику БД (в первую очередь, с помощью триггеров и хранимых процедур);
- определить функциональные характеристики транзакций;
- внедрить механизмы защиты (как минимум, предусмотреть авторизацию пользователей и назначить права доступа к данным).

Таким образом, отношения, разработанные на стадии формирования логической модели данных, преобразуются в таблицы, атрибуты становятся столбцами таблиц, для ключевых атрибутов создаются уникальные индексы, домены преображаются в типы данных, принятые в конкретной СУБД и т.д.

3.3 ER-моделирование

3.3.1 Основные понятия

Рассмотрим процесс концептуального и логического проектирование с использованием исходящего подхода на основе ER-моделирования. В этом подходе проектирование начинается с выявления основных типов сущностей, и затем для сущностей определяются атрибуты.

Начнем с основных понятий.

Сущность – это класс однотипных объектов, информация о которых должна быть учтена в модели. Примерами сущностей могут быть такие классы объектов как «Поставщик», «Сотрудник», «Товар».

Экземпляр сущности – это конкретный представитель данной сущности. Например, представителем сущности «Сотрудник» может быть «Сотрудник Иванов». Экземпляры сущностей должны быть различимы, т.е. сущности должны иметь некоторые свойства, уникальные для каждого экземпляра этой сущности.

Атрибут сущности – это именованная характеристика, являющаяся некоторым свойством сущности. Примерами атрибутов сущности «Сотрудник» могут быть такие атрибуты как «Табельный номер», «Фамилия», «Имя», «Отчество», «Должность», «Зарплата» и т.п.

Сущность и атрибуты сущности в классическом способе построения ER-диаграмм отображаются в виде прямоугольника с наименованием и полями-атрибутами, как показано на рисунке 30.

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.

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

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

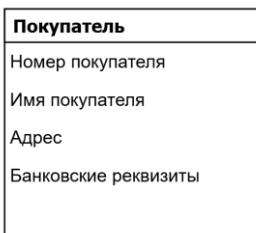


Рисунок 30 – Сущность в ER-диаграмме

Различают 3 типа связей: один к одному, один ко многим и многие ко многим. В связи «один к одному» один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой). 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. Связь типа «многие 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 «многие-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.

Кроме того, каждая связь может иметь одну из двух модальностей связи. 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.

Способ отображения связей на ER-диаграмме показан на рисунке 31.

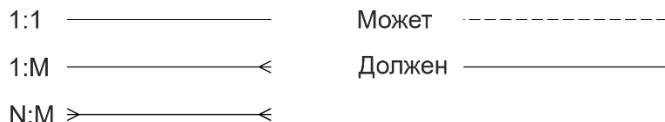


Рисунок 31 – Связи в ER-диаграмме

3.3.2 Пример разработки ER-модели

При разработке ER-моделей необходимо 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. После разработки первого приближенного варианта диаграммы происходит его уточнение путем опроса экспертов предметной области.

Рассмотрим построение ER-диаграммы на примере. Предположим, что перед нами стоит задача разработать информационную систему по заказу некоторой оптовой торговой фирмы, которая должна выполнять следующие действия:

- хранить информацию о покупателях;
- печатать накладные на отпущенные товары;
- следить за наличием товаров на складе.

Выделим потенциальные кандидаты на сущности и атрибуты и проанализируем их. Допустим, были получены следующие варианты:

- покупатель – явный кандидат на сущность;
- накладная – явный кандидат на сущность;
- товар – явный кандидат на сущность;

- (?) склад – надо уточнить, 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 показано на рисунке 32.

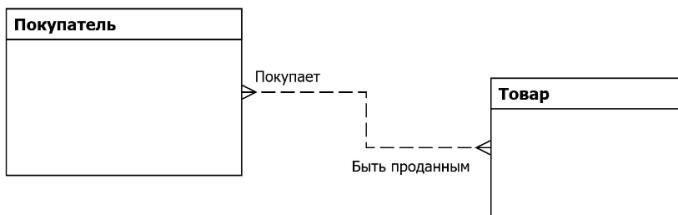


Рисунок 32 – Вариант ER-диаграммы

Далее, было выяснено, что фирма имеет несколько складов. Причем, каждый товар может храниться на нескольких складах и быть проданным с любого склада. Возникает вопрос куда поместить сущности «Накладная» и «Склад» и с чем их связать?

Допустим, что существует следующий процесс. 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 так, как показано на рисунке 33.

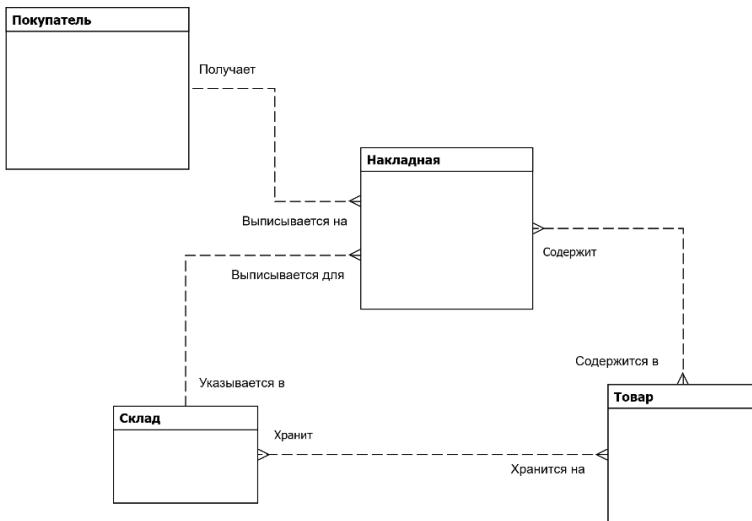


Рисунок 33 – Уточненный вариант ER-диаграммы

На следующем этапе построения ER-модели были определены атрибуты, которые должны храниться в базе данных. Кроме того, необходимо было разделить связи «многие ко многим» на 2 связи типа «один ко многим» с использованием дополнительной сущности. Для этого было создано 2 сущности – «Запись в накладной», чтобы убрать связь между накладной и товарами, и сущность «Товар на складе», чтобы убрать связь «Склад-Товар».

Результат внесения изменения в диаграмму представлен на рисунке 34.

Показанный на рисунке 34 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. На рисунке 35 представлен физический вариант диаграммы, выполненный средствами MySQL Workbench.

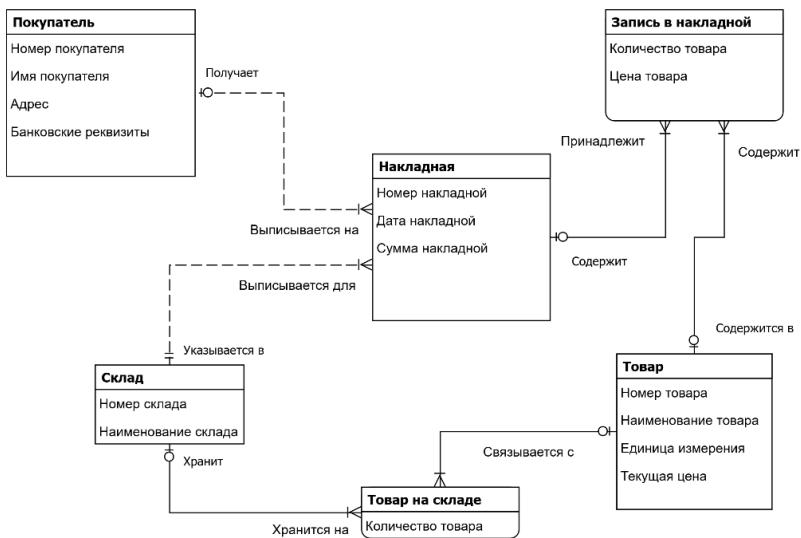


Рисунок 34 – Итоговый вариант ER-диаграммы

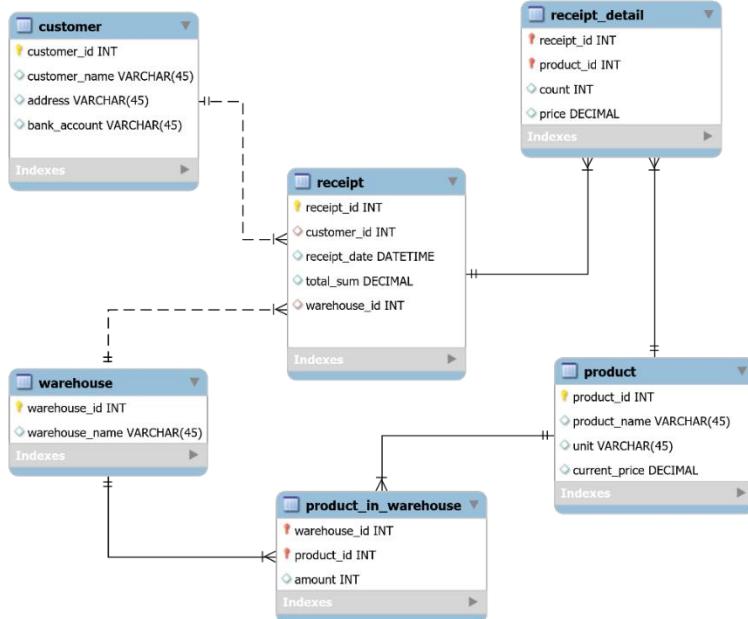


Рисунок 35 – Физический вариант ER-диаграммы

На данной диаграмме каждая сущность представляет собой таблицу базы данных, каждый атрибут становится колонкой соответствующей таблицы. Обратите внимание на то, что во многих таблицах, например, «`receipt_detail`» и «`product_in_warehouse`», соответствующих сущностям «Запись списка» 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.

Полученные таблицы нормализованы и не содержат избыточных данных.

3.4 Нормализация отношений

Процесс нормализации и ER-моделирование выступают разными сторонами одной медали. Оба подхода приводят к созданию модели БД, с той лишь разницей, что ER-моделирование осуществляется на концептуальном уровне и основано на принципе от общего к частному (так называемый нисходящий подход). Нормализация производится на этапе логического проектирования и, в отличие от ER-моделирования, использует идеи восходящего подхода (от частного к общему).

3.4.1 Пример предметной области

Для изучения процесса нормализации отношений, рассмотрим в качестве предметной области организацию, выполняющую некоторые проекты. Модель предметной области опишем следующим неформальным текстом:

- 1) сотрудники организации выполняют проекты;
- 2) проекты состоят из нескольких задач;

- 3) каждый сотрудник может участвовать в одном или нескольких проектах, или временно не участвовать ни в каких проектах;
- 4) над каждым проектом может работать несколько сотрудников, или временно проект может быть приостановлен, тогда над ним не работает ни один сотрудник;
- 5) над каждой задачей в проекте работает ровно один сотрудник;
- 6) каждый сотрудник числится в одном отделе;
- 7) каждый сотрудник имеет телефон, находящийся в отделе сотрудника.

В ходе дополнительного уточнения того, какие данные необходимо учитывать, выяснилось следующее:

- 1) о каждом сотруднике необходимо хранить табельный номер и фамилию. Табельный номер является уникальным для каждого сотрудника;
- 2) каждый отдел имеет уникальный номер;
- 3) каждый проект имеет номер и наименование. Номер проекта является уникальны;
- 4) каждая задача из проекта имеет номер, уникальный в пределах проекта. Задачи в разных проектах могут иметь одинаковые номера.

Отчетная форма, сформированная по неформально описанной модели предметной области, может иметь вид, как показано в таблице 3.1.

Как видно из таблицы, все атрибуты записаны в одну форму. Следует отметить, что данная форма не является отношением, т.к. не выполняется свойство атомарности данных. В примере один проект содержит несколько строк с информацией о сотрудниках.

Таблица 3.1. Отчетная форма

Номер проекта	Название проекта	Номер сотрудника	Фамилия сотрудника	Номер отдела	Телефон	Номер задания
1	Космос	1	Иванов	1	11-22-33	1
		2	Петров	1	11-22-33	2
		3	Сидоров	2	33-22-11	3
2	Климат	3	Сидоров	2	33-22-11	2
		1	Иванов	1	11-22-33	1

Перейдем к нормализации описанных данных.

3.4.2 Первая нормальная форма

Переменная отношения находится в **первой нормальной форме** (1НФ) тогда и только тогда, когда в любом допустимом значении этой переменной каждый кортеж отношения содержит только одно значение для каждого из атрибутов.

Согласно определению отношений, любое отношение автоматически уже находится в 1НФ согласно свойствам отношений:

- в отношении нет одинаковых кортежей;
- кортежи не упорядочены;
- атрибуты не упорядочены и различаются по наименованию;
- все значения атрибутов атомарны.

Данные свойства и являются свойствами первой нормальной формы.

Если упростить определение, таблица приведена к первой нормальной форме, если в ней отсутствуют повторяющиеся группы, и все значения, хранимые в ней, неделимы (атомарны).

Вернемся к примеру. В ходе логического моделирования на первом шаге предложено хранить данные в одном сводном отношении, имеющем следующие атрибуты:

СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ (

 НОМЕР_СОТРУДНИКА, ФАМИЛИЯ, НОМЕР_ОТДЕЛА, ТЕЛЕФОН,

 НОМЕР_ПРОЕКТА, ПРОЕКТ, НОМЕР_ЗАДАЧИ
) ,

где

- НОМЕР_СОТРУДНИКА – табельный номер сотрудника;
- ФАМИЛИЯ – фамилия сотрудника;
- НОМЕР_ОТДЕЛА – номер отдела, в котором числится сотрудник;
- ТЕЛЕФОН – телефон сотрудника;
- НОМЕР_ПРОЕКТА – номер проекта, над которым работает сотрудник;
- ПРОЕКТ – наименование проекта, над которым работает сотрудник;
- НОМЕР_ЗАДАЧИ – номер задачи, над которой работает сотрудник.

Потенциальный ключ – подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности (несократимости).

Т.к. каждый сотрудник в каждом проекте выполняет ровно одну задачу, то в качестве потенциального ключа отношения необходимо взять пару атрибутов {НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТА}.

Отношение в 1НФ показано в таблице 3.2.

Таблица 3.2. Отношение в 1НФ

НОМЕР_СОТРУДНИКА	ФАМИЛИЯ	НОМЕР_ОТДЕЛА	ТЕЛЕФОН	НОМЕР_ПРОЕКТА	ПРОЕКТ	НОМЕР_ЗАДАЧИ
1	Иванов	1	11-22-33	1	Космос	1
1	Иванов	1	11-22-33	2	Климат	1
2	Петров	1	11-22-33	1	Космос	2
3	Сидоров	2	33-22-11	1	Космос	3
3	Сидоров	2	33-22-11	2	Климат	2

Данные в отношении СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ хранятся с большой избыточностью. Во многих строках повторяются фамилии сотрудников, номера телефонов, наименования проектов. Кроме того, в данном отношении хранятся вместе независимые друг от друга данные – и данные о сотрудниках, и об отделах, и о проектах, и о работах по проектам. При изменении состояния базы данных это может привести к ряду проблем. Исторически эти проблемы получили название аномалии обновления.

Аномалия обновления – неадекватность модели данных предметной области или необходимость дополнительных усилий для реализации всех ограничений, определенных в предметной области.

Т.к. аномалии проявляют себя при выполнении операций, изменяющих состояние базы данных, то различают следующие виды аномалий:

- аномалии вставки (INSERT);
- аномалии обновления (UPDATE);
- аномалии удаления (DELETE).

К примеру, в отношение нельзя добавить запись о сотруднике, который пока не участвует ни в одном проекте, т.к. нельзя добавить кортеж со значением NULL атрибута НОМЕР_ПРОЕКТА, поскольку этот атрибут входит в состав потенциального ключа, и, следовательно, не может содержать NULL-значений. Также, из-за дублирования информации невозможно реализовать обновление информации одним действием (например, при изменении названия проекта потребуется менять значения атрибута в нескольких кортежах). При удалении некоторых данных может произойти потеря другой информации.

Учитывая указанные недостатки, можно сделать вывод, что логическая модель данных неадекватна модели предметной области. База данных, основанная на такой модели, будет работать неправильно.

3.4.3 Вторая нормальная форма

Для устранения указанных аномалий необходима дальнейшая нормализация, которая основана на понятии функциональной зависимости атрибутов отношения.

Переменная отношения находится во **второй нормальной форме** (2НФ) тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут неприводимо зависит от (каждого) её потенциального ключа. **Неключевой атрибут** – это атрибут, не входящий в состав никакого потенциального ключа.

Или проще, таблица приведена ко второй нормальной форме, если она соответствует 1НФ и в ней нет частичных зависимостей, т.е. нет неключевых столбцов, зависящих от части первичного ключа.

Если потенциальный ключ является простым, то есть состоит из единственного атрибута, то любая функциональная зависимость

от него является неприводимой (полной). Если потенциальный ключ является составным, то, согласно определению второй нормальной формы, в отношении не должно быть неключевых атрибутов, зависящих от части составного потенциального ключа.

Другими словами, у таблицы в 2НФ не должно быть атрибутов, зависящих только от части первичного ключа.

Рассмотрим отношение в 1НФ, представленное в таблице 3.2. Можно выделить следующие функциональные зависимости:

1) Зависимость атрибутов от **ключа отношения**:

- { НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ } → ФАМИЛИЯ;
- { НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ } → НОМЕР_ОТДЕЛА;
- { НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ } → ТЕЛЕФОН;
- { НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ } → ПРОЕКТ;
- { НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ } → НОМЕР_ЗАДАЧИ.

2) Зависимость атрибутов от **номера сотрудника**:

- НОМЕР_СОТРУДНИКА → ФАМИЛИЯ;
- НОМЕР_СОТРУДНИКА → НОМЕР_ОТДЕЛА;
- НОМЕР_СОТРУДНИКА → ТЕЛЕФОН.

3) Зависимость атрибутов от **номера проекта**:

- НОМЕР_ПРОЕКТА → ПРОЕКТ.

4) Зависимость атрибутов от **номера отдела**:

- НОМЕР_ОТДЕЛА → ТЕЛЕФОН.

Отношение не находится в 2НФ, т.к. есть атрибуты, зависящие от части сложного ключа. Для того, чтобы устраниТЬ зависимость

атрибутов от части сложного ключа, нужно произвести **декомпозицию** отношения на несколько отношений. При этом *те атрибуты, которые зависят от части сложного ключа*, выносятся в отдельное отношение.

Для устранения функциональных зависимостей отношение СОТРУДНИКИ_ОТДЕЛЫ_ПРОЕКТЫ декомпозировано на три отношения: СОТРУДНИКИ_ОТДЕЛЫ, ПРОЕКТЫ, ЗАДАЧИ.

Отношение СОТРУДНИКИ_ОТДЕЛЫ (НОМЕР_СОТРУДНИКА, ФАМИЛИЯ, НОМЕР_ОТДЕЛА, ТЕЛЕФОН) показано в таблице 3.3.

Таблица 3.3. Отношение СОТРУДНИКИ_ОТДЕЛЫ

НОМЕР_СОТРУДНИКА	ФАМИЛИЯ	НОМЕР_ОТДЕЛА	ТЕЛЕФОН
1	Иванов	1	11-22-33
2	Петров	1	11-22-33
3	Сидоров	2	33-22-11

В отношении СОТРУДНИКИ_ОТДЕЛЫ можно выделить следующие функциональные зависимости:

- 1) Зависимость атрибутов от номера сотрудника:
 - НОМЕР_СОТРУДНИКА → ФАМИЛИЯ;
 - НОМЕР_СОТРУДНИКА → НОМЕР_ОТДЕЛА;
 - НОМЕР_СОТРУДНИКА → ТЕЛЕФОН.
- 2) Зависимость атрибутов от номера отдела:
 - НОМЕР_ОТДЕЛА → ТЕЛЕФОН.

Отношение ПРОЕКТЫ (НОМЕР_ПРОЕКТА, ПРОЕКТ) показано в таблице 3.4.

Таблица 3.4. Отношение ПРОЕКТЫ

НОМЕР_ПРОЕКТА	ПРОЕКТ
1	Космос
2	Климат

В отношении ПРОЕКТЫ можно выделить функциональную зависимость НОМЕР_ПРОЕКТА → ПРОЕКТ.

Отношение ПРОЕКТЫ (НОМЕР_ПРОЕКТА, ПРОЕКТ) показано в таблице 3.5.

Таблица 3.5. Отношение ЗАДАЧИ

НОМЕР_СОТРУДНИКА	НОМЕР_ПРОЕКТА	НОМЕР_ЗАДАЧИ
1	1	1
1	2	1
2	1	2
3	1	3
3	2	2

В отношении ЗАДАЧИ можно выделить функциональную зависимость {НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ} → НОМЕР_ЗАДАЧИ.

Отношения, полученные в результате декомпозиции, находятся в 2НФ. Отношения СОТРУДНИКИ_ОТДЕЛЫ и ПРОЕКТЫ имеют простые ключи, следовательно, автоматически находятся в 2НФ, отношение ЗАДАНИЯ имеет сложный ключ, но единственный неключевой атрибут НОМЕР_ЗАДАЧИ функционально зависит от всего ключа {НОМЕР_СОТРУДНИКА, НОМЕР_ПРОЕКТ}.

Часть аномалий устранена, однако, все еще возможны аномалии вставки, обновления и удаления. Например, в отношение СОТРУДНИКИ_ОТДЕЛЫ нельзя вставить кортеж (4, Семенов, 1, 33-22-11), т.к. при этом получится, что два сотрудника из одного отдела имеют разные номера телефонов, а это противоречит модели предметной области. Также, из-за дублирования информации (номера телефонов), обновление базы данных одним действием реализовать невозможно. При удалении некоторых данных по-прежнему может произойти потеря другой информации.

3.4.4 Третья нормальная форма

Атрибуты называются взаимно независимыми, если ни один из них не является функционально зависимым от другого.

Отношение находится в **третьей нормальной форме** (ЗНФ) тогда и только тогда, когда отношение находится во второй нормальной форме и все неключевые атрибуты взаимно независимы.

Иначе, таблица приведена к третьей нормальной форме, если она соответствует 2НФ и в ней отсутствуют транзитивные зависимости.

Транзитивная зависимость $A \rightarrow B \rightarrow C$ устраняется единственным способом – за счет выделения атрибутов $B \rightarrow C$ (или $A \rightarrow B$) в другую таблицу. Т.е. для того, чтобы устраниТЬ зависимость неключевых атрибутов, необходимо произвести декомпозицию отношения на несколько отношений. При этом те неключевые атрибуты, которые являются зависимыми, выносятся в отдельное отношение.

Отношение СОТРУДНИКИ_ОТДЕЛЫ, представленное в таблице 3.3, не находится в ЗНФ, т.к. имеется функциональная зависимость неключевых атрибутов (зависимость номера телефона от номера отдела). Для приведения отношения в третью нормальную форму, декомпозирируем отношение СОТРУДНИКИ_ОТДЕЛЫ на два отношения: СОТРУДНИКИ, ОТДЕЛЫ.

Отношение СОТРУДНИКИ (НОМЕР_СОТРУДНИКА, ФАМИЛИЯ, НОМЕР_ОТДЕЛА) показано в таблице 3.6.

Таблица 3.6. Отношение СОТРУДНИКИ

НОМЕР_СОТРУДНИКА	ФАМИЛИЯ	НОМЕР_ОТДЕЛА
1	Иванов	1
2	Петров	1
3	Сидоров	2

В отношении СОТРУДНИКИ можно выделить следующие функциональные зависимости:

- НОМЕР_СОТРУДНИКА → ФАМИЛИЯ;
- НОМЕР_СОТРУДНИКА → НОМЕР_ОТДЕЛА.

Отношение ОТДЕЛЫ (НОМЕР_ОТДЕЛА, ТЕЛЕФОН) показано в таблице 3.7.

Таблица 3.7. Отношение ОТДЕЛЫ

НОМЕР_ОТДЕЛА	ТЕЛЕФОН
1	11-22-33
2	33-22-11

В отношении ОТДЕЛЫ можно выделить функциональную зависимость НОМЕР_ОТДЕЛА → ТЕЛЕФОН.

Следует обратить внимание, что атрибут НОМЕР_ОТДЕЛА, не являвшийся ключевым в отношении СОТРУДНИКИ_ОТДЕЛЫ, становится потенциальным ключом в отношении ОТДЕЛЫ. Именно за счет этого устраняется избыточность, связанная с многократным хранением одних и тех же номеров телефонов.

Таким образом, все обнаруженные аномалии обновления устраниены. Реляционная модель, состоящая из четырех отношений: СОТРУДНИКИ, ОТДЕЛЫ, ПРОЕКТЫ, ЗАДАНИЯ, – находящихся в третьей нормальной форме, является адекватным описанием модели предметной области.

3.4.5 Алгоритм нормализации

Алгоритм нормализации, т.е. приведения отношений к ЗНФ, описывается следующим образом.

Шаг 1 (Приведение к 1НФ). На первом шаге задается одно или несколько отношений, отображающих понятия предметной об-

ласти. По модели предметной области выписываются обнаруженные функциональные зависимости. Все отношения автоматически находятся в 1НФ.

Шаг 2 (Приведение к 2НФ). Если в некоторых отношениях обнаружена зависимость атрибутов от части сложного ключа, то проводим декомпозицию этих отношений на несколько отношений следующим образом: те атрибуты, которые зависят от части сложного ключа, выносятся в отдельное отношение вместе с этой частью ключа. В исходном отношении остаются все ключевые атрибуты.

Шаг 3 (Приведение к 3НФ). Если в некоторых отношениях обнаружена зависимость некоторых неключевых атрибутов от других неключевых атрибутов, то проводим декомпозицию этих отношений следующим образом: те неключевые атрибуты, которые зависят от других неключевых атрибутов, выносятся в отдельное отношение.

На практике, при создании логической модели данных, как правило, не следуют прямо приведенному алгоритму нормализации. Опытные разработчики обычно сразу строят отношения в 3НФ. Кроме того, основным средством разработки логических моделей данных являются различные варианты ER-диаграмм. Особенность этих диаграмм в том, что они сразу позволяют создавать отношения в 3НФ. Тем не менее, приведенный алгоритм важен по двум причинам. Во-первых, этот алгоритм показывает, какие проблемы возникают при разработке слабо нормализованных отношений. Во-вторых, как правило, модель предметной области никогда не бывает правильно разработана с первого шага. Изменения в модели могут привести к появлению новых зависимостей, которые отсутствовали в первоначальной модели предметной области. В этом случае необходимо использовать алгоритм нормализации хотя бы для того, чтобы убедиться, что отношения остались в 3НФ и логическая модель не ухудшилась.

Существуют и более строгие формы нормализации. **Четвертая нормальная форма** устраниет многозначные зависимости (т.е. связи «многие ко многим») путем введения дополнительной таблицы, которая заменяет связь «многие ко многим» на две связи «один ко многим». **Пятая нормальная форма** предназначена для устранения зависимых сочетаний и учитывается только при декомпозиции отношений.

4 ВВЕДЕНИЕ В SQL

4.1 Стандарт SQL

В середине 70-х годов, после появления реляционной модели данных, началась разработка нового языка, предназначенного для управления данными. Перспективный язык реляционных баз данных должен был позволять:

- создавать базы данных, таблицы и другие объекты БД;
- выполнять основные операции редактирования данных в таблицах (вставка, модификация и удаление);
- выполнять запросы пользователя к данным, преобразующие хранящиеся в таблицах данные в выходные отношения.

В отличие от высокоуровневых языков программирования тех лет, новый язык должен был работать с трехзначной логикой. Наряду с классическими для любого языка программирования понятиями истина/ложь (`true` и `false`), необходимо было обрабатывать третье значение неопределенности `Unknown` (или `NULL`). К тому же, планировалась разработка языка, с которыми смогут работать не только программисты, но и пользователи, т.е. язык должен был быть декларативным, а не процедурным. В соответствии с этим пользователь лишь ставит базе данных задачу (т.е. указывает, что ему нужно от базы данных), а каким образом СУБД станет решать поставленную задачу, пользователю не интересует.

Таким языком стал **язык SQL** (Structured Query Language – «язык структурированных запросов») – декларативный язык программирования, применяемый для создания, модификации и выбора данных в реляционной базе данных, управляемой соответствующей системой управления базами данных [12].

Поскольку к началу 1980-х годов существовало несколько вариантов СУБД разных производителей, причём, каждый из них об-

ладал собственной реализацией языка запросов, было принято решение разработать стандарт языка, который будет гарантировать переносимость программного обеспечения с одной СУБД на другую (при условии, что они будут поддерживать этот стандарт).

Стандартом SQL стал в 1986 году благодаря Американскому национальному институту стандартов (ANSI) и Международной организации стандартизации (ISO).

Изначально SQL был основным способом работы пользователя с базой данных и позволял выполнять следующий набор операций:

- создание в базе данных новой таблицы;
- добавление в таблицу новых записей;
- изменение и удаление записей;
- выборка записей из одной или нескольких таблиц (в соответствии с заданным условием);
- изменение структуры таблиц.

Со временем SQL усложнился: обогатился новыми конструкциями, обеспечил возможность описания и управления хранимыми объектами (например, индексами, представлениями, триггерами и хранимыми процедурами), – и стал приобретать черты,ственные языкам программирования.

После появлению первой версии стандарта в 1986 году, он постоянно менялся и дополнялся (таблица 4.1). В 1990-х годах официально действующим и общепризнанным стал считаться стандарт SQL:92. Практически любая серьезная компания, разрабатывающая СУБД, старается поддерживать требования SQL:92.

В 1999 году был выпущен очередной стандарт SQL-3 (SQL:99). В стандарте была добавлена поддержка триггеров, базовые процедурные расширения, поддержка регулярных выражений. Кроме того, стандарт сделал важный шаг в направлении поддержки объектных БД. В частности, здесь были объявлены структурные определяемые пользователем данные и типизированные таблицы.

Таблица 4.1. Стандарт SQL

Год	Название	Изменения
1986	SQL-86	Первый вариант стандарта, принятый институтом ANSI и одобренный ISO в 1987 году.
1989	SQL-89	Немного доработанный вариант предыдущего стандарта.
1992	SQL-92	Значительные изменения (ISO 9075).
1999	SQL:1999 / SQL-3	Добавлена поддержка регулярных выражений, рекурсивных запросов, поддержка триггеров, базовые процедурные расширения, нескалярные типы данных и некоторые объектно-ориентированные возможности.
2003	SQL:2003	Введены расширения для работы с XML-данными, оконные функции, генераторы последовательностей и основанные на них типы данных.
2006	SQL:2006	Функциональность работы с XML-данными значительно расширена. Появилась возможность совместно использовать в запросах SQL и XQuery.
2008	SQL:2008	Улучшены возможности оконных функций, устраниены некоторые неоднозначности стандарта SQL:2003.
2011	SQL:2011	Реализована поддержка хронологических баз данных (PERIOD FOR), поддержка конструкции FETCH.
2016	SQL:2016	Защита на уровне строк, полиморфные табличные функции, JSON.

Однако многие специалисты указывали на незавершенность стандарта. Во многом по этой причине в 2003 году к вопросу модернизации SQL вернулись вновь. В обновленный стандарт с необходимыми изменениями вошли все части прежнего стандарта и были представлены расширения для взаимодействия с языком Java и XML-данными.

После 2003 года очередные версии стандарта SQL выходили примерно раз в 3–5 лет, последним стандартом считается SQL:2016, который вобрал в себя наработки всех своих предшественников. В стандарт были добавлены оконные функции, поддержка JSON и другие возможности.

По традиции, как и со многими стандартами в ИТ-индустрии, с языком SQL возникла проблема: на каком-то этапе многие производители использующего SQL программного обеспечения (ПО) решили, что функциональность в текущей (на тот момент времени) версии стандарта недостаточна, и её желательно расширить. В результате у разных производителей систем управления базами данных в ходу разные диалекты SQL, в общем случае между собой несовместимые.

Как следствие стандарт в свою очередь не успевает за производителями, а это неминуемо ведет к появлению различных ветвей языка, что с каждым годом все более и более минимизирует вероятность появления редакции SQL, однозначно на 100 % поддерживающей всеми разработчиками ПО.

Можно выделить следующие преимущества языка SQL:

- 1) независимость от конкретной СУБД. Несмотря на наличие диалектов и различий в синтаксисе, в большинстве своём SQL-запросы могут быть достаточно легко перенесены из одной СУБД в другую. Естественно, что при применении некоторых специфичных для реализации возможностей такой переносимости добиться уже очень трудно;
- 2) наличие стандартов. Наличие стандартов и набора тестов для выявления совместимости и соответствия конкретной реализации SQL общепринятому стандарту способствует «стабилизации» языка. Правда, стоит обратить внимание, что сам по себе стандарт местами слишком формализован;
- 3) декларативность. С помощью SQL программист описывает только то, какие данные нужно извлечь или модифицировать. То, каким образом это сделать, решает СУБД непосредственно при обработке SQL-запроса. Но, конечно, полезно представлять, как СУБД будет разбирать текст запроса. Чем сложнее сконструирован запрос, тем больше он

допускает вариантов написания, различных по скорости выполнения, но одинаковых по итоговому набору данных.

К недостаткам языка SQL можно отнести следующие:

- 1) несоответствие реляционной модели данных. Создатели реляционной модели и их сторонники указывают на то, что SQL не является истинно реляционным языком. В частности, они указывают на следующие дефекты SQL с точки зрения реляционной теории:
 - допущение строк-дубликатов в таблицах и результатах выборок, что в рамках реляционной модели данных невозможно и недопустимо;
 - поддержка неопределённых значений (NULL), создающая фактически многозначную логику;
 - значимость порядка столбцов, возможность ссылок на столбцы по номерам (в реляционной модели столбцы должны быть равноправны);
 - допущение столбцов без имени, дублирующихся имён столбцов;
- 2) сложность языка. Хотя SQL и задумывался как средство работы конечного пользователя, позже он стал настолько сложным, что превратился в инструмент программиста;
- 3) отступления от стандартов. Несмотря на наличие международного стандарта ANSI, многие разработчики СУБД вносят изменения в язык SQL, применяемый в СУБД, тем самым отступая от стандарта. Таким образом появляются специфичные для каждой конкретной СУБД диалекты языка SQL;
- 4) сложность работы с иерархическими структурами. Ранее диалекты SQL большинства СУБД не предлагали способа манипуляции древовидными структурами. Некоторые поставщики СУБД предлагали свои решения. В настоящее время в ANSI стандартизована рекурсивная конструкция WITH.

4.2 Возможности SQL

Рассмотрим возможности, которые предоставляет язык SQL. Сразу отметим, что SQL не рассматривает задачи, стоящие перед прикладным и тем более системными программистами, такие как реализация низкоуровневых операций ввода-вывода, вопросы построения пользовательского интерфейса, организации работы с периферийными устройствами и тому подобное. Т.е. SQL это не замена традиционным языкам программирования.

Язык SQL выступает неотъемлемой частью реляционных СУБД и применяется только в задачах обработки данных. В общем случае можно выделить следующие задачи SQL: определение данных, манипулирование данными, ограничение доступа к данным, управление курсором и управление транзакцией (рисунок 36).

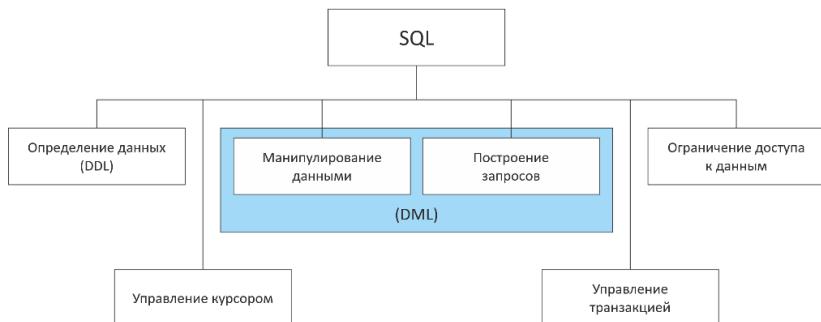


Рисунок 36 – Возможности SQL

Операции определения данных реализуются средствами подъязыка определения данных (Data Definition Language). Язык нацелен на решение вопросов создания и удаления базы данных и ее объектов: таблиц, представлений, индексов, курсоров. Используются операторы CREATE, ALTER и DROP.

Манипулирование данными (Data Manipulation Language) обеспечивает проведение операций вставки, редактирования и удаления

данных из таблиц базы данных. Для модификации данных используются команды INSERT, UPDATE и DELETE. Наиболее востребованная часть DML, основанная на инструкции SELECT, позволяет извлекать данные из одной или нескольких таблиц.

Ограничение доступа к данным – это определение набора прав пользователей при работе с объектами базы данных. Основными командами являются команды GRANT, REVOKE и DENY.

Управление курсором позволяет обрабатывать данные построчно. Задача решается за счет команд DECLARE, OPEN, FETCH и CLOSE.

Наконец, команды управления транзакцией включают инструкции SET TRANSACTION, BEGIN TRANSACTION, COMMIT и ROLLBACK. Язык позволяет определять уровень изоляции транзакции, начинать, фиксировать или возвращать транзакцию в исходное состояние.

4.3 Типы данных в SQL

Знакомство с языком начнем с рассмотрения поддерживаемых им типов данных. На рисунке 37 представлен перечень стандартных типов данных SQL. СУБД, поддерживающие стандарт SQL:92, реализуют набор типов данных, который включает в себя точные числовые типы, приближенные числовые типы, типы для работы с датой и временем, временные интервалы, логические типы данных, строки символов и битовые строки.

Если СУБД совместима с более новыми стандартами, то к перечню добавляется еще несколько типов. В стандарте SQL:2003 их называют непредопределеными и даже более жестко – типами данных, не относящимися к SQL. С некоторой степенью допущения их можно причислить к структурным типам данных, имеющимся в большинстве высокоуровневых языков программирования: это кол-

лекции, последовательности, типы данных, определяемые пользователем, и ссылочные типы.

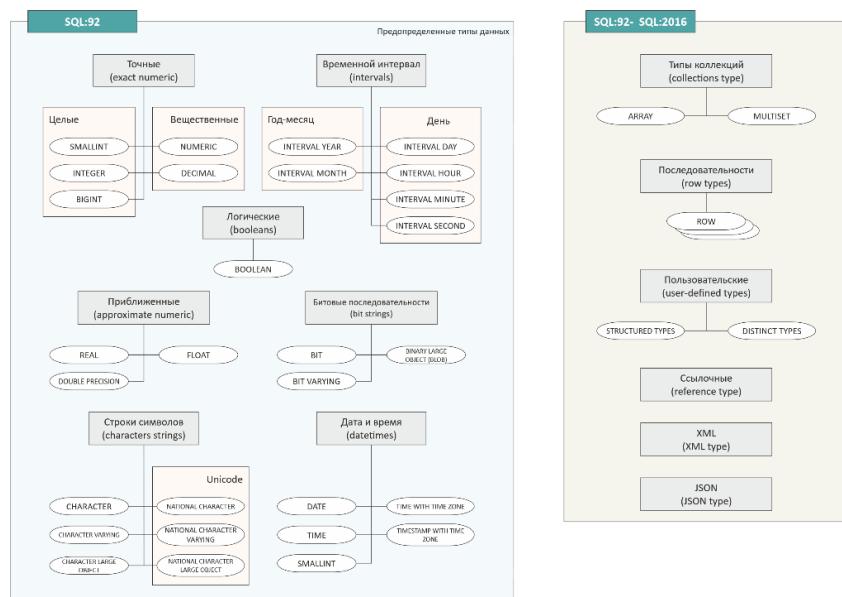


Рисунок 37 – Типы данных

Кроме того, современные стандарты SQL способны работать с популярными сегодня форматами данных XML и JSON.

Типы данных используются, во-первых, при определении столбцов таблиц. Во-вторых, при объявлении переменных в хранимых процедурах, функциях, определяемых пользователем, и триггерах. В-третьих, при организации обмена данными между базой данных и клиентским приложением.

4.3.1 Точные числовые типы

Точные числовые типы предназначены для обработки целочисленных значений и значений, имеющих дробную часть, без потерь точности. При задании такого типа данных необходимо указать два

аргумента: точность (n) и масштаб (m). Точность задает общее число значащих цифр, используемых при отображении числа. Масштаб определяет число значащих цифр справа от десятичной точки.

Единственным дополнением к набору точных числовых данных, существовавших с первых версий стандарта SQL, стал введенный в 1999 году тип данных BIGINT (таблица 4.2).

Таблица 4.2. Точные числовые типы

Спецификация	Описание
NUMERIC [(n[,m])]	Точное число, описываемое аргументами n и m.
DECIMAL [(n[,m])] или DEC [(n[,m])]	В отличие от NUMERIC, способно хранить число дальше с большей точностью, чем определено в аргументе m. Поэтому говорят, что NUMERIC задает реальное значение точности, а DECIMAL – минимальное значение точности.
BIGINT	Тип данных, предназначен для хранения больших целых чисел (обычно 64 бит). Типы данных BIGINT, INTEGER и SMALLINT являются частным случаем типа данных NUMERIC, у которого масштаб установлен в 0, а точность определена возможностями СУБД.
INTEGER или INT	Тип данных, предназначен для хранения целых чисел (обычно 32 бит).
SMALLINT	Тип данных, предназначен для хранения малых целых чисел (обычно 16 бит).

4.3.2 Приближенные числовые типы

Приближенные числовые типы представляют собой тип данных для определения чисел с плавающей точкой, осуществляющий хранение числа в научном формате (мантиssa плюс порядок) (таблица 4.3).

Типы данных `real` и `double precision` хранят приближённые числовые значения с переменной точностью. Обычно эти типы реализуют стандарт для двоичной арифметики с плавающей

точкой (с одинарной и двойной точностью соответственно), в той мере, в какой его поддерживают процессор, операционная система и компилятор.

Неточность здесь выражается в том, что некоторые значения, которые нельзя преобразовать во внутренний формат, сохраняются приближённо, так что полученное значение может несколько отличаться от записанного.

Таблица 4.3. Приближенные числовые типы

Спецификация	Описание
REAL	Точность и предел значений зависят от СУБД. Как правило, занимает в памяти 6 байт и в состоянии хранить число в интервале от $-3,4E - 38$ до $+3,4E + 38$ с точностью до 7 цифр после запятой.
FLOAT [(n)]	Аргументом n определяется минимальное значение точности. Обычно подразумевается хранение чисел с одинарной точностью, 4 байта.
DOUBLE PRECISION	Точность определяется версией СУБД, превышает точность REAL (обычно, 8 байт). Тип данных способен хранить число в интервале от $1,7E - 308$ до $+1,7E - 308$.

4.3.3 Логический тип

Логический тип данных языка SQL отличается от соответствующего типа в стандартных языках программирования. Особенность заключается в том, что два классических элемента (true/ false) булевой логики здесь дополнены третьим значением – неопределенностью Unknown (или NULL). Как следствие логика становится более сложной – трехзначной.

Понятие NULL-значения вводилось в разделе 2.4.1 при рассмотрении вопросов целостности данных. Таблицы истинности для трехзначной логики были представлены на рисунке 15.

4.3.4 Строковые типы данных

Строковые типы данных, определенные в стандарте, представлены в таблице 4.4. CHAR представляет собой строку фиксированной длины. Если в поле типа CHAR записывается текстовое значение меньшего размера, чем размерность поля, то оставшиеся позиции символов заполняются пробелами.

VARCHAR представляет собой строку переменной длины. Преимущество такого типа данных над типом CHAR заключается в том, что здесь пустые позиции не заполняются пробелами и, соответственно, таблица требует меньшего размера оперативной и дисковой памяти.

Таблица 4.4. Строковые типы

Спецификация	Описание
CHARACTER [n], или сокращенно CHAR[n]	Тип данных предназначен для создания текстовой строки фиксированной длины, дополненная пробелами. Количество символов в строке определяется в квадратных скобках после указания типа данных.
CHARACTER VARYING [n], или сокращенно VARCHAR[n]	Текстовая строка переменной длины. Максимальный размер строки определяется в квадратных скобках.
NCHAR[n], NCHAR VARYING [n]	Строки национального символьного набора (NATIONAL). При использовании следует указать спецификацию набора символов, воспользовавшись командой CHARACTER SET.
(NATIONAL) CHARAC- TER LARGE OBJECT [n], или CLOB[n]	Предназначен для определения столбцов таблиц, хранящих большие группы символов.

В стандарте также определены типы данных для хранения национального символьного набора. Для такого типа данных необходимо задавать используемую кодировку. И типы данных для хранения текстовой информации большого объема.

И конечно разные СУБД поддерживают дополнительные типы, например, тип данных TEXT в MySQL и PostgreSQL.

4.3.5 Битовые последовательности

Битовая последовательность предназначена для хранения любой двоичной информации. Тип данных универсален и позволяет описывать как простейшие логические данные, так и сложные объекты, например, файлы мультимедиа.

Так же, как и для строковых типов данных, поддерживаются битовые последовательности фиксированной длины, переменной длины и последовательности, предназначенные для хранения больших объектов (таблица 4.5). Особенность типа данных фиксированной длины BIT состоит в том, что попытка записать в поле этого типа значения меньшей длины, чем указано в аргументе n, приведет к ошибке.

Таблица 4.5. Битовые последовательности

Спецификация	Описание
BIT [n]	Битовая последовательность фиксированной длины. Аргумент n устанавливает длину последовательности в битах. Если аргумент отсутствует (или установлен в 1), то тип данных используется для полей логического типа.
BIT VARYNG [n]	Битовая последовательность переменной длины, n – максимальное значение битовой последовательности.
BINARY LARGE OBJECT [n], BLOB [n]	Тип данных предназначен для хранения больших объектов. Например, файлов мультимедиа и изображений.

4.3.6 Дата и время

За описание значений даты и времени отвечает пять типов данных. Соответственно, можно хранить только дату, только время, дату и время одновременно, а также дополнительно можно указать

временную зону (таблица 4.6). Дата представляется в формате общепринятого в большинстве стран мира григорианского календаря.

Таблица 4.6. Дата и время

Спецификация	Описание
DATE	Тип данных включает три поля: YEAR (год) – от 0001 до 9999; MONTH (месяц) – от 01 до 12; DAY (день) – от 01 до 31. Формат записи: «yyyy-mm-dd».
TIME [(n)]	Тип данных включает три поля: HOUR (часы), MINUTE (минуты), SECOND (секунды). Если аргумент точности (n) не определен, то полное число позиций (вместе с разделителями) равно 8. Формат записи: «hh:mm:ss».
TIMESTAMP [(n)]	Метка даты-времени, представляющая собой комбинацию типов данных DATE и TIME.
TIME WITH TIME ZONE	Тип данных аналогичен TIME плюс два дополнительных значения, характеризующих смещение от Гринвичского меридиана в часах TIMEZONE_HOUR и минутах TIMEZONE_MINUTE.
TIMESTAMP WITH TIME ZONE	Метка даты-времени плюс смещение от Гринвича.

4.3.7 Непредопределенные типы данных

Большинство нестандартных, или, как их еще называют, непредопределенных типов данных, вошло в состав SQL сравнительно недавно. В SQL:1999 появилась спецификация коллекции и массива, а в SQL:2003 к стандарту добавилось мультимножество. Кроме того, стандарт добавил такие типы, как последовательности, пользовательский тип, тип данных XML, JSON и ссылочный тип.

Основная особенность непредопределенных типов в том, что даже в действующем стандарте SQL их называют типами данных, не соответствующими SQL. Несоответствий много, но главное –

нарушение требования к атомарности данных, которое предписывает, чтобы в одной ячейке таблицы хранилось единственное неделимое значение.

Массив, мультимножество и последовательность нельзя рассматривать как атомарный тип, и если такой тип данных определяет колонку таблицы, то сразу возникнут проблемы нормализации данных. Указанное противоречие появилось из-за стремления разработчиков стандарта расширить возможности реляционных баз по хранению данных и, в первую очередь, разрешить хранить в базе данных сложные объекты, чтобы позволить разработчикам создавать объектно-реляционные базы данных.

4.4 Операторы

В SQL существует стандартный набор операторов, применяемых для осуществления математических, логических операций, операций сравнения и других операций.

Операция присваивания: «=», реже «:=».

Арифметические операторы:

- «+» сложение;
- «–» вычитание;
- «*» умножение;
- «/» деление;
- «DIV» целочисленного деление;
- «%» (MOD) остатка от деления.

Многие диалекты SQL поддерживают операторы целочисленного деления и остатка от деления. За небольшим исключением, в качестве operandов должны выступать только числа. Кроме того, арифметические операторы могут использоваться при работе с такими типами данных, как дата, время и интервал.

Язык SQL также поддерживает стандартный перечень логических операторов:

- AND – логическое умножение («И»);
- OR – логическое сложение («ИЛИ»);
- NOT – логическое отрицание («НЕ»);
- XOR – исключающее «ИЛИ».

Операторы сравнения также являются стандартными:

- «<» меньше;
- «>» больше;
- «<=» меньше или равно;
- «>=» больше равно;
- «==» проверка равенства;
- «<=>» проверка равенства с учетом NULL;
- «!=», «<>» проверка неравенства.

При осуществлении операции сравнения необходимо учитывать, что среди сравниваемых значений может оказаться NULL. Результатом оператора сравнения может являться 1 – истина, 0 – ложь или NULL.

Многие СУБД определяют свои функции и операторы сравнения, например, сравнение строк по паттерну или проверка попадания значения в некоторый диапазон.

Вialectах SQL различных производителей обычно предусмотрено несколько способов проверки значения на неопределенность. Обычно используется оператор IS NULL:

<значение> IS [NOT] NULL.

Кроме того, часто встречается функция ISNULL().

4.5 Встроенные функции

Классический SQL определяет сравнительно небольшой набор встроенных функций: получение длины битовой последовательности или строки, получение текущего времени и даты, поиск подстроки в строке, преобразование к нижнему или верхнему регистру и другие, перечисленные в таблице 4.7.

Таблица 4.7. Встроенные функции

Спецификация	Описание
BIT_LENGTH(битовая строка)	Возвращает длину строки в битах.
CAST(значение AS тип данных)	Функция преобразования исходного значения к новому типу данных.
CHAR_LENGTH(символьная строка)	Возвращает длину строки в символах.
CURRENT_DATE	Возвращает текущую дату.
CURRENT_TIME(точность)	Возвращает текущее время с указанной точностью.
CURRENT_TIMESTAMP(точность)	Возвращает текущую дату и время с указанной точностью.
LOWER(строка)	Преобразование текстовой строки к нижнему регистру.
POSITION(подстрока IN строка)	Возвращает позицию, с которой начинается вхождение подстроки в строку.
SUBSTRING(строка FROM n FOR длина)	Возвращает часть строки, начиная с n-го символа с указанной длиной.
TRANSLATE(строка USING функция)	Преобразование строки с использованием указанной функции.
TRIM(LEADING TRAILING BOTH символ FROM строка)	Удаление из строки всех первых (LEADING), последних (TRAILING) или первых и последних (BOTH) символов.
UPPER(строка)	Преобразование текстовой строки к верхнему регистру.

Современные СУБД значительно расширяют этот список, добавляя функции работы с числами: такие как округление, возведение в степень, нахождения квадратного корня и другие, функции работы с датой и временем, например, получение месяца даты или минуты времени, а также дополнительные функции работы со строками.

5 SQL. ОПЕРАТОРЫ ОПРЕДЕЛЕНИЯ ДАННЫХ

Перейдем к изучению первой большой группы операторов языка SQL – операторов определения данных данных (Data Definition Language, DDL). Иногда эту группу операторов называют подъязыком DDL. Эти операторы позволяют создавать, редактировать и удалять основные объекты БД, такие как схемы, домены, таблицы и т. д. За создание, изменение и удаление объекта БД отвечают три команды CREATE, ALTER и DROP соответственно.

5.1 Базы данных

Для создания базы данных используется команда CREATE DATABASE. Она имеет следующий синтаксис:

```
CREATE DATABASE [IF NOT EXISTS] <имя_базы_данных>;
```

В квадратных скобках указывается необязательная часть команды. Соответственно, у команды есть две формы. Первая форма CREATE DATABASE попытается создать базу данных, но если такая база данных уже существует, то операция возвратит ошибку. Вторая форма CREATE DATABASE IF NOT EXISTS попытается создать базу данных, если на сервере отсутствует база с таким именем.

Хотя команда CREATE DATABASE поддерживается производителями СУБД, в стандарте SQL она в явном виде отсутствует, а вместо нее предусмотрена синтаксическая конструкция CREATE SCHEMA, которая имеет следующий формальный вид:

```
CREATE SCHEMA <имя_схемы> [AUTHORIZATION имя владельца]  
[DEFAULT CHARACTERSET набор символов по умолчанию]  
[PATH <символьный путь к файлам БД> ]  
[ <дополнительные инструкции>... ]
```

После создания базы данных с ней будут проводится различные операции: создание таблиц, добавление и получение данных и т.д. Чтобы производить эти операции, надо установить определенную базу данных в качестве используемой. Для этого применяется оператор USE:

```
USE <имя_базы_данных>;
```

Для удаления базы данных применяется команда DROP DATABASE, которая имеет следующий синтаксис:

```
DROP DATABASE [IF EXISTS] <имя_базы_данных>;
```

Как и в случае с командой создания базы данных, команда удаления имеет две формы. Первая форма DROP DATABASE попытается удалить базу данных, но если такая база отсутствует на сервере, то операция возвратит ошибку. Вторая форма DROP DATABASE IF EXISTS попытается удалить базу данных, если на сервере существует база с таким именем.

В свою очередь, в стандарте SQL определена команда DROP SCHEMA:

```
DROP SCHEMA <имя_схемы> [CASCADE | RESTRICT];
```

5.2 Таблицы

5.2.1 Создание таблиц

Для создания таблиц используется команда CREATE TABLE. Эта команда применяет ряд операторов, которые определяют столбцы таблицы и их атрибуты. Упрощенный формальный синтаксис команды CREATE TABLE имеет следующий вид:

```
CREATE TABLE <название_таблицы> (
    название_столбца1 <тип_данных> <атри-
    буты_1>,
```

```
    название_столбца2 <тип_данных> <атри-
    буты_2>,
    ...
    название_столбцаN <тип_данных> <атри-
    буты_N>,
    атрибуты_уровня_таблицы
);
```

После команды CREATE TABLE идет название таблицы. Имя таблицы выполняет роль ее идентификатора в базе данных, поэтому оно должно быть уникальным. Затем в скобках перечисляются названия столбцов, их типы данных и атрибуты. В самом конце можно определить атрибуты для всей таблицы. Атрибуты столбцов, а также атрибуты таблицы указывать необязательно.

В качестве примера рассмотрим скрипт создания простейшей таблицы:

```
CREATE DATABASE productsdb;
USE productsdb;

CREATE TABLE customers (
    id INT,
    age INT,
    first_name VARCHAR(20),
    last_name VARCHAR(20)
);
```

Т.к. таблица не может существовать сама по себе, а должна быть создана в определенной базе данных, в скрипте сначала создается база productsdb. Затем применяется команда USE, чтобы указать, что все дальнейшие операции, в том числе создание таблицы, будут производиться с этой базой данных. Далее собственно идет команда создания таблицы клиентов с названием customers. В таблице определено четыре столбца: `id`, `age`, `first_name`, `last_name`. Первые два столбца представляют собой идентификатор клиента и его возраст и имеют тип INT, то есть будут хранить

целочисленные значения. Следующие столбцы хранят имя и фамилию клиента и имеют тип VARCHAR (20), то есть представляют собой строку длиной не более 20 символов. В данном случае для каждого столбца определены имя и тип данных, при этом атрибуты столбцов и таблицы в целом отсутствуют.

С помощью атрибутов можно настроить поведение столбцов. Могут использоваться атрибуты, представленные в таблице 5.1.

Таблица 5.1. Атрибуты столбцов

Ключ	Описание
NOT NULL	Запрет на вставку в столбец неопределенного значения NULL.
UNIQUE	Значение столбца должно быть уникальным.
AUTO_INCREMENT	Значение столбца будет автоматически увеличиваться при добавлении новой строки.
DEFAULT	Определяет значение по умолчанию для столбца.
PRIMARY KEY	Признак первичного ключа. Значение поля должно быть уникальным, оно не может содержать NULL, в таблице это ограничение может использоваться только один раз.
CHECK	Ограничение-проверка на допустимое значение. В скобках за оператором CHECK указывается предикат, проверяющий допустимость значения.
FOREIGN KEY	Ограничение внешнего ключа для таблицы. Ограничения внешнего ключа требуют, чтобы все значения, присутствующие во внешнем ключе, соответствовали значениям родительского ключа (обеспечение ссылочной целостности)

Рассмотрим пример создания таблицы с использованием дополнительных атрибутов.

```
CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    age INT DEFAULT 18 CHECK(age > 0 AND age <
    100),
```

```
first_name VARCHAR(20) NOT NULL,  
last_name VARCHAR(20) NOT NULL,  
email VARCHAR(30) NULL,  
phone VARCHAR(20) UNIQUE,  
CHECK ((email != '') AND (phone != ''))  
);
```

В примере создается таблица `customers` с указанием, что столбец `id` является первичным ключом. Кроме того, этот столбец является автоинкрементным, т.е. явно задавать значение идентификатора не требуется, значение столбца `id` после каждой новой добавленной строки будет увеличиваться на единицу. Для столбца `age` указан атрибут `DEFAULT`, т.е. если явно не указать значение этого столбца при добавлении новых данных, по умолчанию будет добавлено значение 18. Кроме того, у столбца `age` указан атрибут `CHECK`, после которого добавлено условие, что возраст клиентов должен быть больше 0 и меньше 100. Также `CHECK` можно использовать на уровне таблицы. В данном примере проверяется, что текстовые поля заполнены. Далее указано, что имя и фамилия клиента не могут содержать неопределенные значения `NULL`. Столбец `phone`, который представляет телефон клиента, может хранить только уникальные значения. Т.е. будет нельзя добавить в таблицу две строки, у которых значения в этом столбце будут совпадать.

Рассмотрим еще один пример с заданием первичного ключа таблицы. Первичный ключ уникально идентифицирует строку в таблице. Первичный ключ может быть установлен как на уровне столбца, так и на уровне таблицы. Первичный ключ также может быть составным: такой ключ будет использовать сразу несколько столбцов, чтобы уникально идентифицировать строку в таблице. В следующем примере составной ключ создается для таблицы заказов `orders`.

```
CREATE TABLE orders (
    order_id INT,
    product_id INT,
    quantity INT,
    price DECIMAL(10, 2),
    PRIMARY KEY (order_id, product_id)
);
```

В таблице поля `order_id` и `product_id` вместе выступают как составной первичный ключ. То есть в таблице `orders` не может быть двух строк, где для обоих из этих полей одновременно были бы одни и те же значения.

Таким же образом на уровне таблицы можно указывать атрибуты `CHECK` и `UNIQUE`.

5.2.2 Внешние ключи

Внешние ключи (`FOREIGN KEY`) позволяют установить связи между таблицами. Внешний ключ устанавливается для столбцов из зависимой, дочерней таблицы, и указывает на один из столбцов главной или родительской таблицы. Как правило, внешний ключ указывает на первичный ключ из связанной главной таблицы. Общий синтаксис установки внешнего ключа на уровне таблицы имеет следующий вид:

```
[CONSTRAINT <имя_ограничения>]
FOREIGN KEY (<столбец1>, <столбец2>, ..., <столбецN>)
REFERENCES <главная_таблица> (<столбец1>, <столбец2>, ..., <столбецN>)
[ON DELETE <действие>]
[ON UPDATE <действие>]
```

Для создания ограничения внешнего ключа после `FOREIGN KEY` указывается столбец таблицы, который будет представлять внешний ключ. После ключевого слова `REFERENCES` указывается имя связанной таблицы, а затем в скобках имя связанного столбца,

на который будет указывать внешний ключ. После выражения REFERENCES идут выражения ON DELETE и ON UPDATE, которые задают действие при удалении и обновлении строки из главной таблицы соответственно.

Рассмотрим создание внешнего ключа на примере. Определим таблицы `customers` и `orders`. Таблица `customers` является главной и представляет клиента. Таблица `orders` является зависимой и представляет заказ, сделанный клиентом.

```
CREATE TABLE customers (
    id INT PRIMARY KEY AUTO_INCREMENT,
    age INT,
    first_name VARCHAR(20),
    last_name VARCHAR(20)
);

CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    created_at DATE,
    FOREIGN KEY (customer_id) REFERENCES customers
(id)
);
```

Столбец `customer_id` в таблице `orders` является внешним ключом, который указывает на столбец `id` из таблицы `customers`, который является первичным ключом.

При создании внешнего ключа с помощью выражений ON DELETE и ON UPDATE можно установить действия, которые выполняются соответственно при удалении и изменении связанной строки из главной таблицы. В качестве действия могут использоваться опции, перечисленные в таблице 5.2. Например, каскадное удаление позволяет при удалении строки из главной таблицы автоматически удалить все связанные строки из зависимой таблицы.

Таблица 5.2. Опции ON DELETE / ON UPDATE

Ключ	Описание
CASCADE	Изменение значения первичного ключа приводит к автоматическому изменению соответствующих значений внешнего ключа.
SET NULL	При изменении значения или удалении первичного ключа все значения в связанных с внешним ключом колонках устанавливаются в NULL. В результате в дочерней таблице появляются «брошенные» строки, утерявшие связь с соответствующей записью из главной таблицы.
RESTRICT	Отклоняет удаление или изменение строк в главной таблице при наличии связанных строк в зависимой таблице.
NO ACTION	То же самое, что и RESTRICT.
SET DEFAULT	При удалении связанной строки из главной таблицы устанавливает для столбца внешнего ключа значение по умолчанию, которое задается с помощью атрибута DEFAULT.

5.2.3 Модификация таблиц

Для изменения таблицы применяется команда ALTER TABLE. Ее сокращенный формальный синтаксис имеет следующий вид:

```
ALTER TABLE <название_таблицы>
{
    ADD <название_столбца> <тип_данных> [<атрибуты>]
    |
    DROP COLUMN <название> RESTRICT | CASCADE |
    MODIFY COLUMN <название> <тип_данных> [<атри-
    буты>] |
    ALTER COLUMN <название> SET DEFAULT <значение> |
    ADD [CONSTRAINT] <определение_ограничения> |
    DROP [CONSTRAINT] <имя> RESTRICT | CASCADE
}
```

С помощью этой команды можно добавить, изменить или удалить столбцы таблицы, а также добавить к таблице новое или удалить уже существующее ограничение. Параметры команды опи-

саны в таблице 5.3. Полные возможности команды можно посмотреть в официальной документации используемой СУБД.

Таблица 5.3. Параметры команды модификации таблицы

Ключ	Описание
ADD [COLUMN]	Добавляет в таблицу новый столбец. Новый столбец определяется так же, как в операторе CREATE TABLE.
ALTER [COLUMN]	Используется для создания или отмены значения по умолчанию для столбца.
DROP [COLUMN]	Удаляет из таблицы столбец. При использовании параметра RESTRICT перед удалением столбца СУБД проверит наличие ссылок на него из других таблиц и представлений. Если таковые имеются, то столбец не будет удален. Наоборот, при использовании параметра CASCADE вместе со столбцом будут удалены все объекты, ссылающиеся на него.
ADD CONSTRAINT	Позволяет добавить к таблице новое ограничение.
DROP CONSTRAINT	Удаляет уже существующие ограничения. Если в предложении задан параметр RESTRICT, то в этот момент столбец не должен использоваться как родительский ключ для внешнего ключа другой таблицы. Если передается параметр CASCADE, то внешние ключи, имеющие ссылки или ограничения FOREIGN KEY, уничтожаются.

Для полного удаления данных, т.е. очистки таблицы, применяется команда TRUNCATE TABLE:

TRUNCATE TABLE <имя_таблицы>;

Для удаления таблицы из базы данных применяется команда DROP TABLE, после которой указывается название удаляемой таблицы:

DROP TABLE <имя_таблицы>;

6 SQL. ОПЕРАТОРЫ МАНИПУЛИРОВАНИЯ ДАННЫМИ

Следующий блок SQL-операторов предназначен для манипулирования данными, т.е. для выборки данных из таблиц, вставки, редактирования и удаления данных в таблицах.

6.1 Добавление данных. Инструкция INSERT

Начнем с добавления данных. Для вставки в таблицу одной или нескольких строк используется инструкция **INSERT**. Практически все диалекты SQL полностью поддерживают синтаксическую конструкцию, рекомендованную стандартом:

```
INSERT INTO {имя_таблицы | имя_представления}  
{имя_столбца [, ...]}  
{VALUES (значение_1 [, ...]) | инструкция SELECT}
```

После ключевых слов **INSERT INTO** указывается имя таблицы, в которую необходимо добавить данные. Далее идет перечисление столбцов таблицы, в которые добавляются новые данные. Если список столбцов окажется неполным, т.е. в запросе будут указаны не все столбцы таблицы, то в оставшиеся поля записывается значение по умолчанию или **NULL**. Если, в свою очередь, значение по умолчанию не было определено заранее (на этапе создания таблицы) и на столбец наложено ограничение **NOT NULL**, то операция добавления данных завершится с ошибкой. Также ошибкой завершится попытка явной записи данных в столбец, который самостоятельно генерирует новое значение (например, в столбцы автоинкрементного типа).

После ключевого слова **VALUES** указываются данные, которые требуется добавить в таблицу. Другим вариантом является добавление строк, возвращаемых в результате выполнения инструкции выборки данных **SELECT**, которая будет рассмотрена далее.

Для демонстрации примеров использования команды `INSERT` создадим базу данных `productsdb` и в ней создадим таблицу `products`, которая содержит автоинкрементный первичный ключ `id`, название товара и производителя, а также количество товара и его цену.

```
CREATE DATABASE productsdb;
USE productsdb;

CREATE TABLE products (
    id INT NOT NULL AUTO_INCREMENT,
    product_name VARCHAR(45) NOT NULL,
    manufacturer VARCHAR(45) NOT NULL,
    product_count INT DEFAULT 0,
    price DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (id)
);
```

Рассмотрим наиболее распространенную форму применения инструкции `INSERT INTO`, в которой нам требуется добавить новую строку в таблицу `products`. Значения будут записываться в столбцы с учетом позиций.

```
INSERT INTO products
(product_name, manufacturer, product_count, price)
VALUES
('iPhone 13', 'Apple', 5, 94990);
```

Заметим, что в тексте запроса отсутствует обязательное значение для столбца первичного ключа таблицы `id`. Т.к. поле `id` объявлено с атрибутом `AUTO_INCREMENT`, то новое значение первичного ключа будет сгенерировано автоматически.

При реализации команды `INSERT` некоторые СУБД расширили стандартные возможности SQL, позволяя за одно обращение к команде `INSERT` вставлять в таблицу несколько строк.

```
INSERT INTO products
(product_name, manufacturer, price, product_count)
VALUES
('Galaxy S21', 'Samsung', 3, 67000),
('P50', 'Huawei', 1, 115000),
('12 Pro', 'Xiaomi', 3, 115000);
```

Такое решение весьма удобно, особенно когда требуется добавить все строки в таблицу в рамках единственной транзакции.

6.2 Выборка данных. Инструкция SELECT

6.2.1 Базовый синтаксис

Инструкция SELECT, предназначенная для выборки данных из одной или нескольких таблиц или представлений, является основным элементом языка SQL. В простейшем случае результат запроса может полностью повторять содержимое одной таблицы, может представлять собой объединение нескольких таблиц, определенную выборку из исходных данных и т.д. Помимо этого, запросы SELECT активно используются при создании ряда ключевых объектов базы данных (представлений, хранимых процедур и триггеров).

Базовая синтаксическая конструкция запроса выглядит следующим образом:

```
SELECT [DISTINCT|ALL]
{ имя_столбца [AS псевдоним] [, ...]
| функция_агрегирования [AS псевдоним] [, ...]
| выражение_для_вычисления_значения [AS псевдоним]
[, ...]
| спецификатор.* }
FROM
{ имя_таблицы [AS псевдоним] [, ...]
| имя_представления [AS псевдоним] [, ...] }
[WHERE условие_отбора]
[GROUP BY имя_столбца [, ...]] [HAVING условие]
[ORDER BY имя_столбца [ASC | DESC] [, ...]]
```

Следует отметить, что представленный шаблон запроса описывает выборку данных из одной таблицы. Запросы SELECT, которые работают с несколькими таблицами или представлениями, будут рассмотрены далее.

Запросы на выборку данных всегда начинаются с ключевого слова SELECT, затем следует перечень столбцов, которые планируется включить в результат выполнения запроса. В перечень, кроме названий столбцов таблиц, могут входить агрегирующие функции, выражения и просто константы. За списком столбцов следует еще одно обязательное слово FROM, а за ним – имена таблиц или представлений, из которых будет производиться отбор данных.

Для иллюстрации примеров использования оператора SELECT будет использоваться созданная таблица products.

Для выборки всех данных таблицы может использоваться следующий запрос:

```
SELECT * FROM products;
```

Символ «*» указывает, что что в результирующее отношение попадут все столбцы из таблицы. Стоит отметить, что применение «*» для получения данных считается не очень хорошей практикой, так как выборка всех столбцов значительно повышает объем запрашиваемых данных и замедляет выполнение запроса. Однако использование символа «*» может быть предпочтительным тогда, когда названия столбцов не известны.

Если необходимо получить данные не из всех, а из каких-то конкретных столбцов, тогда спецификации этих столбцов перечисляются через запятую после SELECT, например:

```
SELECT product_name, manufacturer FROM products;
```

Если результирующее отношение содержит повторяющиеся строки, которые необходимо удалить из выборки, то сразу после инструкции SELECT следует вставить предикат DISTINCT:

```
SELECT DISTINCT manufacturer FROM products;
```

Обратный предикат ALL указывает на то, что необходимо вернуть все строки данных, включая и дубликаты. Впрочем, в явном применении ALL необходимости нет, так как это режим по умолчанию.

6.2.2 Использование псевдонимов

Спецификация столбца необязательно должна представлять его название. Это может быть любое выражение, например, результат арифметической операции.

В тех случаях, когда запрос достаточно сложен и содержит длинные названия столбцов и таблиц, а также при использовании агрегирующих функций или столбцов, появившихся в результате выполнения различных операций, можно ввести псевдонимы столбцов и таблиц. Для этого применяется ключевое слово AS.

В следующем примере результат запроса содержит 3 столбца: идентификатор, полное название товара, являющееся конкатенацией названия и производителя, а также совокупную стоимость товара. Кроме того, выполнено переименование как столбца таблицы, так и столбцов – результатов операций.

```
SELECT
    id AS ProductID,
    CONCAT('Товар:', product_name,
    '.Производитель:', manufacturer) AS Title,
    product_count * price AS TotalSum
FROM products;
```

6.2.3 Сортировка. Предложение ORDER BY

Полученный в результате выполнения запроса набор строк может быть упорядочен. Порядок сортировки данных определяется при помощи предложения ORDER BY и перечня столбцов, принимающих участие в сортировке.

В следующем примере выполняется упорядочивания выборки из таблицы products по столбцу price по возрастанию:

```
SELECT * FROM products  
ORDER BY price;
```

По умолчанию данные сортируются по возрастанию, однако с помощью ключевого слова DESC можно задать сортировку по убыванию. Кроме того, можно производить сортировку данных по псевдониму столбца, который определяется с помощью оператора AS. Также в качестве критерия сортировки также можно использовать сложно выражение на основе столбцов.

В следующем примере происходит сортировка по совокупной стоимости, причем значение вычисляется непосредственно в операторе ORDER BY:

```
SELECT product_name  
FROM products  
ORDER BY product_count * price DESC;
```

При сортировке сразу по нескольким столбцам все эти столбцы указываются через запятую после оператора ORDER BY. Для разных столбцов можно определить сортировку по возрастанию или по убыванию с помощью ключевых слов ASC и DESC соответственно.

```
SELECT product_name, manufacturer, price  
FROM products  
ORDER BY manufacturer ASC, product_name DESC;
```

6.2.4 Условие отбора данных. Предложение WHERE

Зачастую необходимо извлекать не все данные из базы данных, а только те, которые соответствуют определенному условию. Для фильтрации данных в команде SELECT применяется оператор WHERE, после которого указывается условие фильтрации. Совместно с WHERE применяется следующий перечень типов ограничений:

- сравнение: проводится сравнение результатов вычисления одного выражения с другим с использованием операторов сравнения (`<>`, `<>`, `<=`, `>=`, `=`, `!=`, `<>`). Результат сравнения может принимать значения TRUE, FALSE и UNKNOWN;
- попадание в диапазон: проверяется, попадает ли результат вычисления выражения в определенный диапазон значений;
- соответствие шаблону: проверяется, соответствует ли некоторое строковое значение заданному шаблону;
- неопределенность: содержит ли поле неопределенное значение NULL;
- проверка существования: проверяется факт существования значения в выходных результатах вложенных подзапросов к другим отношениям базы данных;

Все перечисленные ограничения описываются сразу после инструкции WHERE. В одном запросе допускается комбинировать несколько ограничений, используя логические операторы AND, OR и NOT.

Сравнение – наиболее распространенное условие ограничения результирующего набора данных. В следующем примере запрос возвращает все товары, количество которых меньше трех.

```
SELECT * FROM products
WHERE product_count < 3;
```

Условие запроса может содержать сложные составные выражения с использованием арифметических и логических операторов, например:

```
SELECT * FROM products
WHERE manufacturer = 'Samsung' OR (price >
70000 AND price < 100000)
ORDER BY price DESC;
```

В формулировании условий можно объединять несколько выражений, включающих различные операции с помощью логических операций. При использовании сложных выражений следует учитывать приоритет операций. В таблице 6.1 указаны операции в порядке убывания приоритета. Приоритет операций можно переопределить с помощью скобок.

Таблица 6.1. Приоритет операций

Операция	Описание
-, ~	Унарный минус, битовая инверсия.
^	Побитовое исключающее ИЛИ.
*, /, DIV, %, MOD	Умножение, деление, остаток от деления, целочисленное деление.
+, -	Сложение, вычитание.
<<, >>	Побитовые сдвиги.
&	Побитовое И.
	Побитовое ИЛИ.
=, <>, <=, >=	Операции сравнения.
NOT	Логическое отрицание.
AND	Логическое И.
XOR	Логическое исключающее ИЛИ.
OR	Логическое ИЛИ.

Оператор подзапроса **IN** производит проверку, входит ли результат вычисления в заданное множество. Множество может описываться как поэлементно, так и быть результатом другого, вложенного подзапроса:

<значение> [**NOT**] **IN** (подзапрос | элементы_множества) .

В примере выполняется поиск товаров, у которых производитель – один из элементов множества:

```
SELECT * FROM products
WHERE manufacturer IN ('Samsung', 'HTC',
'Huawei');
```

Предикат **BETWEEN** выполняет проверку, что интересующий нас результат входит в определенный диапазон значений. Указывается ключевое слово **BETWEEN**, затем начальное и конечное значения диапазона, разделенные ключевым словом **AND**:

<значение> [**NOT**] **BETWEEN** <начало_диапазона> **AND**
<конец_диапазона>.

Чтобы найти строки, которые не попадают в указанный диапазон, можно воспользоваться оператором **NOT BETWEEN**. Кроме того, в запросе можно использовать и сложные выражения.

Предикат **LIKE** определяет условия поиска в форме шаблона:

<значение> [**NOT**] **LIKE** <шаблон_подстроки>

В шаблоне подстроки, помимо набора интересующих нас символов, допускается применять два трафаретных символа:

- символ подчеркивания «_», который используется вместо любого одиночного символа в выражении;
- символ процента «%», заменяющий последовательность любых символов.

В следующем примере выполняется поиск товаров, чьи производители удовлетворяют шаблону. Шаблон подстроки содержит два символа подчеркивания на второй и четвертой позиции и символ % в конце строки:

```
SELECT * FROM products  
WHERE product_name LIKE 'i_h_ne%';
```

Для нахождения строк, содержащих в столбце неопределенное значение NULL, используется предикат IS NULL, предназначенный для проверки факта отсутствия или наличия значений в столбце отношения. Если же требуется найти строки, которых не содержат неопределенных значений, необходимо использовать предикат IS NOT NULL.

Для получения диапазона строк в разных СУБД представлены разные конструкции. В MySQL и PostgreSQL можно воспользоваться операторами LIMIT и OFFSET, которые имеют немного отличающийся синтаксис. В SQL Server для выбора диапазона строк используется команда TOP.

```
MySQL: LIMIT {[offset,] row_count | row_count  
OFFSET offset}  
PostgreSQL: LIMIT {row_count | ALL} [OFFSET offset]
```

Предикат LIMIT определяет количество извлекаемых строк. OFFSET указывает смещение относительно начала (то есть сколько строк нужно пропустить).

В следующем примере выполняется выборка двух записей, начиная с четвертой.

```
SELECT * FROM products ORDER BY id  
LIMIT 2 OFFSET 3;
```

6.2.5 Агрегатные функции

Агрегатные (обобщающие) функции в качестве исходных параметров принимают значения, указанные в запросе (после слова SELECT), и вычисляют результат. Основные агрегатные функции перечислены в таблице 6.2. Как правило, эти функции применяются в запросах группировки с использованием команды GROUP BY.

Таблица 6.2. Агрегатные функции

Функция	Описание
COUNT	Возвращает количество строк. Тип возвращаемого значения – целое число.
AVG	Вычисляет среднее арифметическое для указанных элементов. Используется только для полей цифровых типов данных. Тип возвращаемого значения – вещественное число.
SUM	Вычисляет сумму значений. Применяется только для цифровых типов данных.
MAX	Возвращает наибольшее из всех значений.
MIN	Возвращает наименьшее из всех значений.

Все агрегатные функции работают с единственным полем таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к текстовым полям. Функции SUM и AVG должны применяться только к числовым полям. Совместно с перечисленными функциями разрешено применять предикат DISTINCT.

Все агрегатные функции за исключением COUNT (*) игнорируют значения NULL.

В следующем примере выполняет расчет средней совокупной стоимости всех товаров определённого производителя с использованием функции AVG:

```
SELECT AVG(price * product_count) AS avg_total_price
```

```
FROM products  
WHERE manufacturer = 'Samsung';
```

Функция COUNT вычисляет количество строк в выборке. Есть две формы этой функции. Первая форма COUNT (*) подсчитывает число строк в выборке. Вторая форма функции с указанием имени столбца вычисляет количество строк по определенному столбцу, при этом строки со значениями NULL игнорируются, например:

```
SELECT COUNT(product_count)  
FROM products;
```

Остальные функции работают аналогично. Значения NULL игнорируются и не учитываются при агрегации значений.

Все функции могут комбинироваться, чтобы в одном запросе выводились результаты применений нескольких функций к столбцам таблицы или вычисляемым выражениям.

В следующем примере вычисляется минимальное и максимальное значение цены для всех товаров, а также общая стоимость всех имеющихся товаров:

```
SELECT MIN(price), MAX(price),  
       SUM(price * product_count)  
FROM products;
```

По умолчанию все вышеперечисленных пять функций учитывают все строки выборки для вычисления результата. Но выборка может содержать повторяющие значения. Если необходимо выполнить вычисления только над уникальными значениями, исключив из набора значений повторяющиеся данные, то для этого применяется оператор DISTINCT.

6.2.6 Группировка данных. Предложение GROUP BY

Предложение GROUP BY предназначено для осуществления

группировки выходных строк по какому-либо признаку, например, по равенству значений каких-либо столбцов. Предложение GROUP BY имеет следующий синтаксис:

```
GROUP BY имя_столбца_1 [, имя_столбца_2, ...]
```

Запрос, в котором присутствует GROUP BY, называется **группирующим** запросом, поскольку в нем группируются данные, полученные в результате выполнения операции SELECT, после чего для каждой отдельной группы создается единственная суммарная строка. При наличии в операторе SELECT предложения GROUP BY каждый элемент списка в предложении SELECT должен иметь единственное значение для всей группы. Более того, предложение SELECT может включать только следующие типы элементов: имена полей, агрегатные функции, константы и выражения, включающие комбинации перечисленных выше элементов.

Все имена полей, приведенные в списке предложения SELECT, должны присутствовать и в предложении GROUP BY за исключением случаев, когда имя столбца используется в агрегатной функции. Обратное правило не является справедливым – в предложении GROUP BY могут быть имена столбцов, отсутствующие в списке инструкции SELECT.

В качестве примера рассмотрим запрос, который группирует товары по производителю и считает количество товаров каждого производителя.

```
SELECT manufacturer, COUNT(*) AS count  
FROM products  
GROUP BY manufacturer;
```

Данный запрос сгруппирует все строки, имеющие одинаковое значение поля «производитель» (manufacturer), в одну строку.

Все имена полей, перечисленные в SELECT, участвуют либо в предложении GROUP BY, либо в агрегатной функции.

Если в выражении SELECT производится выборка по одному или нескольким столбцам, и также используются агрегатные функции, то необходимо использовать выражение GROUP BY.

Следует учитывать, что выражение GROUP BY должно идти после выражения WHERE, но до выражения ORDER BY, т.к. сначала делается фильтрация записей, удовлетворяющих условию WHERE, а потом выполняется группировка строк. Результат группировки сортируется по условию, указанному в ORDER BY.

Оператор GROUP BY также может выполнять группировку по нескольким столбцам.

6.2.7 Фильтрация групп. Предложение HAVING

Оператор HAVING позволяет выполнить фильтрацию групп, то есть определяет, какие группы будут включены в выходной результат:

[**HAVING** условия_отбора]

Использование HAVING во многом аналогично применению WHERE. Только если WHERE применяется для фильтрации строк, то HAVING - для фильтрации групп. В одном запросе можно сочетать выражения WHERE и HAVING.

В следующем примере запроса сначала фильтруются строки: выбираются те товары, общая стоимость которых больше ста тысяч. Затем выбранные товары группируются по производителям. На заключительном шаге фильтруются сами группы: выбираются те группы, которые содержат больше одной модели.

```
SELECT manufacturer, COUNT(*) AS models_count
FROM products
```

```
WHERE price * product_count < 500000  
GROUP BY manufacturer  
HAVING COUNT(*) > 1;
```

Если при этом необходимо провести сортировку, то выражение ORDER BY идет после выражения HAVING.

6.3 Вложенные запросы

Часто невозможно решить поставленную задачу путем одного запроса. Это особенно актуально, когда при использовании условия поиска в предложении WHERE значение, с которым надо сравнивать, заранее не определено и должно быть вычислено в момент выполнения оператора SELECT. В таком случае используются операторы SELECT, вложенные в тело другого оператора SELECT.

Инструкция SELECT допускает вложение в запрос неограниченного количества подзапросов, благодаря которым можно создавать весьма сложные многоярусные конструкции выборки данных.

- Внутренний подзапрос представляет собой также оператор SELECT, а кодирование его предложений подчиняется тем же правилам, что и основного оператора SELECT.
- Внешний оператор SELECT использует результат выполнения внутреннего оператора для определения содержания окончательного результата всей операции.
- Внутренние запросы могут быть помещены непосредственно после оператора сравнения в предложения WHERE и HAVING внешнего оператора SELECT – они получают название подзапросов или вложенных запросов. Внутренние операторы SELECT могут применяться в операторах INSERT, UPDATE и DELETE.

Вложенный запрос (подзапрос) – это инструмент создания временной таблицы, содержимое которой извлекается и обрабатывается внешним оператором. Текст подзапроса должен быть заключен в скобки. К подзапросам применяются следующие правила и ограничения:

- предложение ORDER BY не используется, хотя и может присутствовать во внешнем подзапросе;
- список в предложении SELECT состоит из имен отдельных столбцов или составленных из них выражений – за исключением случая, когда в подзапросе присутствует ключевое слово EXISTS;
- по умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в предложении FROM. Однако допускается ссылка и на столбцы таблицы, указанной в предложении FROM внешнего запроса, для чего применяются квалифицированные имена столбцов (т.е. с указанием таблицы);
- если подзапрос является одним из двух operandов, участвующих в операции сравнения, то запрос должен указываться в правой части этой операции.

По типу результирующего набора данных можно выделить два типа подзапросов:

- скалярный подзапрос, который возвращает единственное значение;
- табличный подзапрос, который возвращает множество значений, т.е. значения одного или нескольких столбцов таблицы, размещенные в более чем одной строке. Табличный подзапрос может применяться везде, где допускается наличие таблицы.

Подзапросы могут применяться во всех командах выборки и модификации данных.

При выборке данных (в инструкции SELECT) вложенные запросы могут применяться:

- в качестве спецификации столбца после выражения SELECT;
- в качестве таблицы для выборки в выражении FROM;
- для фильтрации строк и групп в предложениях WHERE и HAVING соответственно.

При добавлении данных в выражении INSERT вложенные запросы могут применяться для определения значения, которое вставляется в один из столбцов таблицы.

При обновлении данных в выражении UPDATE подзапросы применяются:

- для установки нового значения после оператора SET;
- как часть условия фильтрации в выражении WHERE.

При удалении данных в выражении DELETE вложенные запросы применяются как часть условия в выражении WHERE.

Чтобы рассмотреть выполнение вложенных запросов на примерах, создадим в базе данных таблицу заказов `orders` (дополнительно к существующей таблице товаров `products`, описанной в разделе 6.1):

```
CREATE TABLE orders(
    id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT NOT NULL,
    product_count INT DEFAULT 1,
    created_at DATE NOT NULL,
    FOREIGN KEY (product_id) REFERENCES
    products(id) ON DELETE CASCADE
);
```

Таблица `orders` является дочерней и связана с главной с использованием внешнего ключа.

В первом примере будем использовать вложенный запрос для определения значений при добавлении данных в предложении

`INSERT`. Добавим информацию о заказах, указав идентификатор товара, на который сделан заказ, количество товаров в заказе и дату заказа:

```
INSERT INTO orders (product_id, created_at,  
product_count)  
VALUES  
( (SELECT id FROM products WHERE  
product_name='iPhone 13'), '2022-05-21', 2 );
```

При добавлении данных в таблицу `orders` используется подзапрос для получения информации об идентификаторе нужного товара из таблицы `products`. В примере основной запрос и подзапросы выполнялись к разным таблицам.

Основной и вложенный запрос могут обращаться к одной таблице. Рассмотрим пример использования подзапроса в предложении `SELECT`:

```
SELECT * FROM products  
WHERE price = (SELECT MAX(price) FROM products);
```

В примере осуществляется поиск всех товаров, которые имеют максимальную цену. В таком запросе для вывода товаров с максимальной ценой нельзя использовать простое ограничение `LIMIT`, т.к. количество записей заранее неизвестно, поэтому используется вложенный запрос. В подзапросе сначала находится максимальная цену товара, используя агрегатную функцию `MAX`. На втором шаге в основном запросе выполняется поиск всех товаров с такой ценой. Т.е. в этом примере подзапрос выполняется как часть условия фильтрации в предложении `WHERE`.

В следующем примере необходимо найти информацию о заказах самого дорогого товара:

```
SELECT * FROM orders
WHERE product_id IN (
    SELECT id FROM products
    WHERE price = (
        SELECT MAX(price) FROM products
    )
);
```

В примере используется два подзапроса: сначала выполняется подзапрос, который возвращает максимальную стоимость товара, затем запрос, который возвращает идентификаторы товаров, имеющих максимальную стоимость, и только после этого отрабатывает внешний запрос, который возвращает заказы с самым дорогим товаром. Т.к. в таблице `products` может существовать несколько товаров с максимальной ценой, то подзапрос получения идентификаторов вернет множество значений. Поэтому для поиска заказов выполняется не просто проверка на равенство, а используется оператор `IN`, который выполняет фильтрацию по попаданию значения в множество.

6.3.1 Оператор EXISTS

Оператор `EXISTS` проверяет, возвращает ли подзапрос какое-либо значение. Как правило, этот оператор используется для индикации того, что как минимум одна строка в таблице удовлетворяет некоторому условию. Поскольку возвращения набора строк не происходит, то подзапросы с подобным оператором выполняются достаточно быстро.

Применение оператора имеет следующий формальный синтаксис:

```
WHERE [NOT] EXISTS (<подзапрос>)
```

В следующем примере используется запрос нахождения всех товаров из таблицы `products`, на которые есть заказы в таблице `orders`, с использованием оператора `EXISTS`:

```
SELECT * FROM products
WHERE EXISTS (
    SELECT * FROM orders
    WHERE orders.product_id = products.id
);
```

Стоит отметить, что для получения подобного результата можно было бы использовать и оператор `IN`. Но поскольку при применении `EXISTS` не происходит выборка строк, то его использование более эффективно, чем использование оператора `IN`.

6.3.2 Многократное сравнение

При выполнении подзапроса `IN` проверяется равенство некоторого значения, обрабатываемого основным запросом, какому-то из значений, содержащихся в столбце результатов вложенного запроса. В случае совпадения данная запись добавляется в результирующий набор запроса. В SQL предусмотрены операторы, осуществляющие более сложные правила сравнения, поддерживающие не только элементарную проверку на равенство, но и остальные операторы сравнения – это предикаты `ALL` и `ANY` | `SOME`.

Предикат `ALL` имеет следующий синтаксис:

<сравниваемое значение> <оператор> **ALL** (<подзапрос>) .

Проверяемое значение поочередно сравнивается с каждым элементом, возвращаемым подзапросом:

- если все сравнения дают `TRUE`, то и вся проверка `ALL` возвратит `TRUE`;

- если хотя бы один результат сравнения равен FALSE, то общим результатом также станет FALSE.

Еще одна особенность проверки ALL заключается в том, что если вложенный подзапрос вернет пустой набор данных, то предикат ALL возвратит TRUE.

В следующем примере выполняется поиск всех товаров, цена которых меньше, чем у любого товара указанной компании:

```
SELECT * FROM products
WHERE price < ALL (
    SELECT price FROM products
    WHERE manufacturer = 'Apple'
);
```

Оператор ALL в своем роде является эквивалентом логической операции AND («И»), он возвращает значение TRUE, если подзапрос не возвратит ни одной записи или, наоборот, когда все строки в наборе выдерживают сравнение. Соответственно, результат FALSE вернется в случае, когда сравнение не пройдет хотя бы одна строка в наборе.

Предикаты ANY и SOME равнозначны, их синтаксис выглядит следующим образом:

```
<сравниваемое значение> <оператор отношения>
ANY | SOME (<подзапрос>)
```

Как и в случае с ALL, проверяемое значение последовательно сравнивается с элементами, возвращенными подзапросом:

- если хотя бы одно из сравнений даст положительный результат, то проверка ANY (SOME) также завершится с результатом TRUE;
- иначе результат проверки FALSE.

Если вложенный подзапрос вообще не возвратит данные, то сравнение ANY (SOME) заканчивается значением FALSE.

6.3.3 Коррелирующие подзапросы

Также стоит отметить, что подзапросы классифицируются как коррелирующие и некоррелирующие:

- **некоррелирующий** подзапрос не зависит от основного запроса и выполняется один раз для всего внешнего запроса;
- **коррелирующий** подзапрос (correlated subquery) – подзапрос, который содержит ссылку на столбцы из включающего его основного запроса. Коррелирующий подзапрос будет выполняться для каждой строки основного запроса, так как значения столбцов основного запроса будут меняться.

В следующем примере запрос выбирает информацию о заказах из таблицы `orders`, добавив к ней информацию о товаре:

```
SELECT created_at, product_count,
       (SELECT product_name
        FROM products
        WHERE products.id = orders.product_id) AS
product
FROM orders;
```

В рассмотренном примере подзапрос является спецификацией столбца. Запрос возвращает три столбца, причем в третьем столбце содержится название товара, полученное выполнение вложенного коррелирующего запроса.

Конечно, такой способ объединения данных из нескольких таблиц в одном запросе практически не используется, это просто пример, демонстрирующий, как можно использовать коррелирующие подзапросы. Более эффективный способ для извлечения данных из других таблиц заключается в использовании оператора соединения `JOIN`, который будет рассмотрен в разделе 6.4.

Следует учитывать, что т.к. коррелирующие подзапросы выполняются для каждой отдельной строки выборки, то выполнение таких подзапросов может замедлять выполнение всего запроса в целом.

6.4 Многотабличные запросы

Помимо использования подзапросов, в SQL предусмотрено еще два способа построения запросов к нескольким таблицам – это соединения, осуществляемые с помощью ключевых слов WHERE и JOIN, и слияние таблиц с использованием оператора UNION (рисунок 38).

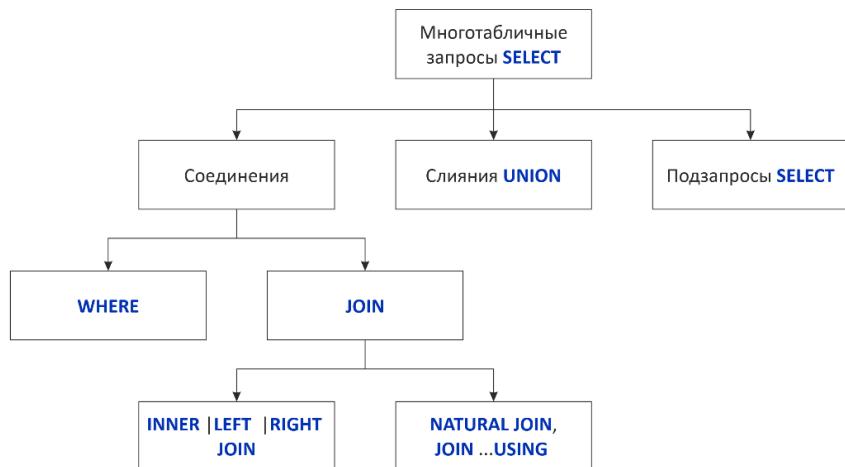


Рисунок 38 – Классификация многотабличных запросов

Для иллюстрации примеров будут использоваться определенные ранее таблицы товаров `products` и заказов `orders` и новая таблица клиентов `customers`:

```
CREATE TABLE customers (
    customer_id INT NOT NULL AUTO_INCREMENT,
```

```
    first_name VARCHAR(30) NOT NULL,  
    PRIMARY KEY (customer_id)  
);
```

Кроме того, в таблице `products` был переименован первичный ключ с `id` на `product_id`. Переименование необязательно для связи таблиц, но так будет проще проиллюстрировать некоторые свойства отношений. В таблицу `orders` был добавлен новый столбец `customer_id` и внешний ключ, указывающий на таблицу `customers`.

6.4.1 Декартово произведение таблиц

Работа с данными нескольких таблиц основывается на операциях теории множеств и реляционной алгебры, которые были рассмотрены в разделе 2.5. Рассмотрим эти операции применительно к таблицам базы данных.

В следующем примере выполняется декартово произведение таблиц `orders` и `customers`:

```
SELECT *  
FROM orders, customers;
```

При такой выборке каждая строка из таблицы `orders` будет соединяться с каждой строкой из таблицы `customers`. Но вряд ли это тот результат, который хотелось бы видеть. Тем более, каждый заказ связан с конкретным покупателем, а не со всеми возможными покупателями.

6.4.2 Внутреннее соединение WHERE

Чтобы решить задачу связи записей по идентификаторам (т.е. по конкретному условию), необходимо использовать предложение `WHERE` для фильтрации строк.

Рассмотрим пример запроса, который выводит информацию о заказах, включая информацию о клиенте и заказанному товару:

```
SELECT customers.first_name, products.product_name,  
       orders.created_at  
FROM orders, customers, products  
WHERE orders.customer_id = customers.customer_id  
AND orders.product_id = products.product_id;
```

Так как здесь нужно соединить три таблицы, то применяются как минимум два условия. Ключевой таблицей остается `orders`, из которой извлекаются все заказы, а затем к ней подсоединяется данные по клиенту по условию равенства значений столбцов `customer_id` из таблиц `orders` и `customers` и данные по товару по условию равенства значений столбцов `product_id`.

В данном примере названия таблиц сильно увеличивают код запроса, но его можно сократить за счет использования псевдонимов таблиц. Ключевое слово `AS` при определении псевдонимов можно опустить:

```
SELECT c.first_name, p.product_name, o.created_at  
FROM orders AS o, customers c, products p  
WHERE o.customer_id = c.customer_id  
AND o.product_id = p.product_id;
```

Результат запроса будет эквивалентен результату предыдущего запроса.

6.4.3 Соединение JOIN

С выходом стандарта SQL-92 вместо соединения с использованием предложения `WHERE` используются синтаксические конструкции, использующие предложение `JOIN`. В SQL таких конструкций несколько.

Во-первых, поддерживается внутреннее соединение INNER JOIN или просто JOIN (ключевое слово INNER является необязательным, по умолчанию используется INNER JOIN):

```
SELECT имя_столбца [, ...]  
FROM <левая_таблица>  
[INNER] JOIN <правая_таблица> [ON <условие>]  
[WHERE условия_отбора]  
...
```

После ключевого слова ON указывается условие соединения, которое устанавливает, как две таблицы будут сравниваться. В большинстве случаев для соединения применяется первичный ключ главной таблицы и внешний ключ зависимой таблицы.

Во-вторых, в SQL осуществлена поддержка так называемого внешнего соединения, для которого требуется явным образом указывать, какого типа соединение требуется получить: левое (LEFT), правое (RIGHT):

```
SELECT имя_столбца [, ...]  
FROM <левая_таблица>  
{LEFT | RIGHT} JOIN <правая_таблица> [ON <условие>]  
[WHERE условия_отбора]
```

При написании запроса важную роль играет последовательность соединения таблиц. По правилам построения запроса JOIN имя левой таблицы следует сразу после ключевого слова FROM, имя правой таблицы – после JOIN.

Если рассмотреть пару таблиц А и В, то с использованием соединения JOIN можно получить различные комбинации данных, как показано на рисунке 39.

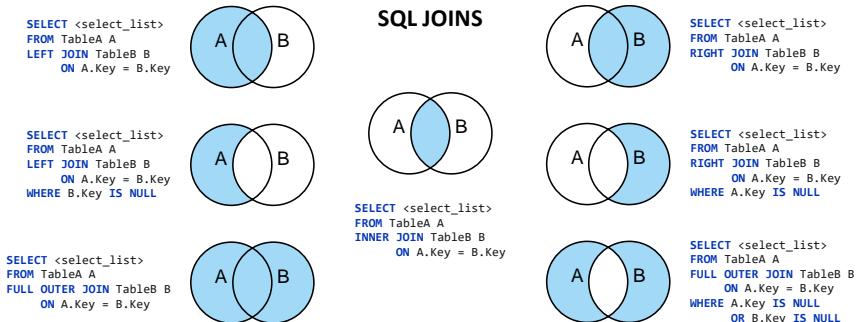


Рисунок 39 – Типы соединений

`INNER JOIN` используется для получения пересечения двух таблиц.

Левое внешнее соединение `LEFT JOIN` возвратит все (даже несвязанные) строки левой таблицы и дополнит их связанными строками правой таблицы. Для несвязанных строк недостающие значения будут заполнены значением `NULL`. Правое внешнее соединение, наоборот, вернет все строки из правой таблицы, дополнив их строками из левой таблицы либо значениями `NULL`.

В ряде СУБД поддерживается полное соединение (`FULL OUTER JOIN`), которое возвращает все строки из левой и правой таблиц.

Рассмотрим операции соединения на примерах.

Используя `JOIN`, выберем все заказы и добавим к ним информацию о товарах. Дата создания заказа и количество товаров в заказе выбираются из таблицы `orders`, название товара выбирается из таблиц `products`. Для сокращения кода запроса используются псевдонимы таблиц:

```
SELECT o.created_at, o.product_count, p.product_name
FROM orders o
JOIN products p ON p.product_id = o.product_id;
```

Стандарт SQL постоянно совершенствуется, добавляя дополнительные синтаксические конструкции использования JOIN. Первая конструкция NATURAL JOIN – естественное соединение.

Суть конструкции естественного соединения заключается в том, что SQL самостоятельно выбирает способ соединения таблиц. В первую очередь проверяются одноименные столбцы в левой и правой таблицах если таковые отсутствуют, то соединение осуществляется по однотипным столбцам конструкции JOIN. Но к использованию такого соединения следует относиться с большой осторожностью, т.к. можно получить неожиданный результат, например, если соединение произойдет по двум одноименным столбцам.

Синтаксическая конструкция JOIN ... USING – соединение с явным указанием столбца. В этом случае указывается имя столбца, которое присутствует в обеих таблицах:

```
SELECT orders.created_at, orders.product_count,  
       products.product_name  
FROM orders  
JOIN products USING (product_id);
```

По своей сути соединения NATURAL JOIN и JOIN ... USING выступают просто сервисными расширениями традиционных конструкций JOIN и предназначены для использования только в том случае, если при проектировании таблиц строго соблюдались правила именования столбцов первичного и внешнего ключей и не было других таблиц с одинаковыми именами, которые не должны участвовать в соединении.

К запросам соединения также можно применять более комплексные условия с фильтрацией и сортировкой.

В отличие от INNER JOIN внешнее соединение возвращает все строки одной или двух таблиц, которые участвуют в соединении.

Левое внешнее соединение LEFT JOIN возвратит все (даже несвязанные) строки левой таблицы и дополнит их связанными строками правой таблицы.

В следующем примере выполняется соединение таблиц customers и orders. В случае, если для клиентов нет связанных заказов, соответствующие столбцы из таблицы orders в выборке будут заполнены значением NULL:

```
SELECT first_name, created_at, product_count  
FROM customers  
LEFT JOIN orders ON orders.customer_id =  
                  customers.customer_id;
```

Правостороннее соединение выбирает все записи из правой таблицы и дополняет их сведениями из левой таблицы или значение NULL.

В следующем примере делается выборка всех товаров с информацией о заказах при их наличии. В запросе используется правостороннее соединение:

```
SELECT o.created_at, o.product_count,  
              p.price, p.product_name  
FROM orders o  
RIGHT JOIN products p USING (product_id);
```

В ряде СУБД поддерживается оператор FULL [OUTER] JOIN, который осуществляет операцию полного внешнего соединения, возвращая все строки из правой и левой таблиц соединения.

И конечно в одном запросе могут комбинироваться различные типы соединений, условия фильтрации, группировки и сортировки.

6.4.4 Объединение UNION

Оператор UNION используется, когда требуется объединить результаты нескольких запросов. Объединение происходит построчно, новых столбцов, в отличии от использования операции соединения, не появляется.

Синтаксис операции имеет следующий вид:

```
SELECT выражение_1  
UNION [ALL]  
SELECT выражение_2  
[UNION [ALL] SELECT выражение_N]
```

По умолчанию, повторяющиеся строки при объединении удаляются. Чтобы сохранить строки-дубликаты, необходимо использовать ключевое слово ALL.

Объединение UNION может осуществляться только у совместимых по объединению отношений. Допустимо объединять отношения только при условии совпадения количества и типа данных объединяемых столбцов.

Для примера использования объединения создадим еще одну таблицу сотрудников employees, совпадающую по структуре с таблицей клиентов. Тогда запрос вывода информации о всех клиентах и сотрудниках в одном результирующем отношении может иметь следующий вид:

```
SELECT first_name FROM customers  
UNION  
SELECT first_name FROM employees;
```

При объединении количество выбираемых столбцов и их тип совпадают для обеих выборок. Результат запроса является объединением двух запросов.

6.5 Модификация данных. Инструкция UPDATE

Для обновления уже существующих в таблице строк применяется команда UPDATE. Она имеет следующий формальный синтаксис:

```
UPDATE {имя_таблицы | имя_представления}
SET {{имя_столбца= выражение | значение | NULL |
      DEFAULT | ARRAY | ROW=строка} [, ...]}

[WHERE условие_отбора | WHERE CURRENT OF имя_кур-
сора]
```

Обычно используется упрощенный синтаксис изменения таблицы, при которых перечисляются названия столбцов со значениями и указывается условие фильтрации строк, к которым будут применены изменения:

```
UPDATE имя_таблицы
SET имя_столбца_1 = значение_1,
    имя_столбца_2 = значение_2, ...,
    имя_столбца_N = значение_N
[WHERE условие_отбора]
```

В следующем примере выполняется изменение значений двух столбцов у строк таблицы, которые удовлетворяют условию, указанному в предложении WHERE:

```
UPDATE products
SET manufacturer = 'Samsung',
    product_count = product_count + 3
WHERE manufacturer = 'Samsung Inc.');
```

При обновлении вместо конкретных значений и выражений могут использоваться ключевые слова DEFAULT и NULL для установки соответственно значения по умолчанию или NULL.

6.6 Удаление данных. Инструкция DELETE

Инструкция DELETE предназначена для удаления строк из таблиц данных:

```
DELETE FROM имя_таблицы  
[WHERE условие_отбора | WHERE CURRENT OF имя_курсора]
```

В большинстве СУБД операция удаления не приводит к немедленному физическому стиранию строки. Вместо этого специальный служебный бит удаления строки помечается соответствующим образом, после чего сервер начинает полагать, что строки не существует.

В следующем примере удаляются строки таблицы, удовлетворяющие составному условию, объединенному логическим оператором AND.

```
DELETE FROM products  
WHERE manufacturer = 'Apple' AND price < 90000;
```

Для удаления всех строк таблицы можно воспользоваться командой DELETE, не указывая предикат фильтрации данных, т.е. без предложения WHERE:

```
DELETE FROM products;
```

Если необходимо удалить все строки из таблицы, можно также использовать команду TRUNCATE:

```
TRUNCATE TABLE products;
```

Отличия в использовании операторов DELETE и TRUNCATE:

- TRUNCATE выполняется быстрее;
- действие команды TRUNCATE нельзя отменить оператором отката транзакции;
- если для таблицы определены автоинкрементные поля, то значение их счетчиков обнуляется после использования команды TRUNCATE.

7 ПРЕДСТАВЛЕНИЯ

7.1 Понятие представления

Представление (VIEW) – объект базы данных, являющийся результатом выполнения запроса к базе данных, определенного с помощью оператора SELECT, в момент обращения к представлению.

Представления иногда называют «виртуальными таблицами». Такое название связано с тем, что представление доступно для пользователя как таблица, но само оно не содержит данных, а извлекает их из таблиц в момент обращения к нему. Если данные изменены в базовой таблице, то пользователь получит актуальные данные при обращении к представлению, использующему данную таблицу; кэширования результатов выборки из таблицы при работе представлений не производится. При этом, механизм кэширования запросов (query cache) работает на уровне запросов пользователя безотносительно к тому, обращается ли пользователь к таблицам или представлениям.

Можно отметить следующие преимущества использования представлений:

- 1) дает возможность гибкой настройки прав доступа к данным за счет того, что права даются не на таблицу, а на представление. Это позволяет назначить права на отдельные строки таблицы или дает возможность получать не сами данные, а результат каких-то действий над ними;
- 2) позволяет разделить логику хранения данных и программного обеспечения. Можно изменять структуру данных, не затрагивая программный код. При изменении схемы базы данных достаточно создать представления, аналогичные таблицам, к которым раньше обращались приложения. Это

- удобно, когда нет возможности изменить программный код, или к одной базе данных обращаются несколько приложений с различными требованиями к структуре данных;
- 3) удобство в использовании за счет автоматического выполнения таких действий, как доступ к определенной части строк и/или столбцов, получение данных из нескольких таблиц и их преобразование с помощью различных функций;
 - 4) поскольку SQL-запрос, выбирающий данные представления, зафиксирован на момент его создания, СУБД получает возможность применить к этому запросу оптимизацию или предварительную компиляцию, что может повысить производительность выполнения запросов.

Упрощенный синтаксис создания представлений в СУБД MySQL имеет следующий вид:

```
CREATE [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Данное выражение создает новое представление или заменяет существующее, если используется выражение OR REPLACE. Выражение select_statement определяет представление. SELECT может относиться как к таблицам, так и к другим представлениям, т.е. поддерживается использование вложенных представлений.

Представления должны иметь уникальные имена столбцов. По умолчанию имена столбцов, полученные выражением SELECT, используются для имени столбца представления. Явные имена для столбцов представления могут быть заданы с помощью выражения column_list как список разделяемых запятой идентификаторов.

Опция ALGORITHM определяет, как СУБД обрабатывает представление:

- MERGE: текст запроса к представлению объединяется с текстом запроса самого представления к таблицам БД, результат выполнения объединенного запроса возвращается пользователю;
- TEMP TABLE: содержимое представления сохраняется во временную таблицу, которая затем используется для выполнения запроса;
- UNDEFINED (значение по умолчанию): СУБД самостоятельно выбирает, какой алгоритм использовать при обращении к представлению.

При использовании в определении представления конструкции WITH [CASCADED | LOCAL] CHECK OPTION все добавляемые или изменяемые строки будут проверяться на соответствие определению представления.

Рассмотрим пример представления из базы данных sakila (учебной базы данных, поставляемой с СУБД MySQL), возвращающего продажи по каждой категории фильма.

```
CREATE ALGORITHM=TEMPTABLE
VIEW sales_by_film_category AS

SELECT c.name AS category, SUM(p.amount) AS sales
FROM payment p
JOIN rental r ON p.rental_id = r.rental_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
```

```
GROUP BY c.name  
ORDER BY total_sales DESC;
```

Т.е. в примере создается представление sales_by_film_category с алгоритмом обработки TEMPTABLE. Представление определяется запросом, объединяющим данные из шести таблиц, с группировкой данных.

В следующем примере выполняется запрос к представлению:

```
SELECT *  
FROM sales_by_film_category  
ORDER BY category  
LIMIT 5;
```

Работа с представлениями происходит так же, как и с таблицами – поэтому представления иногда и называют виртуальными таблицами. В запросе вместо обращения к таблице делается обращение к представлению.

7.2 Обновляемые представления

Представление называется **обновляемым**, если к нему могут быть применимы операторы UPDATE и DELETE для изменения данных в таблицах, на которых основано представление.

Для того, чтобы представление было обновляемым, должны быть выполнены некоторые условия, в т.ч.:

- отсутствие функций агрегации в представлении;
- отсутствие следующих в представлении выражений DISTINCT, GROUP BY, HAVING, UNION;
- отсутствие подзапросов в списке выражения SELECT;
- столбцы представления должны быть простыми ссылками на поля таблиц (а не, например, арифметическими выражениями) и т.д.

Обновляемое представление может допускать добавление данных (выполнение инструкции `INSERT`), если все поля таблицы-источника, не присутствующие в представлении, имеют значения по умолчанию.

Для представлений, основанных на нескольких таблицах, операция добавления данных (`INSERT`) выполняется только в случае, если происходит добавление в единственную реальную таблицу. Удаление данных (`DELETE`) для таких представлений не поддерживается.

При использовании в определении представления конструкции `WITH [CASCADED | LOCAL] CHECK OPTION` все добавляемые или изменяемые строки будут проверяться на соответствие определению представления:

- изменение данных (`UPDATE`) разрешено только для строк, удовлетворяющих условию `WHERE` в определении представления. Кроме того, новые значения строки также должны удовлетворять значениями удовлетворяет условию `WHERE`;
- добавление данных (`INSERT`) разрешено, только если новая строка удовлетворяет условию `WHERE` в определении представления.

Ключевые слова `CASCADED` и `LOCAL` определяют глубину проверки для представлений, основанных на других представлениях:

- для представления с модификатором `LOCAL` происходит проверка условия `WHERE` только в собственном определении представления;
- для представления с модификатором `CASCADED` происходит проверка для всех представлений, на которых основано данное представление. Значением по умолчанию является `CASCADED`.

В следующем примере создается обновляемое представление, которое возвращает товары только для одного производителя:

```
CREATE VIEW apple_products AS
SELECT *
FROM products
WHERE manufacturer = 'apple';
```

Использование такого представления может быть полезно в случае, если необходимо разграничить использование таблицы разными пользователями или приложениями и не предоставлять доступ к товарам сразу всех производителей. Т.к. в данном примере не определено никаких ограничений на представление, то возможно выполнить обновление данных в представлении и, соответственно, в таблице, изменив производителя.

В следующем примере создается вложенное представление, для которого изменяемые строки будут проверяться на соответствие определению представления:

```
CREATE VIEW iphone_products AS
SELECT *
FROM apple_products
WHERE product_name LIKE 'iPhon%'
WITH CASCADED CHECK OPTION;
```

Представление объявлено с модификатором WITH CASCADED CHECK OPTION; т.е. проверка строк будет проводиться для всех представлений, на которых основано данное представление.

При попытке обновления данных в этом представлении на те, которые не будут удовлетворять условиям в предложении WHERE представлений, запрос завершится с ошибкой.

8 ИНДЕКСИРОВАНИЕ

8.1 Понятие индекса

Рассмотрим на примере, для чего необходимо выполнять индексирование таблиц. Допустим, база данных содержит телефонный справочник с указанием фамилии и телефонного номера владельца, причем данные хранятся неупорядоченно. Если потребуется найти номер телефона конкретного человека, то необходимо будет выполнить последовательный просмотр всех строк таблицы для поиска нужного значения. Такой последовательный перебор записей может занимать много времени, особенно, если таблица содержит большое число строк. Конечно, можно реализовать упорядоченное хранение строк данных на диске по фамилии владельца, как в бумажном телефонном справочнике. Однако, если нам понадобится решить обратную задачу – найти владельца телефонного номера – без перебора всех строк опять не обойтись.

Для решения быстрого поиска данных в СУБД используется механизм индексирования данных. **Индекс** – это объект базы данных, создаваемый с целью повышения производительности поиска данных.

Реляционная таблица может одновременно содержать несколько индексов:

- индекс первичного ключа создается автоматически в момент создания первичного ключа таблицы (PRIMARY KEY). Индекс первичного ключа должен гарантировать, что во входящих в состав первичного ключа столбцах будут храниться уникальные данные;
- индексы внешнего ключа создаются автоматически в момент создания внешнего ключа таблицы (FOREIGN KEY).

Основная задача индекса внешнего ключа – быстрый поиск записей во внешних таблицах для соединения с этими таблицами;

- пользовательские индексы создаются вручную. Они описывают дополнительные способы упорядочивания строк таблицы, ускоряют поиск данных и могут проверять данные на уникальность.

Быстрый доступ к данным – это главное преимущество индекса, благодаря которому в базе данных реализуется несколько важных сервисных возможностей:

- ускорение упорядочивания строк отношений;
- ускорение поиска данных по полному или частичному совпадению;
- ускорение соединения таблиц, связанных по первичному и внешнему ключам;
- возможность поддержки уникальности данных за счет защиты от ввода повторяющихся значений.

Но, конечно, за прирост в скорости поиска приходится платить.

Для хранения индексов в базе данных приходится создавать дополнительные структуры данных. В этих структурах хранятся код данных и указатель на местоположение этих данных для каждой строки проиндексированной таблицы. Чем больше индексов, тем больше размерность дополнительных структур. Может случиться так, что суммарный размер служебной информации, отводимой для хранения индексов, превысит размер полезных данных. Кроме того, индексы также увеличивают время выполнения запросов на добавление, изменение и удаление данных из-за необходимости обновления индекса после выполнения соответствующих запросов.

Существуют различные типы индексов, наиболее популярные из которых:

- индексы, базирующиеся на технологии хеширования (hash);

- индексы на основе сбалансированных В-деревьев (B-tree);
- пространственные индексы (Spatial grid, R-tree);
- битовые индексы;
- полнотекстовые индексы.

8.2 Типы индексов

8.2.1 Индексы на основе хеширования

Хеширование (hashing) полезно в том случае, когда требуется большой объем значений сохранить в сравнительно небольшой по размерности таблице (точнее, хеш-таблице) и обеспечить быстрый доступ к этим значениям.

Существует множество методов хеширования данных. Их общей идеей является применение к данным некоторой хеш-функции, по определенному алгоритму вырабатывающей значение меньшего размера (хеш-значение). Например, на вход функции поступает текст (допустим, название товара), а на выходе получаем какое-то число. Свертка исходных данных – односторонний процесс: получив хеш-значение, не получится восстановить по нему исходные данные, но при построении индексов в этом и нет необходимости. Главное, чтобы хеш-функция на одни и те же входные данные всегда возвращала одно и то же выходное значением.

Для хранения индексных данных СУБД создает отдельную хеш-таблицу, состоящую из двух столбцов: хеш-значения и адресного поля, содержащего указатель на строку индексируемой таблицы (рисунок 40). Изначально адресное поле не заполнено и хранит неопределенный указатель NULL. Система начинает процесс индексирования – последовательно читает названия товаров и передает их на вход хеш-функции. Хеш-функция в соответствии с заложенным в нее математическим выражением осуществляет вычисляет числовое хеш-значение.

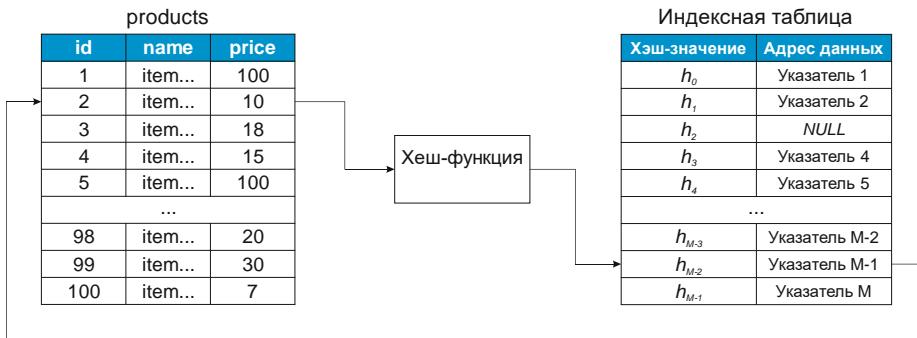


Рисунок 40 – Индекс на основе хеширования

Допустим, что хеширование текстовой строки с $id=2$ дало хеш-значение h_{M-2} . Переходим к строке с порядковым номером h_{M-2} в индексной таблице (хеш-таблице) и в ее столбец «Адрес данных» вносим указатель на физическую запись с нужным идентификатором индексируемой таблицы. Эта операция повторяется до тех пор, пока не построится индекс для всей таблицы. По завершении индексирования в хеш-таблице буду содержаться пары: хеш-значение и соответствующий этому значению указатель на строку с реальными данными таблицы

Благодаря индексам существенно упрощается поиск данных: теперь вместо перебора всех строк исходной таблицы с целью нахождения совпадения система передаст содержание интересующей нас строки хеш-функции. Хеш-функция вычисляет хеш-значение и возвращает номер строки в индексной хеш-таблице. После этого остается перейти к искомой строке по хранящемуся в индексной таблице указателю.

После создания индекса СУБД обязана поддерживать его в актуальном состоянии. Это означает, что при вводе в базовую таблицу нового кортежа, редактировании и удалении строки СУБД индексирует изменения и обновляет данные в индексной таблице. Объем вычислительных ресурсов, затрачиваемых на обновление индексов,

прямо пропорционален количеству модифицированных записей и в отдельных случаях может существенно снизить производительность БД.

В процессе построения индекса на основе технологии хеширования ключевая роль отводится хеш-функции, отвечающей за свертку исходных данных в хеш-значение заведомо меньшей размерности.

Идеальных функций хеширования не существует, кроме того, размер хеш-таблицы существенно меньше исходной таблицы, поэтому всегда имеется вероятность того, что в результате хеширования два (или более) различных входных значения свернутся в одно и то же хеш-значение и, как следствие, будут направлены в одну и ту же строку хеш-таблицы. Однако строка хеш-таблицы обладает единственным полем указателя, поэтому одновременно не способна ссылаться на несколько записей исходной таблицы.

Одно из наиболее распространенных решений данной проблемы – разрешение коллизий с помощью цепочек (рисунок 41).

Индексная таблица

Хеш-значение	Адрес данных
h_0	Указатель 1
h_1	Указатель 2
h_2	NULL
h_3	Указатель 4
h_4	Указатель 5
...	
h_{M-3}	Указатель M-2
h_{M-2}	Указатель M-1
h_{M-1}	Указатель M

The diagram illustrates the resolution of collisions in an index table. It shows a table with columns 'Хеш-значение' (Hash value) and 'Адрес данных' (Address of data). The 'Address of data' column contains pointers to external locations. A horizontal arrow points from the 'Address of data' column to a linked list structure. The list consists of two boxes connected by a blue arrow. The first box contains a black dot and a black square. The second box contains a black dot and the text 'NULL'. A bracket above the second box indicates it continues. A red arrow points from the 'Address of data' row for h_3 to the first box in the list.

Рисунок 41 – Разрешение коллизий

В этом случае предусматривается возможность прикрепления к каждой из ячеек хеш-таблицы дополнительной структуры, например связного списка. Как только в результате коллизии в уже заня-

тую ячейку попадает ссылка на еще одну строку, механизм построения индекса создает дополнительный элемент связного списка (который состоит из двух полей – поля данных и поля – указателя на следующий элемент списка) и заносит указатель на строку с исходными данными уже не в ячейку таблицы, а во вновь созданный элемент списка. В результате выстраивается цепочка из элементов списка, а в ячейке хеш-таблицы теперь хранится указатель не на строку исходной реляционной таблицы, а на первый элемент списка

Наличие цепочки несколько замедляет процесс поиска данных, ведь теперь приходится просматривать все элементы связного списка, но это все равно намного быстрее, чем перебирать все записи неиндексированной таблицы.

Преимуществом использования хеш-индексов является быстрый доступ к данным, если нет большого количества коллизий.

К недостаткам хеш-индексов можно отнести следующие:

- нельзя использовать данные в индексе, чтобы избежать чтения строк;
- нельзя использовать для сортировки, поскольку строки в индексе не хранятся в отсортированном порядке;
- хеш-индексы не поддерживают поиск по частичному ключу, т.к. хеш-коды вычисляются для всего индексируемого значения;
- хеш-индексы поддерживают только операцию сравнения на равенство;
- некоторые операции обслуживания индекса могут оказаться медленными, если количество коллизий велико.

8.2.2 Индексы на основе В-деревьев

Семейство индексов на основе В-деревьев – это наиболее часто используемый тип индексов, организованных как сбалансирован-

ное дерево упорядоченных ключей. Они поддерживаются практически всеми СУБД как реляционными, так и не реляционными, и практически для всех типов данных.

При построении индекса будет создана многоуровневая сбалансированная иерархическая структура. Страницы верхних уровней В-дерева станут выступать в роли индекса, а листья самого нижнего уровня станут хранить данные и указатели на записи в таблице (рисунок 42).

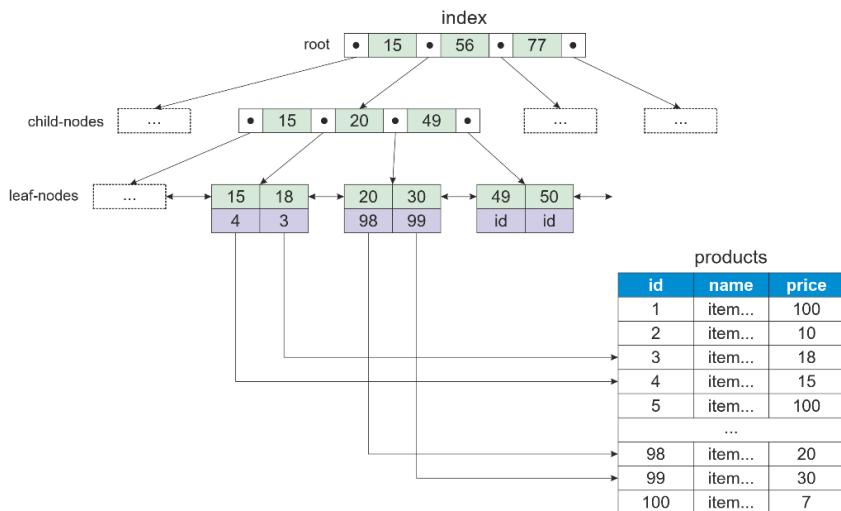


Рисунок 42 – Разрешение коллизий

Как только возникает задача поиска какой-нибудь записи, считывается индекс корневой страницы и по указателю выполняется переход к нужной дочерней странице, затем считывается индекс у дочерней страницы и выполняется переход к следующему узлу. Процесс продолжается до тех пор, пока не доберемся к требуемому листу с данными. Благодаря тому, что В-дерево сбалансировано, поиск в индексе осуществляется за одинаковое количество переход-

дов по индексным страницам, вне зависимости от того, какие данные необходимо найти. Такая особенность В-деревьев хорошо сказывается на производительности СУБД.

Индексы на основе В-деревьев обладают рядом преимуществ по сравнению с хеш-индексами:

- поддерживается поиск по полному значению;
- поддерживается поиск по префиксу столбца;
- поддерживается поиск по диапазону значений;
- поддерживается поиск по полному совпадению одной части и диапазону в другой части;
- если индекс хранит в себе отдельные поля таблицы, а не только указатели на строки, то запросы на эти поля можно выполнять только по индексу, не переходя к строкам таблицы.

В качестве недостатков можно указать следующие:

- если рассматривается составной индекс, то нельзя искать по правой части ключа, без использования левой части ключа;
- нельзя пропускать столбцы составного ключа;
- нельзя выполнять оптимизацию после поиска в диапазоне, т.е. необходимо просматривать весь диапазон;
- большие затраты на перестроение индекса, в особенности, если было изменено, добавлено или удалено большое количество записей. Это приводит к расщеплению большого количества переполненных вершин или, наоборот, слиянию пустых или полупустых вершин и листьев.

8.2.3 Пространственные индексы

Пространственные индексы необходимы для индексирования пространственных данных. Пространственные данные представляют сведения о физическом расположении (т.е. координатах) и

форме геометрических объектов. Этими объектами могут быть объекты простых типов, такие как точки, линии, полигоны, а также более сложные объекты, представляемые коллекциями объектов. В данный момент все крупные СУБД имеют пространственные типы данных и функции для работы с ними.

Для пространственных типов данных существуют особые методы индексирования на основе сеток (grid-based spatial index: MS SQL Server) и R-деревьев (R-Tree index: MySQL, PostgreSQL).

Spatial grid (или пространственная сетка) – это древовидная структура, подобная B-дереву, которая используется для организации доступа к пространственным данным, то есть для индексации многомерной информации, такой, например, как географические данные с двумерными координатами (широтой и долготой). В этой структуре узлами дерева выступают ячейки пространства (рисунок 43).

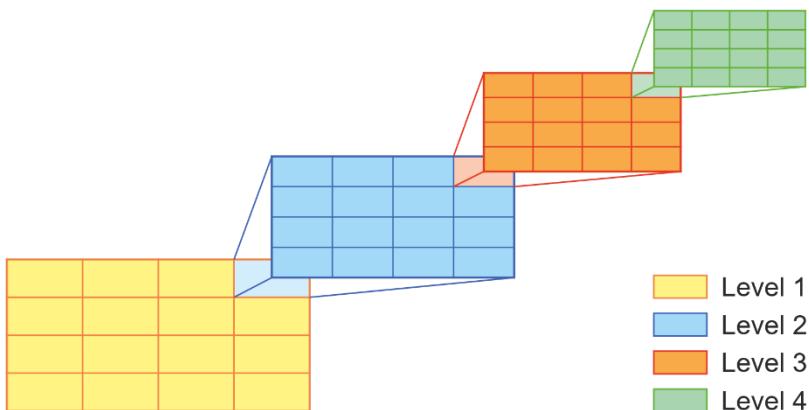


Рисунок 43 – Пространственная сетка

Например, для двухмерного пространства сначала вся родительская площадь будет разбита на сетку строго определенного разрешения, затем каждая ячейка сетки, в которой количество объектов превышает установленный максимум объектов в ячейке, будет

разбита на подсетку следующего уровня. Этот процесс будет продолжаться до тех пор, пока не будет достигнут максимум вложенности (если установлен), или пока все не будет разделено до ячеек, не превышающих максимум объектов.

Следующий тип пространственных индексов – индексы на основе R-деревьев. R-деревья – это древовидная структура данных, подобная Spatial Grid. Эта структура данных также разбивает пространство на множество иерархически вложенных ячеек, но которые, в отличие от Spatial Grid, не обязаны полностью покрывать родительскую ячейку и могут пересекаться (рисунок 44).

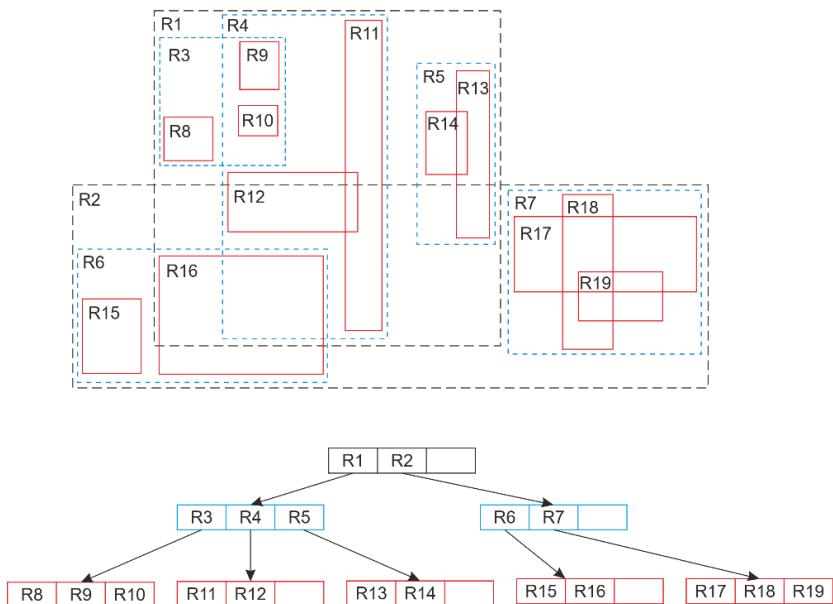


Рисунок 44 – Индекс на основе R-дерева

Для расщепления переполненных вершин могут применяться различные алгоритмы, что порождает деление R-деревьев на подтипы: с квадратичной и линейной сложностью.

Типичным запросом с использованием R-деревьев может являться следующий: «Найти все объекты в пределах 2 километров от моего текущего местоположения».

8.2.4 Битовые индексы

Основное назначение битовых индексов (bitmap indexes) – индексирование столбцов с небольшим количеством различных значений.

Классический пример – пол человека. С точки зрения базы данных его можно задать тремя значениями: «М», «Ж» и NULL. В подобных индексах для представления существования определенного значения столбца целесообразно использовать битовую последовательность (строку из нулей и единиц), как показано на рисунке 45.

Данные

Id	last_name	first_name	gender
1	Орехов	Владимир	Мужской
2	Петровский	Александр	Мужской
3	Кульгина	Оксана	Женский
4	Самойлов	Евгений	Мужской
5	Кузьмина	Ирина	Женский

Битовые маски

value	first-id	bitmask
Женский	1	00101
Мужской	1	11010

Рисунок 45 – Битовый индекс

Если индекс на основе В-деревьев хранит однозначное соответствие между строкой таблицы и записью в индексе, то битовый индекс обеспечивает ссылку на большое количество строк одновременно. Индексы на основе битовых карт занимают малый размер и поэтому обеспечивают гораздо более высокую скорость выборки.

8.2.5 Полнотекстовый индекс

Полнотекстовый индекс используется для полнотекстового поиска. Полнотекстовый поиск – поиск по содержимому текстовых документов больших объемов.

Полнотекстовый индекс – словарь, в котором перечислены все слова и указано, в каких местах документа (строках таблицы) они встречаются. При наличии такого индекса достаточно осуществить поиск нужных слов в индексе и тогда сразу же будет получен список документов, которые содержат искомую информацию.

Преимущества полнотекстового поиска по сравнению с оператором LIKE:

- поддержка морфологии;
- выдача результатов по релевантности;
- наличие модификаторов для настройки полнотекстового поиска;
- стоп-слова, которые будут проигнорированы при поиске.

8.2.6 Кластерные индексы

Кластерный индекс — это такой способ организации индекса, при котором значения индекса хранятся вместе с данными, им соответствующими. И индексы, и данные при такой организации упорядочены. Некластерный индекс хранится отдельно от данных, и, соответственно, в этом случае упорядоченный индекс ссылается на неупорядоченные строки таблицы.

Как правило, если для таблицы определен кластерный индекс, то некластерный ссылается на него. С точки зрения эффективности использования кластерные индексы эффективнее, т.к. после выполнения поиска не надо делать соответствующую выборку данных из таблицы. Это позволяет минимизировать количество операций чтения с диска при работе с таким индексом. В таблице может быть только один кластерный индекс.

8.3 Селективность индекса

Под **избирательностью** (селективностью) индекса понимается число кортежей, которые могут быть выбраны по каждому значению индекса во время поиска. Более избирательный индекс учитывает больше значений и отбирает меньший объем данных одним своим значением, и наоборот – индекс с низкой избирательной способностью (имеющий всего несколько значений в большой таблице) может оказаться бесполезным в запросах.

Формула расчета селективности индекса S имеет следующий вид:

$$S = \frac{1}{\text{число уникальных значений в столбце(ах)}}.$$

Чем меньше результат, тем лучше с точки зрения оптимизатора. Например, в случае с уникальным индексом селективность с увеличением строк в таблице стремится к 0. Здесь каждому значению индекса соответствует единственная запись в таблице, а повторяющихся ключей нет. Если же индекс не уникален и допускает хранение значений-дубликатов, то селективность падает. При самом плохом развитии ситуации (когда все значения ключей одинаковы) селективность стремится к 1. Показатель селективности индекса в первую очередь важен для оценки быстродействия индекса. Чем выше селективности, тем производительнее индекс.

В большинстве СУБД индекс основан на структуре В-дерева, основным достоинством которого считается сбалансированность. Благодаря сбалансированности мы, находясь в корне дерева, достигнем любого из листов нижнего уровня за одно и то же количество шагов и быстро обнаружим искомый ключ. Но только при одном условии – если индекс уникален. В противном случае на самом нижнем уровне дерева появляются цепочки дубликатов (рисунок 46).

Каждая цепочка хранит только одинаковые элементы-дубликаты. С появлением очередного дубликата он вставляется в начало цепочки, а остальные элементы смещаются. Таким образом, чем больше в таблице повторяющихся значений, тем длиннее их цепочка. Теперь при поиске нужного значения индекс будет вынужден затрачивать дополнительные усилия на просмотр соответствующего списка элементов.

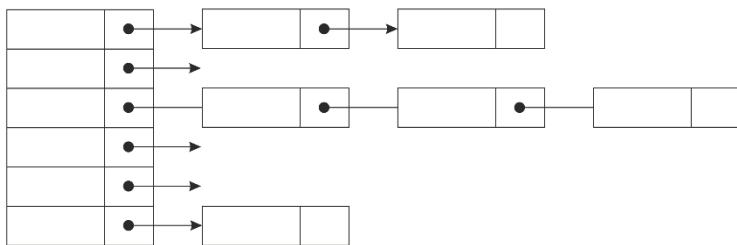


Рисунок 46 – Лист нижнего уровня В-дерева с цепочками дубликатов

Т.е. здесь возникают те же проблемы, что и с коллизиями в хеш-индексе.

8.4 Рекомендации по созданию индексов

Т.к. первичные индексы и индексы внешних ключей создаются автоматически, рассмотрим, когда вручную необходимо создавать пользовательские индексы. В первую очередь следует индексировать столбцы отношений, которые наиболее часто используются в запросах на выборку данных.

Если столбцы используются в сортировке или группировке данных, последовательность столбцов в составном индексе должна соответствовать последовательности столбцов, следующих в инструкции SELECT сразу после предложения ORDER BY или GROUP BY.

Наиболее эффективны индексы, накладываемые на столбцы, в которых не хранятся повторяющиеся значения. В большинстве случаев не стоит индексировать столбцы с небольшим количеством многократно повторяющихся значений или столбцы с часто изменяющимися значениями.

Важно помнить, что индексы предполагают дополнительные операции записи на диск. При каждом обновлении или добавлении данных в таблицу, происходит также обновление данных в индексе. Поэтому создавать следует только необходимые индексы, чтобы не расходовать ресурсы сервера. Не имеет смысла создавать индексы на таблицах, число записей в которых меньше нескольких тысяч. Для таких таблиц выигрыш от использования индекса будет практически незаметен.

Не следует создавать индексы заранее. Индексы должны устанавливаться под форму и тип нагрузки работающей системы. Создание индексов следует начинать с самых частых запросов.

Неиспользуемые индексы необходимо удалять.

8.5 Создание индексов

Система индексирования используется во всех СУБД без исключения, но в стандартном SQL синтаксическая конструкция создания индексов отсутствует. Вместе с тем диалекты SQL большинства производителей весьма схожи и придерживаются схожей конструкции создания индексов.

Упрощенный синтаксис создания индекса в MySQL имеет следующий вид:

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name  
[index_type]  
ON tbl_name (key_part,...)
```

```
key_part: {col_name [(length)] | (expr)} [ASC |  
DESC]
```

```
index_type:  
    USING {BTREE | HASH}
```

Для удаления индекса используется оператор определения данных **DROP**:

```
DROP INDEX index_name ON tbl_name
```

9 ПРОЦЕДУРНЫЙ SQL

При разработке языка SQL ставилась цель разработать декларативный язык, которым могли бы пользоваться не только программисты, но и пользователи базы данных. В соответствии с таким подходом оператору было достаточно лишь поставить СУБД правильно сформулированную задачу, а каким образом СУБД станет решать эту задачу, пользователя не касалось.

Современный SQL имеет все признаки декларативного языка. Например, рассмотрим простейший SQL-запрос, найти минимальное значение цены товара:

```
SELECT MIN(price) FROM products;
```

По сути, запрос звучит так: «Выбрать минимальное из значений столбца `price` из таблицы `products`». Другими словами, задача СУБД поставлена на естественном языке. Если схожая задачу решалась на классическом процедурном языке программирования, необходимо было бы объявить ряд переменных и организовать цикл, в котором бы осуществлялся последовательный перебор всех значений столбца и проводились операции сравнения текущего значения с предыдущим.

Однако возможности декларативного языка не безграничны. В тех ситуациях, когда необходимая операция с данными хотя бы на немного отходит от привычной для анализатора SQL структуры, язык перестает справляться. В результате во всех диалектах SQL стали появляться процедурные элементы.

9.1 Процедурные расширения SQL

В 1996 году институтом ANSI был принят стандарт SQL/PSM для постоянно хранимых модулей в качестве расширения SQL.

Стандарт поддерживает процедурное программирование в дополнение к выражениям запроса языка SQL. Расширение SQL/PSM стандартизирует процедурное расширение для SQL, включая следующие функции:

- управление потоком выполнения;
- обработка условий;
- обработка флагов состояний;
- поддержка курсоров и локальных переменных, а также присваивание выражений переменным и параметрам;
- формализация объявления и поддержка постоянных подпрограмм языков баз данных (например, «хранимых процедур»).

Практически каждый производитель SQL стал разрабатывать свои процедурные расширения: PL/SQL в Oracle, Transact-SQL в Microsoft SQL Server, SQL/PSM в MySQL, PL/pgSQL в PostgreSQL.

В дополнение к стандартным расширениям SQL/PSM и proprietарным расширениям SQL процедурное и объектно-ориентированное программирование доступно на многих платформах SQL посредством интеграции СУБД с другими языками. Стандарт SQL определяет расширения для поддержки кода Java в базах данных SQL. PostgreSQL позволяет пользователям писать функции на широком различные языки, включая Perl, Python, C и другие.

Основными процедурными расширениями являются хранимые процедуры, функции и триггеры. **Хранимые процедуры, функции и триггеры** – это объекты базы данных, представляющие собой набор SQL-инструкций, который компилируется и хранится как самостоятельный исполняемый код в системном каталоге базы данных.

Таким образом, эти объекты можно рассматривать как подпрограммы, в которых могут производиться числовые вычисления и операции над символьными данными, результаты которых могут

присваиваться переменным и параметрам. Они поддерживают стандартные операции работы с базами данных, в них возможны циклы, ветвления и работа с переменными.

Кроме того, хранимые процедуры и функции могут иметь входные параметры и возвращаемые значения.

Ключевое отличие процедур и функций, с одной стороны, и триггеров – с другой, заключается в том, что первые могут вызываться с любого места инструкции SQL с помощью оператора CALL (EXECUTE) или SELECT, а также программы, написанной на высокуюровневом языке, а триггеры запускаются СУБД автоматически по определенному событию без какого-либо вмешательства извне. Например, триггер может быть вызван перед добавлением новой записи в таблицу базы данных или после удаления записи.

В использовании хранимых процедур и функций есть ряд преимуществ:

- *повышение производительности.* В базе данных хранимые процедуры хранятся в откомпилированном и оптимизированном виде. Как следствие, выполнение хранимой процедуры происходит быстрее, чем запуск аналогичного кода динамического SQL;
- *снижение объема передаваемых данных.* Для вызова хранимой процедуры достаточно указать ее имя и значения параметров, в любом случае это меньше, чем отправка по сети полного текста SQL-запроса. Как следствие можно сократить сетевой трафик;
- *обеспечение безопасности данных.* Хранимые процедуры могут использоваться для защиты от несанкционированного доступа. Код хранимых процедур и функций известен, в то время как SQL-запрос может быть сформирован злоумышленником и содержать вредоносные команды. На выполнение хранимой

- процедуры могут быть назначены права отдельным пользователям, которые даже не будут знать, какие таблицы задействованы для выполнения поставленной задачи;
- *повторное использование кода.* Хранимые процедуры, выполняющие однотипные действия, могут переноситься между базами данных с незначительной модификацией.

9.2 Управляющие конструкции SQL

Рассмотрим управляющие конструкции языка SQL на примере СУБД MySQL.

9.2.1 Переменные

Наиболее распространенный способ хранения данных, используемых в промежуточных вычислениях, реализуется с использованием переменных. Переменные позволяют размещать в памяти значения различных типов (целые и вещественные числа, символы, текстовые строки и т.д.).

Можно выделить три категории переменных: локальные, пользовательские и системные переменные.

Наиболее часто используются **локальные** переменные, которые применяются в хранимых процедурах, функциях и триггерах. Для объявления переменных используют синтаксическую конструкцию следующего вида:

```
DECLARE var_name [, var_name, ...] type [DEFAULT  
value]
```

Для объявления используется оператор **DECLARE**, после которого указывается имя и тип переменной. Чтобы задать значение по умолчанию для переменной следует использовать предложение **DEFAULT**. Значение может быть задано в виде выражения; оно не

обязательно должно быть константой. Если предложение DEFAULT отсутствует, начальным значением переменной будет NULL.

Пример объявления целочисленной переменной и инициализации ее значением 0:

```
DECLARE count INT 0;
```

Для присваивания значения переменной используют оператор SET:

```
SET variable = expr [, variable = expr] ...
```

```
SET count = 10;
```

Переменные объявляются в первых строках кода хранимой процедуры, сразу за ключевым словом BEGIN и будут доступными до END, закрывающего тело процедуры.

Пользовательские переменные предназначены для решения локальных задач с применением интерактивного SQL, они доступны исключительно в рамках отдельной сессии. Имена переменных, определяемых пользователем, должны начинаться с символа «@». Отдельного объявления пользовательской переменной не требуется. Для определения переменной и присваивания значения используются ключевые слова SET или SELECT и операторы присваивания «==» или «:=»:

```
SET @start = 1;
SELECT @finish := @start + 10;
```

Основное назначение **системных** переменных заключается в информировании оператора об особенностях конфигурации и работы базы данных и для изменения параметров сервера. Начинаются с двух символов @. Следующий пример выводит версию сервера:

```
SELECT @@version;
```

9.2.2 Условный оператор IF...THEN...ELSE

Для построения конструкции ветвления (если...то...иначе) в MySQL предусмотрен классический условный оператор IF, который имеет следующий синтаксис:

```
IF УСЛОВИЕ1 THEN  
{... выполняется, когда УСЛОВИЕ1 принимает TRUE ...}  
[ ELSEIF УСЛОВИЕ2 THEN  
{... выполняется, когда УСЛОВИЕ2 принимает TRUE ...} ]  
[ ELSE  
{... выполняется, когда УСЛОВИЕ1 и УСЛОВИЕ2 принимаю-  
т FALSE ...} ]  
END IF;
```

Ветви ELSEIF и ELSE – необязательные. Условие ELSEIF используется, если необходимо выполнить набор операторов, когда второе условие (т.е. УСЛОВИЕ2) принимает значение TRUE. Условие ELSE используется, когда необходимо выполнить набор операторов в случае, если ни одно из условий IF или ELSEIF не равно TRUE.

9.2.3 Оператор выбора CASE

Во многих высокоуровневых языках программирования оператор выбора CASE введен для повышения удобства работы. В процедурном SQL решаемая оператором CASE задача схожая – осуществить множественный выбор, но особенностью CASE в SQL является то, что здесь оператор обладает двумя синтаксическими формами. При желании оператор CASE может без заменен конструкцией IF...THEN...ELSE.

Первая форма CASE практически ничем не отличается от аналогичных конструкций в обычных высокоуровневых языках программирования:

```

CASE выражение
WHEN значение_1 THEN
{ выполняется код, когда выражение равно значе-
ние_1}
[ WHEN значение_2 THEN
{ выполняется код, когда выражение равно значе-
ние_2} ]
[ WHEN значение_n THEN
{ выполняется код, когда выражение равно значе-
ние_n} ]
[ ELSE
{ выполняется код, если выражение не равно значениям
} ]
END CASE;

```

Селектор последовательно сравнивает выражение со значениями и в случае совпадения выполняет соответствующий код. Если совпадений не было обнаружено, выполнению подлежит код секции ELSE.

Вторая форма CASE обладает большей гибкостью:

```

CASE
WHEN условие_1 THEN
{ выполняется код, когда условие_1 = TRUE }
[ WHEN условие_2 THEN
{ выполняется код, когда условие_2 = TRUE } ]
[ WHEN условие_n THEN
{ выполняется код, когда условие_n = TRUE } ]
[ ELSE
{ выполняется код, когда все условия = FALSE } ]
END CASE;

```

Здесь вместо простого указания единственного контрольного значения сравнение описывается непосредственно для каждой из

конструкции WHEN. В качестве выражения сравнения может приниматься любой набор операторов, при условии, что он возвратит TRUE или FALSE. Как только выражение сравнения возвращает значение TRUE, выполняется соответствующее выражение.

9.2.4 Циклы

В большинствеialectов SQL допускается применение нескольких типов циклов. Наиболее популярны циклы WHILE (цикл с предусловием) и REPEAT (цикл с постусловием). Несколько реже встречается цикл LOOP – петлевой цикл, для выхода из которого обязательно используется оператор LEAVE.

Синтаксис оператора WHILE в MySQL имеет следующий вид:

```
[метка:] WHILE условие_цикла DO  
    операторы_цикла  
END WHILE [метка]
```

Тело цикла выполняется, пока условие_цикла принимает значение TRUE.

Цикл с постусловием основан на операторе REPEAT. В MySQL синтаксис цикла выглядит следующим образом:

```
[метка:] REPEAT  
    операторы_цикла  
UNTIL условие_цикла  
END REPEAT [метка]
```

Цикл REPEAT имеет две особенности:

- 1) цикл выполняется, пока его условие ложно. Т.е. цикл завершится, когда условие_цикла станет равно TRUE;
- 2) вне зависимости от состояния условия цикл выполнит, по крайней мере, одну итерацию, ведь проверяемое условие находится в последней строке его кода.

Основная особенность петлевого цикла LOOP заключается в том, что в нем (в отличие от циклов WHILE и REPEAT) отсутствует условие выхода. Поэтому необходимо предусмотреть способ прерывания петли, иначе получится бесконечный цикл. Синтаксис цикла выглядит следующим образом:

```
[метка:] LOOP  
операторы_цикла  
END LOOP [метка]
```

Все типы циклов могут расширять свои возможности за счет включения необязательных операторов, которые позволяют осуществлять досрочный выход из цикла (оператор LEAVE или BREAK) и пропуск итерации (ITERATE или CONTINUE).

9.3 Хранимые процедуры

Большую часть примеров создания и использования хранимых процедур, функций и триггеров будем рассматривать на примере СУБД MySQL. В других СУБД используются те же объекты баз данных, но может отличаться синтаксис их определения, способы использования управляющих конструкций и т.д.

Упрощенный синтаксис создания хранимой процедуры в СУБД MySQL имеет следующий вид:

```
CREATE PROCEDURE [IF NOT EXISTS] sp_name  
([proc_parameter [, . . .]])  
[characteristic ...] routine_body  
proc_parameter:  
    [ IN | OUT | INOUT ] param_name type  
type:  
    валидный тип данных MySQL  
characteristic: {  
    COMMENT 'string'  
    | [NOT] DETERMINISTIC
```

```
| { CONTAINS SQL | NO SQL | READS SQL DATA |
MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
}
routine_body:
    валидный оператор SQL
```

Команда начинается с классического оператора языка определения данных CREATE, после которого указывается тип объекта – процедура. При необходимости можно указать ключевые слова IF NOT EXISTS, т.е. процедура будет создана только в том случае, если процедуры с указанным именем в базе данных не существует. Дальше указывает имя процедуры `sp_name` и список параметров. Процедура может не иметь параметров.

Каждый параметр процедуры определяется модификатором IN | OUT | INOUT, именем параметра `param_name` и типом `type`. Модификатор определяет способ передачи параметра. При передаче параметра с модификатором IN процедура может читать и менять значение параметра, но изменение не будет видно вызывающей стороне, когда процедура выполнится. Параметр OUT передает значение из процедуры обратно вызывающей стороне. Его начальное значение равно NULL в процедуре, и его значение видно вызывающей стороне, когда процедура выполнится. Параметр INOUT инициализируется вызывающей стороной, может быть изменен процедурой, и любое изменение, внесенное процедурой, будет видно вызывающей стороне, когда процедура выполнится. По умолчанию используется модификатор IN.

Типом параметра является любой валидный тип данных.

Процедура также может быть описана с набором необязательных характеристик. Характеристика COMMENT – комментарий – может использоваться для описания хранимой процедуры.

Дальше идут модификаторы DETERMINISTIC или NOT DETERMINISTIC. Процедура считается «детерминированной», если

она всегда дает один и тот же результат для одних и тех же входных параметров, и «недетерминированной» в противном случае. Если ни DETERMINISTIC, ни NOT DETERMINISTIC не указаны в определении процедуры, по умолчанию используется значение NOT DETERMINISTIC. СУБД не проверяет, что процедура, объявленная DETERMINISTIC, не содержит операторов, которые дают недетерминированные результаты, и наоборот. Однако неправильное указание типа процедуры может повлиять на результаты или производительность.

Дальше идут несколько характеристик, которые предоставляют информацию о характере использования данных процедурой. В MySQL эти характеристики носят только рекомендательный характер. Сервер не использует их для ограничения того, какие операторы разрешено выполнять в процедуре.

CONTAINS SQL указывает, что процедура не содержит инструкций, которые считывают или записывают данные. Это значение по умолчанию. NO SQL указывает, что процедура не содержит операторов SQL. READS SQL DATA указывает, что процедура содержит инструкции для чтения данных (например, SELECT), но не инструкции для записи данных. MODIFIES SQL DATA указывает, что процедура содержит операторы, которые могут записывать данные (например, INSERT или DELETE).

Характеристика SQL SECURITY может иметь значение DEFINER или INVOKER, чтобы указать контекст безопасности; то есть выполняется ли процедура с правами пользователя, который ее создал, или пользователя, который ее вызывает.

Наконец, тело процедуры `routine_body` состоит из выражений языка SQL. Это может быть простой оператор, такой как SELECT или INSERT, или составной оператор, написанный с ис-

пользованием BEGIN и END. Составные операторы могут содержать объявления, циклы и другие управляющие инструкции.

В следующем примере создается хранимая процедура, которая возвращает максимальную цену товара заданного производителя, передаваемого в процедуру во входной переменной p_manufacturer:

```
CREATE PROCEDURE find_max_price (IN p_manufacturer
                                VARCHAR(45), OUT p_max_price DECIMAL(10, 2))
BEGIN
    SELECT MAX(price) INTO p_max_price
    FROM products
    WHERE manufacturer = p_manufacturer;
END
```

Процедура выполняет запрос и сохраняет результат запроса в выходную переменную p_max_price, объявленную с модификатором OUT.

Для вызова созданной процедуры используется команда CALL. Для вывода значения переменной можно воспользоваться оператором SELECT:

```
CALL find_max_price('Apple', @max_price);
SELECT @max_price;
```

9.4 Функции

9.4.1 Создание функции в MySQL

Синтаксис создания функции в MySQL имеет схожий вид с синтаксисом создания процедуры:

```
CREATE FUNCTION [IF NOT EXISTS] sp_name ([func_parameter [, . . . ]])
```

```

RETURNS type
    [characteristic ...] routine_body
func_parameter:
    param_name type
type:
    валидный тип данных MySQL
characteristic:
    COMMENT 'string'
    | [NOT] DETERMINISTIC
    | { CONTAINS SQL | NO SQL | READS SQL DATA |
MODIFIES SQL DATA }
    | SQL SECURITY { DEFINER | INVOKER }
}
routine_body:
    валидный оператор SQL

```

В отличие от хранимых процедур, у функций нет модификаторов параметров IN, OUT и INOUT. Для функций все параметры всегда рассматриваются как параметры IN. Возврат значения из функции происходит с использованием ключевого слова RETURN, как во многих языках программирования. Выражение RETURNS указывает тип возвращаемого значения функции, а тело функции должно содержать оператор RETURN. Если инструкция RETURN возвращает значение другого типа, оно приводится к нужному типу.

В отличие от хранимых процедур, функции могут быть вызваны внутри SQL выражения.

Рассмотрим пример создания функции, которая возвращает количество заказов выбранного товара:

```

CREATE FUNCTION get_orders_count(p_product_id INT)
    RETURNS INT
    READS SQL DATA
BEGIN
    DECLARE count INT DEFAULT 0;
    SELECT COUNT(*) INTO count
    FROM orders

```

```
    WHERE product_id = p_product_id;
    RETURN count;
END
```

В качестве входных данных функции выступает целочисленная переменная – идентификатор товара, результат функции возвращается с использованием оператора RETURN.

В отличие от хранимой процедуры, функция вызывается в запросе SELECT. В следующем примере результат вызова функции, которому присвоен псевдоним count, выводится в запросе дополнительно к столбцам таблицы products:

```
SELECT *, get_orders_count(product_id) AS count
FROM products
ORDER BY orders_count DESC;
```

Для модификации процедур и функций применяют инструкции ALTER:

```
ALTER PROCEDURE имя_процедуры ...;
ALTER FUNCTION имя_функции ...;
```

Синтаксис инструкции ALTER повторяет синтаксис инструкций CREATE PROCEDURE и CREATE FUNCTION. Это объясняется тем, что на самом деле инструкции ALTER сначала удаляют подлежащую редактированию процедуру (или функцию), а затем вновь создают ее.

Для удаления процедур и функций используется команда DROP:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] имя_процедуры(функции);
```

Перед удалением процедуры следует убедиться, что она нигде больше не используется, в т.ч. в различных клиентских приложениях.

9.4.2 Создание функции в PostgreSQL

Для примера рассмотрим синтаксис создания функции согласно официальной документации PostgreSQL:

```
CREATE [ OR REPLACE ] FUNCTION
    имя ( [ [ режим_аргумента ] [ имя_аргумента ]
    тип_аргумента [ { DEFAULT | = } выражение_по_умол-
    чанию ] [, ...] ] )
        [ RETURNS тип_результата | RETURNS TABLE (
    имя_столбца тип_столбца [, ...] ) ]
        { LANGUAGE имя_языка
        | TRANSFORM { FOR TYPE имя_типа } [, ...]
        | WINDOW
        | { IMMUTABLE | STABLE | VOLATILE }
        | [ NOT ] LEAKPROOF
        | { CALLED ON NULL INPUT | RETURNS NULL ON
    NULL INPUT | STRICT }
        | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL
    ] SECURITY DEFINER }
        | PARALLEL { UNSAFE | RESTRICTED | SAFE }
        | COST стоимость_выполнения
        | ROWS строк_в_результате
        | SUPPORT вспомогательная_функция
        | SET параметр_конфигурации { TO значение | =
    значение | FROM CURRENT }
        | AS 'определение'
        | AS 'объектный_файл', 'объектный_символ'
        | тело_sql
    } ...
```

Команда CREATE FUNCTION определяет новую функцию. CREATE OR REPLACE FUNCTION создаёт новую функцию, либо заменяет определение уже существующей. Далее указывается имя функции и параметры, включая модификатор аргумента IN, OUT, INOUT, имя параметра и его тип.

Далее указывается тип возвращаемых данных (возможно, дополненный схемой).

Затем идут параметры функции, имя языка, на котором реализована функция, модификаторы, как функция работает с базой данных, от имени какого пользователя запускается и другие. В конце указывается тело функции.

В следующем примере создается функция, использующая цикл с предусловием WHILE:

```
CREATE FUNCTION calc_cost(val INT) RETURNS INT
    LANGUAGE plpgsql
AS $$

DECLARE
    res INTEGER;
BEGIN
    res = 0;
    WHILE res <= 3000 LOOP
        res = res + val;
    END LOOP;
    RETURN res;
END $$;
```

9.5 Триггеры

Под триггером понимается особая разновидность хранимой процедуры, вызов которой осуществляется автоматически. Триггеры в основном предназначены для обеспечения целостности и не противоречивости данных:

- триггер гарантированно срабатывает только при наступлении определенного события, обычно связанного с модификацией значений в строке таблицы;

- триггер проверяет условия выполнения операции, вызвавшей его срабатывание;
- если условия верны, то триггер выполняет определенные действия (например, разрешает добавить в таблицу новую строку), а если условия ложны – триггер отвергает операцию.

Существенное отличие триггера от хранимой процедуры состоит в том, что за вызов триггера отвечает только сервер SQL, а клиентское приложение не может управлять работой триггеров.

9.5.1 Создание триггера в MySQL

Для создания триггеров в различных СУБД применяются немного отличающиеся синтаксические конструкции. Запрос создания триггера в MySQL выглядит следующим образом:

```
CREATE TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger
```

В запросе trigger_name – имя триггера, trigger_time определяет время выполнения триггера: до или после наступления события trigger_event. В качестве события могут использоваться следующие: INSERT, UPDATE, DELETE.

Предложение FOR EACH ROW указывает, что триггер срабатывает отдельно для каждой из строк таблицы, попавших под действие команды SQL. Такие триггеры называют триггерами уровня кортежа. Некоторые СУБД поддерживают команду FOR EACH STATEMENT, в этом случае будет создан триггер уровня команды. Этот триггер, вне зависимости от количества строк, попавших под действие инструкции SQL, сработает только один раз.

Возможно создание нескольких триггеров для заданной таблицы с одинаковыми значениями trigger_time и trigger_event. Порядок выполнения таких триггеров определяется датой их создания. Порядок можно изменить, используя выражения **FOLLOW**S и **PRECEDES**. С **FOLLOW**S новый триггер активируется после существующего триггера, с **PRECEDES** – перед существующим триггером.

Стандарт SQL рекомендует разработчикам СУБД обеспечить возможность не только описывать триггер, реагирующий на факт свершившегося события (**AFTER**), но и предоставить возможность вызова триггера перед событием (**BEFORE**). Таким образом, в большинстве СУБД возможно создание трех пар триггеров:

- события BEFORE INSERT и AFTER INSERT возникают соответственно перед тем, как новая строка попадет в таблицу, и после того, как эта строка будет сохранена;
- события BEFORE UPDATE и AFTER UPDATE генерируются перед началом и после завершения редактирования данных;
- событие BEFORE DELETE предшествует, а AFTER DELETE завершает операцию удаления данных из таблицы.

Многие СУБД позволяют к одной и той же таблице подключать несколько триггеров, явно указывая порядок их срабатывания. Если условие срабатывания триггера BEFORE верно, то действие выполняется и управление передается триggerу AFTER, который, в свою

очередь, проверяет свое условие и выполняет свое действие. При невыполнении условия в любом триггере BEFORE осуществляется отмена действия, и данные обрабатываемой строки возвращаются в исходное состояние. Если за вызвавшим исключение триггером BEFORE следует триггер AFTER, то управление ему не передается.

Важно помнить еще одну деталь. Если срабатывание триггера произошло в рамках транзакции, до этого выполнившей какой-то перечень операций с данными, и в результате срабатывания триггер пришел к выводу, что условие ошибочно, то генерация исключительной ситуации приведет к откату всей транзакции.

Для того чтобы разработчик триггера мог получать доступ как к новым данным, поступающим в таблицу с очередной строкой, так и к старым (подлежащим модификации) значениям столбцов, в теле триггера в рядеialectов SQL доступны две контекстные переменные OLD и NEW, которые позволяют получить доступ к строкам таблицы, действие над которыми активировали триггер. В случае INSERT триггера может использоваться только NEW.col_name, так как нет старых строк. В DELETE триггере может использоваться только OLD.col_name, так как нет новых строк. В UPDATE триггере можно использовать OLD.col_name для обращения к столбцам строки перед ее обновлением и NEW.col_name для обращения к столбцам строки после ее обновления.

Рассмотрим пример создания триггера в MySQL. Следующий триггер осуществляет проверку валидности данных перед изменением в таблице orders:

```
CREATE TRIGGER update_check BEFORE UPDATE ON orders
FOR EACH ROW BEGIN
    IF NEW.product_count < 0 THEN
        SET NEW.product_count = 0;
    ELSEIF NEW.product_count > 100 THEN
```

```

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT =
'Превышено количество товаров в заказе';
    END IF;
END;

```

В случае, если указано количество товаров (значение столбца `product_count`) в заказе меньше нуля, в таблицу запишется значение ноль. Если количество товаров превысит 100, будет сгенерировано исключение. Состояние 45000 – это общее состояние, представляющее собой «необработанное определяемое пользователем исключение».

9.5.2 Создание триггера в PostgreSQL

В PostgreSQL синтаксис создания триггера выглядит следующим образом:

```

CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER имя {
BEFORE | AFTER | INSTEAD OF } { событие [ OR ... ]
}

ON имя_таблицы
[ FROM ссылающаяся_таблица ]
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY
IMMEDIATE | INITIALLY DEFERRED ] ]
[ REFERENCING { { OLD | NEW } TABLE [ AS ]
имя_переходного_отношения } [ ... ] ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( условие ) ]
EXECUTE { FUNCTION | PROCEDURE } имя_функции (
аргументы )
событие:
    INSERT
    UPDATE [ OF имя_столбца [, ... ] ]
    DELETE
    TRUNCATE

```

Базовые конструкции создания триггеров схожи: используется оператор CREATE TRIGGER, указывается имя триггера, условие выполнения, имя таблицы, для которой создается триггер.

В качестве примера рассмотрим создание триггера, ведущего протокол изменения данных в таблице products. Сначала создадим функцию, которая будет логировать все изменения по событиям INSERT, UPDATE и DELETE:

```
CREATE OR REPLACE FUNCTION product_logger_function()
RETURNS trigger AS $body$
BEGIN
    if (TG_OP = 'INSERT') then
        INSERT INTO products_log (user_name, rec-
ord_time, old_product_name, new_product_name, opera-
tion)
        VALUES (CURRENT_USER, CURRENT_TIMESTAMP, null,
NEW.product_name, 'INSERT');
        RETURN NEW;
    elsif (TG_OP = 'UPDATE') then
        INSERT INTO products_log (user_name, rec-
ord_time, old_product_name, new_product_name, opera-
tion)
        VALUES (CURRENT_USER, CURRENT_TIMESTAMP,
OLD.product_name, NEW.product_name, 'UPDATE');
        RETURN NEW;
    elsif (TG_OP = 'DELETE') then
        INSERT INTO products_log (user_name, rec-
ord_time, old_product_name, new_product_name, opera-
tion)
        VALUES (CURRENT_USER, CURRENT_TIMESTAMP,
OLD.product_name, null, 'DELETE');
        RETURN OLD;
    end if;
END;
$body$
LANGUAGE plpgsql
```

Далее создадим триггер сразу на 3 события:

```
CREATE TRIGGER product_logger_trigger
AFTER INSERT OR UPDATE OR DELETE
ON products
FOR EACH ROW
EXECUTE FUNCTION product_logger_function()
```

Триггер будет вызываться после каждого события модификации данных и вызывать созданную функцию `product_logger_function`.

9.6 Курсыры

Обычный, построенный на основе инструкции `SELECT` запрос сразу возвращает нам некоторый результирующий набор строк. Однако зачастую логика разрабатываемой базы данных требует дополнительной обработки каждой отдельной строки в результирующем наборе, на которую не способна команда `SELECT`.

В таких случаях используется **курсор**, представляющий собой указатель на отдельный кортеж в отношении. С помощью курсора можно последовательно перебирать строки полученного отношения, осуществляя с ними дополнительные операции, например, редактирования.

Курсыры могут быть полезны в следующих случаях:

- сложная логика запроса (наличие формул, необходимость обращения из запроса к процедурам или функциям, необходимость использования условных операторов и исключений);
- операции по денормализации данных, когда из нескольких отношений следует собрать одно (например, в формате временной таблицы) или возвратить результаты в формате текстовой строки, документа XML и т.п.;

- создание перекрестного запроса, например, выполняющего статистические расчеты по двум и более таблицам, которые позднее группируются в виде одной таблицы;
- движение по иерархическому дереву.

Для работы с курсорами последовательно применяется четыре команды SQL:

- 1) объявить курсор (DECLARE CURSOR);
- 2) открыть курсор (OPEN);
- 3) считать данные из курсора (FETCH);
- 4) закрыть курсор (CLOSE).

Диаграмма на рисунке 47 иллюстрирует работу курсора.

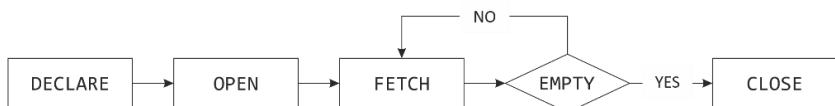


Рисунок 47 – Работа курсора

Рассмотрим синтаксис создания курсора в MySQL. Синтаксис объявления курсора имеет следующий вид:

DECLARE cursor_name CURSOR FOR SELECT_statement;

Для открытия курсора используется выражение OPEN:

OPEN cursor_name;

Оператор OPEN инициализирует результирующий набор для курсора, поэтому он должен быть вызван до начала обхода строк из результирующего набора.

Затем используется оператор FETCH, чтобы получить строку, указанную курсором, и переместить курсор на следующую строку результирующего набора:

FETCH cursor_name INTO variables_list;

Здесь `variables_list` – список переменных, в которые извлекаются данные, описанные в `SELECT_statement`.

После этого выполняется проверка, доступны ли еще строки для извлечения. При работе с курсором необходимо определить, что делать в случае, когда курсор не может найти следующую строку. Для этого следует объявить обработчик события `NOT FOUND` следующим образом:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND statement;
```

Здесь `statement` – выражение, которое будет выполнено при достижении конца результирующего набора, обычно это присваивание значения некоторой переменной, но в общем случае это может быть и комплексное выражение заключенное в `BEGIN...END`. Обработчик событий должен объявляться после объявления курсора.

После завершения работы с курсором используется выражение `CLOSE` для освобождения памяти, используемой курсором, после закрытия курсора работа с ним невозможна:

```
CLOSE cursor_name;
```

10 ТРАНЗАКЦИИ

10.1 Понятие транзакции

Под **транзакцией** будем понимать выполняемую от имени определенного пользователя или процесса последовательность операторов манипулирования данными, переводящая базу данных из одного целостного состояния в другое целостное состояние.

То есть транзакция – это операция, состоящая из одного или нескольких запросов к базе данных. Суть транзакций – обеспечить корректное выполнение всех запросов в рамках одной транзакции, а также обеспечить механизм изоляции транзакций друг от друга для решения проблемы совместного доступа к данным. Любая транзакция либо выполняется полностью, либо полностью отменяется.

Одним из наиболее распространённых наборов требований к транзакциям и транзакционным системам, в т.ч. СУБД, является набор ACID. Данные требования позволяют обеспечить наиболее надёжную и согласованную работу системы. Эти требования включают в себя:

- атомарность (**atomicity**);
- согласованность (**consistency**);
- изолированность (**isolation**);
- долговечность (**durability**).

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Вне зависимости от количества вовлекаемых в транзакцию операторов, транзакция должна представлять собой единую и неделимую логическую единицу работы.

Транзакция выполняется как атомарная операция – либо выполняется вся транзакция целиком, либо она целиком не выполня-

ется. Транзакция не может быть выполнена частично, если в процессе выполнения транзакции возникает какой-то сбой, то все выполненные до этого момента изменения данных должны быть отменены. Транзакция считается успешно выполненной, если в процессе исполнения не возникла какая-либо аварийная ситуация и все проверки ограничений целостности дали положительный результат.

Следующее требование – **согласованность**. Транзакция переводит базу данных из одного согласованного (целостного) состояния в другое согласованное (целостное) состояние. Однако в процессе выполнения транзакции допускается появление несогласованных промежуточных результатов. Согласованность достигается вследствие атомарности, т.к. завершенная транзакция гарантирует согласованность данных.

Говоря о согласованности результатов, имеется в виду, что в результате выполнения транзакции данные будут соответствовать заложенной в базе данных логике и правилам поддержания целостности данных.

Третье требование – **изолированность**. Транзакции должны выполняться независимо одна от другой. Транзакции должны быть недоступны промежуточные результаты других параллельно исполняемых транзакций, а также её промежуточные результаты недоступны другим транзакциям.

Другими словами, изолированная транзакция A может получить доступ к объекту, обрабатываемому транзакцией B , только после завершения B , и наоборот. Таким образом при одновременной работе двух пользователей с одними и теми же данными они не замечают друг друга, как если бы они работали с этими данными строго по очереди. Изолированность – требование дорогое, поэтому в реальных БД существуют режимы, не полностью изолирующие транзакцию – так называемые уровни изоляции.

Четвертое требование – **долговечность**. Если транзакция выполнена, то результаты ее работы должны сохраниться в базе данных и не могут быть утеряны или отменены ни при каких обстоятельствах. В случае успешного выполнения транзакции все изменения гарантировано фиксируются в постоянной памяти даже если в следующий момент произойдет сбой системы. В случае неуспешного завершения транзакции по любой причине гарантировано отсутствие каких-либо связанных с ней изменений во внешней памяти.

То есть независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

Все эти требования призваны обеспечить создание системы, работа которой будет понятная и предсказуема. Все это регламентируется правилами работы с транзакциями.

10.2 Механизмы обеспечения требований к транзакциям

10.2.1 Журнал транзакций

Существуют несколько механизмов, обеспечивающих выполнение ACID требований, начнем с требований атомарности и долговечности.

Для обеспечения требований атомарности и долговечности транзакций используется простой принцип: каждая операция не производится с данными напрямую. Используется принцип двойной записи: сначала данные пишутся в **журнал транзакций**, только потом меняются сами табличные данные. Такой подход позволяет

в любой момент времени вернуть базу данных в исходное состояние (состояние до старта транзакции). Откат транзакции осуществляется в том случае, если хотя бы один из операторов, входящих в состав транзакции, выполняется с ошибкой. Во время отката в обратном порядке отменяются все операции транзакции. Если же все операторы выполняются верно, то результаты транзакции фиксируются – данные из журнала переносятся в таблицы базы данных.

Соответственно **журнализация изменений** (write ahead logging, WAL) – функция СУБД, которая сохраняет информацию, необходимую для восстановления базы данных в предыдущее согласованное состояние в случае логических или физических отказов.

Для того чтобы СУБД смогла гарантировать корректное завершение или успешный откат транзакции, в журнале транзакций приходится хранить существенный объем данных, в т.ч.:

- сведения о старте транзакции и ее идентификатор;
- перечень объектов базы данных, на которые повлияла транзакция;
- описание каждой из входящих в состав транзакции операций обновления данных (вставка, обновление, удаление);
- предыдущее состояние объекта и новое состояние объекта;
- сведения о завершении транзакции.

Формируемая таким образом информация называется журналом изменений базы данных. Журнал содержит отметки начала и завершения транзакции и отметки принятия контрольной точки, когда данные сбрасывались на диск.

Рассмотрим, зачем требуется использовать журнал транзакций.

Основная проблема при обеспечении сохранности данных при записи на диск возникает из-за того, что жесткий диск не имеет атомарной функции записи. Т.е. если будет записываться мегабайт данных и в момент записи будет отключено питание – результат записи

не определён: все данные будут записаны, ничего не будет записано или запишется только какая-то часть.

Вторая проблема – устройства хранения очень медленные по сравнению с оперативной памятью.

Чтобы корректно обрабатывать эти ситуации используется журнал транзакций. Общий алгоритм использования журнала транзакций имеет следующий вид:

- 1) когда начинается транзакция – ей присваивается номер. При изменении данных новые данные не пишутся в таблицы на жесткий диск, а все изменения (состояние до и после) записываются в журнал транзакции, и изменяются страницы базы данных в памяти;
- 2) в журнал транзакций сбрасывается маркер успешного завершения транзакции;
- 3) когда приходит команда подтверждения транзакции – весь журнал транзакции сбрасывается на диск;
- 4) как только приходит подтверждение от операционной системы, что журнал транзакции успешно сброшен – выполняются изменения в файлах, которые соответствуют данным таблиц, и уже они сбрасываются на диск.

После успешного сброса на диск файлов таблиц, данные журнала для этой транзакции больше не нужны, данные таблицы на диске.

Рассмотрим, что происходит в случае сбоя. Если сбой произошел до того, как транзакция успешно завершилась – т.е. изменения записывались в журнал транзакции, произошел сбой, данные не были до конца записаны на диск, в журнале транзакции находятся не валидные данные – при восстановлении базы данных эти данные будут проигнорированы. Если данные были сброшены в журнал транзакций, но не были изменены файлы с данными таблиц, то при

восстановлении базы данных будет сделана проверка журнала транзакций, найдена корректная запись, которая не была записана в файл с данными, и эта транзакция применится к файлу данных – транзакция пройдет успешно. Т.е. принцип двойной записи позволяет корректно обрабатывать ситуации со сбоями.

Сброс данных на жесткий диск – это относительно долгая задача, поэтому СУБД сообщает, что транзакция завершилась успешно, когда на диск успешно записаны файлы журнала транзакций. Данные таблиц меняются в памяти, но не сбрасываются на диск сразу после завершения транзакции. Сброс данных на диск может происходить после выполнения нескольких транзакций. Этот момент помечается контрольной точкой.

Т.к. в журнал транзакций пишутся все данные, которые необходимы для восстановления системы после сбоя, то журнал транзакций может использоваться и для дополнительных целей:

- восстановление на момент времени (point in time recovery) – использование резервной копии и журнала транзакций для восстановления системы после сбоя;
- репликация – механизм синхронизации содержимого нескольких копий данных (в т.ч., содержимого базы данных) для повышения надежности и производительности системы. В этом случае журнал транзакции транслируется по сети от одного сервера к другому для синхронизации данных.

Таким образом, журнал транзакций, помимо того, что он обеспечивает сохранения свойств атомарности и долговечности, также позволяет реализовать механизмы репликации и восстановления базы данных на момент сбоя или на момент выбранной транзакции.

10.2.2 Управление параллельным доступом посредством многоверсионности

Перейдем к следующему требованию – изолированности данных. Рассмотрим ситуацию, когда несколько пользователей работают с одной и той же базой данных. Необходимо установить правила, как действия пользователей будут влиять друг на друга.

Есть два основных подхода для обеспечения изолированности:

- 1) использование блокировок;
- 2) использование механизма управления параллельным доступом посредством многоверсионности (MultiVersion Concurrency Control, MVCC).

При использовании блокировок все данные, которые задействованы приложением в рамках транзакции, блокируются. При начале чтения или изменения данных на них накладывается блокировка. Другое приложение, обращающееся к тем же данным, видит блокировку и ждет окончания операций. Проблема такого подхода состоит в том, что он плохо масштабируется, т.к. все пользователи находятся в состоянии ожидания, пока один клиент работает с базой данных.

Управление параллельным доступом посредством многоверсионности – один из механизмов СУБД для обеспечения параллельного доступа к базам данных, заключающийся в предоставлении каждому пользователю так называемого «снимка» или среза базы, обладающего тем свойством, что вносимые пользователем изменения невидимы другим пользователям до момента фиксации транзакции. Этот способ управления позволяет добиться того, что пишущие транзакции не блокируют читающих, и читающие транзакции не блокируют пишущих. Такой подход имеет следующие преимущества:

- разные пользователи могут одновременно работать с одними и теми же данными;

- каждый пользователь видит свой изолированный срез данных;
- изменения, вносимые пользователем, никому не видны до завершения транзакции.

Механизм MVCC впервые появился в PostgreSQL, затем его поддержка появилась и в других СУБД, в т.ч. в MySQL, в движке InnoDB.

10.3 Проблемы совместного доступа к данным

При параллельном выполнении транзакций возможны следующие проблемы:

- **потерянное обновление** (lost update) – при одновременном изменении одного блока данных разными транзакциями теряются все изменения, кроме последнего. Т.е. в этом случае одна транзакция переписывает изменения, осуществленные другой транзакцией, в результате одно из изменений будет утеряно;
- **«грязное» чтение** (dirty read) – чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтверждается (откатится). В этом случае незафиксированные изменения, осуществленные одной транзакцией, читаются (или обновляются) другой. В случае перезаписи этих промежуточных значений или отката первой транзакции незафиксированные изменения могут быть отменены, а прочитавшая их транзакция с этого момента станет работать с неверными данными;
- **неповторяющееся чтение** (non-repeatable read) – при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными. Возникает тогда, когда транзакция считывает из базы значение, после чего

вторая транзакция обновляет это значение. Если в этот момент времени первая транзакция продолжает выполняться, то имеющиеся в ее распоряжении данные становятся неактуальными;

- **phantom reads** – одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

10.3.1 Уровни изоляции транзакций

Чтобы устранить проблемы, возникающие при параллельном выполнении транзакций, необходимо добиться, чтобы изменения, сделанные одной транзакцией, были невидимы для других транзакций до тех пор, пока текущая транзакция не будет зафиксирована или отменена. Т.е. необходимо обеспечить изоляцию транзакций.

Под «уровнем изоляции транзакций» понимается степень обеспечиваемой внутренними механизмами СУБД защиты от всех или некоторых видов несогласованности данных, возникающих при параллельном выполнении транзакций.

Стандарт SQL-92 определяет шкалу из четырёх уровней изоляции.

1. **READ UNCOMMITTED** (чтение незафиксированных или «грязных» данных) – наименее защищенный уровень изоляции, при котором транзакции способны читать незафиксированные изменения, сделанные другими транзакциями. При этом возможно считывание не только логиче-

- ски несогласованных данных, но и данных, изменения которых ещё не зафиксированы;
2. **READ COMMITTED** (чтение фиксированных данных) – исключается «грязное» чтение, транзакция увидит только изменения, зафиксированные другими транзакциями. Тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных. Большинство промышленных СУБД, в частности, Microsoft SQL Server, PostgreSQL и Oracle по умолчанию используют именно этот уровень изоляции;
 3. **REPEATABLE READ** (повторяющееся чтение) – накладывает блокировки на обрабатываемые транзакцией строки и не допускает их изменения другими транзакциями. В результате транзакция видит только те строки, которые были зафиксированы на момент ее запуска. Однако другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведёт к фантомному чтению. Это уровень изоляции по умолчанию в MySQL;
 4. **SERIALIZABLE** (сериализуемость) – самый надежный уровень изоляции, полностью исключающий взаимное влияние транзакций. Только на этом уровне параллельные транзакции не подвержены эффекту «фантомного чтения».

При изменении данных в рамках транзакции на них требуется накладывать блокировку, чтобы не перезаписать изменения. В некоторых ситуациях для обеспечения изолированности требуется

блокировать еще и чтение. Но поддержание блокировок требует вычислительных ресурсов.

Разные уровни изоляции транзакций выдают разные правила того, какие действия из соседних транзакций будут видны. Более строгие уровни изоляции допускают меньшее влияние транзакций друг на друга. На разных уровнях изоляции транзакции допускаются различные конфликты в данных: грязное чтение, неповторяющееся чтение, фантомное чтение и аномалии сериализации. Потерянное обновление не допускается ни на одном уровне изоляции.

Уровни изоляции транзакций определяют, какие проблемы совместного доступа к данным допустимы (таблица 10.1).

Таблица 10.1. Уровни изоляции транзакций

Уровень изоляции	Потерянное обновление	Грязное чтение	Неповторяющееся чтение	Фантомное чтение	Аномалии сериализации
Read uncommitted	Исключено	Возможно	Возможно	Возможно	Возможно
Read committed	Исключено	Исключено	Возможно	Возможно	Возможно
Repeatable read	Исключено	Исключено	Исключено	Возможно	Возможно
Serializable	Исключено	Исключено	Исключено	Исключено	Исключено

Т.е. на уровне изоляции READ UNCOMMITTED допустимо грязное чтение, неповторяющееся чтение, фантомное чтение и аномалии сериализации. Потерянное обновление исключено.

Стандарт определяет минимальные требования, но СУБД может убрать какие-то проблемы совместного доступа, главное обеспечить необходимый минимум.

В PostgreSQL на уровне READ UNCOMMITTED невозможно грязное чтение, а на уровне REPEATABLE READ невозможно фантомное чтение, которые допускаются стандартом.

10.3.2 Потерянное обновление

Рассмотрим проблемы совместного доступа к данным на примерах.

При потерянном обновлении одна транзакция переписывает изменения, осуществленные другой транзакцией, в результате одно из изменений будет утеряно.

Допустим, в системе выполняются две транзакции, и до завершения первой транзакции во второй транзакции были изменены те же данные, что были изменены в первой транзакции (таблица 10.2).

Таблица 10.2. Потерянное обновление

Транзакция 1	Транзакция 2
<pre>UPDATE accounts SET balance = balance + 100 WHERE name = 'Alice';</pre>	
	<pre>UPDATE accounts SET balance = balance + 500 WHERE name = 'Alice';</pre>

Ситуацию с потерянным обновлением не допускает ни один уровень изоляции транзакции. Проблема решается применением блокировок на запись.

10.3.3 Грязное чтение

Грязное чтение – это ситуации, при которой доступно чтение незафиксированных данных, добавленных или изменённых еще не завершённой транзакцией.

В этом случае можно получить результат промежуточной работы транзакции, при этом будут доступны данные, которые в контексте целостности базы данных не должны быть доступны, т.к. по-

сле завершения транзакции они гарантированно изменятся. Во-первых, эта транзакция может откатиться, во-вторых, эти данные могут еще раз измениться – т.е. для чтения будут доступны так называемые грязные данные. Соответственно, возникает проблема согласованности данных.

Пример получения грязных данных показан в таблице 10.3.

Таблица 10.3. Грязное чтение

Транзакция 1	Транзакция 2
<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 100</pre>	
<pre>UPDATE accounts SET balance = balance + 100 WHERE name = 'Alice';</pre>	
	<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 200;</pre>
<pre>ROLLBACK;</pre>	

В примере первая транзакция обновила данные, а вторая транзакция прочитала обновленные, но не зафиксированные данные. После отката первой транзакции вторая транзакция будет работать с невалидными (грязными) данными.

10.3.4 Неповторяющееся чтение

При неповторяющемся чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Проблема возникает, когда в процессе транзакции доступны данные, которых не было на начало транзакции. Транзакция, которая внесла изменения в эти данные, уже завершилась, но в момент начала исходной транзакции этих данных еще не было.

Пример данной ситуации показан в таблице 10.4.

Таблица 10.4. Неповторяющееся чтение

Транзакция 1	Транзакция 2
	<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 100</pre>
<pre>UPDATE accounts SET balance = balance + 100 WHERE name = 'Alice';</pre>	
<pre>COMMIT;</pre>	
	<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 200;</pre>

В примере показано, что вторая транзакция получила доступ к данным, которых не было на ее начало, т.е. нарушилась изолированность транзакции.

10.3.5 Фантомное чтение

При фантомном чтении возникает ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк. От неповторяющегося чтения эта про-

блема отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (phantomных) данных.

Пример фантомного чтения показан в таблице 10.5.

Таблица 10.5. Фантомное чтение

Транзакция 1	Транзакция 2
	SELECT SUM(balance) FROM accounts; ----- 500
INSERT INTO accounts (name, balance) VALUES ('Dave', 1000);	
COMMIT;	
	SELECT SUM(balance) FROM accounts; ----- 1500

В примере вторая транзакция считает некоторую сумму, после чего первая транзакция добавляет новые данные, после чего вторая транзакция при подсчете суммы получит новое значение.

В чем разница между фантомным чтением и неповторяющимся чтением? В неповторяющемся чтении данные были изменены или удалены, при фантомном чтении – добавлены. Эти проблемы разделяют, поскольку для их решения необходимы разные подходы для обеспечения изолированности в механизме MVCC. При неповторяющемся чтении известно, какие записи были прочитаны. Если эти записи другая транзакция пытается модифицировать – можно доба-

вить блокировок на сами записи. Для фантомных записей блокировку добавить нельзя, т.к. не известно, какие записи можно прочитать в будущем при таком запросе. В этом случае обычно запоминают последнюю добавленную запись на начало транзакции, чтобы отсечь вновь добавленные.

10.3.6 Аномалии сериализации

Последняя проблема – аномалии сериализации, которые не допускаются на наиболее жестком уровне изоляции транзакций – уровне SERIALIZABLE.

Аномалия сериализации – это ситуация, когда две транзакции по какой-то причине при параллельном выполнении выдают результат, отличный от последовательного выполнения этих транзакций.

Пример подобного выполнения транзакций показан в таблице 10.6.

Таблица 10.6. Аномалии сериализации

Транзакция 1	Транзакция 2
<pre>SELECT SUM(balance) FROM accounts WHERE name = 'Bob'; ----- 1000</pre>	<pre>SELECT SUM(balance) FROM accounts WHERE name = 'Alice'; ----- 500</pre>
<pre>INSERT INTO accounts (name, balance) VALUES ('Alice', 1000);</pre>	<pre>INSERT INTO accounts (name, balance) VALUES ('Bob', 500);</pre>
<pre>COMMIT;</pre>	<pre>COMMIT;</pre>

В случае уровня транзакции REPEATABLE READ после завершения транзакций для пользователей с именем Bob сумма будет

1500, для пользователей с именем Alice – тоже 1500. При последовательном выполнении этих двух транзакций получить такой результат невозможно.

Уровень SERIALIZABLE такого выполнения не допускает. В случае выполнения этих транзакций на уровне SERIALIZABLE одна транзакция будет отменена.

Может возникнуть вопрос, зачем вообще нужны уровни изоляции, которые допускают проблемы совместного доступа к данным, если можно всегда использовать уровень изоляции SERIALIZABLE.

Причина в том, что этот уровень наиболее ресурсоемкий, в этом случае практически полностью останавливается параллельная обработка данных, заставляя остальные транзакции простаивать в очереди. Чем более строгие требования предъявляются к обеспечению изолированности транзакций, тем больше шансов, что какие-то транзакции между собой будут конфликтовать и транзакция будет отменена.

Кроме того, далеко не всегда нужны такие гарантии целостности. Часть таких проблем могут решаться, например, на уровне приложения.

В целом, для уменьшения вероятности конфликтов между транзакциями, транзакции должны быть как можно короче, длинных транзакций стоит избегать.

10.4 Явное управление транзакцией

По умолчанию сервер баз данных берет на себя полную ответственность за управление транзакций. Как только пользователь отправляет СУБД инструкцию SQL, сервер самостоятельно начинает транзакцию, осуществляет фиксацию изменений, если в ходе выполнения транзакции не произошло исключительных ситуаций, или осуществляет автоматический откат операций, осуществленных в

транзакции при наличии в них хотя бы одной ошибки. Такое поведение сервера называют **неявным управлением транзакциями**.

Вместе с тем в SQL предусматривается инструментальный набор по **явному управлению транзакциями**. Он складывается из трех инструкций:

- 1) START TRANSACTION (или BEGIN) – используется для запуска новой транзакции;
- 2) COMMIT – фиксация изменений, осуществленных транзакцией;
- 3) ROLLBACK – откат транзакции в исходное состояние.

10.4.1 Управление транзакцией в MySQL

В MySQL для управления транзакцией используется следующие выражения:

```
START TRANSACTION  
COMMIT [AND [NO] CHAIN] [[NO] RELEASE]  
ROLLBACK [AND [NO] CHAIN] [[NO] RELEASE]  
SET autocommit = {0 | 1}
```

По умолчанию, MySQL автоматически фиксирует выполнение транзакции. Чтобы явно отключить режим автоматической фиксации, необходимо использовать следующую инструкцию SET autocommit = 0. После отключения автоматической фиксации изменений необходимо явно вызывать оператор COMMIT для подтверждения транзакции или ROLLBACK для ее отката. Также отключить режим автоматического завершения транзакций для отдельной последовательности операторов можно оператором START TRANSACTION.

Выражение AND CHAIN указывает системе, что с очередной транзакцией следует работать так, как будто она является следующим звеном предыдущей. В результате вторая транзакция

станет использовать общие настройки первой транзакции (например, уровень изоляции).

Ключевое слово RELEASE требует, чтобы после завершения транзакции клиент завершил текущую сессию и отключился от сервера. По умолчанию действует режим NO RELEASE – сессия не прерывается.

Для некоторых операторов нельзя выполнить откат с помощью оператора ROLLBACK. В MySQL это операторы языка определения данных (Data Definition Language). Сюда входят запросы CREATE, ALTER, DROP, TRUNCATE, COMMENT, RENAME.

Соответственно, оператор ALTER TABLE, например, неявно завершает транзакцию, как если бы перед выполнением этого оператора был вызван COMMIT.

PostgreSQL, в отличие от MySQL, поддерживает выполнение операторов языка DDL в рамках транзакции. Т.е. в транзакции можно создать таблицу, изменить ее, а потом, например, откатить изменения.

Транзакция может быть разделена на точки сохранения. Для управления точками сохранения используются следующие команды:

```
SAVEPOINT identifier  
ROLLBACK TO [SAVEPOINT] identifier  
RELEASE SAVEPOINT identifier
```

Оператор **SAVEPOINT identifier_name** устанавливает именованную точку сохранения транзакции с именем идентификатора. Если текущая транзакция имеет точку сохранения с тем же именем, старая точка сохранения удаляется, а новая устанавливается.

Оператор ROLLBACK TO SAVEPOINT *identifier_name* откатывает транзакцию до указанной точки сохранения без прерывания транзакции. Изменения, которые выполняются в текущей транзакции для строк после установки точки сохранения, отменяются при откате.

Для удаления одной или нескольких точек сохранения используется команда RELEASE SAVEPOINT *identifier_name*.

Для явного изменения уровня изоляции транзакции можно воспользоваться оператором

```
SET TRANSACTION ISOLATION LEVEL
```

и выбрать один из рассмотренных уровней изоляции:

```
{READ UNCOMMITTED | READ COMMITTED | REPEATABLE  
READ | SERIALIZABLE }
```

10.4.2 Пример управления транзакцией в MySQL

Рассмотрим пример с проблемой повторяющегося чтения, в котором при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными. Частичные запросы примера были показаны в таблице 10.4, в таблице 10.7 показан полный синтаксис запросов в MySQL.

В примере создаются два подключения, в каждом подключении явно устанавливается уровень изоляции READ COMMITTED, который допускает получение зафиксированных в рамках другой транзакции записей. Далее выполняются запросы выбора и изменения данных. При повторном выборе данных вторая транзакция увидит изменения, зафиксированные первой транзакцией.

Таблица 10.7. Пример управления транзакциями в MySQL

<pre>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; START TRANSACTION;</pre>	<pre>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; START TRANSACTION;</pre>
	<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 100</pre>
<pre>UPDATE accounts SET balance = balance + 100 WHERE name = 'Alice'; ----- UPDATE 1</pre>	
<pre>COMMIT;</pre>	
	<pre>SELECT balance FROM accounts WHERE name = 'Alice'; ----- 200;</pre>
	<pre>COMMIT;</pre>

11 РАСШИРЕННЫЕ ТИПЫ ДАННЫХ

11.1 Пространственные данные

11.1.1 Понятие пространственных данных

Пространственные данные представляют сведения о физическом расположении и форме геометрических объектов. Этими объектами могут быть объекты простых типов, такие как точки, линии, полигоны, а также более сложные объекты, представляемые коллекциями объектов.

Пространственные данные составляют основу информационного обеспечения геоинформационных систем.

Совокупность пространственных данных, записанных (сохранённых) тем или иным образом, называется **пространственной базой данных** (spatial database). Современные пространственные базы данных организуются на платформе специализированного программного обеспечения, позволяющего сохранять, накапливать и обрабатывать (включая пространственный анализ) все компоненты пространственных данных в виде логически единой базы данных.

Большинство современных СУБД поддерживают так называемые пространственные расширения – геометрические типы данных и пространственные индексы.

Пространственные расширения СУБД позволяют создавать, хранить и анализировать следующие объекты:

- типы данных для представления пространственных значений;
- функции, выполняющие операции над пространственными данными;
- пространственные индексы для ускорения времени выполнения пространственных операций.

Большинство СУБД разрабатывают пространственные расширения, основываясь на спецификации Open Geospatial Consortium (OGC). OGC – международная некоммерческая организация, ведущая деятельность по разработке стандартов в сфере геопространственных данных и сервисов.

Базовый набор стандартов OGC содержит более 30 стандартов, в том числе:

- Simple Features Specification For SQL – определение схемы SQL, которая поддерживает хранение, поиск, запрос и обновление простых коллекций геопространственных объектов;
- GML – XML-формат для географической информации;
- SRID – Spatial Reference System Identifier – идентификатор тождественности пространственных систем координат.

Пространственные данные в MySQL

MySQL поддерживает следующие типы данных, которые соответствуют классам OpenGIS:

- GEOMETRY – (абстрактный класс) пространственные значения любых типов;
- POINT – точка – представляет собой объект без измерения, представляющий отдельное местоположение, и кроме координат может содержать значения Z (уровень) и M (мера);
- LINESTRING – линия – одномерный объект, представляющий последовательность точек и соединяющих их линейных сегментов;
- POLYGON – полигон – двухмерная поверхность, хранимая в виде последовательности точек, определяющих внешнее ограничивающее кольцо и внутренние кольца (последние могут отсутствовать).

Кроме того, поддерживаются коллекции объектов:

- MULTIPOINT;
- MULTILINESTRING;
- MULTIPOLYGON;
- GEOMETRYCOLLECTION.

Типы коллекций (MULTIPOINT, MULTILINESTRING и MULTIPOLYGON) представляют множества объектов соответствующих типов (POINT, LINESTRING и POLYGON) а GEOMETRYCOLLECTION позволяет хранить множество объектов любого типа. Примеры объектов различных типов пространственных данных приведены на рисунке 48.

POINT		MULTIPOINT	
LINESTRING		MULTILINESTRING	
POLYGON		MULTIPOLYGON	

Рисунок 48 – Типы пространственных данных

Расширение PostGIS для PostgreSQL

В PostgreSQL по умолчанию нет поддержки пространственных данных, для их использования необходимо установить расширение PostGIS.

PostGIS – программное расширение с открытым исходным кодом, расширяющее пространственную базу данных системы управления PostgreSQL.

Использованием расширения PostGIS в PostgreSQL позволяет применять:

- пространственные индексы для быстрых пространственных запросов;
- пространственные операторы для проведения геопространственных вычислений и определения геопространственного набора операций (объединение, разность, симметричная разность);
- функции создания и управления пространственными данными;
- инструменты для определения пространственных отношений и измерений;
- функции создания и обработки растровых данных и т.д.

11.1.2 Пространственный анализ

Функции работы с пространственными данными можно разделить на несколько основных категорий в зависимости от типа выполняемой ими операции:

- функции, создающие геометрию пространственных объектов в различных форматах (WKT, WKB, внутренний);
- функции, преобразующие геометрию пространственных объектов между форматами;
- функции, которые обращаются к качественным или количественным свойствам геометрии пространственных объектов;
- функции, описывающие отношения между двумя пространственными объектами;
- функции, которые создают новые пространственные объекты из существующих.

Для создания пространственных объектов из текстового описания используется формат WKT (Well-Known Text) – текстовый формат представления векторной геометрии и описания систем коорди-

нат. Для хранения этой же информации в базах данных используется двоичный эквивалентный формат – WKB (Well-Known Binary).

Примеры геометрии в WKT формате показаны в таблице 11.1.

Таблица 11.1. Примеры геометрии в WKT-формате

Тип геометрии	Пример объекта
Point	POINT(15 20)
LineString	LINESTRING(0 0, 10 10, 20 25, 50 60)
Polygon	POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (5 5, 7 5, 7 7, 5 7, 5 5))
MultiPoint	MULTIPOINT(0 0, 20 20, 60 60)
MultiLin- eString	MULTILINESTRING((10 10, 20 20), (15 15, 30 15))
MultiPolygon	MULTIPOLYGON(((0 0, 10 0, 10 10, 0 10, 0 0), ((5 5, 7 5, 7 7, 5 7, 5 5))))
GeometryCol- lection	GEOMETRYCOLLECTION(POINT(10 10), POINT(30 30), LINESTRING(15 15, 20 20))

Функции создания пространственных объектов

Чтобы создать геометрию пространственного объекта по его WKT-описанию, могут использоваться следующие функции:

- ST_GeomFromText(wkt, [, srid [, options]]);
- ST_PointFromText(wkt, [, srid [, options]]);
- ST_LineFromText(wkt, [, srid [, options]]);
- ST_PolygonFromText(wkt, [, srid [, options]]);
- ST_GeomCollFromText(wkt, [, srid [, options]]).

Функция `ST_GeomFromText` выполняет построение геометрического значения произвольного типа с использованием его представления WKT. Остальные функции создают геометрии конкретного типа, т.е. нельзя вызвать функцию `ST_PointFromText`, например, передав в качестве WKT описания геометрию полигона. Такой вызов завершится с ошибкой. Также существуют похожие функции создания мультиточки, мультилинии и мультиполигона.

В PostGIS существуют функции с теми же названиями, в Microsoft SQL Server названия функций немного отличаются, но общие принципы создания пространственных данных остаются теми же.

Обычно помимо WKT описания геометрии функции принимают еще один необязательный параметр SRID. Параметр SRID – это идентификатор пространственной системы координат (Spatial Reference System Identifier) – этот параметр определяет, что координаты объекта заданы в конкретной системе координат. Разделяют два вида систем координат:

- сферические (географические), где координаты точки задаются с помощью широты и долготы;
- плоские (декартовы), где координаты точек задаются как расстояния до осей на некоторой проекции земной поверхности.

Так как в основе различных систем координат лежат различные предположения о форме Земли, а также используются разнообразные допущения и упрощения, то фактически сравнивать координаты двух точек, заданные в различных системах координат, некорректно. Этот же принцип реализуется и при работе с пространственными данными – сравнимыми будут только пространственные объекты, заданные в одной системе координат. По умолчанию, если параметр SRID не задан, он трактуется равным 0, что соответствует случаю обычной прямоугольной системы координат. СУБД MySQL поставляется с более чем 5000 предопределенных систем координат.

В следующем примере создаются несколько пространственных объектов:

```
SELECT ST_LineFromText('LINESTRING(0 0, 10 5, 20  
25, 30 15)';  
SELECT ST_GeomFromText('POLYGON((0 0, 10 0, 10 10,  
0 10, 0 0), 5 5, 7 5, 7 7, 5 7, 5 5))';  
SELECT ST_GeomCollFromText('GEOMETRYCOLLECTION(  
POLYGON((0 0, 5 0, 5 5, 0 5, 0 0)), POINT(-5 5),  
POINT(8 8), LINESTRING(-5 10, 10 0))');
```

Первый запрос вернет линейный объект, созданный с использованием функции `ST_LineFromText`. Во втором запросе создается полигон с использованием общей функции создания геометрических объектов `ST_GeomFromText`. В данном случае полигон имеет внешнее кольцо и внутреннее кольцо. В третьем примере создается один объект, содержащий коллекцию геометрических объектов: полигон, две точки и линию.

Результаты этих запросов показаны на рисунке 49.

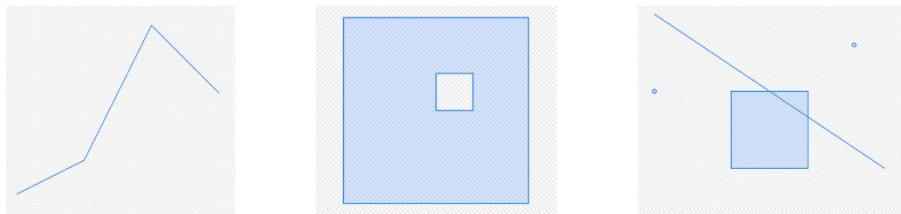


Рисунок 49 – Примеры пространственных объектов

Созданные объекты могут быть добавлены в таблицы, использоваться в функциях пространственного анализа и т.д., а не только возвращаться в предложении `SELECT`.

Для создания объектов можно использовать не только WKT описание. Некоторые СУБД поддерживают форматы GEOJSON и GML:

- `ST_GeomFromGeoJSON(str [, options [, srid]])`;
- `ST_GeomFromGML(text geomgml, integer srid)`.

Пространственные объекты, хранящиеся в СУБД, можно преобразовать в текстовое описание различного формата, в зависимости от поддерживаемых СУБД возможностей. Функции получения описания пространственного объекта в текстовом виде:

- ST_AsText(geometry) : в формате WKT;
- ST_AsGeoJSON(geometry) : в формате GeoJSON;
- ST_AsGML(geometry) : в формате GML.

MySQL позволяет вывести геометрию в формате WKT или GeoJSON, PostgreSQL также поддерживает формат GML.

Операции над пространственными объектами

В различных СУБД реализовано большое число функций для работы с пространственными данными и их анализа. Основные функции в MySQL представлены в таблице 11.2.

Таблица 11.2. Операции над пространственными объектами

Функция	Описание
ST_SRID(geom)	Возвращает идентификатор системы координат пространственного объекта geom.
ST_X(point), ST_Y(point)	Возвращает координаты x и y точки point.
ST_NumPoints(ls)	Возвращает число точек линии ls.
ST_PointN(ls, N)	Возвращает N-ю точку линии ls.
ST_Length(ls)	Возвращает длину линии ls.
ST_Centroid(poly mpoly)	Возвращает математический центроид для полигона или мультиполигона.
ST_IsValid(geom)	Принимает пространственный объект geom, возвращает 1 если геометрия объекта корректна, 0 в противном случае. Корректность геометрии проверяется в соответствии со спецификацией OGC.

Окончание табл. 11.2

Функция	Описание
ST_Validate(geom)	Принимает пространственный объект geom, возвращает объект, если он является корректным пространственным объектом, и NULL в противном случае.
ST_Simplify(geom, max_distance)	Принимает пространственный объект geom и параметр max_distance, возвращает упрощенную геометрию по алгоритму Дугласа-Пекера.
ST_Envelope(geom)	Возвращает минимальный ограничивающий прямоугольник для пространственного объекта geom.
ST_Distance(g1, g2)	Возвращает расстояние между объектами g1 и g2.
ST_Buffer(geom, dist)	Принимает пространственный объект geom и расстояние dist, возвращает объект геометрии, который представляет буфер вокруг объекта.
ST_Intersection(g1, g2)	Возвращает результат пересечения геометрии пространственных объектов g1 и g2.
ST_Union(g1, g2)	Возвращает результат объединения геометрии пространственных объектов g1 и g2.
ST_Difference(g1, g2)	Возвращает результат разности геометрии пространственных объектов g1 и g2.
ST_SymDifference(g1, g2)	Возвращает результат симметрической разности геометрии пространственных объектов g1 и g2.

Рассмотрим примеры использования функций. Первый запрос выводит идентификатор, адрес, координаты объекта и геометрию объекта в формате WKT для объектов из таблицы address учебной базы данных sakila:

```
SELECT address_id, address, ST_X(location) AS
x, ST_Y(location) AS y, ST_AsText(location)
```

```

geom_text
FROM sakila.address;

```

В следующем примере выполняется объединение двух полигонов:

```

SET @g1 = ST_GeomFromText('POLYGON((10 10,10 60,50
60,50 10,10 10))');
SET @g2 = ST_GeomFromText('POLYGON((50 30,50 90,110
90,110 30,50 30))';
SELECT ST_AsText(ST_UNION(@g1, @g2));

```

Пространственные отношения объектов

В задачах анализа пространственных данных очень часто возникает необходимость определить пространственные отношения некоторых объектов. Например, «в границах какого района находится здание?», «имеют ли пересечения земельные участки?» и т.д. Основные функции для анализа геоданных представлены в таблице 11.3.

Таблица 11.3. Функции анализа пространственного отношения объектов

Функция	Описание
ST_Equals (g1, g2)	Возвращает значение 1, если объекты g1 и g2 пространственно-эквивалентны, в противном случае возвращается значение 0.
ST_Intersects(g1, g2)	Возвращает значение 1, если объекты g1 и g2 пересекаются. Объекты пересекаются, если имеют как минимум одну общую точку.
ST_Disjoint(g1, g2)	Возвращает значение 1, если объекты g1 и g2 не пересекаются. Объекты не пересекаются, если не имеют общих точек.
ST_Contains(g1, g2)	Возвращает значение 1, если g1 полностью содержит g2. Объект g1 полностью содержит объект g2, если ни одна точка g2 не лежит вне g1 и как минимум одна из точек g2 является внутренней точкой g1.

Окончание табл. 11.3

Функция	Описание
ST_Within(g1, g2)	Возвращает значение 1, если g1 находится в границах g2. Является аналогом ST_Contains с обратным порядком аргументов.
ST_Touches(g1, g2)	Возвращает значение 1, если объекты касаются. Объекты касаются, если никакие из общих точек их геометрий не пересекают их внутренних частей.
ST_Overlaps(g1, g2)	Возвращает значение 1, если пространственные объекты перекрываются. Два пространственных объекта перекрываются, если они имеют одинаковую размерность и их пересечение имеет ту же размерность, что и сами объекты, и область пересечения не равна ни одному из них.
ST_Crosses(g1, g2)	Возвращает значение 1, если пространственные объекты перекрещиваются. Два пространственных объекта перекрещиваются, если их пересечение имеет размерность меньшую, чем максимальная размерность исходных объектов, и точки пересечения являются внутренними (не граничными) по отношению к обоим исходным объектам.

В качестве примера рассмотрим задачу поиска объектов внутри заданного. В примере анализ будет выполнен в СУБД PostgreSQL с установленным расширением PostGIS на данных OpenStreetMap Приволжского округа.

Создадим функцию на языке plpgsql, которая находит геометрию объекта по заданному описанию и возвращает все объекты, лежащие внутри найденной геометрии. Функция принимает один параметр – имя объекта типа текст и возвращает таблицу, содержащую 4 столбца: идентификатор объекта, его имя, тип границ и геометрию объекта:

```
CREATE OR REPLACE FUNCTION find_objects_within(object_name text)
RETURNS TABLE (osm_id bigint, name text, boundary text, way geometry)
```

```

LANGUAGE 'plpgsql'
AS $BODY$
DECLARE
    sam_region GEOMETRY;
BEGIN
    SELECT p.way INTO sam_region
    FROM planet_osm_polygon p
    WHERE p.name = object_name;

    RETURN QUERY
    SELECT p.osm_id, p.name, p.boundary, p.way
    FROM planet_osm_polygon p
    WHERE ST_Within(p.way, sam_region) AND
p.boundary IS NOT NULL;
END
$BODY$;

```

Т.к. функция возвращает таблицу, то в PostgreSQL можно выполнить запрос следующего вида:

```

SELECT *
FROM find_objects_within('Самарская область');

```

Результат выполнения запроса показан на рисунке 50.

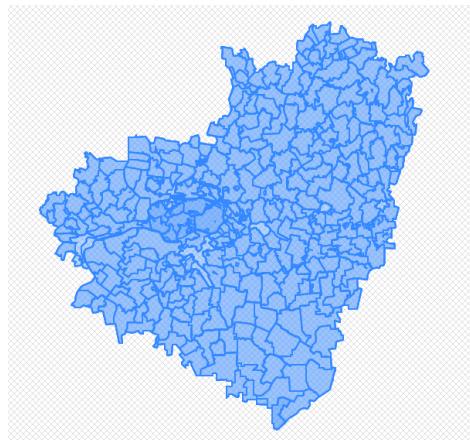


Рисунок 50 – Результат пространственного анализа

11.2 Слабоструктурированные данные

Рассмотрим еще один вид данных – слабоструктурированные данные. Как известно, в реляционных СУБД принято хранить данные в нормализованном виде: с разбиением по таблицам, столбцам и т.д. Однако нормализация не всегда имеет смысл. К примеру, в базе данных необходимо хранить некоторые настройки пользовательского приложения. Какие конкретно настройки и как они используются клиентом – серверу не важно. В этом случае нет необходимости разбирать настройки клиента, проводить нормализацию данных, записывать эти настройки в отдельные таблицы и столбцы и т.д. Сервер работает с этими данными целиком, для него они атомарны.

Другая ситуация – хранение каких-то данных, структура которых часто меняется, добавляются новые атрибуты и прочее. Физическое изменение таблиц, т.е. выполнение команд ALTER TABLE при каждом добавлении или изменении атрибута объекта может быть трудоемким процессом.

В этих случаях иногда отказываются от нормализации данных и данные хранятся в денормализованном виде, условно, в виде JSON документа. Соответственно структура этих JSON документов может отличаться даже в рамках одной таблицы для разных строк. Т.е. можно сказать, что происходит так называемое хранение данных без схемы. Происходит отказ от контроля за целостностью и корректностью данных на уровне СУБД, но появляется возможность легко менять структуру данных.

В этом случае используется хранение данных в каком-то определенном формате:

- двоичный формат (например, Protobuf, MessagePack);
- текстовый формат;
- XML;
- JSON;
- JSONB.

Преимущество хранения в двоичном формате – это компактность. Данные занимают меньше места. Минусы – данные не читаемы пользователем напрямую, к ним нельзя сделать запросы поиска, они просто хранятся в базе. В случае текстового формата для поиска данных можно использовать оператор `LIKE` или полно-текстовый поиск. Но это все еще неструктурированные данные.

Некоторые СУБД, в т.ч. MySQL и PostgreSQL, позволяют хранить данные в формате XML, обеспечивают встроенные функции работы с этим форматом, например, поиск по тегам. В случае, если данные хранятся в форматах XML или JSON, СУБД поддерживает базовую валидацию данных: в столбцы данных типов нельзя записать синтаксически некорректные документы с точки зрения XML или JSON соответственно.

В MySQL поля типа JSON не индексируются. PostgreSQL позволяет хранить JSON объекты также в столбцах типа `JSONB`. Типы данных `json` и `jsonb` принимают на вход почти одинаковые наборы значений, а отличаются они главным образом с точки зрения эффективности. Тип `json` сохраняет точную копию введённого текста, которую функции обработки должны разбирать заново при каждом выполнении запроса, тогда как данные `jsonb` сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, `jsonb` поддерживает индексацию, что тоже может быть очень полезно.

11.2.1 Тип данных JSON

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на JavaScript. Формат считается независимым от языка и может использоваться практически с любым языком программирования. Формат широко используется для обмена данными между программными системами.

В качестве значений в JSON могут быть использованы:

- запись – неупорядоченное множество пар ключ: значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми;
- массив (одномерный) – это упорядоченное множество значений. Массив заключается в квадратные скобки «[]». Значения разделяются запятыми. Массив может быть пустым, т.е. не содержать ни одного значения;
- число (целое или вещественное);
- литералы `true` (логическое значение «истина»), `false` (логическое значение «ложь») и `null`;
- строка – упорядоченное множество из нуля или более символов юникода, заключённое в двойные кавычки.

На рисунке 51 показан пример JSON-документа, хранящего информацию о фильме «Бойцовский клуб».

```
{  
    "_id": {  
        "$oid": "573a139bf29313caabcf4dd0"  
    },  
    "fullplot": "A ticking-time-bomb insomniac and a slippery soap salesman...",  
    "imdb": {  
        "rating": 8.9,  
        "votes": 1191784,  
        "id": 137523  
    },  
    "year": 1999,  
    "plot": "An insomniac office worker, looking for a way to change his life...",  
    "genres": [ "Drama" ],  
    "rated": "R",  
    "metacritic": 66,  
    "title": "Fight Club",  
    "lastupdated": "2015-09-02 00:16:15.833000000",  
    "languages": [ "English" ],  
    "writers": [ "Chuck Palahniuk (novel)", "Jim Uhls (screenplay)" ],  
    "type": "movie",  
    "awards": {  
        "wins": 11,  
        "nominations": 22,  
        "text": "Nominated for 1 Oscar. Another 10 wins & 22 nominations."  
    },  
    "countries": [ "USA", "Germany" ],  
    "cast": [ "Edward Norton", "Brad Pitt", "Helena Bonham Carter", "Meat Loaf" ],  
    "directors": [ "David Fincher" ]  
}
```

Рисунок 51 – Пример JSON-документа

Документ содержит строковые поля, целочисленные значения, вложенные структуры и массивы.

11.2.2 Работа с JSON в MySQL

Для работы с JSON-полями MySQL поддерживает тип данных JSON [13].

Добавление записи, содержащей JSON-поле, может быть выполнено несколькими способами:

- для добавления записи, содержащей поле с JSON-данными, достаточно добавить правильно сформированную JSON-строку в значение этого поля в запросе `INSERT`;
- используя функцию `JSON_OBJECT`, можно сформировать JSON-строку в процессе добавления данных. Эта функция принимает список пар ключ-значение вида:

```
JSON_OBJECT(key1, value1, ... key_n, value_n)
```

и возвращает JSON-объект. При этом если значение является массивом, используется функция:

```
JSON_ARRAY(value1, value2, ..., value_n),
```

которая возвращает массив JSON-объектов.

Для выбора отдельных атрибутов JSON-документа можно использовать:

- функцию `JSON_EXTRACT(column, path)`, которая принимает в качестве аргумента поле таблицы с JSON-данными и путь для перемещения по JSON-объекту;
- эквивалентную конструкцию вида "`column->path`";
- функцию `JSON_CONTAINS(target, candidate[, path])`, которая принимает документ `target`, в котором выполняется поиск, и документ `candidate` для сравнения, возвращает 1, когда найдено совпадение.

Все определения пути в JSON-объекте начинаются с символа \$, за которым следуют другие селекторы:

- точка, за которой следует имя, например, \$.title;
- [N], где N – позиция в массиве с нулевым индексом;
- . [*] возвращает все элементы объекта;
- [*] возвращает все элементы массива;
- prefix**suffix принимает значение всех путей, которые начинаются с префикса имени и оканчиваются суффиксом.

Для модификации JSON-объектов используются следующие функции:

- JSON_SET(doc, path, val[, path, val]...) – вставляет или обновляет данные в документе;
- JSON_INSERT(doc, path, val[, path, val]...) – вставляет данные в документ;
- JSON_REPLACE(doc, path, val[, path, val]...) – заменяет данные в документе;
- JSON.Merge(doc, doc[, doc]...) – объединяет два или более документов;
- JSON_ARRAY_APPEND(doc, path, val[, path, val]...) – добавляет значения в конец массива;
- JSON_ARRAY_INSERT(doc, path, val[, path, val]...) – вставляет массив в документ;
- JSON_REMOVE(doc, path[, path]...) – удаляет данные из документа.

Рассмотрим несколько примеров работы с JSON в MySQL. В первом примере добавим запись в поле data типа JSON, используя функции формирования JSON-строк:

```
INSERT INTO movies (data) VALUES (  
    JSON_OBJECT(  
        'imdb', JSON_OBJECT('rating', 8.7, 'votes',  
        973663, 'id', 167261),  
        'year', 2002,  
        'genres', JSON_ARRAY('Adventure', 'Fantasy'),  
        'title', 'The Lord of the Rings: The Two Tow-  
        ers'  
    )  
) ;
```

В следующем примере добавляется запись, содержащая правильно сформированную JSON-строку:

```
INSERT INTO movies (data) VALUES ('{  
    "imdb": { "rating": 9.3, "votes": 1521105,  
    "id": 111161 },  
    "year": 1994,  
    "genres": [ "Crime", "Drama" ],  
    "title": "The Shawshank Redemption"  
}' );
```

В следующем примере выполняется запрос для выборки всех фильмов с условием «рейтинг более 8.7». Для поиска в JSON используется селектор:

```
SELECT *  
FROM movies  
WHERE data->'$.imdb.rating' > 8.7;
```

12 НЕРЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

В предыдущих разделах рассматривались реляционные базы данных. Достоинством реляционной модели являются:

- простота представления структуры базы данных, отсутствие избыточности данных, поддержка целостности за счет нормализации отношений;
- наличие стандарта SQL; разные диалекты SQL очень похожи между собой, различия проявляются в основном в процедурных расширениях языка;
- выполнение требований ACID к транзакционной системе: атомарность, согласованность, изолированность, долговечность. Выполнение этих требований позволяет обеспечить, в том числе, долговечность данных за счет принципа двойной записи с использованием журнала транзакций, и параллельную обработку данных благодаря принципу изолированности, используемому в механизмах транзакций.

Также можно выделить основные недостатки реляционных баз данных:

- потеря соответствия, т.е. различие между реляционной моделью и структурами данных, находящимися в памяти;
- сложность горизонтального масштабирования базы данных.

12.1 Недостатки реляционной модели

Рассмотрим недостатки реляционной модели немного подробнее.

12.1.1 Потеря соответствия

Реляционная модель данных представляет их в виде отношений и кортежей, причем значения в реляционном кортеже должны

быть простыми. Поэтому, если в базе данных будет храниться в памяти и обрабатываться сложная структура, то необходимо будет преобразовывать реляционное представление в необходимую форму. В итоге возникает потеря соответствия – два разных представления, требующих трансляции.

Пример такого несоответствия показан на рисунке 52. Заказ, который в пользовательском интерфейсе выглядит как единая агрегированная структура, в реляционной базе данных разделяется на множество строк из многих таблиц.

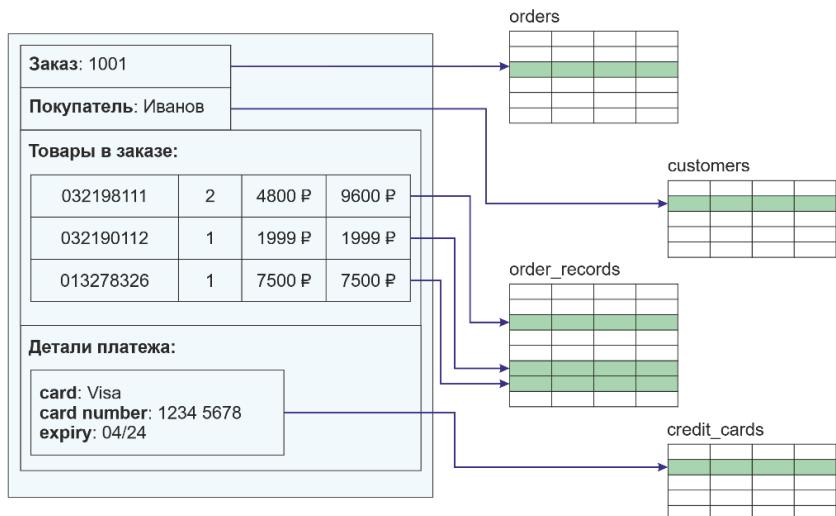


Рисунок 52 – Потеря соответствия в реляционной модели

Хотя многие СУБД стали добавлять поддержку хранения слабоструктурированных данных, например, обработку JSON, но это, по сути, уход от реляционной модели данных.

12.1.2 Сложность масштабирования

Другая проблема – обработка больших объемов данных. Для того чтобы справиться с возрастающим объемом данных и трафика,

потребовались более крупные вычислительные ресурсы для хранения и анализа данных.

Существовали две возможности масштабирования: вертикальное и горизонтальное. **Вертикальное** масштабирование подразумевает увеличение производительности каждого компонента системы с целью повышения общей производительности, т.е. укрупнение компьютеров, увеличение количества процессоров, а также дисковой и оперативной памяти. Но более крупные машины становятся все более и более дорогими, не говоря уже о том, что существует физический предел их укрупнения.

Альтернативой было использование **горизонтального** масштабирования, т.е. разбиение системы на более мелкие структурные компоненты и разнесение их по отдельным физическим машинам, объединенных в кластер. Кроме того, кластер более надежен – отдельные машины могут выйти из строя, но весь кластер продолжит функционирование, несмотря на сбои, обеспечивая высокую надежность.

Когда произошел сдвиг в сторону кластеров, возникла новая проблема – реляционные базы данных не предназначены для работы на кластерах. Кластерные реляционные базы данных основаны на концепции общей дисковой подсистемы. Они используют кластерную файловую систему, выполняющую запись данных в легко доступную дисковую подсистему, но это значит, что дисковая подсистема по-прежнему является единственным источником уязвимости кластера.

Реляционные базы данных также могут работать на разных серверах с разными наборами данных, эффективно выполняя сегментирование (sharding) базы данных. Хотя это позволяет разделить рабочую нагрузку, вся процедура сегментирования должна контролироваться приложением, которое должно следить за тем, к какому серверу базы данных и за какой порцией данных обращаться. Кроме

того, утрачивается возможность управления запросами, целостностью данных, транзакциями и согласованностью между частями кластера.

Это несоответствие между реляционными базами данных и кластерами вынудило некоторые организации рассмотреть альтернативные способы хранения данных. В частности, большое влияние оказали две компании – Google и Amazon, представившие свои решения BigTable от компании Google и Dynamo от компании Amazon для хранения огромных объемов данных.

12.2 NoSQL базы данных

В зависимости от модели данных и подходов к распределённости и репликации в NoSQL-движении выделяются четыре основных типа систем [14]:

- 1) модели «ключ-значение»;
- 2) документоориентированные СУБД;
- 3) системы типа «семейство столбцов»;
- 4) графовые СУБД.

Модель «ключ-значение» является простейшим вариантом, использующим ключ для доступа к значению. Такие базы данных как правило используют хеш-таблицу, в которой находится уникальный ключ и указатель на конкретный объект данных. Хотя базы данных типа «ключ-значение» могут пригодиться в определённых ситуациях, они не лишены недостатков. Первый заключается в том, что модель не предоставляет стандартные возможности баз данных вроде атомарности транзакций или согласованности данных при одновременном выполнении нескольких транзакций. Такие возможности должны предоставляться самим приложением. Второй недостаток в том, что при увеличении объёмов данных поддержание

уникальных ключей может стать проблемой. Для её решения необходимо как-то усложнять процесс генерации строк, чтобы они оставались уникальными среди очень большого набора ключей. Riak, Redis, Amazon DynamoDB – самые популярные СУБД данных такого типа.

Документоориентированные базы данных служат для хранения иерархических структур данных (документов). В основе документоориентированных СУБД лежат документные хранилища, имеющие структуру дерева. Одним из ключевых различий между базами данных типа «ключ-значение» и документоориентированными базами данных является то, что последние включают метаданные, связанные с хранимым содержимым, что даёт возможность делать запросы на основе содержимого. CouchDB, Couchbase и MongoDB – наиболее популярные документоориентированные СУБД.

В системах типа «семейство столбцов» данные хранятся в виде разреженной матрицы, строки и столбцы которой используются как ключи. В сравнении с хранением данных в строках, как в большинстве реляционных баз данных, преимущества хранения в колонках заключаются в быстром поиске/доступе и агрегации данных. Реляционные базы данных хранят каждую строку как непрерывную запись на диске. Разные строки хранятся в разных местах на диске, в то время как колоночные базы данных хранят все ячейки, относящиеся к колонке, как непрерывную запись, что делает операции поиска/доступа быстрее. Самыми известными примерами являются Google BigTable, HBase, Apache Cassandra.

Графовые СУБД применяются для хранения графовых структур. Такие базы данных используют рёбра и узлы для представления данных. Узлы связаны между собой определёнными отношениями, представленными рёбрами между ними. Узлы и рёбра хранят некоторые атрибуты. Первая графовая СУБД Neo4j создана в 2007

году. По состоянию на начало 2020-х годов существуют десятки других графовых СУБД.

12.2.1 Теорема CAP

В 2000 году Э. Брюером было сформулировано эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств [15]:

- согласованность данных (consistency) – во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- доступность (availability) – любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;
- устойчивость к разделению (partition tolerance) – расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

Этот принцип стал известен как теорема CAP (как акроним consistency, availability и partition tolerance) и впоследствии получил широкую популярность и признание в среде специалистов по распределённым вычислениям. Концепция NoSQL, в рамках которой создаются распределённые нетранзакционные системы управления базами данных, зачастую использует этот принцип в качестве обоснования неизбежности отказа от согласованности данных.

С точки зрения теоремы CAP, распределённые системы в зависимости от пары практически поддерживаемых свойств из трёх возможных распадаются на три класса – CA, CP, AP.

Первый класс – CA (Availability + Consistency без поддержки Partition Tolerance), когда данные во всех узлах кластера согласованы и доступны, но не устойчивы к разделению. Это означает, что

реплики одной и той же информации, распределённые по разным серверам друг другу, не противоречат друг другу и любой запрос к распределённой системе завершается корректным откликом. Такие системы возможны при поддержке ACID-требований к транзакциям (атомарность, согласованность, изолированность, долговечность) и абсолютной надежности сети. Классическим примером СА-системы называют распределённую службу каталогов LDAP, а также реляционные базы данных.

СР-система (Consistency + Partition Tolerance – Availability) в каждый момент обеспечивает целостность данных и способна работать в условиях распада в ущерб доступности, не выдавая отклик на запрос. Устойчивость к разделению требует дублирования изменений во всех узлах системы, что реализуется с помощью распределённых пессимистических блокировок для сохранения целостности. По сути, СР – это система с несколькими синхронно обновляемыми мастер-базами. Она всегда корректно отрабатывает транзакцию только в том случае, если изменения удалось распространить по всем серверам. Система продолжает корректно читать данные даже при отказе одного из узлов кластера. Но в этом случае запись будет обрываться или сильно задерживаться, пока система не убедится в своей целостности и согласованности. Из NoSQL-СУБД к СР-системам принято относить Apache HBase, MongoDB, Redis и другие.

АР-система (Availability + Partition tolerance – Consistency) не гарантирует целостность данных, обеспечивая их доступность и устойчивость к разделению, например, как в распределённых веб-кэшах и DNS. Считается, что большинство NoSQL-СУБД относятся к этому классу систем, обеспечивая лишь некоторой уровень согласованности данных в конечном счете (*eventually consistent*). Таким образом, АР-система может быть представлена кластером из нескольких узлов, каждый из которых может принимать данные, но

не обязуется в тот же момент распространять их на другие сервера. Такая система отлично справляется с отказами нескольких узлов, но, когда они снова начинают работать, возможна выдача пользователям старых данных. К AP-системам относят CouchDB, Cassandra, Riak, Amazon DynamoDB.

При всей понятной на первый взгляд концепции тройственной ограниченности, CAP-теорему критикуют за чрезмерное упрощение важных понятий, что приводит к неверному пониманию первоначального смысла модели. В результате этого, теорема из строгого, математически доказанного утверждения превращается в маркетинговый термин с расплывчатым смыслом. Наиболее явно это выражается в следующих случаях:

- **согласованность** в реляционных базах данных как часть требований ACID к транзакционной системе подразумевает, что в системе транзакция фиксирует только допустимые результаты. Согласованность в CAP означает линеаризуемость, а не фиксацию завершённой транзакции. В реальности это значит, что информация во всех репликах, включая кэшированные данные, должна быть одна и та же. Достичь этого не очень просто;
- определение **доступности** имеет две серьёзные проблемы. Первая – нет понятия частичной доступности, или какой-то её степени (проценты например), а есть только полная доступность. Вторая проблема – неограниченное время ответа на запросы, т.е. даже если система отвечает час, она всё ещё доступна;
- **устойчивость к разделению** означает, что для связи используется асинхронная сеть, которая может терять или задерживать сообщения, что характерно для любой интернет-системы.

12.2.2 Альтернативные подходы. BASE. PACELC

Поскольку даже сам автор CAP-теоремы отмечает ее несостоительность для оценки современных распределенных систем, были предложены альтернативные подходы. Наиболее известным подходом считается **BASE** (Basically Available, Soft-state, Eventually consistent) – базовая доступность, неустойчивое состояние, согласованность в конечном счёте. Этот подход к построению распределенных систем был сформулирован во второй половине 2000-х годов.

Базовая доступность означает, что сбой на некоторых узлах приведет к отказу в обслуживании только незначительной части сессий, сохраняя доступность в большинстве случаев.

Неустойчивое состояние подразумевает возможность жертвовать долговременным хранением состояния сессий (промежуточные результаты выборок, информация о навигации и контексте) в пользу фиксации обновлений только критичных операций.

Согласованность в конечном счёте предполагает возможность противоречивости данных в некоторых случаях, но гарантирует итоговую целостность информации в практически обозримое время.

Еще одним альтернативным подходом к построению распределенных систем считается теорема **PACELC**, впервые описанная в 2012 году. Она основана на модели CAP, но, помимо согласованности, доступности и устойчивости к разделению также включает временную задержку (L, Latency) и логическое исключение между сочетаниями этих понятий.

Согласно PACELC, в случае сетевого разделения (P) в распределенной системе необходимо выбирать между доступностью (A) и согласованностью (C), как и в CAP- теореме, но в остальном (E, ELSE), даже при нормальной работе системы без разделения, нужно выбирать между задержкой (L) и согласованностью (C).

В логическом выражении PACELC формулируют следующим образом:

IF P \rightarrow (C or A), ELSE (C or L).

Так PACELC расширяет и уточняет CAP-теорему, регламентируя необходимость поиска компромисса между временной задержкой и согласованностью данных в распределенных системах.

Собственно, все эти подходы говорят, что построить идеальную систему невозможно, чем-то приходится жертвовать.

13 ДОКУМЕНТООРИЕНТИРОВАННАЯ СУБД MONGODB

Рассмотрим NoSQL СУБД на практике на примере СУБД MongoDB [8, 9]. MongoDB – это кроссплатформенная документоориентированная база данных NoSQL с открытым исходным кодом. Она не требует описания схемы таблиц, как в реляционных БД. Данные хранятся в виде коллекций и документов, документы хранятся в бинарном JSON-подобном формате.

Основными возможностями MongoDB являются:

- поддержка запросов по значениям полей документа, в т.ч. поддержка запросов с использованием регулярных выражений;
- поддержка индексации данных, включая вторичные индексы, уникальное, составное, геопространственное и полнотекстовое индексирования;
- поддержка репликации, то есть работа с двумя или более копиями данных на различных узлах. Каждый экземпляр набора реплик может в любой момент выступать в роли основной или вспомогательной реплики. Все операции записи и чтения по умолчанию осуществляются с основной репликой. Вспомогательные реплики поддерживают актуальное состояние копии данных. В случае, когда основная реплика дает сбой, система переключается на второстепенную. Второстепенные реплики могут дополнительно являться источником для операций чтения;
- поддержка горизонтального масштабирования, используя технику сегментирования объектов баз данных – распределение их частей по различным узлам кластера. Благодаря тому, что каждый узел кластера может принимать запросы, обеспечивается балансировка нагрузки. Сегментированием является основным важным достоинством MongoDB перед реляционными решениями;

- поддержка агрегации данных. MongoDB поддерживает аналог SQL-выражения GROUP BY для агрегации данных и предоставляет фреймворк для агрегации на базе концепции конвейеров обработки данных. Кроме того, поддерживается парадигма MapReduce.

К недостаткам MongoDB можно отнести следующие:

- отсутствие схемы, что усложняет разработку типовых приложений. Теперь контролировать схему документа необходимо в приложении. Кроме того, это увеличивает объёмы базы, т.к. каждый документ кроме своих данных, хранит также имена полей;
- отсутствие JOIN и как решение этой проблемы – денормализация. Каждый документ хранит всю необходимую информацию в себе, зачастую дублируя данные. Возможности денормализации при этом ограничиваются размером документа (16 Мб по умолчанию). Существующий аналог JOIN – оператор \$lookup – не работает для сегментированных коллекций;
- в MongoDB добавлена поддержка транзакций, удовлетворяющих требованиям ACID – атомарность, согласованность, изолированность, надежность. Однако заявленная поддержка транзакций ограничена, и выполнение транзакций занимает относительно много времени на сегментированных коллекциях.

13.1 Основные понятия

Документ – это упорядоченный набор ключей со связанными значениями. Документ представляет собой основную единицу данных в MongoDB и приблизительно эквивалентен строке в реляционной системе управления базами данных.

Коллекция – это группа документов. Аналогично, коллекцию можно рассматривать как таблицу с динамической схемой.

База данных – это группа коллекций.

Как и в реляционных СУБД, один экземпляр MongoDB может содержать несколько независимых баз данных, каждая из которых содержит свои собственные коллекции.

У каждого документа есть специальный ключ «`_id`», который является уникальным в рамках коллекции.

Соответствие основных понятий в реляционных СУБД и в MongoDB представлено в таблице 13.1.

Таблица 13.1. Основные понятия MongoDB

Реляционная СУБД	MongoDB
База данных	База данных
Таблица	Коллекция
Кортеж / строка	Документ
Столбец	Поле
Первичный ключ	Первичный ключ (ключ по умолчанию <code>_id</code>)

13.2 Типы данных

Документы в MongoDB можно рассматривать как «JSON-подобные» в том смысле, что они концептуально похожи на JSON-объекты. JSON поддерживает `null`, логический тип данных, число, строка, массив и объект. MongoDB добавляет поддержку ряда дополнительных типов данных, сохраняя при этом нотацию типа «ключ/значение» в JSON.

Наиболее распространенные типы данных в MongoDB представлены в таблице 13.2.

Таблица 13.2. Типы данных MongoDB

Спецификация	Описание
Null	Null можно использовать для обозначения как нулевого значения, так и несуществующего поля: { "x" : null}.
Логический тип	Тип данных, который можно использовать для значений true и false: { "x" : true}.
Числовой тип	По умолчанию используются 64-битные числа с плавающей точкой: { "x" : 3.14}, { "x" : 3}. Для хранения целых чисел используются классы NumberInt или NumberLong, которые обозначают 4-байтовые или 8-байтовые целые числа со знаком соответственно.
Строковый тип	Строка символов в кодировке UTF-8 может быть представлена с использованием строкового типа: { "x" : "foobar" } .
Дата	Дата хранится в виде 64-битных целых чисел, обозначающих миллисекунды с момента эпохи Unix (1 января 1970 г.). Часовой пояс не сохраняется: { "x" : new Date() } .
Массив	Наборы или списки значений могут быть представлены в виде массивов: { "x" : ["a", "b", "c"] }.
Вложенный документ	Документы могут содержать вложенные документы: { "x" : { "foo": "bar" } }.
Идентификатор объекта	Идентификатор объекта – это 12-байтовый идентификатор для документов: { "x" : ObjectId() } .
Двоичные данные	Двоичные данные – это строка из произвольных байтов.
Код	MongoDB также позволяет хранить произвольный код JavaScript в запросах и документах: { "x" : function() { /* ... */ } }.

13.3 Создание документов

Вставка – основной метод добавления данных в MongoDB. Основные методы добавления данных перечислены в таблице 13.3.

Таблица 13.3. Методы создания документов

Метод	Описание
<pre>db.collection.insertOne(<document>, { writeConcern: <document> })</pre>	Добавление одного документа в коллекцию collection.
<pre>db.collection.insertMany([<document_1> , <document_2>, ...], { writeConcern: <document>, ordered: <boolean> })</pre>	Добавление нескольких документов в коллекцию collection.

Чтобы вставить один документ, используется метод коллекции `insertOne`. Метод добавит в документ ключ «`_id`» (если его не указали явно) и сохранит документ в MongoDB. Если коллекция не существует, операции вставки создадут коллекцию.

Параметр `writeConcern` – опциональный, позволяющий узнать, успешно ли завершилась операция вставки, в т.ч. подтвердилась запись в несколько реплик.

Если необходимо вставить несколько документов в коллекцию, можно использовать метод `insertMany`. Этот метод позволяет передавать массив документов в базу данных, что эффективнее добавление документов по одному.

Опциональный параметр `ordered` позволяет указывать, использовать упорядоченное или неупорядоченное добавление. Если указано значение `true` (значение по умолчанию), то документы бу-

дут добавляться в указанном порядке. Также, если добавление массива документов завершится с ошибкой вставки на каком-то документе, ни один документ за пределами этой точки в массиве не будет вставлен. В случае с неупорядоченными вставками MongoDB попытается вставить все документы независимо от того, приводят ли некоторые вставки к ошибкам.

Рассмотрим пример добавления документа:

```
use db_lectures
db.products.insertOne( { item: "card", qty: 15 } );
```

В примере сначала создается база данных db_lectures. В отличие от типичных реляционных СУБД, для этого не используется специальный оператор CREATE DATABASE, а просто указывается, что необходимо использовать базы данных командой use. Если такой базы не существует – она будет создана. Далее, с использование команды insertOne добавляется один документ с полями item и qty в коллекцию products. Если коллекция не существует – она будет создана. Т.к. идентификатор объекта явно не указывался, он будет создан самой СУБД.

13.4 Выборка документов

13.4.1 Базовый синтаксис

Основной метод, используемый для выборки данных из коллекции – это метод find:

```
db.collection.find(query, projection)
```

Метод принимает два опциональных параметра: query, который определяет условия фильтрации документов с использованием операторов запроса, и projection, определяющий поля, которые

должны возвращаться в документах, соответствующих фильтру запроса. Метод возвращает курсор (или указатель) на выбранные документы.

Для выбора всех документов коллекции используется метод `find` с пустым документом-фильтром:

```
db.inventory.find( {} )
```

Такой вызов метода `find` эквивалентен SQL-запросу следующего вида:

```
SELECT * FROM inventory;
```

Рассмотрим, как фильтровать документы с использованием переменной `query`. Для фильтрации документом используются условия следующего вида:

```
{
  <field1>: <value1>,
  <field2>: { <operator>: <value> },
  ...
}
```

В условии фильтрации поле документа либо явно сравнивается со значением, используя синтаксическую конструкцию `<field1>: <value1>`, либо дополнительно используются операторы сравнения через конструкцию `<field2>: { <operator>: <value> }`.

Один документ, описывающий запрос фильтрации, может содержать несколько условий. В качестве оператора в запросе могут использоваться операторы сравнения, логические операторы, операторы работы с полями и массивами. В MongoDB используются стандартные операторы сравнения, представленные в таблице 13.4.

Таблица 13.4. Операторы сравнения

Оператор	Описание
\$eq	Равно (=).
\$ne	Не равно (!=).
\$gt	Больше (>).
\$gte	Больше или равно (>=).
\$lt	Меньше (<).
\$lte	Меньше или равно (<=).
\$in	Соответствие хотя бы одному элементу массива.
\$nin	Несоответствие ни одному элементу массива.

Также поддерживаются стандартные логические операторы, представленные в таблице 13.5.

Таблица 13.5. Логические операторы

Оператор	Описание
\$and	Логическое И.
\$or	Логическое ИЛИ.
\$not	Логическое отрицание.
\$nor	Логическое НЕ-ИЛИ. Возвращает все документы, которые не соответствуют обоим предложениям.

Кроме того, поддерживаются операторы проверки на существование указанного поля в документе, проверки типа поля документа, а также поиска с использованием регулярных выражений (таблица 13.6).

Таблица 13.6. Операторы фильтрации элементов

Оператор	Описание
\$exists	Возвращает документы, которые содержат указанное поле.
\$type	Возвращает документы, если поле имеет указанный тип.
\$regex	Возвращает документы, значения полей которых соответствуют указанному регулярному выражению.

Основные операторы фильтрации по условиям на поля-массивы перечислены в таблице 13.7.

Таблица 13.7. Операторы фильтрации элементов

Оператор	Описание
\$all	Возвращает документы, массивы которых содержат все элементы, указанные в запросе.
\$elemMatch	Возвращает документы, если элемент поля-массива соответствует всем указанным условиям.
\$size	Возвращает документы, если поле-массив имеет указанный размер.

Рассмотрим примеры фильтрации документов. Начнем с базовой синтаксической конструкции документа-фильтра {<field1>: <value1>}, т.е. проверка осуществляется по совпадению значения поля и искомого значения:

```
db.inventory.find( { status: "D" } )
```

В примере осуществляется выборка документов указанного статуса. Указанный запрос эквивалентен SQL-запросу следующего вида:

```
SELECT * FROM inventory WHERE status = 'D';
```

Более сложные запросы используют операторы в конструкции {<field1>: { <operator1>: <value1> }, ... }:

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

В примере осуществляется поиск документов, статусы которых равны A или D. В SQL запрос выглядел бы следующим образом:

```
SELECT * FROM inventory WHERE status IN ('A', 'D');
```

Условия фильтрации могут объединяться, например:

```
db.inventory.find( {  
    status: "A",  
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]  
} )
```

Что эквивалентно SQL-запросу:

```
SELECT * FROM inventory  
WHERE status = 'A' AND (qty < 30 OR item LIKE 'p%');
```

Для фильтрации документов по вложенным полям используется тот же подход, который применялся при работе с JSON-документами в реляционных базах данных. Если необходимо применить условие на вложенное поле uom переменной size, то фильтр будет выглядеть следующим образом:

```
db.inventory.find( { "size.uom": "in" } )
```

По умолчанию запросы в MongoDB возвращают все поля в соответствующих документах. Чтобы ограничить объем данных, которые MongoDB отправляет приложениям, можно использовать параметр projection, чтобы указать или ограничить возвращаемые поля.

Проекция может явно включать несколько полей, установив для поля значение 1 в документе проекции.

Следующая операция возвращает результирующий набор, который содержит только поля item, status и, по умолчанию, поле _id:

```
db.inventory.find( { status: "D" } ,  
                   { item: 1, status: 1 } )
```

Т.е. этот пример аналогичен SQL-запросу следующего вида:

```
SELECT _id, item, status  
FROM inventory  
WHERE status = 'D';
```

13.4.2 Методы курсора

Метод find возвращает курсор – т.е. указатель на набор документов, которые удовлетворяют условию поиска. У результирующего курсора можно вызвать методы для сортировки, ограничения числа возвращаемых записей и другие. Часть основных методов приведена в таблице 13.8.

Таблица 13.8. Методы курсора

Оператор	Описание
cursor.count()	Изменяет курсор, чтобы он возвращал количество документов в результирующем наборе, а не сами документы.
cursor.limit()	Ограничивает размер результирующего набора.
cursor.map()	Применяет функцию к каждому документу в курсоре и возвращает полученные значения в виде массива.
cursor.forEach()	Применяет JavaScript-функцию к каждому документу в курсоре.
cursor.skip()	Возвращает курсор, который начинает возвращать результаты только после пропуска набора документов.
cursor.sort()	Возвращает документы, упорядоченные в соответствии со спецификацией сортировки.

В следующем примере выбираются два документа, начиная с четвертого. Набор документов при этом сортируется с использованием функции `sort`:

```
db.inventory.find( {} )
    .limit( 2 ).skip( 3 )
    .sort( { item: 1, qty: -1 } )
```

Функция `sort` принимает документ – набор пар типа «ключ/значение», где ключи – это имена полей, а значения – направления сортировки. Направление сортировки может быть 1 (по возрастанию) или -1 (по убыванию). Если указано несколько ключей, результаты будут отсортированы в указанном порядке. В примере документы сортируются по названию (поле `item`) по возрастанию и по полю `qty` по убыванию.

Кроме того, можно применить к каждому документу в наборе некоторую функцию и вернуть результат преобразования с использованием функции `map`:

```
db.products.find( ).limit( 5 ).map( function(p) {
    return p.item + ' / ' + p.qty
})
```

В данном случае в функцию `map` передается JavaScript функция, которая принимает документ `p` и возвращает строку – название товара/количество. Результат возвращается в виде массива.

13.4.3 Агрегация данных

Операции агрегации данных обрабатывают набор документов и возвращают результаты обработки. Агрегация данных может быть использована для:

- группировки значений из нескольких документов;

- выполнения операций над сгруппированными данными для получения единственного результата;
- анализа изменения данных в течение определенного промежутка времени.

Для агрегации данных используются:

- конвейеры агрегации, которые являются предпочтительным методом агрегирования данных. Конвейер агрегации выбирает данные из коллекции и пропускает документы через один или несколько этапов, каждый из которых выполняет свою операцию;
- методы агрегации, которые просты, но лишены возможностей конвейера агрегации.

Простые методы агрегации объединяют документы из одной коллекции (таблица 13.9).

Таблица 13.9. Методы агрегации данных

Оператор	Описание
db.collection. estimatedDocumentCount ()	Возвращает приблизительное количество документов в коллекции или представлении.
db.collection.count ()	Возвращает количество документов в коллекции или представлении.
db.collection.distinct ()	Возвращает массив документов, которые имеют различные значения для указанного поля.

Например, чтобы выбрать все статусы документов из коллекции `inventory` без повторений, можно воспользоваться запросом следующего вида:

```
db.inventory.distinct( "status" )
```

Результатом работы будет массив, содержащий значения статусов. Эквивалентный SQL-запрос имеет следующий вид:

```
SELECT DISTINCT status FROM inventory;
```

Однако простые методы агрегации не могут обеспечить выполнение сложных запросов анализа данных, поэтому основным способом агрегации данных является использование конвейера агрегации.

Конвейер агрегации состоит из одного или нескольких этапов обработки документов:

- каждый этап выполняет операцию над входными документами. Например, на этапе может происходить фильтрация документов, группировка документов и вычисление некоторых значений;
- документы, полученные как результат работы этапа, передаются на следующий этап;
- конвейер агрегации может возвращать результаты для групп документов, например, вернуть среднее, максимальное и минимальное значения;
- начиная с MongoDB 4.2, конвейер агрегации может использоваться для обновления документов.

Для агрегации данных из коллекции collection используется метод aggregate, который принимает два параметра: pipeline – последовательность операций или этапов агрегации данных, и options – дополнительные параметры метода:

```
db.collection.aggregate(pipeline, options)
```

В базовом варианте использования в метод aggregate необходимо передать массив stage – этапов агрегации:

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Часть операторов агрегации данных представлена в таблице 13.10.

Таблица 13.10. Операторы агрегации данных

Оператор	Описание
\$count	Возвращает количество документов на данном этапе конвейера агрегации.
\$group	Группирует входные документы по указанному выражению-идентификатору и применяет выражения-аккумуляторы, если они указаны, к каждой группе. Выводит по одному документу для каждой отдельной группы. Выходные документы содержат только поле-идентификатор и, если указано, поля-аккумуляторы.
\$limit	Передает первые n документов без изменений в конвейер, где n – указанное ограничение. Для каждого входного документа выводит либо один документ (для первых n документов), либо ноль документов (после первых n документов).
\$lookup	Выполняет левое внешнее соединение с другой коллекцией в той же базе данных, чтобы фильтровать документы из «объединенной» коллекции для обработки.
\$match	Фильтрует поток документов, позволяя только документам, удовлетворяющим условию, передаваться без изменений на следующий этап конвейера.
\$merge	Записывает результирующие документы конвейера агрегации в коллекцию. Должен быть последним этапом конвейера.
\$project	Изменяет структуру каждого документа в потоке, например, добавляя новые поля или удаляя существующие поля. Для каждого входного документа выводит один документ.
\$set	Добавляет новые поля в документы. Как и \$project, \$set изменяет структуру каждого документа; в частности, путем добавления новых полей в выходные документы, которые содержат как существующие поля из входных документов, так и вновь добавленные поля.
\$skip	Пропускает первые n документов, где n – указанный номер пропуска, и передает остальные документы без изменений в конвейер.

Окончание табл. 13.10

Оператор	Описание
\$sort	Переупорядочивает поток документов по указанному ключу сортировки. Меняется только порядок, документы остаются без изменений.
\$unset	Удаляет/исключает поля из документов.
\$unwind	Преобразует поле массива из входных документов для вывода документа для каждого элемента. Каждый выходной документ заменяет массив значением элемента. Для каждого входного документа выводит n документов, где n – количество элементов массива, которое может быть равно нулю для пустого массива.

Аккумуляторы – еще один тип выражений, использующихся для вычисления значения из значений полей, находящихся в нескольких документах. Аккумуляторы, которые предоставляет фреймворк агрегации, позволяют выполнять такие операции, как суммирование всех значений в определенном поле, вычисление среднего значения и т.д. Основные аккумуляторы представлены в таблице 13.11.

Конвейер агрегации позволяет СУБД предоставлять собственные возможности агрегации, соответствующие многим распространенным операциям агрегации данных в SQL.

Таблица 13.11. Аккумуляторы данных

Оператор	Описание
\$accumulator	Возвращает результат пользовательской функции-аккумулятора.
\$avg	Возвращает среднее числовых значений. Игнорирует нечисловые значения.
\$first	Возвращает значение из первого документа для каждой группы. Порядок определяется только в том случае, если документы отсортированы.

Окончание табл. 13.11

Оператор	Описание
\$last	Возвращает значение из последнего документа для каждой группы. Порядок определяется только в том случае, если документы отсортированы.
\$max	Возвращает наибольшее значение выражения для каждой группы.
\$min	Возвращает наименьшее значение выражения для каждой группы.
\$sum	Возвращает сумму числовых значений. Игнорирует нечисловые значения.

В таблице 13.12 представлен обзор общих терминов, функций и концепций агрегации SQL, а также соответствующих операторов агрегации MongoDB.

Таблица 13.12. Связь операторов агрегации данных с SQL-операторами

SQL-операторы и функции	Операторы агрегации в MongoDB
WHERE	\$match
GROUP BY	\$group
HAVING	\$match
SELECT	\$project
ORDER BY	\$sort
LIMIT	\$limit
SUM()	\$sum
COUNT()	\$sum \$sortByCount
JOIN	\$lookup

Примеры использования конвейера агрегации данных будем рассматривать на коллекции `zipcodes`, каждый документ которой хранит идентификатор `_id`, название города `city`, популяцию `pop`, штат `state` и координаты `loc`.

В первом примере выполняется подсчет числа записей с популяцией больше 1000 для каждого штата:

```
db.zips.aggregate( [ {  
    $match: { pop: { $gt: 1000 } }  
, {  
    $group: {  
        _id: '$state',  
        cities_count: { $count: {} }  
    }  
} ] )
```

План выполнения этого запроса можно представить в виде нескольких этапов. На первом этапе выполняется фильтрация всех документов с популяцией больше 1000, на втором этапе группируются отфильтрованные на первом этапе документы по штатам и подсчитывается количество записей.

Выполнение метода `aggregate` эквивалентно выполнению SQL-запроса следующего вида:

```
SELECT state AS _id, COUNT(*) AS cities_count  
FROM zips  
WHERE pop > 1000  
GROUP BY state;
```

В следующем примере выполняется выборка штатов с населением более 10 миллионов:

```
db.zips.aggregate( [  
    {  
        $group: {  
            _id: "$state",  
            totalPop: {  
                $sum:  
                "$pop"  
            }  
        }  
    }  
])
```

```
    { $match: { totalPop: { $gte: 10 * 1000 * 1000
} } }
] )
```

В запросе на первом этапе выполняется группировка данных по штатам с подсчетом суммарного населения в каждом штате, а затем выполняется фильтрация групп. Несмотря на то, что выполняется фильтрация агрегированных документов, для проверки условия все так же используется оператор `match`.

Тот же запрос на языке SQL мог бы быть записан как

```
SELECT state AS _id, SUM(pop) AS totalPop
FROM zips
GROUP BY state
HAVING totalPop >= 10 * 1000 * 1000;
```

Можно еще раз отметить разницу. В SQL для фильтрации строк и сгруппированных данных использовались разные операторы `WHERE` и `HAVING`. В MongoDB для этого используется один оператор `match`, т.к. нет разницы, обрабатываются ли исходные документы коллекции или документы, представляющие результаты агрегирующих операций.

В следующем примере результат агрегации данных записывается в новую коллекцию:

```
db.zips.aggregate( [
  { $group: { _id: {state: "$state", city:
"$city"},
pop: { $sum: "$pop" } } },
  { $group: { _id: "$_id.state",
avgCityPop: { $avg: "$pop" } } },
  { $merge: "state_avg_data" }
])
```

В примере используются два этапа группировки данных для расчета среднего значения популяции по штатам. Для сохранения результата агрегации используется этап `merge`, который сохраняет документы в коллекцию `state_avg_data`.

В примерах была рассмотрена только малая часть операторов и возможностей конвейера агрегации. В частности, при агрегации могут использоваться геопространственные операторы, а результаты агрегации также могут использоваться для обновления существующих документов.

13.5 Обновление документов

Для обновления документов чаще всего используются функции `updateOne` для обновления значений полей одного документа, `updateMany` для обновления нескольких документов и `replaceOne` для замены всего документа.

Функции принимают в качестве параметров документ с условием фильтрации, документ с параметрами обновления и дополнительные опции (таблица 13.13).

Таблица 13.13. Методы обновления документов

Метод	Описание
<code>db.collection.updateOne(<filter>, <update>, <options>)</code>	Обновление одного документа
<code>db.collection.updateMany(<filter>, <update>, <options>)</code>	Обновление нескольких документов
<code>db.collection.replaceOne(<filter>, <update>, <options>)</code>	Замена документа

Синтаксис метода `updateOne` имеет следующий вид:

```
db.collection.updateOne(  
    <filter>,  
    <update>,  
    {  
        upsert: <boolean>,  
        writeConcern: <document>,  
        collation: <document>,  
        arrayFilters: [ <filterdocument1>, ... ],  
        hint: <document|string>  
    }  
)  
<update> = {  
    <update operator>: { <field1>: <value1>, ... },  
    <update operator>: { <field2>: <value2>, ... },  
    ...  
}
```

Метод принимает следующие параметры:

- `filter` – условие, по которому будет выбран документ для обновления. Могут применяться те же параметры фильтрации, что и в методе `find`. Если по указанному условию будет найдено несколько документов – обновится первый;
- `update` – документ с описанием модификаций. В базовом случае описывает операторы обновления и параметры;
- набор опций. В частности, если указано `upsert: true`, то функция `updateOne` создаст новый документ, если нет документов, соответствующих фильтру. По умолчанию этот параметр равен `false`.

MongoDB поддерживает ряд операторов для модификации стандартных полей и массивов.

Основные операторы работы с полями перечислены в таблице 13.14, с массивами – в таблице 13.15.

Таблица 13.14. Операторы работы с полями

Оператор	Описание
\$currentDate	Устанавливает значение поля на текущую дату.
\$inc	Увеличивает значение поля на указанную величину.
\$min	Обновляет поле только в том случае, если указанное значение меньше существующего значения поля.
\$max	Обновляет поле только в том случае, если указанное значение больше, чем существующее значение поля.
\$mul	Умножает значение поля на указанную величину.
\$rename	Переименовывает поле.
\$set	Устанавливает значение поля в документе.
\$setOnInsert	Задает значение поля, если обновление приводит к вставке.
\$unset	Удаляет указанное поле из документа.

Таблица 13.15. Операторы работы с массивами

Оператор	Описание
\$	Модификатор, указывающий, что необходимо обновить первый элемент, соответствующий условию запроса.
\$ []	Модификатор, указывающий, что необходимо обновить все элементы, соответствующие условию запроса.
\$pop	Удаляет первый или последний элемент массива.
\$pull	Удаляет все элементы, соответствующие указанному запросу.
\$push	Добавляет элемент в массив.
\$pullAll	Удаляет все элементы массива, соответствующие указанным значениям.

В следующем примере выполняется обновление документа указанного статуса:

```

db.inventory.updateOne(
  { status: "D" },
  {
    $inc: { qty : -10 },
    $set: { "size.uom": "cm", item: "paper_up-
dated" },
    $currentDate: { lastModified: true }
  }
)

```

Первый параметр метода updateOne – это документ, описывающий фильтр. Второй параметр содержит документ, описывающий модификацию данных. В данном случае применяются три оператора модификации.

13.6 Удаление документов

Для удаления документов из коллекции используются методы deleteOne и deleteMany для удаления одного или нескольких документов соответственно (таблица 13.16).

Таблица 13.16. Методы обновления документов

Метод	Описание
db.collection.deleteOne(<filter>, <options>)	Удаление одного документа
db.collection.deleteMany(<filter>, <options>)	Удаление нескольких документов

В следующем примере выполняется удаление всех документов указанного статуса:

```
db.inventory.deleteMany (   
    { status: "A" },   
)
```

13.7 Представления

Как и в реляционных базах данных, **представление** (view) – объект базы данных, содержимое которого определяется запросом к данным. В случае MongoDB, представление определяется конвейером агрегации данных из других коллекций или представлений. Отличие представлений в MongoDB заключается в том, что здесь представления предназначены только для чтения, т.е. нельзя редактировать данные через представления, что допустимо в реляционных базах данных.

Как и в реляционных базах данных преимущества использования представлений включают в себя:

- исключение некоторой информации из возвращаемых приложению документов, например, персональных данных;
- добавление вычисляемых полей в представление;
- создание представления с использованием сложного контейнера агрегации и скрытие деталей реализации от приложения.

Для создания представления можно воспользоваться методами `createCollection` или `createView`, указав исходную коллекцию или представление и конвейер агрегации, который будет применен к исходным данным (таблица 13.17).

Таблица 13.17. Методы создания представлений

Метод	Описание
<pre>db.createCollection("<viewName>", { "viewOn": "<source>", "pipeline": [<pipe- line>], "collation": {<colla- tion>} })</pre>	Создание новой коллекции или представления.
<pre>db.createView("<viewName>", "<source>", [<pipeline>], { "collation": {<colla- tion>} })</pre>	Создание нового представления как результата применения указанного конвейера агрегации к исходной коллекции или представлению.

Рассмотрим пример создания представления. Будем использовать запрос подсчета среднего населения в городах с агрегацией данных по штатам:

```
db.createView(
    "avgPopulation",
    "zips",
    [
        { $group: { _id: { state: "$state", city:
"$city" }, pop: { $sum: "$pop" } } },
        { $group: { _id: "$_id.state", avgCityPop:
{ $avg: "$pop" } } },
    ]
)
```

```
        { $sort: { avgCityPop: -1 } }
    ]
)
```

В примере создается новое представление с именем `avgPopulation`, которое используют коллекцию `zips` в качестве источника данных и конвейер агрегации, состоящий из трех этапов, для обработки данных.

Для удаления представления используется метод `drop`, который может применяться как для удаления коллекции, так и представления:

```
db.collection.drop(<options>)
```

Для изменения представления можно использовать один из двух способов:

- удалить представления и создать его заново;
- использовать команду `collMod` на уровне базы данных.

13.8 Индексирование

13.8.1 Типы индексов

Как и в реляционных базах данных, **индекс** – это объект базы данных специальной структуры, создаваемый с целью повышения производительности поиска данных.

MongoDB определяет индексы на уровне коллекции и поддерживает индексы для любого поля основного или вложенного документов в коллекции. Для индексации данных в MongoDB используется структура данных типа В-дерева, рассмотренная в разделе 8.2.2.

На рисунке 53 показано использование индекса на основе В-дерева, построенного по одному полю `score`. В запросе выполняется фильтрация и сортировка по полю `score`.

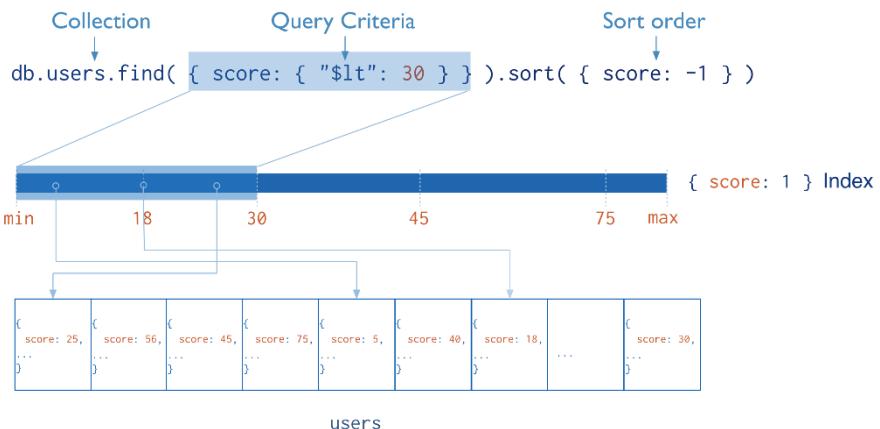


Рисунок 53 – Индекс на основе В-дерева

Типы поддерживаемых индексов в MongoDB похожи на соответствующие индексы в реляционных базах данных:

- индекс по одному полю;
- составной индекс;
- многоключевой (мультиключевой) индекс;
- пространственный индекс;
- текстовый индекс;
- хеш-индекс.

Поддерживается создание определяемых пользователем индексов по возрастанию/убыванию для одного поля документа. По умолчанию, в MongoDB создается уникальный индекс по полю `_id`.

Составные индексы позволяют выполнять индексирование по нескольким полям. Как и в реляционных базах данных, порядок полей, перечисленных в составном индексе, имеет значение. На рисунке 54 проиллюстрирован пример составного индекса, который состоит из идентификатора пользователя `userId` и оценки `score`,

индекс сначала сортируется по идентификатору пользователя, а затем для каждого значения идентификатора пользователя сортируется по оценке.

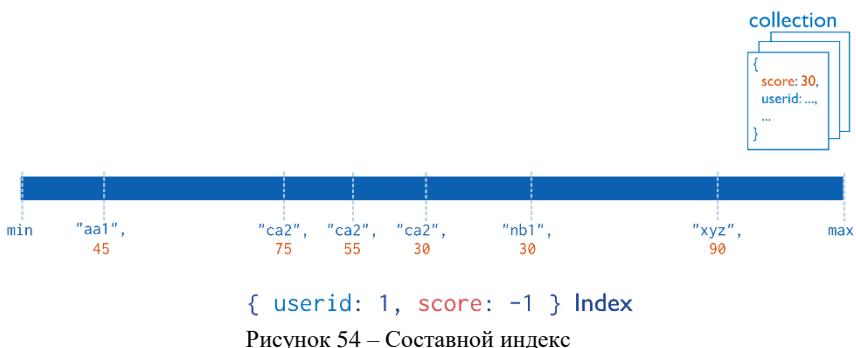


Рисунок 54 – Составной индекс

Многоключевой (мультиключевой) индекс используется для индексации содержимого, хранящегося в массивах. Если индексируется поле, содержащее значение массива, в MongoDB будут созданы отдельные записи индекса для каждого элемента массива. Многоключевые индексы позволяют запросам выбирать документы, содержащие массивы, путем сопоставления элемента или элементов массивов. Этот тип индекса создается автоматически, если индексированное поле содержит значение массива.

Пространственный индекс используется для поддержки эффективных геопространственных запросов. MongoDB предоставляет два специальных индекса: двумерные индексы, которые используют планарную геометрию при возврате результатов, и двумерные индексы, которые используют сферическую геометрию для возврата результатов.

Текстовые индексы используется для устаревшего полнотекстового поиска.

Хеш-индексы используется для поддержки сегментирования на основе хэша.

13.8.2 Создание индекса

Для создания индекса в коллекции используется метод `createIndex`:

```
db.collection.createIndex(keys, options,  
                           commitQuorum)
```

Метод принимает три параметра:

- `keys` – документ, содержащий пары поля и значения, где поле является ключом индекса, а значение описывает тип индекса для этого поля. Для индекса поля с сортировкой поля по возрастанию указывается значение `1`; для индекса по убыванию указывается `-1`;
- `options` – опциональный параметр – документ, содержащий набор параметров, управляющих созданием индекса. Поддерживаются параметры `name` – название индекса, `unique` – для создания уникального индекса и т.д.;
- `commitQuorum` – опциональный параметр – число реплик, которые должны сообщить об успешном создании индекса.

13.9 Транзакции

MongoDB – распределенная СУБД с ACID-совместимыми транзакциями между наборами реплик и/или сегментов.

MongoDB предоставляет два типа API для использования транзакций. Первый – это синтаксис, аналогичный реляционным СУБД (например, `START_TRANSACTION` и `COMMIT_TRANSACTION`), именуемый базовым API, а второй – API обратного вызова, который является рекомендуемым подходом к использованию транзакций. Сравнение этих подходов к выполнению транзакций представлено в таблице 13.18.

Таблица 13.18. API для выполнения транзакций

Базовый API	API обратного вызова
Требует явного вызова для запуска транзакции и ее фиксации.	Запускает транзакцию, выполняет указанные операции и фиксацию (или прерывает работу при возникновении ошибки).
Не включает логику обработки ошибок, вместо этого обеспечивая гибкость, позволяющую включить настраиваемую обработку этих ошибок.	Автоматически включает логику обработки ошибок TransactionError и UnknownTransactionCommitResult.
Требует явного логического сеанса для передачи его в API для конкретной транзакции.	Требует явного логического сеанса для передачи его в API для конкретной транзакции.

Базовый API не предоставляет кода повторного выполнения операции для большинства ошибок и требует от разработчика писать код для операций, функции фиксации транзакции и любой необходимый код повторного выполнения операции и проверки ошибок.

API обратного вызова предоставляет единственную функцию, которая оборачивает большую степень функциональности по сравнению с основным API, включая запуск транзакции, связанной с указанным логическим сеансом, выполнение функции, предоставляемой в качестве функции обратного вызова, и затем фиксацию транзакции (или прерывание при возникновении ошибки). Эта функция также включает в себя код повторного выполнения операции, который обрабатывает ошибки фиксации. API обратного вызова был добавлен в MongoDB версии 4.2, чтобы упростить разработку приложений с транзакциями, а также упростить добавление кода повторного выполнения операции для обработки любых ошибок транзакций.

Оболочка mongo поддерживает получение объекта сессии session, который предоставляет методы startTransaction, commitTransaction и abortTransaction.

Метод startTransaction запускает мультидокументную транзакцию, связанную с сеансом. В любой момент времени для сеанса может быть открыта только одна транзакция.

Метод commitTransaction сохраняет изменения, внесенные операциями в многодокументной транзакции, и завершает транзакцию.

Метод abortTransaction завершает транзакцию с несколькими документами и откатывает любые изменения данных, сделанные операциями внутри транзакции. То есть транзакция завершается без сохранения каких-либо изменений, сделанных операциями в транзакции.

Рассмотрим пример выполнения транзакции:

```
session = db.getMongo().startSession( );
zips_collection =
    session.getDatabase("db_lectures").zips;
inv_collection =
    session.getDatabase("db_lectures").inventory;

session.startTransaction( );

zips_collection.updateOne( { _id: '03101' },
    { $set: { pop: 10000 } } );
inv_collection.insertOne( { item: 'card',
    status: 'C', qty: 10 } );

session.commitTransaction();
session.endSession();
```

В примере сначала запускается новая сессия. Затем, используя объект сессии, в переменную сохраняются ссылки на коллекции, с которыми будет выполняться работа в рамках транзакции. В нашем случае это коллекции `zips` и `inventory` базы данных `db_lectures`. Далее запускается транзакция методом `startTransaction`, выполняются операторы редактирования данных: изменяется один объект в коллекции `zips` и добавляется новый объект в коллекцию `inventory`, транзакция фиксируется и завершается сессия.

Следует отметить, что транзакции в MongoDB работают только если сервер запущен в режиме набора реплик, даже если реплика всего одна. В противном случае операция запуска транзакции завершится с ошибкой.

В заключении следует отметить, что главное достоинство MongoDB – это способность обрабатывать гигантские массивы данных (и огромный поток запросов) за счет репликации и горизонтального масштабирования. Гибкая модель данных упрощает разработку, но может и стать причиной многих проблем, если модель данных уже достаточно зрелая и давно зафиксирована. Стоит учитывать эти моменты при выборе между реляционными и документоориентированными базами данных при решении задач. Реляционные СУБД, в т.ч. поддерживающие хранение JSON данных, во многих случаях остаются предпочтительным выбором.

14 ЗНАКОМСТВО С БОЛЬШИМИ ДАННЫМИ

14.1 Понятие больших данных

С развитием распределенных систем разрабатывались и технологии обработки и хранения данных. Начиная от хранения структурированных данных в реляционных базах, развитие продолжилось в направлении NoSQL баз, заточенных под горизонтальное масштабирование и хранение слабоструктурированных данных. Для хранения структурированных и неструктурированных данных огромных объемов – так называемых больших данных – разрабатываются горизонтально масштабируемые программные инструменты как альтернатива традиционным базам данных.

По результатам исследования компании IDC, в 2020 году в мире было создано 64,2 зеттабайт данных, однако к 2021 году было сохранено менее 2% новых данных, то есть большая часть из них была временно создана или реплицирована для использования, а затем удалена или перезаписана новыми данными (рисунок 55).

По сообщению IDC, объем созданных, потребляемых и передаваемых данных в 2020 году значительно вырос из-за резкого увеличения числа людей, которые на фоне ограничений из-за пандемии COVID-19 были вынуждены работать и учиться дистанционно. В связи с глобальной пандемией также увеличился объем передаваемого мультимедийного контента.

По данным исследований, Интернет вещей является самым быстрорастущим сегментом на рынке данных, не принимая во внимание данные, полученные от систем видеонаблюдения. За ним следуют социальные сети. Считается также, что большие данные могут происходить из внутренней информации предприятий и организаций, из сфер медицины и биоинформатики, из астрономических наблюдений.

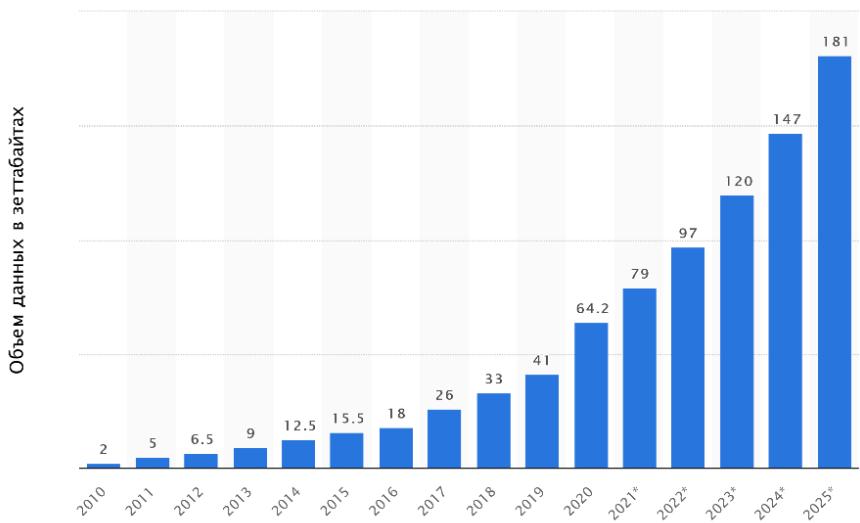


Рисунок 55 – Объемы данных в мире

Использование больших данных нашло применение во многих сферах мировой экономики и управления:

- изучение и анализ больших данных помогает правительствам принимать решения в таких областях, как здравоохранение, занятость населения, экономическое регулирование, борьба с преступностью и обеспечение безопасности, реагирование на чрезвычайные ситуации;
- внедрение инструментов обработки больших данных в промышленности помогает повысить прозрачность промышленных процессов и внедрять «предиктивное производство», позволяющее более точно прогнозировать спрос на продукцию и, соответственно, планировать расходование ресурсов;

- в медицине огромное количество данных, собираемых медицинскими учреждениями и различными электронными приспособлениями, такими, как фитнес-трекеры, позволяет точнее ставить диагнозы, подбирать эффективное лечение и т.д.
- сетевая и электронная торговля использует решения, основанные на анализе больших данных, для персонализации маркетинга и ассортимента товаров;
- промышленные и бытовые приборы, подключенные к интернету вещей, собирают огромное количество данных, на основе анализа которых впоследствии регулируется работа этих приборов.

Также можно упомянуть анализ астрономических данных, решение задач в биоинформатике, спорте, сельском хозяйстве и других сферах жизнедеятельности.

Обычно под понятием **больших данных** (Big Data) обозначают структурированные и неструктурированные данные огромных объёмов и значительного многообразия, эффективно обрабатываемые горизонтально масштабируемыми программными инструментами [16, 17]. Т.е. под термином «большие данные» понимают наборы данных, достаточно большие и сложные для того, чтобы их можно было обработать традиционными средствами работы с данными, например, реляционными СУБД.

Для обработки таких данных была необходима разработка отдельных инструментов, базовым принципом которых является горизонтальная масштабируемость, обеспечивающую обработку данных, распределённых на сотни и тысячи вычислительных узлов, без деградации производительности. Прежде всего эти инструменты включают в себя системы управления базами данных категории NoSQL, алгоритмы MapReduce и реализующими их программными каркасами и библиотеками проекта Hadoop. Но реляционные СУБД продолжают использоваться и для обработки больших данных.

14.2 Характеристики больших данных

В 2001 году было предложено оценивать большие данные с использованием трех характеристик (правило 3V) [18]:

- volume – физический объем данных;
- velocity – скорость прироста данных и скорость быстрой обработки данных с целью получения результатов;
- variety – вариативность, предполагает возможность одновременной обработки различных типов данных. Данные могут быть структурированными, неструктурированными или структурированными частично.

В дальнейшем появились интерпретации с «четырьмя V» (добавлялась veracity – достоверность как самого набора данных, так и результатов его анализа). Далее набор V еще расширялся, добавлялись такие характеристики, как variability – изменчивость форматов, структуры или источников больших данных; value – ценность информации, которую можно получить путем обработки и анализа больших наборов данных, и т.д.

Во всех случаях в этих признаках подчёркивается, что определяющей характеристикой для больших данных является не только их физический объём, но другие категории, существенные для представления о сложности задачи обработки и анализа данных.

Таким образом, технология больших данных на призвана обеспечить:

- 1) хранение и управление огромными объемами данных (сегодня исследователи говорят о необходимости эффективной работы с петабайтами данных);
- 2) организацию неструктурированных и слабоструктурированных данных (такие как тексты, изображения, видео и т.д.);

- 3) получение из больших данных аналитических выводов, которые будут полезны в практической деятельности человека.

Выделяют три базовых принципа работы с большими данными:

- горизонтальная масштабируемость, обеспечивающая обработку данных, предполагает, что данные распределены по узлам таким образом, что их обработка не приводит к снижению производительности системы;
- отказоустойчивость, которая должна позволять свести к минимуму влияние на работу с данными возможные отказы оборудования;
- локальность данных, которая требует, чтобы по возможности данные обрабатывались на том же компьютере, на котором они и хранятся. Иначе временные затраты на передачу данных от места хранения к месту обработки станут слишком большими.

14.3 Инструментарий для обработки больших данных

В настоящее время существует большое количество разных инструментариев для обработки больших данных. На рисунке 56 представлена классификация инструментов по типу решаемых ими задач [16].

Распределенная файловая система похожа на обычную файловую систему, но, в отличие от последней, она работает на нескольких серверах сразу. Распределенные файловые системы обладают рядом преимуществ:

- они способны хранить файлы, размер которых превышает размер диска отдельного компьютера;

- файлы автоматически реплицируются на нескольких серверах для создания избыточности или выполнения параллельных операций;
 - система легко масштабируется.

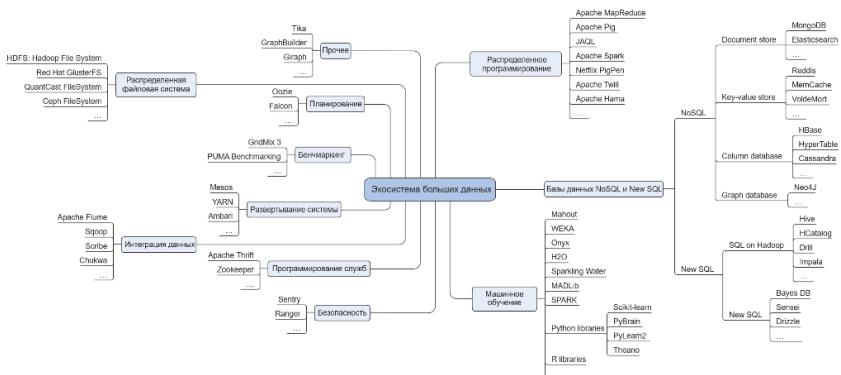


Рисунок 56 – Программное обеспечение для работы с большими данными

В настоящее время одной из наиболее популярных распределенных файловых систем является HDFS – это Hadoop File System. Но существуют и различные альтернативы.

Для использования данных, сохраненных в распределенной файловой системе, используются *инструменты для распределенного программирования*, которые, условно, перемещают код программы к данным, чтобы обеспечить принцип локальной обработки данных. Для обработки данных широко используются инструменты MapReduce и Apache Spark.

Для добавления и перемещения данных из одних источников в другой используются *инструменты интеграции данных*, такие как Apache Flume.

Для анализа данных с использованием методов машинного обучения используются как библиотеки языков Python или R, так и программное обеспечение типа Mahout или Apache Spark.

Для хранения огромных объемов данных предпочтительными являются NoSQL и NewSQL решения. Типы NoSQL баз данных были рассмотрены в главе 12 – это документоориентированные базы данных типа MongoDB, системы хранения типа ключ-значения, такие как Redis, столбцовые базы данных, например, HBase, и графовые базы данных, например, Neo4j.

Инструменты типа SQL в Hadoop позволяют писать пакетные запросы в Hadoop на SQL-подобном языке. Системы типа NewSQL используют интерфейс SQL и реляционные базы данных.

Также существуют инструменты планирования для автоматизации повторяющихся операций и запуска заданий по событиям, например, при появлении нового файла в папке, инструменты развертывания системы, программирования служб и обеспечения безопасности данных.

14.4 Архитектура системы обработки больших данных

14.4.1 Компоненты архитектуры

Рассмотрим основные части архитектуры системы, осуществляющей обработку больших данных. На схеме на рисунке 57 показаны логические компоненты, которые входят в архитектуру. Отдельные решения могут не содержать все компоненты в этой схеме.

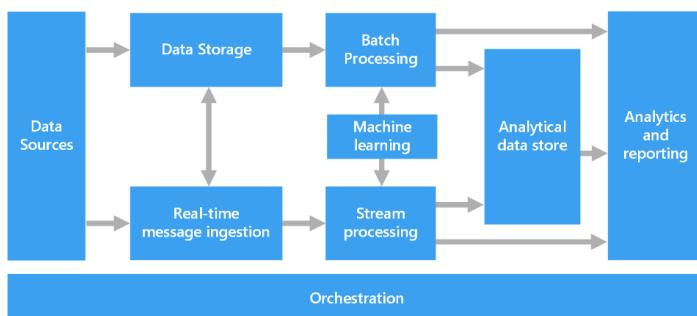


Рисунок 57 – Компоненты архитектуры больших данных

Архитектура содержит следующие компоненты:

- источники данных (data sources). Все решения для обработки больших данных начинаются с одного или нескольких источников данных;
- хранилище данных (data storage). Данные для пакетной обработки обычно хранятся в распределенном хранилище файлов, где могут содержаться значительные объемы больших файлов в различных форматах. Этот тип хранилища часто называют озером данных;
- пакетная обработка (batch processing). Так как наборы данных очень велики, часто в решении обрабатываются длительные пакетные задания. Для них выполняется фильтрация, статистическая обработка и другие процессы. Обычно в эти задания входит чтение исходных файлов, их обработка и запись выходных данных в новые файлы. Для пакетной обработки часто используется подход MapReduce;
- прием сообщений в реальном времени (real-time message ingestion). Если решение содержит источники в режиме реального времени, в архитектуре должен быть предусмотрен способ сбора и сохранения сообщений в режиме реального времени для потоковой обработки данных;
- потоковая обработка (stream processing). Сохранив сообщения, поступающие в режиме реального времени, система выполняет для них фильтрацию, статистическую обработку и другие процессы подготовки данных к анализу;
- хранилище аналитических данных (analytical data store). Во многих решениях для обработки больших данных данные подготавливаются к анализу. Затем обработанные данные структурируются в соответствии с форматом запросов для средств аналитики;

- анализ и создание отчетов (analytics and reporting). Большинство решений по обработке больших данных предназначены для анализа и составления отчетов, что позволяет получить важную информацию;
- оркестрация (orchestration) необходима для управления повторяющимися рабочими процессами, во время которых происходит обработка данных.

14.4.2 Пакетная обработка

Обработка больших данных может выполняться либо в пакетном, либо в потоковом режиме. **Пакетная обработка** – это типичный сценарий работы с большими данными. В этом случае исходные данные загружаются в хранилище данных. Затем данные параллельно обрабатываются локально. В рамках обработки может выполняться несколько итеративных шагов до того, как преобразованные результаты будут загружены в хранилище аналитических данных.

Пакетная обработка может выполняться для очень больших наборов данных и сложных сценариев обработки, которые требуют длительного времени выполнения. Пакетная обработка может выполняться достаточно эффективно если говорить о количестве обрабатываемых записей в единицу времени благодаря современным технологиям формату хранения, способу обработки и т.д. Но результаты обработки доставляются с задержкой.

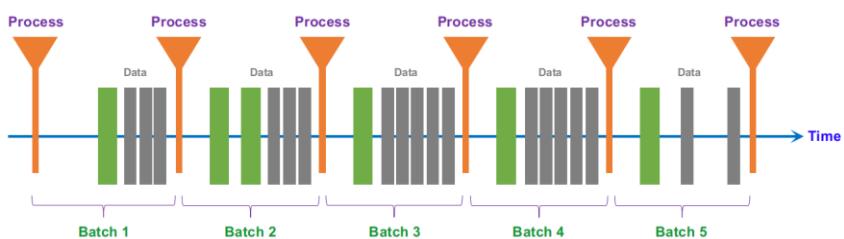


Рисунок 58 – Пакетная обработка данных

На рисунке 58 представлена схематичная схема обработки: данные объединяются в блоки или пакеты, и затем происходит обработка всего пакета данных.

Архитектура системы пакетной обработки имеет следующие логические компоненты [19], показанные на рисунке 59.

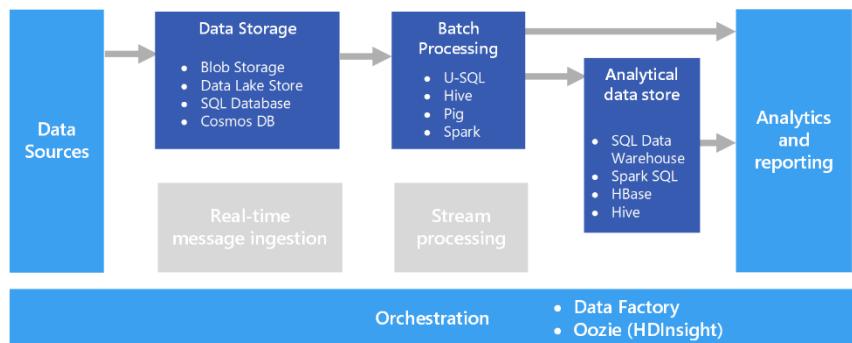


Рисунок 59 – Архитектура системы пакетной обработки данных

В качестве хранилища данных (data storage) обычно выступает распределенное хранилище файлов, которое может служить репозиторием для значительных объемов больших файлов в различных форматах. Также можно использовать различные базы данных. Для пакетной обработки данных (batch processing) может использоваться платформа Spark, которая поддерживает программы пакетной обработки, написанные на разных языках, включая Java, Scala и Python. Spark использует распределенную архитектуру для параллельной обработки данных в нескольких рабочих узлах. Также может использоваться SQL-подобный язык Hive, декларативный язык Pig. В качестве хранилища аналитических данных (analytics data store) могут использоваться NoSQL хранилище типа «семейство столбцов» HBase, система управления базами данных Apache Hive на основе платформы Hadoop и другие.

14.4.3 Потоковая обработка

Потоковая обработка выполняется для потоков данных, получаемых в реальном времени и обрабатываемых с минимальной задержкой для создания отчетов или автоматизированного реагирования в режиме реального времени (или приближенном к реальному времени). Т.е. каждая поступившая порция данных обрабатывается сразу, как показано на рисунке 60.

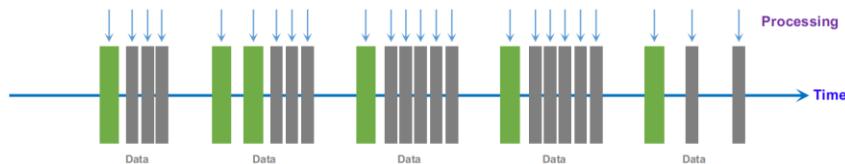


Рисунок 60 – Потоковая обработка данных

Потоковая обработка позволяет обеспечить непрерывную обработку входящего потока данных по мере поступления событий и выдавать результат обработки с минимальным временем задержки в несколько миллисекунд. Но такая обработка не сможет использоваться в сложных сценариях обработки данных или использовать данные из хранилища, т.е. результаты обработки могут быть менее точны или полны, чем при пакетной обработке.

Архитектура обработки в режиме реального времени состоит из следующих логических компонентов: средства сбора и сохранения сообщений в режиме реального времени, средства потоковой обработки, хранилище аналитических данных и средства создания и отчетов (рисунок 61) [20].

Для приема сообщений в реальном времени используются брокеры сообщений, такие, как Apache Kafka или RabbitMQ, которые поддерживают очереди сообщений, потоковую обработку с возможностью масштабирования до нескольких миллионов сообщений в секунду от множества отправителей сообщений, и перенаправление

правление их множеству объектов-получателей. Для потоковой обработки данных могут использоваться системы Apache Storm или Spark Streaming.

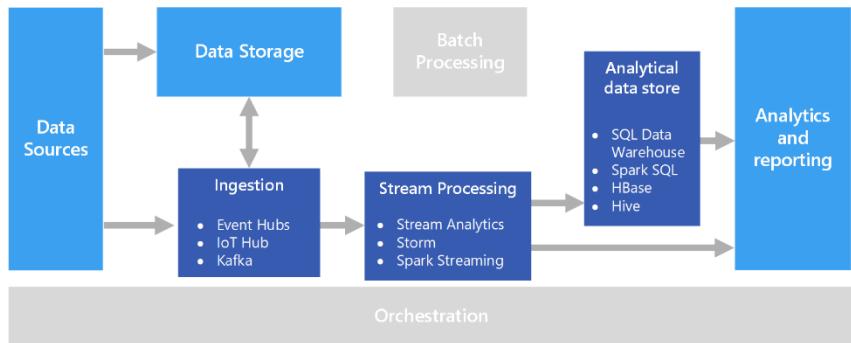


Рисунок 61 – Архитектура системы потоковой обработки данных

14.4.4 Лямбда-архитектура

Часто необходимо совместить оба подхода к обработке данных, т.е. объединить потоковую обработку данных в режиме реального времени с результатами пакетной аналитики. Для этого была предложена лямбда-архитектура, которая является одним из наиболее популярных архитектурных решений для обработки больших данных.

Лямбда-архитектура [2] состоит из трёх взаимодополняющих уровней: уровень пакетной обработки, выполняющий обработку пакетов данных по расписанию, уровень скоростной обработки, выполняющий потоковую обработку данных в реальном времени, и уровень обслуживания, которые объединяют результаты пакетного и потокового представления для предоставления агрегированных данных пользователю по запросу (рисунок 62).

Уровень пакетной обработки представляет собой хранилище «сырых» данных. На этом уровне выполняется обработка по расписанию – через заранее заданные интервалы времени отправляются

запросы к новым данным. Полученные данные добавляются к накопленному ранее архиву, не изменяя его предыдущие копии. Обычно уровень пакетной обработки реализуется на базе Apache Hadoop.

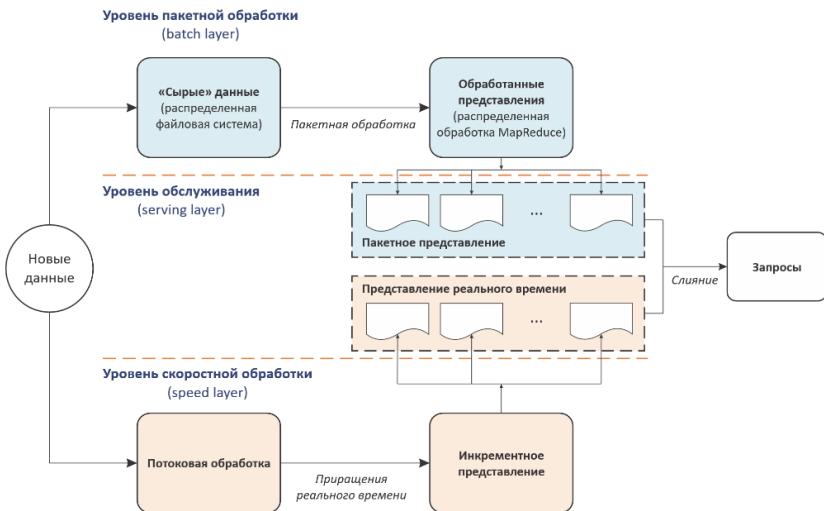


Рисунок 62 – Лямбда-архитектура

Уровень потоковой обработки осуществляет обработку поступающего в реальном времени потока данных. Работа в реальном времени допускает упрощенную обработку данных и использование простых сценариев работы для быстрого анализа данных. В этом случае, можно получить, например, недостаточно точные или неполные данные. Однако эти погрешности с небольшим запозданием компенсируются уровнем пакетной обработки. Уровень скоростной обработки может быть выполнен с использованием инструментов: Apache Spark, Apache Storm и других.

Данные, полученные от уровней пакетной и потоковой обработки, сохраняются на уровне обслуживания. На этом слое выполняется обработка запросов от операторов и возвращение им заранее

подготовленных или подготовленные «на лету» представлений. На стыке сервисного уровня и уровня скоростной обработки может работать как уже рассмотренная ранее документоориентированная СУБД MongoDB, так и ряд других систем, таких как Apache Cassandra или Apache HBase.

14.5 Программное обеспечение для работы с большими данными

14.5.1 Apache Hadoop

Для того чтобы получить возможность хранить большие данные в распределенных системах и при этом обеспечить управляемость всего кластера с данными, необходимо было разработать соответствующее программное обеспечение. Одним из пионеров этого направления стали программисты из организации Apache Software Foundation. Ими в 2005 году был разработан проект под названием Apache Hadoop, который первоначально задумывался как система хранения данных, способная запускать задачи MapReduce. К сегодняшнему дню Hadoop превратился в целый стек компьютерных технологий, способных работать на уровне пакетной обработки в лямбда-архитектуре.

Hadoop пытается достичь следующих целей:

- надежность, которая достигается путем создания нескольких копий данных и повторного применения логики обработки в случае сбоя;
- отказоустойчивость за счет обнаружения сбоев и применения автоматического восстановления;
- масштабируемость, т.к. данные и их обработка распределяются в кластерах (горизонтальное масштабирование);
- портируемость, т.е. возможность установки на различных устройствах и операционных системах.

Базовая инфраструктура состоит из распределенной файловой системы, менеджера ресурсов и системы выполнения распределенных программ. На практике она позволяет работать с распределенной файловой системой почти так же просто, как и с локальной, скрывая реальное расположение данных на разных серверах.

В Apache Hadoop центральное место занимают три компонента:

- распределенная файловая система (Hadoop Distributed File System, HDFS);
- метод распределенных вычислений MapReduce;
- система управления ресурсами кластера YARN, которая осуществляет управление ресурсами и задачами кластера. Для этого создаются контейнеры для приложений, выполняется оценка их потребностей в ресурсах и выделение дополнительных ресурсов при необходимости.

На этой базе сформировалась экосистема приложений, показанная на рисунке 63. В экосистему входят, например, базы данных Hive и HBase, или инфраструктура машинного обучения Mahout. Конечно, в рамках этой инфраструктуры при необходимости могут использоваться и другие компоненты, упомянутые при изучении архитектуры систем обработки больших данных, в т.ч. СУБД MongoDB или Cassandra.

Для реализации параллелизма Hadoop использует технологию MapReduce. Суть парадигмы MapReduce заключается в том, что большая задача разделяется на ряд небольших заданий, каждое из которых может быть выполнено на любом из узлов кластера (подробнее описано в разделе 1.2.7).

Однако алгоритм MapReduce плохо подходит для интерактивного анализа, потому что данные записываются на диск между этапа вычислений. При работе с большими наборами данных запись обходится достаточно дорого.

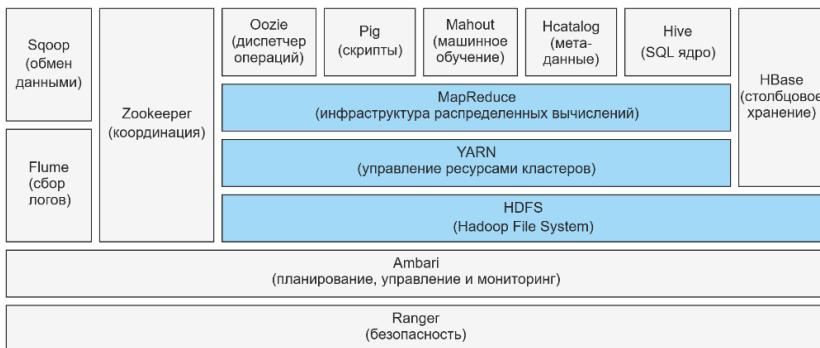


Рисунок 63 – Основные компоненты Apache Hadoop

14.5.2 Apache Spark

Чтобы повысить производительность обработки, был разработан фреймворк Apache Spark [21].

Apache Spark – фреймворк с открытым исходным кодом для реализации распределённой обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop.

В отличие от классического обработчика из ядра Hadoop, реализующего двухуровневую концепцию MapReduce с хранением промежуточных данных на базе дискового хранилища, Spark работает в парадигме резидентных вычислений – обрабатывает данные в оперативной памяти, благодаря чему позволяет получать значительный выигрыш в скорости работы для некоторых классов задач, в частности, возможность многократного доступа к загруженным в память пользовательским данным делает библиотеку привлекательной для алгоритмов машинного обучения.

Проект предоставляет программные интерфейсы для языков Java, Scala, Python, R.

Следует отметить, что Spark не занимается ни хранением файлов в файловой системе, ни управлением ресурсами. Для этого ис-

пользуются системы управления ресурсами YARN или Mesos, поддерживается несколько распределённых систем хранения – HDFS, OpenStack Swift, NoSQL-СУБД Cassandra, Amazon S3. Таким образом, Hadoop и Spark являются взаимодополняющими системами.

Apache Spark состоит из ядра и нескольких расширений, таких как Spark SQL (позволяет выполнять SQL-запросы над данными), Spark Streaming (надстройка для обработки потоковых данных), Spark MLlib (набор библиотек машинного обучения), GraphX (предназначено для распределённой обработки графов).

Стоит отметить, что Spark Streaming, в отличие от, например, Apache Storm, Flink или Samza, не обрабатывает потоки данных целиком. Вместо этого реализуется микропакетный подход (*micro-batch*), когда поток данных разбивается на небольшие пакеты временных интервалов.

Инфраструктура Spark, используемая в сочетании с инфраструктурой Hadoop, показана на рисунке 64.

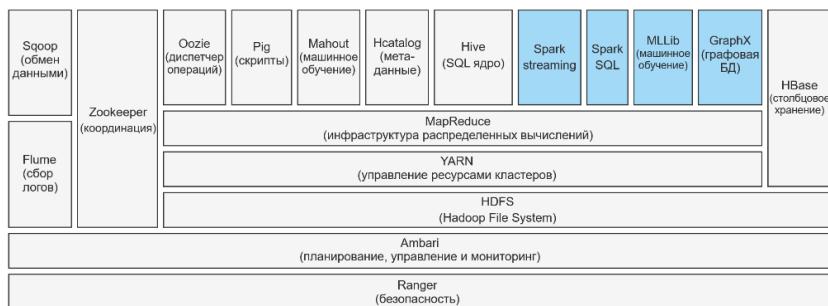


Рисунок 64 – Основные компоненты Apache Spark

14.5.3 Apache Kafka

Apache Kafka – это распределенная система обмена сообщениями с высокой пропускной способностью между компонентами программной системы, работающая по принципу «публикация–подписка» [22].

Можно выделить следующие особенности системы:

- является распределенной, горизонтально-масштабируемой системой, обеспечивающей наращивание пропускной способности как при росте числа и нагрузки со стороны источников, так и количества систем-подписчиков;
- обеспечивается публикация и подписка на потоки записей;
- поддерживается отказоустойчивый способ хранения потоков записей за счет применения техники, сходной с журналами транзакций;
- поток записей обрабатывается по мере появления записей с возможностью временного хранения данных для последующей пакетной обработки.

Использование брокера сообщений упрощает работу, когда в системе несколько источников данных или систем-подписчиков. В этом случае источник данных отправляет сообщений в брокер, которые передаются всем подписчикам, иначе бы источнику пришлось хранить информацию о всех подписчиках и отправлять им сообщения напрямую, решая все те задачи, которые решает брокер, например, надежность доставки сообщений.

Использование инструментов для обработки больших данных позволяет построить горизонтально-масштабируемую систему, способную обрабатывать большие объемы данных как в пакетном, так и в потоковом режиме.