1. **Понятие базы данных, СУБД, виды БД. Определение/Функции СУБД. Архитектура БД. Файловые/клиент-серверные БД. Основные объекты БД. Определение.**

База данных (БД) — совокупность специальным образом организованных данных, хранимых в памяти вычислительной системы и отображающих состояние объектов, и их взаимосвязей в рассматриваемой предметной области.

Система управления базами данных (СУБД) — специальное программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также осуществлять к ней контролируемый доступ.

Виды БД:
1) **Иерархическая** – в таких БД вся информация представлена в виде деревьев, узлы которых называются записями. Например, “Подразделение -> Дочернее предприятие -> Сотрудник №X”. Корневая запись каждого такого дерева обязательно содержит уникальный ключ. Ключи некорневых записей должны быть уникальны только в пределах своего дерева. Каждая запись идентифицируется полным сцепленным ключом (совокупностью ключей всех записей от корня дерева к ней). Каждый экземпляр дерева называется групповым отношением, его корень -владельцем группового отношения, а остальные записи - его членами.
Минусы: нельзя реализовать отношение “многие-ко-многим”, не внося избыточность данных; запросы на поиск информации неоднозначны; медленный поиск. Пример иерархической БД --реестр Windows

2) **Сетевая** - такие БД тоже состоят из групповых отношений, владельца и членов, однако запись может быть членом более одного группового отношения. Главное достоинство: исключение дублирования данных. Главный недостаток: сложность администрирования; медленное выполнение запросов.

3) **Реляционная** - для хранения информации используются таблицы. Подробности см. в вопросах 8 и 9

4) **Постреляционная** - развивает реляционные БД и допускает вложенные таблицы. На практике используется редко

5) **Объектно-ориентированная** - лучший вид БД, хранящий сущности как объекты в стиле ООП. На данный момент распространение этого вида сдерживает отсутствие строгой модели и наличие огромного кол-ва реляционных БД

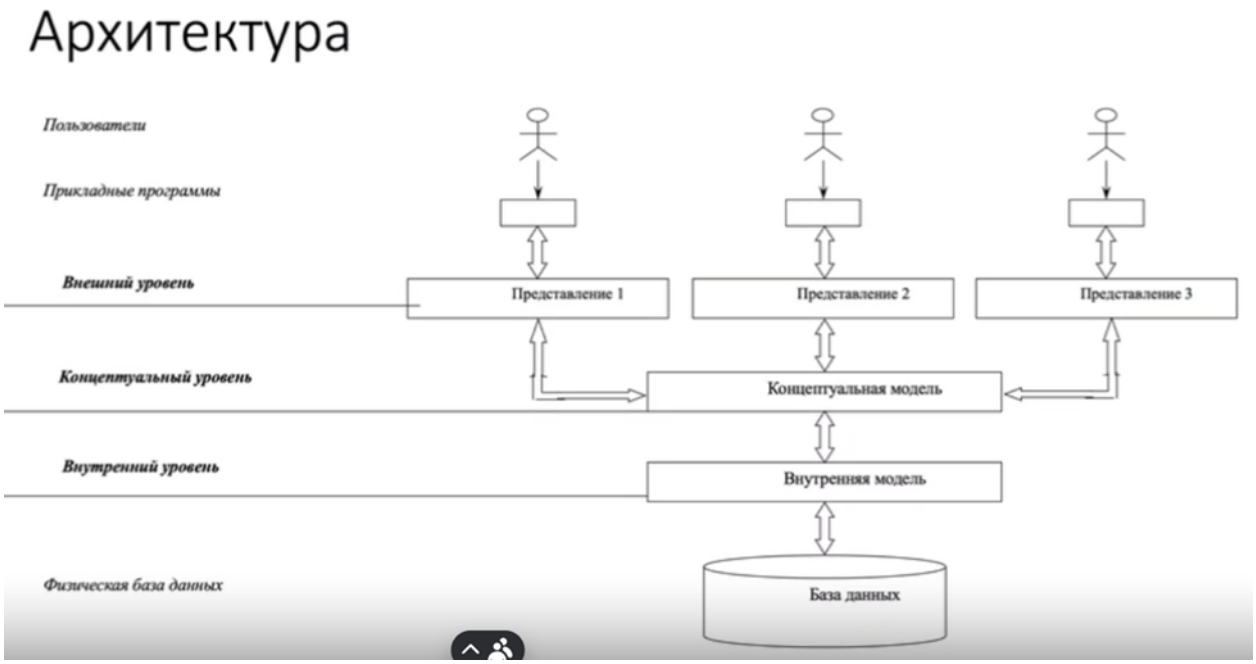
Функции СУБД:

1. **Непосредственное управ данными во внешней памяти.** Предполагает создание необходимых структур во внешней памяти, как для хранения данных непосредственно входящих в БД, так и для служебных целей (например, увеличение скорости доступа к данным).
2. **Управление буферами оперативной памяти**.
3. **Управление транзакциями.** (Транзакция – последовательной операций над БД, рассматриваемая СУБД как единое целое). При использовании транзакций возможны лишь 2 исхода, либо транзакция успешно выполняется и СУБД фиксирует во внешней памяти изменение данных, либо ни одно из производимых действий никак не отображается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. В многопользовательских системах каждая транзакция начинающаяся при целостном состоянии БД оставляет это состояние так же целостным, что делает использование транзакции очень удобным, как единицу активности пользователя по отношению к БД.
4. **Журнализация**. При обеспечении надёжности хранения данных во внешней памяти СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого сбоя. Для восстановления БД необходимо располагать некоторой дополнительной информацией, то есть поддержание надёжности хранения данных в БД требует некоторой избыточности, при чём часть, используемая для восстановления, должна храниться особо надёжно. Наиболее распространённый метод – ведение журнала изменений БД. (Журнал – особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью, в которую поступают записи обо всех изменениях БД)
5. **Поддержка языков базы данных**. Язык определения схемы данных (SDL) и язык манипулирования данными (DML). Первый служит для определения логической структуры базы данных, то есть той

структуры, какой она представляется пользователю. Второй содержит набор операторов манипулирования данными.

6. **Поддержка словаря данных**. Словарь данных сам является БД, он содержит данные о данных (метаданные), то есть содержит определение объектов системы, их свойств и отношения между ними для данной предметной области.

Архитектура БД:



Внешний и концептуальный уровни являются логическими уровнями представления данных, а внутренний – физическим. Цель данной архитектуры – отделение пользовательского представления БД от физического.

Внутренний уровень в архитектуре баз данных представляет собой самый низкий уровень абстракции и отвечает за физическое хранение данных. Он описывает, как данные фактически хранятся на носителях информации, и включает в себя детали, касающиеся структуры файлов, индексов, форматов хранения и других аспектов, связанных с физической реализацией базы данных.

Концептуальный уровень представляет собой абстрактное представление всей базы данных. Он описывает структуру данных, их связи и ограничения, не вдаваясь в детали хранения или реализации.

Внешний уровень представляет собой набор представлений данных, доступных пользователям или приложениям. Это уровень, на котором пользователи взаимодействуют с базой данных.

Файловые базы данных:

- Хранят данные в виде файлов на диске.
- Доступ к данным осуществляется через программы, которые работают с этими файлами.
- Подходят для небольших приложений, но имеют ограничения по многопользовательскому доступу и обеспечению целостности данных.

Файл-серверная. Архитектура систем БД с сетевым доступом предполагает выделение одной из машин сети в качестве центральной (сервер файлов). На такой машине хранится совместно используемая централизованная БД. Все другие машины сети выполняют функции рабочих станций, с помощью которых поддерживается доступ пользовательской системы к централизованной базе данных. Файлы базы данных в соответствии с пользовательскими запросами передаются на рабочие станции, где в основном и производится обработка.

Клиент-серверная. В этой концепции подразумевается, что помимо хранения централизованной базы данных центральная машина (сервер базы данных) должна обеспечивать выполнение основного объёма обработки данных. Запрос на данные, выдаваемый клиентом (рабочей станцией), порождает поиск и извлечение данных на сервере.

Основные объекты БД:

Объекты БД:

БД – контейнер, содержащий таблицы и другие объекты.

Схема – часть БД, в пределах которой все имена объектов уникальны.

Таблица – мультимножество строк, хотя в реляционной теории отношения или математическая модель таблицы определяется как множество кортежей (математическая модель одной строки).

Мультимножество – расширения понятия множества, в котором допускаются повторяющиеся элементы.

Индекс – вспомогательный объект для ускорения поиска данных, однако замедляет операции вставки/обновления/удаления строк.

Представение – именованный запрос на выборку данных, который хранится в БД и выполняется на сервере при любом обращении к нему по имени создавая при этом виртуальную таблицу с отобранными данными. **л**

Хранимые процедуры(функции) – данные объекты БД пишутся на языке процедурного расширения языка SQL, который дополняет его такими управляющими структурами высокоуровневого языка как ветвление и циклы и позволяет реализовать любые алгоритмы обработки данных. Хранимый код постоянно хранится на сервере и выполняется по запросу на его запуск из приложений-клиентов.

Триггер – особый вид хранимой процедуры, который срабатывает автоматически при наступлении определённых событий в БД.

Последовательность – объект для генерации уникальных значений.

Пользователи и роли – пользователи и их права на различные действия в БД. Эти объекты служат для разграничения доступа пользователей к одной БД.

2. **Инфологическое проектирование БД. Диаграмма «Сущность-Связь». Сущность. Атрибут. Виды атрибутов. Ключевые атрибуты. Связь. Показатель кардинальности.**

Инфологическое проектирование базы данных — это процесс, который включает в себя выявление информационных потребностей пользователей и формирование структуры данных, которая будет удовлетворять этим потребностям. Одним из основных инструментов на этом этапе является диаграмма «Сущность-Связь» (ER-диаграмма).

ER-диаграмма визуализирует сущности, их атрибуты и связи между ними. Она помогает понять, как данные будут организованы и как они будут взаимодействовать.

Сущность — это любой конкретный или абстрактный объект в рассматриваемой предметной области, информацию о котором необходимо хранить в базе данных.

Атрибут — это именованная характеристика сущности. Наименование атрибута должно быть уникальным для конкретного типа сущности, но может быть одинаковым для различного типа сущностей. Атрибуты используются для определения того, какая информация должна быть собрана о сущности.

Виды атрибутов:

1. **Простые атрибуты:** Не могут быть разделены на более мелкие компоненты (например, имя, возраст).
2. **Составные атрибуты:** Могут быть разделены на более мелкие компоненты (например, адрес может состоять из улицы, города и почтового индекса).
3. **Однозначные атрибуты**: могут принимать только одно значение
4. **Мультизначные атрибуты**: Могут принимать несколько значений .
5. **Производные атрибуты:** Значения которых могут быть вычислены на основе других атрибутов (например, возраст, который можно вычислить из даты рождения).

Ключевые атрибуты:

- **Ключевой атрибут** — это атрибут (или комбинация атрибутов), который уникально идентифицирует каждую запись в сущности. Например, для сущности «Клиент» ключевым атрибутом может быть номер клиента или адрес электронной почты.
- **Первичный ключ:** Уникальный идентификатор сущности, который не может принимать значение NULL.
- **Внешний ключ:** Атрибут, который используется для установления связи между двумя сущностями, ссылаясь на первичный ключ другой сущности.

Связь — это ассоциирование двух или более сущностей, которое указывает, каким образом связаны сущности. Эта информация необходима для поддержания целостности данных.

Показатель кардинальности

Кардинальность описывает количество экземпляров одной сущности, которые могут быть связаны с экземплярами другой сущности.

Основные типы кардинальности:

1. **Один-к-одному** - Один экземпляр сущности связан с одним экземпляром другой сущности. Пример: один студент имеет один паспорт.
2. **Один-ко-многим** - Один экземпляр сущности связан с несколькими экземплярами другой сущности. Пример: один преподаватель ведет несколько курсов.
3. **Многие-ко-многим** - Несколько экземпляров одной сущности могут быть связаны с несколькими экземплярами другой сущности. Пример: студенты могут быть записаны на несколько курсов, и курсы могут включать несколько студентов.

Типы диаграмм «Сущность-Связь»

- **Концептуальная диаграмма**: Моделирует систему на высоком уровне, без учета технических деталей. Здесь важно понять, какие сущности и связи существуют в системе.
- Л**огическая диаграмма**: Включает более точные детали, такие как атрибуты сущностей и типы связей, но еще не конкретизирует, как эти данные будут храниться в базе.
- **Физическая диаграмма**: Представляет собой схему, которая включает все детали реализации базы данных, такие как типы данных, индексы и связи.

Преимущества диаграммы «Сущность-Связь»

1. **Понимание структуры данных:** Помогает разработчикам, аналитикам и заказчикам визуализировать, как данные связаны между собой.
2. **Упрощение разработки базы данных**: Диаграмма служит основой для дальнейшего проектирования и разработки базы данных.
3. **Упрощение документирования системы**: Служит отличным инструментом для документирования данных и их связей в системе.
4. **Легкость в модификации:** Изменения в требованиях можно быстро отобразить на диаграмме, что помогает в уточнении и доработке модели.

3. Нормализация БД. Преимущества/недостатки. Нормальные формы. (1-5NF + BCNF). Декомпозиция отношений. Декомпозиция без потерь. Функциональные зависимости. Математические свойства ФЗ, теоремы.

Нормализация — процесс преобразования базы данных к виду, отвечающему нормальным формам.

В свою очередь, **нормальная форма** — совокупность требований, которым должно удовлетворять отношение. Нормальная форма предназначена для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц). Избыточность часто является причиной аномалий, возникших при добавлении, редактировании и удалении строк таблицы. Нормализация имеет недостаток - замедление работы СУБД при выполнении запросов на извлечение данных

Денормализация - пренебрежение нормальной формой для улучшения оптимизации работы с БД

Нормальные формы

1. Первая нормальная форма (1NF):
    a. Все атрибуты должны содержать только атомарные (неделимые) значения.
    b. Каждая строка должна быть уникальной.
2. Вторая нормальная форма (2NF):
    a. Должна быть в 1NF.
    b. Все неключевые атрибуты должны зависеть от первичного ключа полностью, а не частично.
3. Третья нормальная форма (3NF):
    a. Должна быть в 2NF.
    b. Все неключевые атрибуты должны быть независимы друг от друга (не должно быть транзитивных зависимостей).
4. Бойс-Кодд нормальная форма (BCNF):
    a. Должна быть в 3NF.
    b. Для каждой функциональной зависимости $X \rightarrow Y$, X должно быть суперключом. (то есть, набором атрибутов, который уникально идентифицирует строки в таблице).
5. Четвертая нормальная форма (4NF):
    a. Должна быть в BCNF.

b. Не должно быть многозначных зависимостей.

6. Пятая нормальная форма (5NF):
    a. Должна быть в 4NF.
    b. Все зависимости должны быть восстановимыми, и каждая зависимость должна быть связана с ключом.

Декомпозиция отношений

Декомпозиция — это процесс разделения отношений (таблиц) на более мелкие, чтобы достичь нормальных форм.

Декомпозиция без потерь

Декомпозиция без потерь означает, что после разбиения таблицы на подтаблицы можно восстановить оригинальные данные без потерь. Это достигается, если:

1. Каждое подмножество данных можно восстановить с помощью соединений (JOIN) без потерь информации.
2. Все функциональные зависимости, существовавшие в оригинальной таблице, должны быть сохранены в декомпозированных таблицах.

Функциональные зависимости (ФЗ) — описывают отношения между атрибутами в реляционных таблицах и помогают определить, как данные могут быть структурированы и организованы.

Функциональная зависимость между двумя наборами атрибутов (A и B) в реляционной таблице обозначается как A → B и читается как "A функционально определяет B". Это означает, что для каждого уникального значения A существует ровно одно значение B.

Математические свойства функциональных зависимостей

1) рефлективность - Если Y является подмножеством X, то X определяет Y

2) дополнение A->B => AC -> BC, где C - любое подмножество атрибутов отношения

3) транзитивность A->B и B->C => A->C

4) самоопределение X->X

5) декомпозиция A->BC => A->B, A->C

6) композиция A->B и C->D => AC->BD

7) Теорема о всеобщей зависимости или теорема всеобщего объединения: Если A->B и C->D => A + (C-B) -> BD

4. **Реляционная модель данных. Атрибуты, домены, кортежи, отношения. Потенциальные ключи. Первичный ключ. Внешний ключ.**

Реляционная модель данных — это способ организации данных в виде таблиц (отношений), где каждая таблица состоит из строк и столбцов. Эта модель обеспечивает структурированный и удобный способ хранения и управления данными.

Реляционная база данных — это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

Ключ или ключевой атрибут — атрибут (столбец) или набор атрибутов, который однозначно идентифицирует сущность/объект/таблицу в базе данных.

Простой ключ – это ключ, состоящий из одного и не более атрибута.

Составной ключ -ключ, состоящий из двух и более атрибутов

1. Атрибуты и домены

Атрибут: **Атрибут** — это колонка в таблице, представляющая собой характеристику или свойство сущности. Например, в таблице "Студенты" атрибутами могут быть СтудентID, Имя, Возраст.

Домены: **Домены** — это набор возможных значений, которые может принимать атрибут. Например, домен атрибута Возраст может быть ограничен целыми числами от 0 до 120, а домен атрибута Имя может включать все строки символов.

2. Кортежи

Кортеж: **Кортеж** — это строка в таблице, представляющая собой конкретный экземпляр данных, который соответствует набору атрибутов. Например, кортеж для таблицы "Студенты" может выглядеть так:

(1, 'Иван', 20)

где 1 — это СтудентID, 'Иван' — это Имя, а 20 — это Возраст.

3. Отношения

Отношение: **Отношение** — это таблица, состоящая из кортежей и атрибутов. Каждое отношение имеет уникальное имя и состоит из множества кортежей, которые имеют одинаковую структуру (то есть одинаковый набор атрибутов). Например, таблица "Студенты" является отношением, состоящим из атрибутов СтудентID, Имя, Возраст и множества кортежей.

4. Потенциальные ключи

Потенциальные ключи: **Потенциальный ключ** — это подмножество атрибутов, которое может быть использовано для уникальной идентификации каждого кортежа в отношении. Например, в таблице "Студенты" потенциальными ключами могут быть СтудентID и комбинация Имя + Возраст, если они уникальны для каждого студента.

5. Первичный ключ

Первичный ключ: **Первичный ключ** — это особый тип потенциального ключа, который используется для уникальной идентификации каждого кортежа в таблице. Он должен быть уникальным и не содержать NULL-значений. Например, в таблице "Студенты" СтудентID может быть выбран в качестве первичного ключа, так как он уникален для каждого студента.

6. Внешний ключ

Внешний ключ: **Внешний ключ** — это атрибут или набор атрибутов в одной таблице, который ссылается на первичный ключ другой таблицы. Он устанавливает связь между двумя таблицами и обеспечивает целостность данных. Например, если у нас есть таблица "Записи о курсах", где СтудентID ссылается на первичный ключ СтудентID в таблице "Студенты", то СтудентID в таблице "Записи о курсах" будет внешним ключом.

Суррогатный ключ — это служебный атрибут, добавленный к уже имеющимся информационным атрибутам отношения. Предназначение суррогатного ключа - служить первичным ключом отношения. Значение этого атрибута генерируется искусственно.

Суперключ — это атрибут или множество атрибутов, которое единственным образом идентифицирует кортеж данного отношения. Он может включать дополнительные атрибуты. Суперключ не обладает свойством неизбыточности.

5. Понятие целостности в реляционной модели.

Целостность в реляционной модели данных относится к набору правил и ограничений, которые обеспечивают корректность, согласованность и надежность данных в базе данных. Понятие целостности охватывает несколько ключевых аспектов, включая:

1) Целостность сущностей
    a) Определение: **Целостность сущностей** гарантирует, что каждое значение первичного ключа в таблице уникально и не содержит NULL-значений. Это позволяет однозначно идентифицировать каждую запись в таблице.
    b) Пример: В таблице "Студенты" атрибут СтудентID должен быть уникальным для каждого студента. Если два студента имеют одинаковый СтудентID, это нарушает целостность сущностей.
2) Целостность ссылок (референциальная целостность)
    a) Определение: **Целостность ссылок** обеспечивает, что все внешние ключи в таблице ссылаются на существующие записи в другой таблице. Это предотвращает наличие "висячих" ссылок (т.е. ссылок на несуществующие записи).
    b) Пример: Если в таблице "Записи о курсах" есть внешний ключ СтудентID, который ссылается на СтудентID в таблице "Студенты", то все значения СтудентID в "Записи о курсах" должны соответствовать существующим СтудентID в таблице "Студенты".
3) Целостность данных
    a) Определение: **Целостность данных** включает в себя ограничения на значения атрибутов, чтобы гарантировать, что данные остаются корректными и логичными. Это может включать ограничения на типы данных, диапазоны значений и уникальность.
    b) Пример: В таблице "Студенты" атрибут Возраст может иметь ограничение, что он должен быть целым числом в диапазоне от 0 до 120. Если попытаться вставить значение -5 или 150, это нарушит целостность данных.
4) Целостность доменов
    a) Определение: **Целостность доменов** гарантирует, что значения атрибутов соответствуют заданным доменам (набору допустимых значений). Это может включать ограничения на тип данных и форматы.
    b) Пример: Если атрибут Имя в таблице "Студенты" должен содержать только строки, то попытка вставить число или специальный символ нарушит целостность доменов.
5) Целостность бизнес-правил
    a) Определение: **Целостность бизнес-правил** включает в себя дополнительные ограничения, которые могут быть

специфичными для конкретного приложения или бизнеса. Это может включать сложные правила, которые определяют, как данные могут быть связаны или изменены.

b) Пример: В учебном заведении может быть правило, что студент не может быть зачислен на курс, если у него нет предварительных условий (например, он должен пройти курс предшествующий).

Определитель NULL - Значение Null обозначает тот факт, что значение не определено. Null не принадлежит никакому типу данных и может присутствовать среди значений любого атрибута, определенного на любом типе данных. Двуместная «арифметическая» операция с Null даёт Null. Операция сравнения с Null даёт UNKNOWN. Два NULLзначения никогда не равны друг другу.

6. Реляционная алгебра и ее операции.

Реляционная алгебра — это формальная система, используемая для манипулирования и извлечения данных из реляционных баз данных. Она предоставляет набор операций, которые могут быть применены к отношениям (таблицам) для получения новых отношений. Основные операции реляционной алгебры включают:

1) Селекция (σ)
   a) Описание: Операция **селекция** (или выборка) используется для извлечения строк из отношения, которые удовлетворяют определенному условию.
   b) Синтаксис: σ(condition)(Relation)
   c) Пример: σ(Возраст > 18)(Студенты) — извлекает все записи студентов старше 18 лет.
2) Проекция (π)
   a) Описание: Операция **проекция** используется для извлечения определенных столбцов из отношения, устраняя дубликаты.
   b) Синтаксис: π(Attribute1, Attribute2, ...)(Relation)
   c) Пример: π(Имя, Возраст)(Студенты) — извлекает имена и возраста всех студентов.
3) Объединение (∪)
   a) Описание: Операция **объединения** используется для объединения двух отношений, которые имеют одинаковую структуру (одинаковое количество и типы атрибутов).
   b) Синтаксис: Relation1 ∪ Relation2

c) Пример: Студенты_1 ∪ Студенты_2 — объединяет записи из двух таблиц студентов.

4) Пересечение (∩)
   a) Описание: Операция **пересечения** используется для извлечения строк, которые присутствуют в обоих отношениях.
   b) Синтаксис: Relation1 ∩ Relation2
   c) Пример: Студенты_1 ∩ Студенты_2 — извлекает студентов, которые есть в обеих таблицах.
5) Разность (−)
   a) Описание: Операция **разности** используется для извлечения строк из одного отношения, которые отсутствуют в другом.
   b) Синтаксис: Relation1 − Relation2
   c) Пример: Студенты_1 − Студенты_2 — извлекает студентов, которые есть только в первой таблице.
6) Декартово произведение (×)
   a) Описание: Операция **декартова произведения** комбинирует каждую строку одного отношения с каждой строкой другого, создавая новое отношение.
   b) Синтаксис: Relation1 × Relation2
   c) Пример: Студенты × Курсы — создает новую таблицу, где каждая запись студента комбинируется с каждой записью курса.
7) Соединение (⋈)
   a) Описание: Операция **соединения объединяет** строки из двух отношений на основе общего атрибута. Существует несколько типов соединений, включая внутреннее, левое, правое и полное соединение.
   b) Синтаксис: Relation1 ⋈ Relation2 ON condition
   c) Пример: Студенты ⋈ Записи_о_курсах ON СтудентID = Студенты.СтудентID — извлекает записи студентов и их курсов.
8) Упорядочение (τ)
   a) Описание: Операция **упорядочения используется** для сортировки строк в отношении по одному или нескольким атрибутам.
   b) Синтаксис: τ(Attribute1, Attribute2, ...)(Relation)
   c) Пример: τ(Возраст)(Студенты) — сортирует студентов по возрасту.

7. Команды DDL для работы с таблицами.

DDL (Data Definition Language) — это подмножество SQL, используемое для определения структуры базы данных, включая создание, изменение и удаление таблиц и других объектов. Основные команды DDL для работы с таблицами включают:

1. **CREATE TABLE**

Описание: Создает новую таблицу в базе данных.

Синтаксис:

```
CREATE TABLE имя_таблицы (
    имя_столбца1 тип_данных [ограничения],
    имя_столбца2 тип_данных [ограничения],
    ...
);
```

Пример:

```
CREATE TABLE Студенты (
    СтудентID INT PRIMARY KEY,
    Имя VARCHAR(100),
    Возраст INT,
    Специальность VARCHAR(100)
);
```

2. **ALTER TABLE**

Описание: Изменяет существующую таблицу, позволяя добавлять, изменять или удалять столбцы и ограничения.

Синтаксис:

Добавление столбца:

```
ALTER TABLE имя_таблицы ADD имя_столбца тип_данных;
```

Изменение столбца:

```
ALTER TABLE имя_таблицы ALTER COLUMN имя_столбца тип_данных;
```

Удаление столбца:

```
ALTER TABLE имя_таблицы DROP COLUMN имя_столбца;
```

Пример:

```
ALTER TABLE Студенты ADD Email VARCHAR(255);
ALTER TABLE Студенты ALTER COLUMN Возраст INT NOT NULL;
ALTER TABLE Студенты DROP COLUMN Специальность;
```

3. **DROP TABLE**

Описание: Удаляет таблицу и все данные, которые в ней содержатся.

Синтаксис:

DROP TABLE имя_таблицы;

Пример:

DROP TABLE Студенты;

4. **TRUNCATE TABLE**

Описание: Удаляет все строки из таблицы, но сохраняет структуру таблицы. Это более быстрый способ очистки таблицы по сравнению с DELETE, поскольку не генерирует журналы для каждой удаленной строки.

Синтаксис:

TRUNCATE TABLE имя_таблицы;

Пример:

TRUNCATE TABLE Студенты;

5. **RENAME TABLE**

Описание: Изменяет имя существующей таблицы.

Синтаксис:

RENAME TABLE старое_имя TO новое_имя;

Пример:

RENAME TABLE Студенты TO Учащиеся;

8. Команды манипулирования данными. SELECT. INSERT/UPDATE/DELETE. Способы соединения таблиц в sql-запросах.

1. **SELECT**

Описание: Используется для извлечения данных из одной или нескольких таблиц.

Синтаксис:

```
SELECT столбец1, столбец2, ...
FROM имя_таблицы
WHERE условие
ORDER BY столбец ASC|DESC;
```

Пример:

```
SELECT Имя, Возраст
FROM Студенты
WHERE Возраст > 18
ORDER BY Имя ASC;
```

2. **INSERT**

Описание: Используется для добавления новых строк в таблицу.

Синтаксис:

```
INSERT INTO имя_таблицы (столбец1, столбец2, ...)
VALUES (значение1, значение2, ...);
```

Пример:

```
INSERT INTO Студенты (Имя, Возраст, Специальность)
VALUES ('Иван', 20, 'Программирование');
```

3. **UPDATE**

Описание: Используется для изменения существующих строк в таблице.

Синтаксис:

```
UPDATE имя_таблицы
SET столбец1 = значение1, столбец2 = значение2, ...
WHERE условие;
```

Пример:

```
UPDATE Студенты
SET Возраст = 21
WHERE Имя = 'Иван';
```

4. **DELETE**

Описание: Используется для удаления строк из таблицы.

Синтаксис:

```
DELETE FROM имя_таблицы
WHERE условие;
```

Пример:

```
DELETE FROM Студенты
WHERE Имя = 'Иван';
```

Способы соединения таблиц в sql-запросах.

1. **INNER JOIN**

Описание: Возвращает только те строки, которые имеют соответствия в обеих таблицах. Если в одной из таблиц нет соответствующей строки, она не будет включена в результат.

Синтаксис:

```
SELECT столбцы
FROM таблица1
INNER JOIN таблица2 ON таблица1.столбец = таблица2.столбец;
```

Пример:

```
SELECT Студенты.Имя, Курсы.Название
FROM Студенты
INNER JOIN Записи_о_курсах ON Студенты.СтудентID =
    Записи_о_курсах.СтудентID;
```

2. **LEFT JOIN (или LEFT OUTER JOIN)**

Описание: Возвращает все строки из левой таблицы и соответствующие строки из правой таблицы. Если в правой таблице нет соответствующей строки, возвращаются NULL для правой таблицы.

Синтаксис:

```
SELECT столбцы
FROM таблица1
LEFT JOIN таблица2 ON таблица1.столбец = таблица2.столбец;
```

Пример:

```
SELECT Студенты.Имя, Курсы.Название
FROM Студенты
LEFT JOIN Записи_о_курсах ON Студенты.СтудентID =
    Записи_о_курсах.СтудентID;
```

3. **RIGHT JOIN (или RIGHT OUTER JOIN)**

Описание: Возвращает все строки из правой таблицы и соответствующие строки из левой таблицы. Если в левой таблице нет соответствующей строки, возвращаются NULL для левой таблицы.

Синтаксис:

```
SELECT столбцы
```

FROM таблица1
RIGHT JOIN таблица2 ON таблица1.столбец = таблица2.столбец;

Пример:
SELECT Студенты.Имя, Курсы.Название
FROM Студенты
RIGHT JOIN Записи_о_курсах ON Студенты.СтудентID =
Записи_о_курсах.СтудентID;

4. **FULL JOIN (или FULL OUTER JOIN)**

Описание: Возвращает все строки из обеих таблиц. Если в одной из таблиц нет соответствующей строки, возвращаются NULL для недостающих значений.

Синтаксис:
SELECT столбцы
FROM таблица1
FULL JOIN таблица2 ON таблица1.столбец = таблица2.столбец;

Пример:
SELECT Студенты.Имя, Курсы.Название
FROM Студенты
FULL JOIN Записи_о_курсах ON Студенты.СтудентID =
Записи_о_курсах.СтудентID;

5. **CROSS JOIN**

Описание: Возвращает декартово произведение двух таблиц, то есть все возможные комбинации строк из обеих таблиц. Обычно используется с ограничением WHERE, чтобы отфильтровать результаты.

Синтаксис:
SELECT столбцы
FROM таблица1
CROSS JOIN таблица2;

Пример:
SELECT Студенты.Имя, Курсы.Название
FROM Студенты
CROSS JOIN Курсы;

6. **SELF JOIN**

Описание: Это соединение таблицы самой с собой. Используется, когда нужно сравнить строки внутри одной таблицы.

Синтаксис:

```
SELECT a.столбец1, b.столбец2
FROM таблица a
INNER JOIN таблица b ON a.столбец = b.столбец;
```

Пример:

```
SELECT a.Имя AS Студент1, b.Имя AS Студент2
FROM Студенты a
INNER JOIN Студенты b ON a.Специальность = b.Специальность
    AND a.СтудентID <> b.Ст
```

9. Вложенные запросы. Кореллированные и некореллированные запросы.

Вложенные подзапросы. Один SELECT-запрос может быть помещён в другой SELECT-запрос. В этом случае первый запрос является вложенным, а второй — главным. Вложенные подзапросы заключаются в круглые скобки.

1. Некорреллированные подзапросы

Описание: **Некорреллированные подзапросы** независимы от внешнего запроса. Они выполняются один раз, и результат используется во внешнем запросе. Эти подзапросы могут возвращать одно или несколько значений.

Синтаксис:

```
SELECT столбцы
FROM таблица
WHERE столбец IN (SELECT столбец FROM другая_таблица
    WHERE условие);
```

Пример:

```
SELECT Имя
FROM Студенты
WHERE СтудентID IN (SELECT СтудентID FROM Записи_о_курсах
    WHERE КурсID = 1);
```

В этом примере подзапрос выбирает всех студентов, записанных на курс с ID = 1, и внешний запрос выбирает имена этих студентов.

2. Корреллированные подзапросы

Описание: **Корреллированные подзапросы** зависят от внешнего запроса. Они выполняются для каждой строки внешнего запроса. Это означает, что подзапрос может ссылаться на столбцы из внешнего запроса, и он будет выполняться многократно — один раз для каждой строки внешнего запроса.

Синтаксис:

```
SELECT столбцы
FROM таблица a
WHERE столбец > (SELECT MAX(другой_столбец) FROM
    другая_таблица b WHERE a.столбец = b.столбец);
```

Пример:

```
SELECT Имя
FROM Студенты s
WHERE EXISTS (SELECT 1 FROM Записи_о_курсах z WHERE
    z.СтудентID = s.СтудентID AND z.КурсID = 1);
```

В этом примере подзапрос проверяет, существует ли запись о курсе с ID = 1 для каждого студента. Если такая запись существует, имя студента будет включено в результирующий набор.

Основные отличия

Выполнение: Некорреллированные подзапросы выполняются один раз, тогда как корреллированные подзапросы выполняются для каждой строки внешнего запроса.

Зависимость: Некорреллированные подзапросы не зависят от внешнего запроса, в то время как корреллированные подзапросы ссылаются на столбцы внешнего запроса.

10.Представления (View). Создание. Удаление. Обновление.

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

Создание представлений

Для создания представления используется оператор CREATE VIEW, имеющий следующий синтаксис:

```
CREATE [OR REPLACE]

[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

VIEW view_name [(column_list)]

AS select_statement

[WITH [CASCADED | LOCAL] CHECK OPTION]
```

view_name — имя создаваемого представления. select_statement — оператор SELECT, выбирающий данные из таблиц и/или других представлений, которые будут содержаться в представлении

Для **удаления** представления используется оператор DROP VIEW. Его синтаксис в PostgreSQL: DROP VIEW [IF EXISTS] view_name [, ...] [CASCADE | RESTRICT]

Для **обновления** представления используется команда REPLACE VIEW. 3

Пример обновления возраста для клиента Ramesh: UPDATE CUSTOMERS_VIEW SET AGE = 35 WHERE name = 'Ramesh';

(хз не нашёл норм инфу)

11.Хранимые процедуры и функции. Триггеры.

Хранимые процедуры представляют собой набор команд SQL, которые могут компилироваться и храниться на сервере. Таким образом, вместо того, чтобы хранить часто используемый запрос, клиенты могут ссылаться на соответствующую хранимую процедуру. Это обеспечивает лучшую производительность, поскольку данный запрос должен анализироваться только однажды и уменьшается трафик между сервером и клиентом. Концептуальный уровень можно также повысить за счет создания на сервере библиотеки функций.

CREATE PROCEDURE – создать хранимую процедуру.

Функция — это набор инструкций SQL, которые принимают только входные параметры и выдают выходные данные в виде одного значения или табличной формы. К функциям можно обращаться из триггеров, хранимых процедур и из других программных компонентов.

CREATE FUNCTION – создать хранимую функцию.

Триггер представляет собой хранимую процедуру, которая активизируется при наступлении определенного события. Например, можно задать хранимую процедуру, которая срабатывает каждый раз при удалении записи из транзакционной таблицы - таким образом, обеспечивается автоматическое удаление соответствующего заказчика из таблицы заказчиков, когда все его транзакции удаляются.

Синтаксис создания триггера

CREATE
[DEFINER = { имя_ пользователя | CURRENT_USER }]
TRIGGER имя_триггера время_триггера событие_срабатывания_триггера
ON имя_таблицы FOR EACH ROW
выражение_выполняемое_при_срабатывании_триггера

Виды триггеров:пре

Виды триггеров:

- **BEFORE**: выполняется перед операцией (например, перед вставкой или обновлением данных).
- **AFTER**: выполняется после завершения операции.
- **INSTEAD OF**: используется для замены стандартных операций (например, замена `INSERT` на `UPDATE`).

Когда что использовать:

- **Хранимые процедуры** — для выполнения комплексных операций, таких как обновления или вычисления, которые могут быть повторно использованы.
- **Функции** — для выполнения вычислений или преобразования данных, когда результат должен быть использован в SQL-запросах.
- **Триггеры** — для автоматического выполнения действий на основе изменений данных, например, поддержания целостности данных или аудита.

12.Разграничение доступа. Привилегии и роли.

Разграничение доступа в базе данных — это процесс управления правами пользователей и групп пользователей, что позволяет контролировать, кто и какие операции может выполнять с данными в базе.

Для авторизации в приложениях с разным доступом рекомендуется использовать одного суперпользователя в БД, но это может привести к атакам и ошибкам. PostgreSQL позволяет определить правила доступа для всех служб, ограничивая привилегии пользователей и повышая безопасность.

Привилегии — это права, которые предоставляются пользователям или ролям для выполнения определенных операций на объектах базы данных.

- **Глобальными**: предоставляются для всей базы данных или сервера.
- **Локальными**: применяются только к определенному объекту (например, к таблице или представлению).

Назначение чтения: GRANT SELECT ON сотрудники TO user_name;
Отзыв - REVOKE SELECT ON сотрудники FROM user_name;

Роли — это совокупность привилегий, которые могут быть назначены пользователю или группе пользователей. Роли позволяют упростить управление доступом, так как вместо назначения привилегий каждому

пользователю отдельно, можно создать роль и назначить её группе пользователей.

Создание роли - CREATE ROLE имя_роли;
Назначение роли для менеджера - CREATE ROLE Менеджер;
Назначение привилегий роли - GRANT SELECT, INSERT, UPDATE ON сотрудники TO Менеджер;
Назначение роли пользователю - GRANT Менеджер TO user_name;
Вложенные роли: роль Администратор может включать роль Менеджер, что позволит администраторам иметь доступ к привилегиям, присваиваемым менеджерам.
CREATE ROLE Администратор;
GRANT Менеджер TO Администратор;

Типы ролей:

3. Типы ролей

1. **Глобальные роли** — применяются ко всем базам данных в сервере (например, роль администратора).
2. **Локальные роли** — действуют только в пределах одной базы данных.

Роли могут быть как **фиксированными**, которые предоставляются СУБД по умолчанию, так и **пользовательскими**, которые создаются администратором для конкретных нужд.

Пример фиксированных ролей:

- **DBA (Database Administrator)** — роль администратора базы данных с полными привилегиями.
- **PUBLIC** — роль, которая по умолчанию присваивается всем пользователям. Используется для предоставления общих привилегий.

13. Транзакции. Уровни изоляции. Феномены, возникающие при работе с данными. Свойства транзакций. Журнал транзакций. Механизм блокировок.

Транзакция — неделимая последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), приводящая к одному из двух возможных результатов: либо последовательность выполняется, если все операторы правильные, либо вся транзакция откатывается, если хотя бы один оператор не может быть успешно выполнен.

Уровни изоляции транзакций определяют степень защиты от несогласованностей данных, возникающих при параллельном выполнении транзакций. Стандарт SQL-92 определяе**т четыре уровня изоляции**:

1. **Read uncommitted** — чтение незафиксированных данных. На этом уровне разрешено «грязное» чтение, поэтому одна транзакция может видеть ещё не зафиксированные изменения, совершённые другими транзакциями.
2. **Read committed** — чтение фиксированных данных. Позволяет транзакции считывать данные, считанные до этого, но не изменённые другой транзакцией, не ожидая завершения выполнения этой другой транзакции.
3. **Repeatable read** — повторяемость чтения. Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. При повторном чтении они будут находиться в первоначальном состоянии.
4. **Serializable** — упорядочиваемость. Самый высокий уровень, при котором транзакции полностью изолированы друг от друга.

Феномены, возникающие при работе с данными:

«Грязное» чтение. Транзакция читает данные, записанные параллельной незавершённой транзакцией.

Неповторяемое чтение. Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

Фантомное чтение. Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

Аномалия сериализации. Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Свойства транзакций (ACID):

1) Atomicity (атомарность). Транзакция неделима. Результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии БД либо не должно быть отражено действие ни одной операции.

2) Consistency (согласованность). Транзакция не нарушает логическую целостность данных, переводит БД из одного согласованного состояния в другое.
3) Isolation (изолированность). Транзакция изолирована, её результаты самодостаточны.
4) Durability (устойчивость, долговечность). Транзакция устойчива (долговечна), её действие постоянно даже при сбое системы.

Журнал транзакций обеспечивает журнализацию всех изменений в базе данных для атомарности и долговечности. В простейшем случае в нём записываются идентификатор транзакции, объект, подвергшийся изменению, предыдущее состояние объекта и его новое состояние.

Блокировки в СУБД — это механизм, который предотвращает одновременное изменение одних и тех же данных несколькими пользователями или процессами, что могло бы привести к ошибкам или несогласованности данных. Когда один процесс изменяет данные, другие процессы могут быть "заблокированы" от доступа к этим данным до тех пор, пока процесс, который получил блокировку, не завершит свою работу.

Механизм блокировок предполагает, что когда транзакция изменяет часть данных, она сохраняет определённые блокировки, защищающие изменение до конца транзакции. Блокировки имеют разные режимы, такие как общие (S) или эксклюзивные (X). Режим блокировки определяет уровень подчинения данных транзакции.

Типы блокировок

1. Блокировка на уровне строк: Блокировка ограничивается одной строкой в таблице, позволяя другим пользователям работать с остальными строками той же таблицы.
2. Блокировка на уровне таблиц: Блокировка целой таблицы, что препятствует другим пользователям выполнять любые операции с этой таблицей, пока текущий процесс не завершит свои изменения.
3. Блокировка на уровне страниц: СУБД делит данные на страницы, и блокировка может быть наложена на страницу данных, а не на всю таблицу.
4. Эксклюзивная блокировка: когда процесс получает эксклюзивный доступ к ресурсу, другие процессы не могут читать или изменять этот ресурс.

5. Совместная блокировка: Несколько процессов могут читать данные, но никто не может их изменить, пока не освободится блокировка.

CREATE ALTER DROP TABLE – накладывают полную блокировку на уровне таблицы
INSERT DELETE UPDATE – накладывает эксклюзивную блокировку только на ту строку, которую она обрабатывает в данный момент. Одновременно накладывается разделяемая блокировка на всю таблицу

Предмет базы данных
В контексте базы данных термин "предмет" может означать объект или сущность, к которой применяется блокировка. Это может быть:
1. **Таблица** — набор данных, организованный в виде строк и столбцов.
2. **Запись** (или строка) — единичный элемент данных в таблице.
3. **Страница данных** — минимальная единица хранения данных в СУБД.
4. **Индекс** — структура данных, помогающая ускорить поиск по базе данных.

14. Индексы. Использование хэш-таблиц, битовых карт и индексных таблиц при работе с индексами. Использование B-деревьев при хранении индексов. Операции добавления/изменения/удаления.

Индексы бывают двух видов: простые и составные. Составной индекс включает более чем один столбец. Таким образом, можно совместно проиндексировать 2 или более столбца, каждый из которых обладает низкой селективностью, а пара их значений – высокой.

Индексы — специальные структуры в базе данных, которые помогают оптимизировать поиск информации. Они работают аналогично каталогу в библиотеке и позволяют быстро находить необходимые данные без перебора всех записей. Применение индексов сокращает время поиска и уменьшает алгоритмическую сложность этого процесса.

Как правило, в индексах хранятся значения индексируемых столбцов таблицы и физические адреса строк для каждого из хранимых значений столбцов

Хеш-индексы строятся на основе **хеш-таблицы** и полезны для точного поиска с указанием всех столбцов индекса. Для каждой строки подсистема

хранения вычисляет хешкод индексированных столбцов — сравнительно короткое значение, которое, скорее всего, будет различно для строк с разными значениями ключей. В индексе хранятся хеш-коды и указатели на соответствующие строки.

Битовые индексы используют **битовые карты** для указания значения индексированного столбца. Это идеальный индекс для столбца с низкой кардинальностью (число уникальных записей в таблице мало) при большом размере таблицы. Эти индексы обычно не годятся для таблиц с интенсивным обновлением, но хорошо подходят для приложений хранилищ данных

Использование B-деревьев при хранении индексов:
B-дерево — это структура данных самобалансирующегося дерева, которая часто используется в качестве алгоритма индексирования в базах данных. Каждый узел дерева состоит из набора ключей и указателей на дочерние узлы; хранение данных осуществляется в иерархической структуре. Деревья B-узлов упорядочены таким образом, что позволяют быстро выполнять поиск, вставку и удаление данных.

Структура внутренней страницы:
N1 ключ(1) N2 ключ(2) N3 ключ(3) … Nn ключ(n) N(n+1) ключ(n+1)
Листовая страница:
ключ(1) сп(1) ключ(2) сп(2) … ключ(t) сп(t), где сп – упорядоченный список идентификаторов кортежей, включающих значения ключа.
Листовые страницы связаны одно- или двунаправленным списком.

Поиск в B-Дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. Поскольку деревья сильноветвящиеся и сбалансированные, то для выполнения поиска по любому значения ключа потребуется одно и тоже, и, как правило, небольшое число обменов с внешней памятью.

Операции добавления, изменения и удаления:
Вставка: при добавлении нового узла в B-дерево алгоритм сначала подыскивает подходящий узел-лист, в который нужно вставить ключ. B-дерево разделяет заполненный узел-лист на два новых узла и перемещает медианный ключ в родительский узел.

Удаление: когда ключ удаляется из B-дерева, алгоритм ищет узел, который изначально хранил ключ. Если узел-лист хранил ключ, то ключ извлекается и узел может нуждаться в перебалансировке.

Изменение: при изменении значений, по которому построен индекс, его нужно обновлять.

(тоже хз чё ещё тут написать можно)

15. Планировщик. Методы доступа. Способы соединения наборов строк. Планы запросов. Способы управления планировщиком.

Планировщик запросов — это компонент системы управления базами данных (СУБД), который отвечает за выбор наилучшего плана выполнения SQL-запроса.
Когда пользователь отправляет запрос в СУБД, планировщик анализирует запрос, генерирует несколько возможных вариантов его выполнения и выбирает наиболее эффективный в зависимости от множества факторов, таких как структура данных, наличие индексов и статистика.

Основные этапы работы планировщика:
1. Разбор запроса (парсинг) — Сначала SQL-запрос анализируется и разбивается на элементы: таблицы, условия, операции.
2. Определение возможных стратегий выполнения — Планировщик генерирует несколько вариантов выполнения запроса с разными методами доступа и соединения таблиц.
3. Оценка стоимости выполнения — Каждый план оценивается по стоимости выполнения. Стоимость может включать время выполнения, использование ресурсов (например, CPU и памяти) и другие факторы.
4. Выбор оптимального плана — Планировщик выбирает наилучший план выполнения запроса на основе оценки стоимости.

Для чего нужен планировщик?

1. Оптимизация выполнения запросов: Планировщик помогает сократить время выполнения запросов, минимизируя ресурсы, необходимые для поиска данных.
2. Эффективное использование индексов: Планировщик анализирует, какие индексы могут быть использованы для ускорения выполнения запроса.

3. Обработка сложных запросов: При наличии нескольких таблиц или подзапросов планировщик выбирает оптимальную последовательность соединений и операций.
4. Автоматическое управление производительностью: Планировщик автоматически выбирает наилучший вариант выполнения без вмешательств пользователя.

Методы доступа - это способы, с помощью которых СУБД получает данные из таблиц. Эти методы определяют, как будет происходить поиск строк в таблице и какие операции будут использованы для извлечения данных.

1. **Seq Scan - последовательный просмотр**. Применяется при небольшом размере таблицы, низкой селективности столбца или при невозможности использования других методов. Осуществляется полностью без использования индексов.
2. **Index Scan - просмотр по индексу.** Использует обращение и к индексам, и к непосредственно к таблице. Используется при высокой селективности столбца
3. **Index Only Scan - просмотр исключительно по индексу**. Использует только обращение к индексам. Возможен при наличии карты видимости таблицы, в которой отмечены те страницы таблицы, которые содержат строки, видные всем транзакциям. Обращение к самой таблице не нужно, если строка видна из индекса. Метод эффективен, если выбираемые данные редко изменяются, и возможен только если все столбцы таблицы индексируемые
4. **Bitmap Heap Scan - просмотр на основе битовой карты**. Осуществляет поиск по индексу как во 2-ом методе, но после формирует битовую карту, где указано, в каких страницах таблицы содержатся выбираемые строки. При таком подходе из каждой строки за один раз выбирают все строки, которые нужно из нее выбрать.

Основные методы доступа:

1. Полное сканирование таблицы (Full Table Scan)

   В этом случае СУБД просматривает все строки таблицы, чтобы найти те, которые соответствуют условиям запроса.

   Это наименее эффективный метод, так как требует времени на обработку всей таблицы.

   Применяется, когда в таблице нет индексов или запрос охватывает все строки таблицы.
2. Использование индексов

- Индексный поиск: Индексы позволяют ускорить поиск данных, так как они хранят ссылки на строки таблицы в упорядоченном виде.
- Планировщик может выбрать индекс для быстрого поиска по столбцам, которые участвуют в условиях запроса.
- B-tree индексы — наиболее распространенные для поиска по диапазонам значений.
- Хеш-индексы — эффективны для точных поисков.
- Гистограммы и битовые индексы — используют для более сложных случаев.

3. Хеширование (Hash Join)

   Метод хеширования используется для соединений таблиц. Один из наборов данных хешируется, а затем этот хеш используется для поиска соответствующих строк в другой таблице.

   Этот метод часто применяется для соединений с большими объемами данных.

4. Бинарный поиск

   Применяется в случае, когда данные в таблице или индексе отсортированы.

   Бинарный поиск позволяет значительно ускорить поиск по отсортированным данным.

5. Кэширование

   СУБД может кэшировать результаты часто выполняемых запросов или промежуточных результатов, чтобы избежать повторного вычисления и ускорить выполнение аналогичных запросов в будущем.

6. Итеративный метод

   В случае сложных запросов планировщик может выбирать итеративные методы доступа для улучшения производительности при многократном обращении к данным.

   - Полное сканирование - применяется, когда таблица маленькая или когда нет возможности использовать индекс.
   - Индексы - используются, когда необходимо быстро искать данные по конкретным столбцам, особенно при поиске по равенству или диапазону.
   - Хэширование - эффективен для соединений больших таблиц.
   - Бинарный поиск - применяется, когда данные отсортированы.

Способы соединения наборов строк:

1. **Nested Loop** - вложенный цикл. Обычно используется при декартовом произведении, когда в секции FROM перечисляется несколько таблиц (например, FROM a, b). При этом вторая соединяемая таблица может быть помечена дополнительным узлом с ключевым словом Materialize. Метод эффективен для небольших выборок, допускает все виды условий, не требует подготовки. Например:

```
QUERY PLAN
Nested Loop  (cost=0.00..10050.83 rows=800700 width=156)
  ->  Seq Scan on sale  (cost=0.00..25.70 rows=1570 width=24)
  ->  Materialize  (cost=0.00..17.65 rows=510 width=132)
        ->  Seq Scan on creative_project  (cost=0.00..15.10 rows=510 width=132)
```

2. **Hash <X> Join** - хеширование. Часто используется при соединении Join на условиях равенства. При этом вместо <X> указывается вид Join-соединения (Right, Left, Full). При этом вторая соединяемая таблица может быть помечена дополнительным узлом с ключевым словом Hash (пример смотри в вопросе 41). В данном методе первая таблица становится хеш-таблицей, в которой ключом является столбец, по которому выполняется соединение, а строк в 1-ой таблице обычно меньше, чем во 2-ой. Метод эффективен для больших выборок, но допускает только условия равенства
3. **Merge <X> Join** - соединение слиянием. Часто используется при большом размере таблиц, соединяемых через Join. Вместо <X> указывается вид Join-соединения. Наборы строк предварительно сортируются по соединяемым столбцам, а затем их строки параллельно читаются и сравниваются. Метод является самым эффективным при больших выборках

Селективность столбца — это процент строк, имеющих важность для запроса выборки Select. Селективность высокая, если в столбце мало одинаковых значений, и низкая, если много.

План запроса - это последовательность операций, которые СУБД использует для выполнения SQL-запроса. План запроса определяет, в каком порядке и каким способом будут обрабатываться таблицы и операции над ними.

Основные компоненты и этапы создания плана запроса:

1. Парсинг запроса:

При получении SQL-запроса СУБД сначала выполняет его разбор (парсинг), превращая запрос в абстрактное синтаксическое дерево (AST).

Этот этап включает определение структуры запроса и его составных частей (например, таблицы, операторы, условия).

2. Реализация запроса:

После разбора запроса планировщик решает, как именно выполнить операции. Для этого он:

1) Оценить доступные методы доступа (например, полный скан таблицы или использование индекса)

2) Выберет способ соединения таблиц (например, внутреннее соединение, левое соединение и т. д.)

3) Учитывает возможные подзапросы, их объединение и порядок выполнения.

3. Оценка стоимости выполнения:

Каждый возможный план оценивается по стоимости, которая может включать время выполнения запроса, использование ресурсов (память, процессорное время) и другие параметры.

Планировщик использует статистику, такую как количество строк в таблице, наличие индексов и так далее, для оценки стоимости.

4. Выбор оптимального плана:

Планировщик выбирает план с наименьшей стоимостью, то есть наибольшей эффективностью выполнения запроса. Это может включать выбор индексов, способ соединения таблиц и оптимизацию порядка операций.

5. Исполнение плана запроса:

После выбора плана выполнения запрос начинается. Если план выполнен корректно, то возвращается результат.

Типы планов запросов:

1. План с полным сканированием:

СУБД использует полный скан таблицы, чтобы найти нужные строки. Это дешевле, когда таблица маленькая или запрос охватывает все строки.

2. План с использованием индекса:

Когда для таблицы есть индекс, планировщик может использовать индекс для быстрого поиска строк, что значительно ускоряет выполнение запроса.

3. План с хешированием:

Для соединений таблиц, когда одна из таблиц хешируется для быстрого поиска соответствий в другой таблице.

4. План с объединением сортированных строк:

   При соединении отсортированных таблиц используется метод слияния, который более эффективен, чем хеширование или сканирование.

5. План с операциями с подзапросами:

   Если запрос включает подзапросы, планировщик может включить их в основной запрос или выполнить отдельно, в зависимости от того, что более эффективно.

Как выбирается план запроса:

- Планы выбираются на основе статистики, оценок стоимости и доступных методов доступа.
- Планировщик может использовать алгоритмы, такие как генетический или вероятностный поиск, для выбора наиболее эффективного плана, если количество возможных вариантов велико. Использование индексов в плане запроса:
- Индексы ускоряют выполнение запросов, поэтому планировщик может предпочесть использовать индексный поиск, если он существует и подходит для условий запроса.

Способы управления планировщиком включают:

1. изменение исходного кода запроса;
2. обновление статистики, на основе которой планировщик строит планы;
3. денормализацию: создание временных таблиц или индексов;
4. изменение параметров планировщика, управляющих выбором порядка соединения наборов строк;
5. изменение параметров планировщика, управляющих выбором метода доступа к данным;
6. изменение параметров планировщика, управляющих способом соединения наборов строк;
7. изменение параметров планировщика, управляющих использованием ряда операций: агрегирование на основе хеширования, материализация временных наборов строк, выполнение явной сортировки при наличии других возможностей.

Способы управления планировщиком с помощью явных предложений JOIN:

Планировщик запросов в СУБД может соединять таблицы в любом порядке, что влияет на производительность запроса. Использование явных предложений JOIN помогает управлять этим порядком соединения для оптимизации выполнения запроса. Когда число таблиц становится слишком большим, планировщик переходит к генетическому вероятностному поиску для выбора плана, что быстрее, но не гарантирует оптимальности.

Ограничения при внешних соединениях (LEFT JOIN):
При использовании внешних соединений планировщик имеет меньше гибкости в выборе порядка соединений, так как результат для каждой строки из первой таблицы должен быть найден, даже если для неё нет соответствия во второй таблице.

Явные предложения JOIN:
Использование явных предложений JOIN (например, INNER JOIN, CROSS JOIN) помогает ограничить порядок соединений, улучшая производительность при пранировании.

Параметр join_collapse_limit:
Этот параметр управляет тем, как планировщик обрабатывает порядок соединений. Установив его значение 1, можно заставить планировщик соблюдать порядок соединений, что ускоряет планирование при определённых условиях.

Подзапросы в родительском запросе:
Включение подзапросов в основной запрос может улучшить план, так как это позволяет избежать избыточных операций. Однако это также увеличивает сложность планирования, так как количество вариантов соединений растёт.

Параметры from_collapse_limit и join_collapse_limit:
Эти параметры управляют “сносом” подзапросов и соединений в основной запрос, помогая сбалансировать время планирования и выполнения запроса.

План SQL-запроса можно получить с помощью команды EXPLAIN, предшествующей непосредственно запросу (Select). Стоит помнить, что команда EXPLAIN может быть применена к любому оператору, который имеет план выполнения (Select, Insert, Delete, Update, Values, Execute, Declare, Create Table As)
EXPLANE - просмотр сборки SQL запроса

Структура плана SQL-запроса представляет собой дерево, состоящее из узлов плана. Самые нижние узлы отвечают за просмотр и выдачу строк таблицы, осуществляемые с помощью определенного метода доступа (Seq, Index, Index Only, Bitmap Heap. После обязательно идет слово ключевое имя Scan и имя таблицы, к которой выполняется запрос)

Агрегирование (GroupAggregate, HashAggregate), соединение таблиц определенным методом (Nested Loop, Hash Join, Merge Join), сортировка (Sort) и оконные функции (WindowAgg) создают в дереве плана отдельные узлы, которые стоят выше и выполняются после выполнения нижележащего.

Если в плане запроса есть подзапрос, то он обозначается ключевым словом Subquery.

Если в исходном запросе есть секция Where, то в плане запроса будет присутствовать ключевое слово Filter, которое будет следовать после того запроса, к которому оно применяется.

Каждый узел обязательно содержит в круглых скобках значения трех параметров, если не применен параметр команды EXPLAIN COSTS OFF:

1) cost=x..y - здесь x - это оценка ресурсов, требуемых для начала вывода данных, y - оценка общей стоимости выполнения запроса (включая время обработки нижележащих узлов и вывод). Оценки приведены в условных единицах. Значение имеет только их соотношение
2) rows - это примерное количество строк, используемых на данном узле
3) width - это средний размер строки в байтах

Стрелка слева указывает на то, что вышеописанный узел выполняется над результатами того, на который указывает стрелка. Читать запрос необходимо снизу вверх:

Именованное отношение - это переменная отношения, определённая в СУБД (системе управления базами данных) посредством оператора **CREATE** (CREATE TABLE, CREATE BASE RELATION, CREATE VIEW, CREATE SNAPSHOT).

Базовое отношение - это именованное отношение, которое не является

производным. Существование базового отношения не зависит от существования других отношений.

Производное отношение - это отношение, которое определено через другие именованные отношения. Производное отношение зависит от существования других - базовых - отношений.

Выражаемое отношение - это отношение, которое можно получить из набора именованных отношений посредством некоторого реляционного выражения.

Каждое именованное отношение является выражаемым отношений, но не наоборот.

Примеры выражаемых отношений - базовые отношения, представления, снимки, промежуточные и окончательные результаты.

Множество всех выражаемых отношений - это множество всех базовых и всех производных отношений.

Снимки (snapshot) - это то же, что и представление, но с физическим сохранением и с периодическим обновлением.

Результат запроса - это неименованное производное отношение.

СУБД не обеспечивает постоянного существования результатов запросов.

Для сохранения результата запроса его можно присвоить какому-либо именованному отношению.

Промежуточный результат - это неименованное производное отношение, являющееся результатом подзапроса, вложенного в большее выражение.

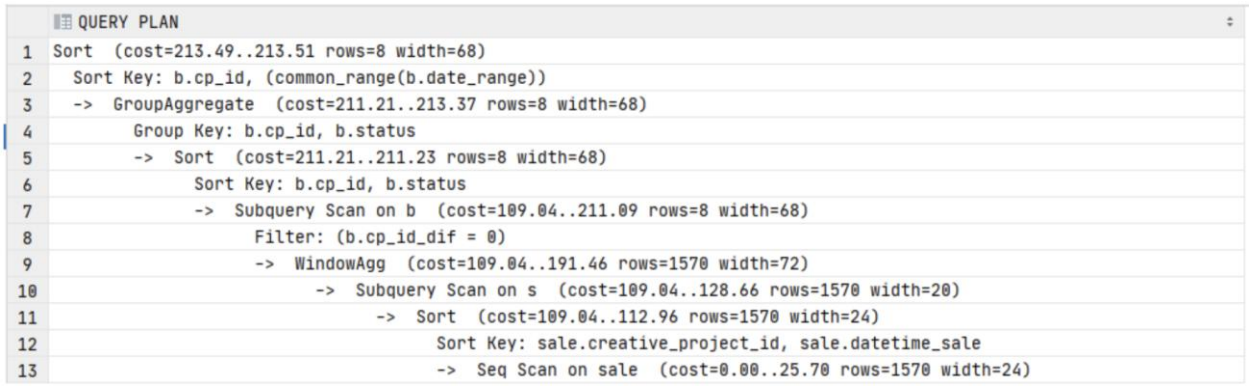
Хранимые процедуры. Пример
Для примера давайте напишем хранимую процедуру, которая будет добавлять новую запись, т.е. новый товар в нашу тестовую таблицу. Для

этого мы определим три входящих параметра: @CategoryId – идентификатор категории товара, @ProductName — наименование товара и @Price – цена товара, данный параметр будет у нас необязательный, т.е. его можно будет не передавать в процедуру (например, мы не знаем еще цену), для этого в его определении мы зададим значение по умолчанию. Эти параметры в теле процедуры, т.е. в блоке BEGIN…END можно использовать, так же как и обычные переменные (как Вы знаете, переменные обозначаются знаком @). В случае если Вам нужно указать выходные параметры, то после названия параметра указывайте ключевое слово OUTPUT (или сокращённо OUT).

```
--Создаем процедуру
CREATE PROCEDURE TestProcedure
(
	--Входящие параметры
	@CategoryId INT,
	@ProductName VARCHAR(100),
	@Price MONEY = 0
)
AS
BEGIN
--Инструкции, реализующие Ваш алгоритм

--Обработка входящих параметров
--Удаление лишних пробелов в начале и в конце текстовой строки
SET @ProductName = LTRIM(RTRIM(@ProductName));

--Добавляем новую запись
INSERT INTO TestTable(CategoryId, ProductName, Price)
	VALUES (@CategoryId, @ProductName, @Price)

--Возвращаем данные
	SELECT * FROM TestTable
	WHERE CategoryId = @CategoryId
END

GO
```

План SQL-запроса. Общие сведения и практика

Рассмотрим следующий план запроса:



```
QUERY PLAN
Sort  (cost=213.49..213.51 rows=8 width=68)
  Sort Key: b.cp_id, (common_range(b.date_range))
  ->  GroupAggregate  (cost=211.21..213.37 rows=8 width=68)
        Group Key: b.cp_id, b.status
        ->  Sort  (cost=211.21..211.23 rows=8 width=68)
              Sort Key: b.cp_id, b.status
              ->  Subquery Scan on b  (cost=109.04..211.09 rows=8 width=68)
                    Filter: (b.cp_id_dif = 0)
                    ->  WindowAgg  (cost=109.04..191.46 rows=1570 width=72)
                          ->  Subquery Scan on s  (cost=109.04..128.66 rows=1570 width=20)
                                ->  Sort  (cost=109.04..112.96 rows=1570 width=24)
                                      Sort Key: sale.creative_project_id, sale.datetime_sale
                                      ->  Seq Scan on sale  (cost=0.00..25.70 rows=1570 width=24)
```

1) Seq Scan on sale (cost=0.00..25.70 rows=1570 width=24) - Scan означает, что происходит сканирование таблицы и выборка конкретных строк, Seq - что используется метод доступа к таблице “Последовательный просмотр”, sale - название таблицы. cost=0.00..25.70 указывает на то, что для начала вывода данных требуется затратить 0.00 условных единиц, а вывести их - 25.70. rows=1570 - что примерное количество просматриваемых строк равно 1570, а width=24 - что их средний размер равен 24 байта.

2) После отбора строк выполняется вышележащий узел Sort, на котором производится сортировка отобранных ранее строк. Сортировка производится по столбцам creative_project_id и datetime_sale таблицы sale.

3) Subquery Scan on s … - означает, что выполненное ранее является подзапросом с именем s

4) WindowAgg … - означает, что к результатам подзапроса была применена оконная функция

5) Subquery Scan on b … - означает, что это был еще один подзапрос с именем b. Слово Filter указывает, что из него были отобраны только строки, удовлетворяющие условию cp_id_dif = 0, где cp_id_dif - столбец подзапроса b

6) Выше располагаются два узла: Sort и GroupAggregate. Это означает, что к внешнему запросу была применена группировка по столбцам cp_id и status подзапроса b, которая вызвала сортировку строк внешнего запроса по этим же столбцам

7) Самым верхним узлом является Sort, что говорит о том, что результаты внешнего запроса были отсортированы по столбцам cp_id и common_range(date_range), где date_range и cp_id - столбцы подзапроса b, взятые во внешнем запросе, а common_range - агрегатная функция (соответственно, common_range(date_range) -- это столбец, полученные после применения агрегатной функции common_range к столбцу date_range).

Рассмотрим другой запрос, в котором дерево ветвится (есть стрелочки, находящиеся на одном уровне отступа справа. Обычно такое бывает в случае наличия секции Join):

```
Sort  (cost=100.33..101.23 rows=360 width=186)
  Sort Key: (count(s.id)) DESC, m.id
  ->  HashAggregate  (cost=81.45..85.05 rows=360 width=186)
        Group Key: m.id
        ->  Hash Right Join  (cost=39.58..73.60 rows=1570 width=182)
              Hash Cond: (cp.member_id = m.id)
              ->  Hash Right Join  (cost=21.48..51.33 rows=1570 width=8)
                    Hash Cond: (s.creative_project_id = cp.id)
                    ->  Seq Scan on sale s  (cost=0.00..25.70 rows=1570 width=8)
                    ->  Hash  (cost=15.10..15.10 rows=510 width=8)
                          ->  Seq Scan on creative_project cp  (cost=0.00..15.10 rows=510 width=8)
              ->  Hash  (cost=13.60..13.60 rows=360 width=178)
                    ->  Seq Scan on member m  (cost=0.00..13.60 rows=360 width=178)
```

Эмпирическое правило: для того, чтобы найти, к какой таблице применяется запрос (эта таблица будет в секции FROM), нужно спускаться по дереву плана запроса до тех пор, пока не будет встречено ключевое слово Scan. Первое из них будет указывать на таблицу, используемую в секции FROM самого первого и глубокого запроса

1) Согласно эмпирическому правилу, запрос применяется к таблице sale. Узел Hash после указывает на то, что параллельно выполняется выборка строк из таблицы creative_project (из следующего узла).
2) Узел Hash Right Join над узлом Seq Scan on sale s… указывает на то, что строки двух ранее выбранных таблиц sale и creative_project соединяются методом хеширования. При этом ключевое слово Right указывает, что выполнятся Right Join. Соединение происходит по условию s.creative_project_id = cp.id, на которое указывают ключевые слова Hash Cond.
3) Строки полученной на этом узле таблицы соединяются со строками таблицы member, получаемой из следующего узла на том же уровне вложенности, по условию cp.member_id = m.id.
4) Следующий узел выше указывает на то, что по полученной таблице производится группировка строк хешированием (HashAggregate) по столбцу m.id
5) И самый верхний узел указывает на то, что полученные результаты сортируются по убыванию (есть ключевое слово DESC) по столбцу count(s.id) (агрегатная функция count к столбцу s.id) и по возрастанию по столбцу m.id

> ORM (Object Relational Mapping, объектно-реляционное отображение) — это технология, которая позволяет работать с базами данных так, как если бы это были объекты из языков программирования.
> ORM — как бы прослойка между базой данных и языками программирования.

С его помощью можно:

- представить информацию из базы как объекты, с которыми может работать язык программирования;
- без ошибок сопоставить поля из базы данных и свойства полученных объектов друг с другом;
- избавиться от необходимости использовать SQL;
- создать интерфейс, который позволит выполнять задачи CRUD — к ним относятся создание, чтение, модификация и удаление данных из базы.

Преимущества использования ORM

Возможность писать на любом языке. Частый язык для работы с ORM — Python. На нём же написаны многие ORM-библиотеки. Но технически можно применять и другие языки — такие, которые удобны конкретному разработчику или используются в проекте. Не нужно постоянно «переключаться» между SQL и более привычными языками. Можно просто писать запросы с ORM.

Ускорение разработки. Разработчику не приходится писать много лишнего кода для перевода записей базы данных в сущности, понятные

языку программирования, — и наоборот. Мэппинг, то есть сопоставление, происходит за счет ORM.

Независимость от СУБД. Идея ORM — чтобы разработчику вообще не приходилось думать о базе и системе управления, он мог просто брать данные и работать с ними. Кроме того, ORM обычно поддерживает разные СУБД: MySQL, SQLite, PostgreSQL и так далее. Поэтому переходить с одной СУБД на другую довольно легко — выбранное ORM-решение может поддерживать обе системы.

Широкие возможности. ORM реализует большинство функций, нужных разработчику при взаимодействии с базой. Кроме стандартных CRUD-задач, ORM поддерживает транзакции, миграции и другие полезные возможности.

Недостатки ORM

Сложная первоначальная настройка. Чаще всего недостаточно просто скачать ORM-фреймворк, чтобы он работал «из коробки». Его нужно подключить и настроить, а это дополнительная работа, время и возможные сложности.

Неэффективность в некоторых ситуациях. Иногда писать SQL-запросы вручную оказывается эффективнее: они срабатывают быстрее и используют меньше ресурсов компьютера. Это не всегда критично — порой скорость работы программиста важнее. Но эту особенность нужно учитывать, если для программы важна эффективность.

Меньшая прозрачность. При работе с ORM у разработчика меньше понимания, что происходит в базе на самом деле. Если он до этого не имел дела с SQL и базами данных, то в нештатной ситуации может просто запутаться из-за непонимания, как что работает. Поэтому рекомендуется всё же изучать теорию работы с базами, чтобы иметь представление, что происходит «внутри».