

1. **Что такое База Данных?** База данных (БД) — это структурированный, организованный набор взаимосвязанных данных, предназначенный для долговременного хранения в компьютерной системе и эффективного управления (поиск, извлечение, обновление, удаление). Ключевые характеристики БД включают: *структурированность* (данные организованы по определенной модели), *интегрированность* (данные объединены в единое целое с минимальной избыточностью), *независимость* (отделение логической структуры данных от физического хранения и прикладных программ), *управляемость* (данные поддаются централизованному контролю и администрированию), *целостность* (обеспечение правильности и непротиворечивости данных) и *многопользовательский доступ* (поддержка одновременной работы многих пользователей). БД представляют собой основу для информационных систем различного масштаба — от простых списков контактов до глобальных систем бронирования.

## 2. Что такое Система Управления Базами Данных?

Система Управления Базами Данных (СУБД) — это комплекс программных и языковых средств, предназначенный для создания, ведения и совместного использования баз данных многими пользователями и приложениями. СУБД выступает посредником между базой данных, прикладными программами и пользователями. Ее основные функции: *управление данными* (определение структуры данных - DDL, манипулирование данными - DML, запрос данных - DQL), *управление хранением* (эффективная организация данных на физических носителях), *управление транзакциями* (обеспечение свойств ACID), *управление параллельным доступом* (контроль одновременных операций для предотвращения конфликтов), *восстановление данных* (после сбоев), *обеспечение безопасности* (аутентификация, авторизация, шифрование) и *поддержка целостности данных* (проверка правил). Примеры СУБД: Oracle, MySQL, PostgreSQL, Microsoft SQL Server, SQLite.

## 3. Когда и кем создана первая СУБД, как она называлась? Назовите несколько известных современных СУБД.

Понятие "первой СУБД" исторически дискуссионно, но одной из первых коммерчески успешных и признанных СУБД, основанной на **навигационной** (иерархической и сетевой) модели данных, была **Integrated Data Store (IDS)**, разработанная **Чарльзом Бахманом** (Charles Bachman) в компании General Electric в начале 1960-х годов (первая версия - 1963 г.). Работа Бахмана была

революционной и легла в основу стандарта CODASYL (Conference on Data Systems Languages) для сетевых баз данных. Его вклад был отмечен премией Тьюринга в 1973 году. **Современные известные СУБД** чрезвычайно разнообразны: *Реляционные (RDBMS)*: Oracle Database, Microsoft SQL Server, MySQL, PostgreSQL, IBM Db2, SQLite; *NoSQL*: MongoDB (документная), Cassandra (колоночная), Redis (ключ-значение), Neo4j (графовая); *НовыеSQL*: Google Spanner, CockroachDB.

#### 4. Чем СУБД принципиально отличаются друг от друга?

СУБД принципиально отличаются по нескольким ключевым аспектам:

- **Модель данных:** Фундаментальное различие. Реляционная (таблицы), Документная (JSON/XML-подобные документы), Ключ-Значение (пары ключ-значение), Колоночная (хранение по столбцам, а не строкам), Графовая (узлы и ребра), Иерархическая (деревья), Сетевая (графы общего вида).
- **Архитектура и назначение:** OLTP (оперативная обработка транзакций), OLAP (аналитическая обработка), HTAP (гибридная), распределенные vs централизованные, in-memory (в памяти).
- **Согласованность и доступность (CAP/BASE):** Традиционные RDBMS фокусируются на строгой согласованности (ACID). Многие NoSQL СУБД жертвуют строгой согласованностью в пользу доступности и устойчивости к разделению сети (BASE - Basically Available, Soft state, Eventually consistent).
- **Схема:** Схема на запись (традиционные RDBMS) vs Схема на чтение/Бессхемные (многие NoSQL, схема определяется при чтении данных).
- **Масштабируемость:** Вертикальное (увеличение мощности сервера) vs Горизонтальное (добавление серверов). NoSQL СУБД обычно легче масштабируются горизонтально.
- **Языки запросов:** SQL (стандартизирован, но с диалектами) vs Специфические языки (например, MongoDB Query Language) vs API.
- **Лицензия и модель распространения:** Проприетарные (Oracle, MS SQL) vs Открытые (PostgreSQL, MySQL).

## 5. Что такое модель данных и в чем различие моделей?

**Модель данных** — это совокупность концепций, правил и структур, используемых для описания организации данных в базе, включая их структуру, операции над ними и ограничения целостности. Это абстракция, позволяющая представить данные понятным для СУБД и пользователей способом. **Различия между моделями** определяются тем, как они *структурируют данные и какие отношения между ними поддерживают*:

- **Иерархическая:** Древовидная структура (родитель-потомок). Один родитель, много потомков. Проста, но неэффективна для сложных связей. Пример: IMS.
- **Сетевая:** Обобщение иерархической. Узел может иметь много родителей. Гибче, но сложнее в управлении. Пример: IDS, CODASYL.
- **Реляционная:** Данные в таблицах (отношениях). Связи устанавливаются через значения данных (ключи). Математическая основа (теория множеств, реляционная алгебра). Доминирующая модель. Примеры: все RDBMS.
- **Объектно-ориентированная (ОО):** Представляет данные как объекты с атрибутами и методами. Пытается интегрировать БД и ООП. Сложна, не получила широкого распространения. Пример (гибридные): Oracle Object-Relational.
- **Объектно-реляционная (ОР):** Расширяет реляционную модель поддержкой объектов, наследования и т.д. Примеры: PostgreSQL, Oracle.
- **NoSQL-модели:** Специализированы под конкретные задачи и паттерны доступа. Включают Документную (MongoDB), Ключ-Значение (Redis), Колоночную (Cassandra), Графовую (Neo4j). Отличаются гибкой/отсутствующей схемой, горизонтальной масштабируемостью, ослабленной согласованностью.

## 6. Какие бывают виды идентификации объектов данных?

Идентификация объектов данных (сущностей, записей) необходима для их уникального определения и установления связей. Основные виды:

- **Естественный ключ (Natural Key):** Уникальный атрибут или комбинация атрибутов, которые существуют в реальном мире и естественным образом идентифицируют объект (например, паспортный

номер человека, ISBN книги, VIN автомобиля). Может меняться, иногда не является стабильным или компактным.

- **Суррогатный ключ (Surrogate Key):** Искусственно созданный уникальный идентификатор, не имеющий смысла в реальном мире. Вводится исключительно для целей идентификации в БД (например, автоинкрементное целое число, UUID, GUID). Стабилен, неизменяем, компактен. Широко используется в реляционных БД как первичный ключ.
- **Составной ключ (Composite Key):** Ключ, состоящий из двух или более атрибутов, комбинация значений которых уникально идентифицирует объект (например, **НомерРейса** + **ДатаВылета** для записи о рейсе). Может быть как естественным, так и суррогатным по природе своих атрибутов.
- **Внешний ключ (Foreign Key):** Атрибут(ы) в одной таблице, который ссылается на первичный ключ (или уникальный ключ) другой таблицы. Используется для установления связей между сущностями и поддержания ссылочной целостности. Сам по себе не идентифицирует объект в своей таблице уникально, но связывает его с уникальным объектом в другой таблице.

## 7. Кто считается автором реляционной модели? В каком году была опубликована самая известная его работа?

Автором реляционной модели данных единогласно признан **Эдгар Франк (Тед) Кодд (Edgar Frank "Ted" Codd)**. Будучи ученым-компьютерщиком, работавшим в IBM, он заложил теоретические основы реляционных баз данных. **Самой известной его работой** является статья **"A Relational Model of Data for Large Shared Data Banks"** (Реляционная модель данных для больших совместно используемых банков данных), которая была **опубликована в журнале Communications of the ACM в июне 1970 года** (том 13, № 6). Эта основополагающая работа представила математически строгую модель (основанную на теории множеств и логике предикатов первого порядка) для организации и манипулирования данными с использованием отношений (таблиц), свободную от навигационных зависимостей, присущих сетевым и иерархическим моделям. За свои достижения Кодд получил премию Тьюринга в 1981 году.

## 8. Перечислите пять-шесть основных требований, которые предъявляются к СУБД.

К современным СУБД предъявляются следующие ключевые требования:

1. **Надежность (Reliability):** Способность системы безотказно выполнять свои функции в течение длительного времени и корректно восстанавливаться после сбоев (аппаратных, программных, операторских).
2. **Масштабируемость (Scalability):** Способность системы обрабатывать растущие объемы данных и увеличивающееся количество пользователей/запросов, как путем увеличения мощности сервера (вертикальное масштабирование), так и, что особенно важно, путем добавления новых серверов (горизонтальное масштабирование).
3. **Производительность (Performance):** Способность системы обрабатывать запросы и транзакции с минимальной задержкой (latency) и максимальной пропускной способностью (throughput), эффективно используя ресурсы (CPU, память, диск, сеть).
4. **Безопасность (Security):** Обеспечение конфиденциальности, целостности и доступности данных. Включает аутентификацию, авторизацию (права доступа), аудит, шифрование данных на диске и в сети.
5. **Управляемость и Простота администрирования (Manageability):** Наличие удобных инструментов для установки, настройки, мониторинга, резервного копирования, восстановления, настройки производительности и общего управления системой.
6. **Совместимость и Соответствие стандартам (Compatibility & Standards):** Поддержка общепринятых стандартов (особенно SQL), совместимость с различными платформами (ОС, оборудование), языками программирования и фреймворками. Возможность миграции данных между системами.

## 9. Что представляет собой требование целостности данных?

**Целостность данных (Data Integrity)** — это фундаментальное требование к базе данных, означающее **корректность, точность и непротиворечивость данных на протяжении всего их жизненного цикла в БД**. Это гарантия того, что данные соответствуют определенным бизнес-правилам, семантическим ограничениям и логике предметной области. Требование целостности включает в себя несколько аспектов:

- **Структурная целостность:** Обеспечение правильности структуры данных (типы данных, форматы, обязательность полей - **NOT NULL**).
- **Ссылочная целостность (Referential Integrity):** Гарантия того, что связи между таблицами (через внешние ключи) всегда действительны. Не должно быть "висячих" ссылок (внешний ключ указывает на несуществующую запись), а удаление или изменение родительской записи должно обрабатываться по определенным правилам (**CASCADE**, **SET NULL**, **RESTRICT**).
- **Семантическая (пользовательская) целостность:** Обеспечение соответствия данных бизнес-правилам, которые не покрываются структурными и ссылочными ограничениями (например, **Зарплата >= 0**, **ДатаОкончания > ДатаНачала**, **Возраст >= 18**). Реализуется через **CHECK**-ограничения, триггеры, хранимые процедуры.
- **Целостность сущности (Entity Integrity):** Гарантия уникальности и ненулевости первичного ключа для каждой строки таблицы (**PRIMARY KEY** constraint). СУБД обеспечивает целостность с помощью механизмов ограничений (constraints), триггеров, хранимых процедур и транзакций.

## 10. Что называют согласованностью данных в базе, какими механизмами она поддерживается?

**Согласованность данных (Data Consistency)** в контексте баз данных имеет два основных аспекта:

1. **Согласованность в смысле целостности (Consistency в ACID):** Это состояние, при котором данные в БД удовлетворяют всем заданным правилам целостности (ограничениям) *после выполнения каждой отдельной транзакции*. Транзакция переводит БД из одного согласованного состояния в другое согласованное состояние. Нарушение любого правила целостности приводит к откату транзакции.
  2. **Согласованность в распределенных системах (Consistency в CAP):** В контексте распределенных БД и NoSQL это гарантия того, что *каждое чтение данных получит самую последнюю запись или ошибку*. Это самый строгий вид согласованности, который часто противостоит доступности при разделении сети (CAP-теорема). Существуют более слабые формы (eventual consistency, read-your-writes consistency и т.д.).
- Механизмы поддержки согласованности (в основном для ACID-согласованности):**

- **Транзакции:** Основной механизм. Группировка операций в атомарную единицу работы, обеспечивающую переход между согласованными состояниями.
- **Ограничения целостности (Constraints):** PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL - автоматически проверяются СУБД при вставке, обновлении, удалении.
- **Триггеры (Triggers):** Автоматически выполняемый код на сервере БД при возникновении определенных событий (INSERT/UPDATE/DELETE). Могут использоваться для реализации сложных правил целостности и каскадных изменений.
- **Хранимые процедуры (Stored Procedures):** Позволяют инкапсулировать сложную бизнес-логику и проверки целостности на сервере БД, гарантируя их выполнение.
- **Блокировки (Locking):** Механизмы управления параллельным доступом (пессимистичные, оптимистичные блокировки), предотвращающие конфликты при одновременной модификации одних и тех же данных, которые могут привести к нарушению целостности.
- **Журналирование (Logging - Write-Ahead Logging - WAL):** Запись всех изменений в журнал перед их фиксацией на диск. Критично для восстановления согласованного состояния после сбоя.

## 11. Что такое нормализация в реляционной модели?

**Нормализация** — это процесс **структурирования (декомпозиции) отношений (таблиц)** в реляционной базе данных с целью:

- **Устранения избыточности данных:** Хранение каждого факта в одном месте, что уменьшает объем хранилища и риск аномалий обновления.
- **Устранения аномалий обновления:**
  - *Аномалия вставки:* Невозможность вставить данные без наличия связанных данных (например, нельзя добавить отдел без сотрудников).
  - *Аномалия обновления:* Необходимость обновлять одни и те же данные в нескольких местах, что может привести к противоречивости (например, изменение адреса отдела во всех записях сотрудников).
  - *Аномалия удаления:* Непреднамеренная потеря данных при удалении других данных (например, удаление последнего сотрудника отдела приводит к потере информации об отделе).

- **Обеспечения гибкости структуры данных:** Упрощение модификации схемы БД.
- **Улучшения целостности данных:** Более четкое определение зависимостей между данными. Процесс нормализации заключается в последовательном приведении таблиц к **нормальным формам (НФ)**, каждая из которых имеет более строгие требования, чем предыдущая:
- **1НФ:** Значения всех атрибутов атомарны (неделимы), нет повторяющихся групп.
- **2НФ:** Находится в 1НФ, и каждый неключевой атрибут полностью функционально зависит от *всего* первичного ключа (актуально для составных ключей).
- **3НФ:** Находится в 2НФ, и ни один неключевой атрибут не находится в транзитивной функциональной зависимости от первичного ключа (т.е. неключевые атрибуты не зависят функционально друг от друга).
- **НФ Бойса-Кодда (BCNF):** Усиленная 3НФ. Каждая детерминанта (атрибут, от которого функционально зависит другой атрибут) является потенциальным ключом. Устраняет оставшиеся аномалии, не устраненные 3НФ.
- **4НФ, 5НФ:** Устраняют многозначные зависимости и зависимости соединения. Применяются реже на практике. Целью обычно является достижение 3НФ или BCNF. Денормализация (обратный процесс) иногда применяется осознанно для повышения производительности запросов в ущерб идеальной структуре.

## 12. Что такое отношение, кортеж и домен – в терминах SQL-модели?

В терминах реляционной модели и ее практической реализации SQL:

- **Отношение (Relation):** Это фундаментальная структура данных в реляционной модели, представленная в SQL как **Таблица (Table)**. Отношение представляет собой *множество* однотипных записей, описывающих сущность предметной области (например, "Сотрудники", "Отделы"). Оно имеет имя и строго определенную структуру (заголовок - набор атрибутов). *Важно:* В теории отношений порядок строк не определен, в SQL таблицах порядок строк также не гарантирован без **ORDER BY**.
- **Кортеж (Tuple):** Это *упорядоченный* набор значений атрибутов, описывающий один конкретный экземпляр сущности. В SQL кортежу соответствует **Строка (Row)** или **Запись (Record)** таблицы. Каждая



строка содержит фактические данные об одном объекте (например, конкретном сотруднике: 'Иванов', 'Программист', 50000).

- **Домен (Domain):** Это *множество* допустимых значений, которые может принимать определенный атрибут. Домен определяет *тип данных* и *семантические ограничения* для атрибута. В SQL домену соответствует **Тип данных столбца (Data Type)** (`INTEGER`, `VARCHAR(50)`, `DATE`, `BOOLEAN`) и, возможно, дополнительные ограничения (`CHECK`, `NOT NULL`). Например, домен "Возраст" может быть `INTEGER` с ограничением `CHECK (Возраст >= 18 AND Возраст <= 65)`. Домен обеспечивает целостность на уровне атрибута.

### 13. Чем отличаются инфологический, даталогический и физический уровни представления моделей данных?

Это три уровня абстракции архитектуры ANSI/SPARC (или трехуровневой архитектуры СУБД), разделяющих представление данных для разных категорий пользователей:

- **Инфологический уровень (External Level / View Level):**
  - *Цель:* Представить данные с точки зрения *конкретного пользователя* или *прикладной программы*. Отражает требования и способы использования данных конечными пользователями или разработчиками приложений.
  - *Что описывает:* Подмножество всей базы данных, видимое конкретному пользователю. Определяет *логические структуры данных* (сущности, атрибуты, связи), *ограничения целостности*, *правила обработки*, *форматы данных*, *терминологию* предметной области. Может использовать модели (ER-диаграммы, UML).
  - *Абстракция:* Наиболее высокоуровневая, ориентирована на бизнес-логику.
- **Даталогический уровень (Conceptual Level / Logical Level):**
  - *Цель:* Представить *общую, интегрированную модель* всех данных предметной области без привязки к физическому хранению или конкретным приложениям. Является "посредником" между инфологическим и физическим уровнями.
  - *Что описывает:* *Полную структуру* всей базы данных: все сущности, атрибуты, связи, типы данных, ограничения целостности (первичные/внешние ключи, уникальность, проверки), правила

безопасности. Часто описывается на языке определения данных (DDL) СУБД или концептуальной ER-диаграммой.

- **Абстракция:** Независима от физического хранения и конкретных внешних представлений. Фокусируется на *что* хранится, а не *как*.
  - **Физический уровень (Internal Level / Physical Level):**
    - **Цель:** Описать *реализацию* базы данных на физических носителях информации. Определяет, как данные физически хранятся и организуются на диске.
    - **Что описывает:** Структуры хранения (файлы, страницы, блоки), методы доступа (индексы - B-деревья, хеши), стратегии размещения данных (кластеризация, секционирование), методы сжатия, стратегии управления памятью (буферный пул), управление свободным пространством. Описывается на языке низкоуровневых спецификаций СУБД.
    - **Абстракция:** Наиболее низкоуровневая, скрыта от программистов и пользователей. Фокусируется на *как* хранится и извлекается.
- Ключевое отличие:** Инфологический уровень - вид *пользователя/приложения*, Даталогический - *общая логическая структура*, Физический - *реализация на диске*. Независимость данных достигается за счет изоляции изменений на одном уровне от других уровней (логическая независимость: изменение концептуальной схемы не влияет на внешние схемы; физическая независимость: изменение физической схемы не влияет на концептуальную и внешние схемы).

#### 14. Что понимают под сущностью в реляционной системе?

В реляционной системе (и в моделировании данных ER) **сущность (Entity)** — это объект, предмет, явление, событие или концепция из *предметной области*, информацию о котором необходимо хранить в базе данных. Сущность обладает *набором характеристик (атрибутов)*, которые ее описывают, и должна быть *однозначно идентифицируема* среди экземпляров того же типа. В реляционной модели:

- **Тип сущности (Entity Type / Entity Set):** Описывает класс однородных объектов (например, "Сотрудник", "Заказ", "Товар"). В реляционной базе типу сущности соответствует **Таблица (Table)**.
- **Экземпляр сущности (Entity Instance):** Конкретный представитель типа сущности (например, сотрудник "Иван Иванов", заказ №12345). В

реляционной базе экземпляру сущности соответствует **Строка (Row)** или **Запись (Record)** в таблице.

◦ **Ключевые характеристики:**

- *Идентифицируемость:* Каждый экземпляр должен иметь уникальный идентификатор (первичный ключ).
- *Наличие атрибутов:* Сущность описывается своими атрибутами (столбцами таблицы).
- *Существование независимо от других сущностей:* Сущность имеет самостоятельное значение (хотя и может быть связана с другими). Примеры: Сотрудник, Проект, Автомобиль, Пациент, Счет.

## 15. Что такое атрибут, чем он отличается от сущности?

**Атрибут (Attribute)** — это характеристика, свойство или описатель, который определяет или уточняет **сущность** (или **связь**). Атрибут представляет собой отдельный элемент данных, который хранит конкретное значение для каждого экземпляра сущности. В реляционной модели атрибуту соответствует **Столбец (Column)** таблицы. **Ключевые характеристики атрибута:**

- *Имя:* Уникальное в пределах сущности/таблицы.
- *Тип данных (Домен):* Определяет вид хранимых значений (число, текст, дата, логическое) и возможные операции над ними.
- *Возможные ограничения:* **NOT NULL** (обязательность), **UNIQUE** (уникальность), **DEFAULT** (значение по умолчанию), **CHECK** (проверка значения).
- *Простой vs Составной:* Простой атрибут неделим (например, **Имя**). Составной атрибут можно разложить на более мелкие компоненты (например, **Адрес** -> **Улица**, **Дом**, **Квартира**; на логическом уровне это составной, на физическом в реляционной модели обычно разбивается на отдельные столбцы).
- *Однозначный vs Многозначный:* Однозначный атрибут содержит одно значение для экземпляра (например, **ДатаРождения**). Многозначный атрибут может содержать несколько значений (например, **Телефон** у человека). В чистой реляционной модели многозначные атрибуты не допускаются (их нужно выносить в отдельную таблицу).
- *Производный (Вычисляемый):* Значение может быть вычислено на основе других атрибутов (например, **Возраст** = ТекущаяДата -

**ДатаРождения**). Может храниться или вычисляться "на лету". **Отличие от Сущности:**

- **Сущность** — это объект предметной области, о котором хранится информация (Сотрудник, Заказ). **Атрибут** — это *свойство* этого объекта (**ФИО**, **Должность**, **ДатаЗаказа**, **Сумма**).
- Сущность имеет *самостоятельное значение* и *уникально идентифицируется*. Атрибут *описывает* сущность и не имеет *самостоятельного значения* вне контекста сущности (или связи).
- В реляционной модели Сущность -> Таблица, Атрибут -> Столбец.

## 16. Что такое связь? Перечислите главные ее характеристики.

**Связь (Relationship)** в модели данных — это **ассоциация**, устанавливаемая между двумя или более сущностями, которая представляет собой важное для предметной области взаимодействие или зависимость между ними. Связи показывают, как экземпляры одной сущности соотносятся с экземплярами другой сущности. В реляционной модели связи реализуются через **внешние ключи (Foreign Keys)**. **Главные характеристики связи:**

1. **Тип (Степень) связи (Relationship Degree):** Количество сущностей, участвующих в связи.

- **Бинарная (Binary):** Связь между двумя сущностями (наиболее распространенный тип). Пример: Сотрудник работает в Отделе.
- **Тернарная (Ternary):** Связь между тремя сущностями. Пример: Поставщик (S) поставляет Деталь (P) на Проект (J). Связь "Поставка" требует всех трех участников.
- **N-арная:** Связь между **N** сущностями (редко используется).

2. **Мощность связи (Cardinality):** Количественное соотношение экземпляров сущностей, которые могут участвовать в связи. Основные типы:

- **Один-к-одному (1:1):** Один экземпляр сущности А связан ровно с одним экземпляром сущности Б, и наоборот. Пример: Сотрудник - СлужебныйАвтомобиль (если у каждого сотрудника ровно одна машина и каждая машина закреплена за одним сотрудником).
- **Один-ко-многим (1:M):** Один экземпляр сущности А может быть связан с *многими* экземплярами сущности Б, но каждый экземпляр сущности Б связан *только с одним* экземпляром сущности А.

Пример: Отдел (1) - Сотрудник (М) (В одном отделе много сотрудников, один сотрудник работает в одном отделе).

- *Многие-ко-многим (M:N)*: Один экземпляр сущности А может быть связан с *многими* экземплярами сущности Б, и наоборот. Пример: Студент (М) - Курс (N) (Один студент посещает много курсов, один курс посещают много студентов). В реляционной модели реализуется через *ассоциативную (связующую) таблицу*.

3. **Обязательность (Participation / Optionality)**: Указывает, *должен* ли экземпляр сущности участвовать в связи.

- *Обязательная (Total) связь*: Каждый экземпляр сущности *должен* участвовать хотя бы в одной связи. Обозначается сплошной линией на ERD. Пример: Каждый Заказ *должен* быть сделан Клиентом (Клиент обязателен).
- *Необязательная (Partial) связь*: Экземпляр сущности *может* существовать без участия в связи. Обозначается пунктирной линией на ERD. Пример: Сотрудник *может* не иметь служебного автомобиля (Автомобиль необязателен).

4. *\*Роль (Role)*: Имя, которое описывает функцию сущности в связи (особенно важно, если сущность связана сама с собой - рекурсивная связь). Пример: Сущность "Сотрудник", связь "Подчиняется", роли "Начальник" и "Подчиненный".

17. **Для чего в реляционной модели используются первичный и внешний ключи?**

**Первичный ключ (Primary Key - PK) и Внешний ключ (Foreign Key - FK)** являются фундаментальными механизмами реляционной модели для обеспечения целостности данных и установления связей между таблицами.

◦ **Первичный ключ (PK)**:

- *Назначение 1: Уникальная идентификация строки в таблице.* Гарантирует, что каждая строка таблицы может быть однозначно выделена из всех остальных строк этой же таблицы.
- *Назначение 2: Обеспечение целостности сущности (Entity Integrity).* Ограничение **PRIMARY KEY** автоматически подразумевает **UNIQUE** и **NOT NULL** для всех столбцов, входящих в ключ. Это предотвращает дублирование строк и наличие строк с неопределенным (NULL) идентификатором.

- **Состав:** Может состоять из одного столбца (простой ключ) или нескольких столбцов (составной ключ). Значения в столбцах РК должны быть уникальны в комбинации и не содержать NULL.
- **Пример:** В таблице **Сотрудники** столбец **EmployeeID** (идентификатор сотрудника) является РК.
- **Внешний ключ (FK):**
  - **Назначение 1: Установление связи** между двумя таблицами (родительской и дочерней). FK в дочерней таблице ссылается на РК (или уникальный ключ) в родительской таблице. Это определяет отношение "один-ко-многим" или "один-к-одному".
  - **Назначение 2: Обеспечение ссылочной целостности (Referential Integrity).** FK гарантирует, что значение в столбце(ах) FK дочерней таблицы либо соответствует какому-либо значению РК в родительской таблице, либо равно NULL (если FK столбец допускает NULL). Это предотвращает появление "висячих" ссылок (orphaned records) в дочерней таблице.
  - **Поведение при изменении/удалении:** Ограничение FK определяет действия при попытке обновления или удаления записи в родительской таблице, на которую ссылаются:
    - **NO ACTION / RESTRICT:** Запретить действие, если существуют ссылающиеся записи (по умолчанию).
    - **CASCADE:** Каскадно обновить/удалить ссылающиеся записи.
    - **SET NULL:** Установить столбец(цы) FK в NULL у ссылающихся записей.
    - **SET DEFAULT:** Установить столбец(цы) FK в значение по умолчанию у ссылающихся записей.
  - **Пример:** В таблице **Заказы** (дочерняя) столбец **CustomerID** является FK, ссылающимся на столбец **CustomerID** (РК) в таблице **Клиенты** (родительская). Это связывает заказ с клиентом, его сделавшим, и гарантирует, что нельзя создать заказ для несуществующего клиента.

## 18. Чем графически отличаются друг от друга два самых известных вида диаграмм сущность-связь (нотация Чена и нотация Эвереста)?

Две наиболее распространенные нотации для ER-диаграмм (Entity-Relationship Diagrams) — **нотация Питера Чена (Chen, 1976)** и **нотация Ричарда Баркера** (иногда называемая "нотацией воронки" или

"нотацией Эвереста", так как Баркер работал в CACI, ранее CACI-Everest) / Information Engineering (IE) Crow's Foot — имеют существенные визуальные различия:

- **Сущности (Entities):**

- *Чен*: Изображаются **прямоугольниками**. Имя сущности пишется внутри прямоугольника.
- *Баркер (Crow's Foot)*: Изображаются **прямоугольниками с закругленными углами** (иногда просто прямоугольниками). Имя сущности пишется внутри, обычно в единственном числе и заглавными буквами.

- **Атрибуты (Attributes):**

- *Чен*: Изображаются **овалами**, соединенными линией с соответствующей сущностью (или связью). Ключевые атрибуты подчеркиваются. Многозначные атрибуты заключаются в двойной овал, составные атрибуты соединяются линиями с составляющими их простыми атрибутами.
- *Баркер (Crow's Foot)*: Атрибуты **перечисляются внутри прямоугольника сущности** (под именем сущности). Ключевые атрибуты помечаются символом **PK** или **(PK)**, внешние ключи - **FK** или **(FK)**. Обязательные атрибуты (NOT NULL) часто помечаются \* или **(M)**, необязательные - **o** или **(O)**. Составные и многозначные атрибуты обычно не визуализируются явно на диаграмме высокого уровня, подразумевается их декомпозиция на отдельные столбцы в таблице.

- **Связи (Relationships):**

- *Чен*: Изображаются **ромбами**. Имя связи пишется внутри ромба. Линии соединяют ромб с участвующими сущностями. На линиях указывается мощность связи (**1** или **M, N**) и обязательность (одна черта - обязательная, две черты - необязательная; или **(min, max)**). Роли могут указываться на линиях.
- *Баркер (Crow's Foot)*: Изображаются **линиями**, непосредственно соединяющими сущности. Имя связи пишется *над линией* (глагол). **Мощность и обязательность кодируются с помощью символов на концах линии:**
  - *Один (1)*: Одиночная перпендикулярная черта | рядом с сущностью.



- *Многие (M/N)*: "Воронья лапка" (три расходящиеся линии) >< или < рядом с сущностью.
- *Обязательно (Total)*: Сплошная линия, перпендикулярная черта или "лапка" касается сущности.
- *Необязательно (Partial)*: Круг 0 на линии рядом с сущностью.

*Примеры концов линии:*

- | --- | : Один-к-одному (1:1), обязательная с обеих сторон.
  - | --- 0 : Один-к-одному (1:1), слева обязательная, справа необязательная.
  - | --- < : Один-ко-многим (1:M), "один" (обязательный) слева, "многие" (обязательные) справа.
  - 0 > --- < : Многие-ко-многим (M:N), необязательная с обеих сторон (часто реализуется через ассоциативную таблицу, которая не показана).
- **Ключевое графическое отличие:** Чен использует отдельные символы для сущностей (прямоуг.), атрибутов (овалы) и связей (ромбы), четко их разграничивая, но диаграммы могут стать громоздкими. Баркер (Crow's Foot) помещает атрибуты внутрь сущностей и использует линии с символами на концах для связей и их мощности/обязательности, что делает диаграммы компактнее и ближе к физической реализации в реляционных таблицах, особенно для больших схем. Crow's Foot стала де-факто стандартом в современных CASE-средствах (например, ERwin, Toad Data Modeler, MySQL Workbench).

## 19. Перечислите основные виды объектов реляционной базы данных.

Основные объекты, из которых состоит реляционная база данных (RDBMS):

1. **Таблица (Table):** Основная структура хранения данных. Состоит из строк (записей) и столбцов (полей). Представляет сущность предметной области.
2. **Столбец (Column / Field):** Атрибут сущности, хранящий определенный тип данных (INTEGER, VARCHAR, DATE и т.д.). Имеет имя и может иметь ограничения (NOT NULL, UNIQUE, DEFAULT).
3. **Строка (Row / Record / Tuple):** Набор значений столбцов, представляющий один конкретный экземпляр сущности, описанной таблицей.



4. **Ограничение (Constraint):** Правило, накладываемое на данные в таблице для обеспечения целостности.
- **PRIMARY KEY** (Первичный ключ)
  - **FOREIGN KEY** (Внешний ключ)
  - **UNIQUE** (Уникальность)
  - **NOT NULL** (Запрет NULL-значений)
  - **CHECK** (Проверка условия)
  - **DEFAULT** (Значение по умолчанию)
5. **Индекс (Index):** Вспомогательная структура данных, создаваемая на одном или нескольких столбцах таблицы для ускорения поиска (**SELECT**) и сортировки (**ORDER BY**) данных. Замедляет операции вставки/обновления/удаления (**INSERT/UPDATE/DELETE**). Основные типы: В-дерево, Хеш, Bitmap.
6. **Представление (View):** Виртуальная таблица, результат выполнения предопределенного SQL-запроса (**SELECT**). Не хранит данные физически, а представляет "окно" в данные из одной или нескольких базовых таблиц. Используется для безопасности (ограничение доступа), упрощения сложных запросов, обеспечения логической независимости данных.
7. **Хранимая процедура (Stored Procedure):** Именованный набор SQL-операторов и логики (ветвления, циклы), хранящийся на сервере БД и выполняемый как единое целое. Используется для инкапсуляции бизнес-логики, повышения производительности (предкомпиляция), безопасности.
8. **Функция (Function):** Похожа на хранимую процедуру, но всегда возвращает значение (скалярное или табличное). Может использоваться в SQL-выражениях (**SELECT dbo.MyFunction(...)**).
9. **Триггер (Trigger):** Специальный вид хранимой процедуры, который автоматически выполняется сервером БД в ответ на определенное событие (**INSERT, UPDATE, DELETE**) над указанной таблицей (или представлением). Используется для аудита, обеспечения сложной целостности, каскадных изменений.
10. **Последовательность (Sequence) / Автоинкремент (Identity):** Объект, генерирующий уникальные последовательные числовые значения, часто используемые для суррогатных первичных ключей.
11. **Схема (Schema):** Контейнер (пространство имен) внутри базы данных для группировки других объектов (таблиц, представлений, процедур и т.д.). Управляет правами доступа на уровне схемы.

## 20. Чем представление отличается от таблицы?

**Представление (View) и Таблица (Table)** являются фундаментально разными объектами реляционной базы данных:

### ◦ Таблица:

- *Физическое хранение:* Хранит данные непосредственно на диске. Имеет выделенное пространство для строк.
- *Структура:* Определяется при создании (**CREATE TABLE**) и включает имена и типы столбцов, ограничения, индексы.
- *Операции:* Поддерживает все операции манипулирования данными (**INSERT**, **UPDATE**, **DELETE**, **SELECT**), если не ограничено ключами, триггерами и т.д.
- *Данные:* Содержит фактические данные.
- *Изменения:* Изменение структуры таблицы (**ALTER TABLE**) может быть сложным и затрагивать данные. Добавление данных меняет содержимое таблицы.

### ◦ Представление:

- *Физическое хранение:* **Не хранит данные физически.** Это виртуальная таблица.
- *Структура:* Определяется SQL-запросом (**SELECT**), указанным при создании (**CREATE VIEW**). Столбцы представления - это столбцы результата этого запроса. Структура динамически зависит от базовых таблиц и самого запроса.
- *Операции:* В основном используется для **SELECT** (чтения данных). Операции **INSERT**, **UPDATE**, **DELETE** могут быть разрешены через представление (с помощью **INSTEAD OF** триггеров или если представление удовлетворяет определенным условиям - "updatable view": обычно на одну таблицу, без агрегаций, **DISTINCT**, **GROUP BY**, объединений в некоторых СУБД и т.д.), но выполняются они на самом деле над базовыми таблицами, на которые ссылается запрос представления.
- *Данные:* Не содержит собственных данных. При запросе к представлению (**SELECT \* FROM view\_name**) выполняется его базовый SQL-запрос к *актуальным* данным в базовых таблицах.
- *Изменения:* Изменение представления (**ALTER VIEW**) обычно означает переопределение его SQL-запроса. Это не затрагивает данные в базовых таблицах. Данные, видимые через

представление, меняются автоматически при изменении данных в базовых таблицах. **Ключевое отличие:** Таблица - это *физическое хранилище* данных. Представление - это *сохраненный запрос*, предоставляющий *виртуальное окно* в данные, физически хранящиеся в других таблицах (или даже других представлениях).

## 21. Что общего и в чем различие между триггером и хранимой процедурой?

### Общее:

- **Хранение на сервере БД:** Оба являются объектами базы данных, хранящимися и выполняемыми на сервере СУБД.
  - **Написание на SQL/процедурном SQL:** Обычно пишутся на расширенном SQL, включающем элементы процедурных языков (PL/pgSQL, T-SQL, PL/SQL) - переменные, ветвления (**IF**), циклы (**LOOP**, **WHILE**), обработка ошибок.
  - **Инкапсуляция логики:** Позволяют перенести бизнес-логику и сложные проверки целостности с клиентского приложения на сервер БД.
  - **Повышение производительности:** Могут повышать производительность за счет снижения сетевого трафика (выполняются на сервере) и предварительной компиляции/оптимизации (в разной степени).
  - **Безопасность:** Могут использоваться для реализации сложных правил безопасности и аудита.
- | Характеристика               | Хранимая Процедура (Stored Procedure)   | Триггер (Trigger)  |
|------------------------------|---|--|
| <b>Запуск</b>                | Вызывается <b>явно</b> прикладной программой или администратором командой <b>EXECUTE/CALL</b> . Запускается <b>автоматически</b> (неявно) сервером БД в <b>ответ на событие</b> ( <b>INSERT</b> , <b>UPDATE</b> , <b>DELETE</b> ), происходящее с <b>конкретной таблицей/представлением</b> . | Вызывается <b>автоматически</b> сервером БД в <b>ответ на событие</b> ( <b>INSERT</b> , <b>UPDATE</b> , <b>DELETE</b> ), происходящее с <b>конкретной таблицей/представлением</b> .  |
| <b>Время выполнения</b>      | В любое время по запросу пользователя/приложения. Только <b>до</b> ( <b>BEFORE</b> ), <b>после</b> ( <b>AFTER</b> ) или <b>вместо</b> ( <b>INSTEAD OF</b> ) события DML, к которому он привязан.  | В любое время по запросу пользователя/приложения. Только <b>до</b> ( <b>BEFORE</b> ), <b>после</b> ( <b>AFTER</b> ) или <b>вместо</b> ( <b>INSTEAD OF</b> ) события DML, к которому он привязан.   |
| <b>Возвращаемое значение</b> | Может возвращать значения (скалярные, табличные) через <b>OUTPUT</b> параметры или <b>RETURN</b> (для функций). <b>Не возвращает</b> значения явно. Может модифицировать данные через <b>NEW/OLD</b> псевдозаписи (в DML триггерах) или выполнять другие действия.                            | Может возвращать значения (скалярные, табличные) через <b>OUTPUT</b> параметры или <b>RETURN</b> (для функций). <b>Не возвращает</b> значения явно. Может модифицировать данные через <b>NEW/OLD</b> псевдозаписи (в DML триггерах) или выполнять другие действия. |
| <b>Аргументы</b>             | Может принимать входные ( <b>IN</b> ), выходные ( <b>OUT</b> ) и входо-выходные ( <b>INOUT</b> ) параметры. <b>Не имеет явных</b>   | Может принимать входные ( <b>IN</b> ), выходные ( <b>OUT</b> ) и входо-выходные ( <b>INOUT</b> ) параметры. <b>Не имеет явных</b>  |

**параметров.** Информация о событии доступна через специальные псевдозаписи (**INSERTED**, **DELETED** в T-SQL; **NEW**, **OLD** в Oracle/PostgreSQL). | | **Связь с данными** | Может обращаться к любым таблицам в БД (при наличии прав). | Связан с **конкретной таблицей (или представлением)**. Обычно работает с данными *этой* таблицы (через **NEW/OLD**). | | **Основное назначение** | Выполнение сложных операций, инкапсуляция бизнес-логики, пакетная обработка. | Обеспечение сложной целостности данных, реализация каскадных изменений, аудит (логирование изменений), запрет операций. | | **Транзакционность** | Обычно выполняется в контексте транзакции, вызвавшей ее. | Выполняется **в рамках той же транзакции**, что и вызвавшее его DML-событие. Откат триггера приводит к откату всей транзакции. | | **Рекурсия/Вложенность** | Может вызывать другие процедуры и функции. | Может запускать другие триггеры (каскадные триггеры), но *обычно* не может выполнять DML над таблицей, на которой он сам висит (во избежание бесконечной рекурсии), если это не разрешено явно. |

## 22. Что влияет на "стоимость выполнения запроса" в реляционных СУБД?

**"Стоимость выполнения запроса" (Query Cost)** в реляционных СУБД — это оценка, которую оптимизатор запросов вычисляет для каждого возможного плана выполнения. Она отражает *прогнозируемые ресурсные затраты* (время CPU, операции ввода/вывода (I/O), объем используемой памяти, сетевой трафик в распределенных системах) на выполнение запроса по данному плану. Факторы, влияющие на стоимость:

- **Объем обрабатываемых данных:** Количество строк в таблицах, к которым обращается запрос. Сканирование большой таблицы дороже сканирования маленькой.
- **Сложность запроса:**
  - Количество **JOIN** (особенно неэквисоединений - **NON-EQUI JOIN**) и их типы.
  - Количество и сложность условий в **WHERE** и **HAVING**.
  - Наличие агрегатных функций (**SUM**, **COUNT**, **AVG**), **GROUP BY**, **DISTINCT**, **ORDER BY**, **TOP/LIMIT**.
  - Наличие подзапросов (особенно коррелированных).
  - Использование оконных функций (**OVER()**, **PARTITION BY**).
- **Доступные индексы:**

- Наличие подходящего индекса для условий **WHERE** (предикатов доступа) и **JOIN** (предикатов соединения) *крайне важно*.
- Может ли индекс покрывать запрос (**Covering Index** - содержать все необходимые столбцы, чтобы избежать чтения самой таблицы - **Index Only Scan**).
- Селективность индекса (какой процент строк он отбирает).
- Тип индекса (B-дерево, Hash, Bitmap и т.д.).
- **Статистика по данным:** Точность статистической информации, собираемой СУБД о распределении данных в таблицах и индексах (количество строк, уникальные значения, гистограммы распределения). Оптимизатор сильно зависит от этой статистики для оценки количества строк на выходе каждой операции плана (Cardinality Estimation).
- **Типы операций доступа к данным:**
  - *Сканирование таблицы (Table Scan / Full Scan):* Чтение всей таблицы. Очень дорого для больших таблиц.
  - *Сканирование индекса (Index Scan):* Чтение всего индекса (листовых страниц).
  - *Поиск по индексу (Index Seek):* Прямой доступ к нужным записям индекса по ключу. Самый эффективный способ, если предикат высокоселективен.
  - *Ключ поиска (Bookmark Lookup / RID Lookup):* Дополнительная операция для получения полной строки таблицы по идентификатору, найденному в индексе (если индекс не покрывающий). Может быть дорогой при большом количестве строк.
- **Типы операций соединения (Join Algorithms):** Стоимость разных алгоритмов (**Nested Loops**, **Hash Join**, **Merge Join**) сильно различается в зависимости от объема данных и наличия индексов.
- **Ресурсы системы:** Производительность CPU, скорость дисков (SSD vs HDD), объем доступной оперативной памяти (буферный пул), нагрузка на систему.
- **Настройки СУБД:** Размер буферного пула, настройки параллелизма, параметры оптимизатора.
- **Параметризация запроса:** Использование параметров вместо литералов может помочь или помешать повторному использованию кэшированных планов и оценке селективности. Цель оптимизатора — найти план выполнения с *наименьшей* оцененной стоимостью.

## 23. Перечислите основные стадии оптимизации запросов системой управления БД.

Процесс оптимизации запроса в СУБД — это сложный многостадийный процесс, выполняемый компонентом **Оптимизатор Запросов (Query Optimizer)**. Основные стадии:

### 1. Парсинг (Parsing):

- *Синтаксический анализ*: Проверка правильности синтаксиса SQL-запроса.
- *Лексический анализ*: Разбиение запроса на лексемы (ключевые слова, идентификаторы, операторы).
- *Результат*: Построение синтаксического дерева запроса (Parse Tree).

### 2. Верификация (Validation) / Проверка (Checking):

- *Семантический анализ*: Проверка существования объектов (таблиц, столбцов), прав доступа пользователя к этим объектам, соответствия типов данных в операциях.
- *Результат*: Валидное синтаксическое дерево, привязанное к объектам БД.

### 3. Преобразование в внутреннее представление / Препроцессинг:

- Запрос преобразуется в более удобное для оптимизации внутреннее представление (часто на основе реляционной алгебры - операторы **Select**, **Project**, **Join**).
- *Нормализация предикатов*: Упрощение условий **WHERE** (например, **NOT (A > 5) -> A <= 5**).
- *Преобразование представлений*: Подстановка определения представления (**VIEW**) в основной запрос.
- *Семантическая оптимизация*: Устранение избыточных условий, удаление неиспользуемых столбцов/таблиц, вычисление константных выражений.

### 4. Генерация альтернативных планов выполнения (Plan Generation):

- На основе внутреннего представления оптимизатор генерирует множество *логически эквивалентных* вариантов выполнения запроса. Используются правила преобразования (например, коммутативность/ассоциативность соединений, изменение порядка операций **WHERE**, **JOIN**, **GROUP BY**).



- Для каждого варианта определяется набор *физических операторов* (алгоритмов) для реализации логических операций (какой тип **JOIN** использовать, какой индекс сканировать, как сортировать и т.д.).

#### 5. Оценка стоимости планов (Cost Estimation):

- Для каждого сгенерированного физического плана выполнения оптимизатор *оценивает его стоимость*.
- Используется *статистика* по данным (количество строк в таблицах, селективность индексов, распределение значений - гистограммы) для прогнозирования:
  - *Кардинальность (Cardinality)*: Ожидаемое количество строк на выходе каждой операции плана.
  - *Стоимость (Cost)*: Оценка затрат CPU, I/O, памяти для выполнения операции.
- Стоимость всего плана — сумма стоимостей всех его операций.

#### 6. Выбор оптимального плана (Plan Selection):

- Оптимизатор сравнивает оценки стоимости всех сгенерированных альтернативных планов.
- Выбирается план с **наименьшей оцененной стоимостью**.
- *Примечание*: Из-за сложности перебора всех возможных планов для больших запросов оптимизаторы часто используют эвристики и ограничивают пространство поиска.

#### 7. Кэширование плана (Plan Caching - Optional but Common):

- Выбранный план выполнения (или его скомпилированная форма) помещается в кэш планов СУБД.
- При повторном поступлении *того же самого* (или параметризованного аналогично) запроса, СУБД может использовать кэшированный план, минуя дорогостоящие стадии оптимизации, что значительно ускоряет выполнение.

#### 8. Выполнение запроса (Query Execution):

- Исполнительный механизм СУБД (Query Executor) выполняет выбранный план.
- План состоит из набора физических операторов (Table Scan, Index Seek, Hash Join, Sort, Aggregate и т.д.).
- Данные передаются по конвейеру от одного оператора к другому.
- Результат возвращается клиенту.

## 24. Приведите два-три примера рекомендаций по оптимальному написанию запросов.

Оптимальное написание запросов помогает СУБД эффективно их выполнять:

### 1. Используйте конкретные имена столбцов вместо **SELECT \***: Всегда явно перечисляйте нужные столбцы. **SELECT \***:

- Извлекает лишние данные (перегружает сеть и память).
- Может помешать использованию покрывающих индексов (Covering Index), если индекс не включает все столбцы таблицы.
- Делает запрос хрупким (изменение структуры таблицы может сломать код приложения).
- *Оптимально*: **SELECT CustomerID, FirstName, LastName FROM Customers;**

### 2. Пишите селективные условия в **WHERE** и используйте индексы:

- Формулируйте условия так, чтобы они могли использовать существующие индексы (слева от оператора =, >, <, **BETWEEN**, **IN** должен быть столбец или выражение над ним, с которым работает индекс).
- Избегайте функций и вычислений над индексированными столбцами в условиях **WHERE** (например, **WHERE YEAR(OrderDate) = 2023** не использует индекс на **OrderDate**; лучше **WHERE OrderDate >= '2023-01-01' AND OrderDate < '2024-01-01'**).
- Старайтесь, чтобы условия были как можно более селективными (отбирали меньший % строк).
- *Оптимально (если есть индекс на **LastName**)*: **SELECT \* FROM Employees WHERE LastName = 'Smith';**
- *Неоптимально*: **SELECT \* FROM Employees WHERE UPPER(LastName) = 'SMITH';** (если нет индекса по **UPPER(LastName)**).

### 3. Избегайте ненужных сложных операций на больших объемах данных на стороне сервера:

- *Сортировка (**ORDER BY**)*: Сортировка больших результирующих наборов очень ресурсоемка. Сортируйте только при необходимости. Используйте индексы для поддержки сортировки.



- ***DISTINCT** и **GROUP BY***: Убедитесь, что они действительно нужны. Иногда они появляются из-за неверно написанных **JOIN** (дублирование строк). Агрегация также требует ресурсов.
- *Курсоры*: Почти всегда есть более эффективный операторный (set-based) способ решения задачи, чем построчная обработка курсором. Курсоры очень медленны.
- *Коррелированные подзапросы*: Часто можно переписать с использованием **JOIN** или оконных функций, что обычно эффективнее.
- *Неоптимально*: **SELECT DISTINCT c.CustomerID, c.Name FROM Customers c JOIN Orders o ON c.CustomerID = o.CustomerID**; (если **DISTINCT** нужен только из-за того, что у клиента много заказов, а нужен просто список клиентов, сделавших заказы).
- *Оптимальнее*: **SELECT c.CustomerID, c.Name FROM Customers c WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID = c.CustomerID)**; (часто быстрее, особенно при наличии индексов).

## 25. Какие три основных уровня защиты данных обеспечивает СУБД?

СУБД обеспечивает защиту данных на нескольких уровнях, но три основных фундаментальных уровня:

### 1. Аутентификация (Authentication):

- *Цель*: Убедиться, что пользователь (или приложение) **является тем, за кого себя выдает**.
- *Механизмы*: Проверка учетных данных при подключении к СУБД.
- *Примеры*: Логин/пароль, встроенная аутентификация ОС (Windows Authentication в SQL Server), аутентификация по сертификатам, биометрическая аутентификация, интеграция с системами единого входа (SSO - Single Sign-On) типа Kerberos, LDAP.
- *Защита от*: Неавторизованного доступа к серверу БД.

### 2. Авторизация (Authorization):

- *Цель*: Определить, **что именно** аутентифицированный пользователь **имеет право делать** с данными и объектами в базе данных.
- *Механизмы*: Система привилегий (прав доступа) и ролей.
- *Примеры*:

- *Привилегии*: **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **EXECUTE**, **CREATE**, **ALTER**, **DROP** и т.д. на уровне объектов (таблиц, представлений, процедур) или схем/баз данных.
- *Роли*: Группировка привилегий в именованные роли (например, **db\_datareader**, **db\_datawriter**), которые затем назначаются пользователям или другим ролям. Упрощает управление правами.
- *Модели*: Дискреционное управление доступом (DAC - пользователи владеют объектами и назначают права другим), Ролевое управление доступом (RBAC - права назначаются ролям, роли - пользователям), Мандатное управление доступом (MAC - метки безопасности, реже в коммерческих СУБД).
- *Защита от*: Несанкционированных операций с данными (чтение, изменение, удаление) или объектами БД.

### 3. Шифрование (Encryption):

- *Цель*: Защитить **конфиденциальность данных** от несанкционированного доступа, даже если злоумышленник получил физический доступ к файлам базы данных или резервным копиям (Data at Rest), либо перехватывает сетевой трафик (Data in Transit).
- *Механизмы*:
  - *Шифрование данных на диске (Transparent Data Encryption - TDE)*: Шифрование файлов данных, журналов транзакций и резервных копий на уровне страниц/файлов. Прозрачно для приложений. Защищает от кражи дисков/файлов.
  - *Шифрование столбцов (Column-level Encryption)*: Шифрование отдельных чувствительных столбцов (например, номера кредитных карт, пароли в хэшированном виде - это не шифрование, но тоже защита). Может быть прозрачным или требовать явного шифрования/дешифрования в приложении. Защищает от несанкционированного доступа DBA к конкретным данным.
  - *Шифрование соединений*: Использование протоколов SSL/TLS для шифрования сетевого трафика между клиентом и сервером БД. Защищает от sniffingа сети.
  - *Всегда зашифровано (Always Encrypted)*: Продвинутая схема (MS SQL, другие подхватывают), где ключи шифрования

хранятся *только* на стороне клиента. Сервер БД видит только зашифрованные данные и не имеет ключей для их расшифровки. Максимальная защита от DBA и взлома сервера.

- *Защита от:* Раскрытия конфиденциальной информации при компрометации носителей данных или сетевого канала.

## 26. Для чего в SQL используются операторы GRANT и REVOKE?

Операторы **GRANT** и **REVOKE** являются основными средствами **управления правами доступа (авторизации)** в SQL.

### ◦ GRANT:

- *Назначение:* **Предоставить** пользователю или роли определенные **привилегии** на выполнение операций с объектами базы данных (таблицы, представления, процедуры, схемы и т.д.) или на выполнение определенных действий.
- *Синтаксис (общий вид):*

```
GRANT <privilege> [, <privilege> ...]
ON <object_type> <object_name>
TO <user_or_role> [, <user_or_role> ...]
[WITH GRANT OPTION]; -- Опционально: позволяет получателю
самому предоставлять это право другим
```

- *Примеры:*

- **GRANT SELECT, INSERT ON TABLE Customers TO user\_sales;** (Дать пользователю **user\_sales** право читать и добавлять записи в таблицу **Customers**).
- **GRANT EXECUTE ON PROCEDURE CalculateBonus TO role\_managers;** (Дать роли **role\_managers** право выполнять процедуру **CalculateBonus**).
- **GRANT CREATE TABLE TO user\_dev;** (Дать пользователю **user\_dev** право создавать таблицы в текущей базе/схеме).
- **GRANT SELECT ON SCHEMA Sales TO PUBLIC;** (Дать всем пользователям (**PUBLIC**) право читать объекты в схеме **Sales**).

### ◦ REVOKE:

- **Назначение: Отозвать** (аннулировать) ранее предоставленные **привилегии** у пользователя или роли.
- **Синтаксис (общий вид):**

```
REVOKE [GRANT OPTION FOR] <privilege> [, <privilege> ...]  
ON <object_type> <object_name>  
FROM <user_or_role> [, <user_or_role> ...]  
[CASCADE | RESTRICT]; -- Опционально: поведение при отзыве  
прав, выданных WITH GRANT OPTION
```

- **Примеры:**
  - **REVOKE INSERT ON TABLE Customers FROM user\_sales;**  
(Отозвать право на вставку в **Customers** у **user\_sales**, право **SELECT** остается).
  - **REVOKE ALL PRIVILEGES ON SCHEMA Sales FROM user\_intern;** (Отозвать все права на схему **Sales** у **user\_intern**).
  - **REVOKE EXECUTE ON PROCEDURE CalculateBonus FROM role\_managers;**
- **CASCADE vs RESTRICT:**
  - **CASCADE:** Автоматически отозвать привилегию и у всех тех, кому она была делегирована данным пользователем/ролью с помощью **WITH GRANT OPTION**.
  - **RESTRICT** (по умолчанию в большинстве СУБД): Отменить операцию **REVOKE**, если есть зависимые привилегии, выданные получателем (чтобы избежать "висячих" прав). Требуется сначала отозвать зависимые права вручную. **Итог:** **GRANT** дает права, **REVOKE** забирает их обратно. Это ключевые инструменты для реализации модели безопасности Discretionary Access Control (DAC) в реляционных базах данных.

## 27. Перечислите основные виды индексов, используемые в базах данных.

Индексы — это специальные структуры данных, ускоряющие поиск и извлечение данных из таблиц. Основные виды:

### 1. В-дерево (B-Tree) и его варианты (B+, B):\*

- **Структура:** Сбалансированное дерево поиска. Узлы хранят ключи и указатели на дочерние узлы или на данные. Листовые узлы (в B+

деревьях) содержат все ключи и указатели на фактические строки данных (или их идентификаторы - RID/RowID), связанные в упорядоченный список.

- *Оптимален для:* Точечных запросов (**WHERE key = value**), запросов по диапазону (**WHERE key BETWEEN value1 AND value2**), запросов с префиксом (**WHERE key LIKE 'ABC%'**), сортировки (**ORDER BY key**).
- *Поддерживает:* Уникальность (**UNIQUE** constraint).
- *Распространенность:* Наиболее универсальный и распространенный тип индекса в реляционных СУБД. Часто тип по умолчанию.

## 2. Хеш-индекс (Hash Index):

- *Структура:* Использует хеш-таблицу. Ключ индекса преобразуется хеш-функцией в номер корзины (bucket), в которой хранится указатель на строку(и) с данным ключом.
- *Оптимален для:* **Только точечных запросов** на точное совпадение (**WHERE key = value**). Очень быстр для **=**.
- *Не поддерживает:* Запросы по диапазону, запросы с префиксом, сортировку. Требуется, чтобы хеш-функция давала равномерное распределение.
- *Использование:* Часто используется в in-memory базах данных или для определенных оптимизаций. Реже как основной индекс на диске из-за ограничений.

## 3. Битовый индекс (Bitmap Index):

- *Структура:* Для каждого уникального значения в индексируемом столбце создается битовая карта (bitmap). Каждый бит в карте соответствует строке таблицы (1 - строка содержит это значение, 0 - нет).
- *Оптимален для:* Столбцов с **низкой селективностью** (малое количество уникальных значений - Low Cardinality), например, пол, статус заказа, флаги. Очень эффективны для сложных комбинаций условий **AND**, **OR**, **NOT** над несколькими такими столбцами (быстрое побитовое **AND/OR** карт).
- *Не эффективен:* Для столбцов с высокой селективностью (High Cardinality), для частых операций **INSERT/UPDATE/DELETE** (дорогое обновление битовых карт).
- *Использование:* Широко используется в хранилищах данных (Data Warehousing - OLAP) и системах с преобладанием сложных

запросов и редким обновлением данных.

#### 4. Инвертированный индекс (Inverted Index - для полнотекстового поиска):

- *Структура:* Словарь уникальных слов (лексем) с указанием списка документов (или идентификаторов строк), в которых каждое слово встречается, и часто позиций в тексте.
- *Оптимален для:* **Полнотекстового поиска** (`WHERE CONTAINS(text_column, 'word')`, `LIKE '%word%'` - но гораздо эффективнее `LIKE`), поиска по ключевым словам, ранжирования релевантности.
- *Использование:* Специализированные индексы для текстовых данных. Поддерживаются расширениями СУБД (FTS - Full-Text Search в SQL Server, PostgreSQL; Text Indexes в MongoDB).

#### 5. Кластеризованный индекс (Clustered Index) vs Некластеризованный индекс (Non-Clustered Index): Это не отдельные *виды* структур, а способ организации данных относительно индекса:

- *Кластеризованный:* **Определяет физический порядок хранения данных в таблице.** Сами строки таблицы упорядочены по значениям ключа кластеризованного индекса. Таблица может иметь **только один** кластеризованный индекс. Самый быстрый для диапазонных запросов по ключу. Вставки могут быть медленнее из-за необходимости поддержания порядка.
- *Некластеризованный:* Это отдельная структура (например, B-дерево), которая содержит копию части данных таблицы (ключ индекса + включенные столбцы или указатель на строку данных - RID/Кластеризованный ключ). Таблица может иметь **много** некластеризованных индексов. Быстр для поиска по ключу, но требует дополнительного шага (Bookmark Lookup) для получения полной строки, если не является покрывающим.

#### 6. Покрывающий индекс (Covering Index): Это не отдельный тип структуры, а *свойство* использования индекса. Если индекс содержит все столбцы, запрашиваемые в `SELECT` и используемые в `JOIN/WHERE` для конкретного запроса, то СУБД может выполнить запрос, обратившись только к индексу (`Index Only Scan`), не читая саму таблицу. Это значительно ускоряет запрос.

#### 28. Перечислите характеристики b-дерева, позволяющие отличить его от других древовидных структур.

В-дерево (и его наиболее распространенный вариант В+-дерево) обладает следующими ключевыми характеристиками, отличающими его от других деревьев (бинарных, красно-черных, AVL):

1. **Сбалансированность (Balanced):** Все листовые узлы находятся **на одном уровне** (глубине) от корня. Это гарантирует, что время доступа к любой записи предсказуемо (логарифмично от числа записей  $O(\log n)$ ), независимо от распределения ключей или порядка вставки.
2. **Высокая степень ветвления (High Branching Factor):** Каждый узел (кроме корня и листьев) содержит **много** ключей (от  $M/2$  до  $M$ , где  $M$  - порядок дерева) и указателей на дочерние узлы (от  $M/2$  до  $M$ ). Это резко снижает высоту дерева по сравнению с бинарными деревьями при большом объеме данных, что критично для минимизации дорогостоящих дисковых операций ввода-вывода (I/O) (каждый узел обычно соответствует странице диска).
3. **Хранение ключей и данных разделено (в В+-дереве):** Ключи хранятся во *всех* узлах (корневом, внутренних, листовых). **Фактические данные** (или указатели на них - RowID) хранятся **только в листовых узлах**. Это главное отличие от классического В-дерева, где данные могли храниться и во внутренних узлах. В+-дерево:
  - Делает диапазонные запросы и полное сканирование очень эффективным (все данные последовательно в листьях).
  - Уменьшает размер внутренних узлов, позволяя хранить больше ключей и увеличивая степень ветвления.
  - Упрощает управление данными.
4. **Листовые узлы связаны в список:** Листовые узлы связаны между собой одно- или двунаправленными указателями. Это позволяет очень эффективно выполнять **диапазонные запросы** (**BETWEEN**, **>**, **<**, **ORDER BY**) и **полные сканирования индекса**, так как данные можно последовательно читать с диска, переходя от одного листового узла к следующему по ссылке, без необходимости рекурсивного обхода дерева от корня.
5. **Динамический рост и сжатие:** В-дерево автоматически поддерживает свою сбалансированность при вставках и удалениях за счет операций **расщепления (split)** переполненных узлов и **слияния (merge)** или **перераспределения (redistribution)** ключей в недоополненных узлах. Это обеспечивает стабильную производительность.



6. **Оптимизация для внешней памяти (диска):** Высокая степень ветвления и принцип хранения узлов, соответствующих страницам диска, минимизируют количество дисковых чтений, необходимых для поиска записи (обычно требуется всего 3-4 I/O для таблиц в миллиарды строк). Это исторически основная причина его использования в СУБД.

29. **В чем главное достоинство и в чем главный недостаток использования индексов?**

- **Главное достоинство: Значительное ускорение операций ЧТЕНИЯ данных (**SELECT**), особенно поиска (**WHERE**) и сортировки (**ORDER BY**).**
  - *Механизм:* Индекс позволяет СУБД быстро находить нужные строки, избегая полного сканирования таблицы (Table Scan), которое имеет сложность  $O(n)$ . Используя структуру индекса (обычно B-дерево), СУБД может найти данные за логарифмическое время  $O(\log n)$  или даже константное  $O(1)$  для точечных запросов по хеш-индексу.
  - *Эффект:* Чем больше таблица и селективнее запрос, тем больше выигрыш в производительности от правильного индекса. Индексы критичны для быстрой работы приложений, особенно для поиска по неключевым полям, соединениям (**JOIN**) и сортировке.
- **Главный недостаток: Замедление операций ЗАПИСИ данных (**INSERT**, **UPDATE**, **DELETE**) и увеличение занимаемого места на диске.**
  - *Механизм:* При каждой вставке новой строки, обновлении индексируемого столбца или удалении строки СУБД должна **обновить все индексы**, связанные с этой таблицей. Это включает:
    - Вставку новых ключей (и, возможно, RID или значений включенных столбцов) в структуру индекса (B-дерево, хеш-таблицу и т.д.).
    - Обновление существующих записей индекса при изменении индексируемых столбцов.
    - Удаление записей из индекса при удалении строк.
    - Поддержание сбалансированности структуры индекса (расщепления/слияния узлов в B-дереве).
  - *Эффект:*



- Операции **INSERT/UPDATE/DELETE** становятся **медленнее**, так как помимо изменения самой таблицы требуется обновить все ее индексы. Чем больше индексов на таблице, тем больше накладные расходы на запись.
- **Увеличивается объем занимаемого места на диске:**  
Каждый индекс представляет собой отдельную структуру данных, хранящую копии ключевых столбцов (и, возможно, включенных столбцов) и служебную информацию. Для больших таблиц с многими индексами размер индексов может в разы превышать размер самой таблицы. **Итог:** Индексы — это классический пример компромисса "**Space-Time Tradeoff**". Они жертвуют пространством (диском) и скоростью записи ради значительного выигрыша в скорости чтения. Задача администратора БД и разработчика — создавать индексы *разумно*: только там, где они действительно нужны для ускорения критичных запросов, и избегать избыточных или редко используемых индексов, которые лишь замедляют запись и занимают место.

### 30. Что такое транзакция?

**Транзакция (Transaction)** в контексте баз данных — это **логическая единица работы**, выполняемая над данными, которая должна быть выполнена как единое целое. Это последовательность одной или нескольких операций (SQL-команд - **INSERT**, **UPDATE**, **DELETE**, **SELECT**, вызовы процедур), которые либо выполняются **все**, либо **ни одна из них** не выполняется, оставляя базу данных в согласованном состоянии. Транзакция представляет собой атомарное (неделимое) действие с точки зрения его воздействия на состояние БД. Основные этапы жизни транзакции:

1. **Начало (Begin):** Транзакция начинается неявно (с первой SQL-команды, изменяющей данные) или явно командой **BEGIN TRANSACTION/START TRANSACTION**.
2. **Выполнение (Execution):** Выполняются SQL-операции в рамках транзакции. Изменения пока видны только текущей транзакции (уровень изоляции влияет на видимость для других).
3. **Фиксация (Commit):** Если все операции выполнены успешно и соблюдены все ограничения целостности, транзакция фиксируется командой **COMMIT**. Все изменения, сделанные в транзакции, становятся

постоянными (persistent), видимыми другим транзакциям (согласно уровню изоляции) и не могут быть отменены.

4. **Откат (Rollback):** Если во время выполнения возникает ошибка (нарушение целостности, сбой системы, логическая ошибка в приложении) или транзакция явно отменяется командой **ROLLBACK**, все изменения, сделанные в рамках этой транзакции, отменяются. База данных возвращается к состоянию, которое было на момент начала транзакции (или последней точки сохранения внутри нее - **SAVEPOINT**). Транзакции являются основным механизмом обеспечения согласованности данных при параллельной работе многих пользователей и при возникновении сбоев.

### 31. Перечислите базовые свойства транзакции.

Базовые свойства транзакций известны под акронимом **ACID**:

#### 1. Атомарность (Atomicity):

- Гарантирует, что транзакция выполняется как **единое, неделимое целое**.
- Либо выполняются **все** операции, входящие в транзакцию, и их результаты фиксируются в БД.
- Либо **ни одна** операция из транзакции не выполняется, и БД остается в состоянии, предшествующем началу транзакции.
- *Механизмы:* Журналирование (Write-Ahead Logging - WAL), управление откатом (Undo Logs).

#### 2. Согласованность (Consistency):

- Гарантирует, что успешное завершение транзакции (**Commit**) переводит базу данных из **одного согласованного состояния в другое согласованное состояние**.
- "Согласованное состояние" означает, что данные удовлетворяют всем заданным **правилам целостности** (ограничениям - constraints), бизнес-правилам и семантике предметной области.
- *Важно:* Согласованность обеспечивается *внутри* транзакции (она сама должна писать правильные данные) и *после* ее завершения (фиксации). Если транзакция нарушает правила целостности, она не может быть зафиксирована (выполняется **ROLLBACK**).
- *Механизмы:* Ограничения целостности (**PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE, NOT NULL**), триггеры, семантические проверки в коде приложения/процедур.

### 3. Изолированность (Isolation):

- Гарантирует, что **параллельно** выполняющиеся транзакции **не влияют** друг на друга и не видят промежуточных, незафиксированных результатов работы друг друга.
- Каждая транзакция выполняется так, как будто она **единственная** в системе.
- *Механизмы:* Системы управления параллельным доступом (Concurrency Control): блокировки (Locking - пессимистичные), управление версиями (Multi-Version Concurrency Control - MVCC, оптимистичные), снимки данных (Snapshot Isolation).
- *Уровни изоляции:* СУБД обычно предоставляют несколько уровней изоляции (Read Uncommitted, Read Committed, Repeatable Read, Serializable), которые определяют степень "видимости" изменений из параллельных транзакций и баланс между изоляцией и производительностью.

### 4. Долговечность (Durability):

- Гарантирует, что если транзакция успешно **зафиксирована** (**COMMIT**), то ее результаты **станут постоянными** в базе данных, **даже в случае последующего сбоя системы** (аппаратного, программного, отключения питания).
- Фиксированные изменения не могут быть потеряны.
- *Механизмы:* Запись изменений в **журнал транзакций (Transaction Log)** на **устойчивое запоминающее устройство** (диск) **до** фиксации изменений в основных файлах данных (Write-Ahead Logging - WAL). Использование отказоустойчивого оборудования (RAID). Регулярное резервное копирование журналов транзакций (Transaction Log Backup).

## 32. Какие характеристики позволяют хранилищам данных позиционировать себя как NoSQL системы?

Термин "NoSQL" (Not Only SQL) охватывает широкий спектр систем управления данными, которые отличаются от традиционных реляционных СУБД (RDBMS) следующими ключевыми характеристиками, позволяющими им позиционироваться как NoSQL:

### 1. Отказ от реляционной модели и SQL (как основного языка):

- Используют альтернативные модели данных: документную, ключ-значение, колоночную, графовую.

- Могут не поддерживать SQL вообще или иметь свой специфический язык запросов/API (например, MongoDB Query Language, Cassandra CQL - близкий к SQL синтаксически, но работающий с другой моделью).

## 2. Схема на чтение (Schema-on-Read) / Бессхемность (Schemaless):

- Данные не требуют строго predefined схемы перед записью. Структура может меняться от записи к записи (например, документы в MongoDB могут иметь разные поля).
- Схема определяется и применяется при *чтении* данных приложением, а не при вставке СУБД. Это обеспечивает гибкость для неструктурированных или полуструктурированных данных и быструю эволюцию модели.

## 3. Горизонтальная масштабируемость (Horizontal Scalability) как основной подход:

- Акцент на распределенных архитектурах, способных масштабироваться "вширь" (scale-out) путем добавления дешевых стандартных серверов (нод) в кластер.
- Используют методы сегментирования/партиционирования (Sharding) для распределения данных и нагрузки по нодам.
- Предпочтение перед вертикальным масштабированием (scale-up - увеличение мощности одного сервера).

## 4. Ослабленная согласованность и фокус на доступности и устойчивости к разделению (CAP Theorem & BASE):

- Отказ от строгой согласованности ACID (в масштабе всей распределенной системы) в пользу более слабых моделей согласованности для достижения высокой доступности (Availability) и устойчивости к разделению сети (Partition tolerance) в соответствии с CAP-теоремой.
- Принцип **BASE**:
  - **Basically Available**: Система в основном доступна (даже при частичных сбоях).
  - **Soft state**: Состояние системы может меняться со временем без ввода новых данных (из-за репликации, задержек).
  - **Eventually consistent**: Система гарантированно придет к согласованному состоянию через некоторое время после прекращения ввода новых данных.
- Поддержка настраиваемых уровней согласованности (например, в Cassandra: **ONE**, **QUORUM**, **ALL**).

## 5. Простота дизайна и отказ от сложных функций JOIN и транзакций:

- Многие NoSQL системы сознательно отказываются от сложных операций (многотабличные JOIN, сложные транзакции, охватывающие несколько сущностей/нод) в пользу простоты и производительности. Это требует иного подхода к моделированию данных (денормализация, дублирование, агрегированные документы).
- Транзакции, если поддерживаются, часто ограничены одной записью/документом или одной партицией/шардом.

6. **Открытость и сообщество:** Многие популярные NoSQL системы имеют открытый исходный код и развиваются активным сообществом. **Важно:** Границы между SQL и NoSQL размываются. Многие RDBMS теперь поддерживают горизонтальное масштабирование, JSON/документные типы, ослабленную согласованность. Многие NoSQL системы добавляют поддержку SQL-подобных языков и транзакций. Термин "Not Only SQL" становится все более актуальным.

## 33. Перечислите модели данных, наиболее распространенные в мире NoSQL систем.

Четыре основные модели данных, доминирующие в мире NoSQL:

### 1. Документная (Document):

- *Структура:* Данные хранятся как **документы**, обычно в формате JSON, BSON (бинарный JSON), XML или аналогичных. Документ — это самостоятельная сущность, содержащая пары "ключ-значение" (как словари/объект). Значения могут быть вложенными (массивы, объекты).
- *Примеры СУБД:* **MongoDB**, Couchbase, CouchDB, Amazon DocumentDB, Azure Cosmos DB (Document API).
- *Использование:* Каталоги товаров, профили пользователей, контент-менеджмент, данные с гибкой схемой. Хорошо подходит, когда объект можно представить как самодостаточный документ.
- *Особенности:* Гибкая схема, поддержка вложенности и массивов, индексация по полям документа, запросы по структуре документа. Часто поддерживают агрегации.

### 2. Ключ-Значение (Key-Value):

- *Структура:* Самая простая модель. Данные хранятся как **пары "ключ-значение"**. Ключ — уникальный идентификатор (строка).

Значение — произвольный блок данных (blob), непрозрачный для СУБД. Нет схемы для значения.

- *Примеры СУБД:* **Redis**, **Amazon DynamoDB** (основа), Riak, Memcached, Azure Cosmos DB (Table API), etcd.
- *Использование:* Кэширование, сессии пользователей, профили настроек, корзины покупок, счетчики, очереди (Redis). Идеально для простого доступа по уникальному ключу.
- *Особенности:* Очень высокая производительность и масштабируемость для простых операций **get/put/delete** по ключу. Redis расширяет модель сложными структурами данных (списки, множества, хеши, строки с битовыми операциями) и функциями (pub/sub, Lua скрипты).

### 3. Колоночная (Column-Family / Wide-Column):

- *Структура:* Данные хранятся в **таблицах**, но организованы по **колонок** (семействам колонок - Column Families), а не по строкам. Каждая строка может иметь разный набор колонок (разреженная матрица). Физически колонки, относящиеся к одному семейству, хранятся вместе на диске.
- *Модель данных:* Часто описывается как "двумерный распределенный хеш-массив" или "семейства строк, где каждая строка — это отображение ключ-значение". Основные понятия: Keyspace (аналог БД), Таблица (Column Family), Ключ строки (Row Key), Колонки (имя колонки + значение), временные метки версий.
- *Примеры СУБД:* **Apache Cassandra**, **HBase**, **Amazon Keyspaces** (Managed Cassandra), Google Bigtable, ScyllaDB, Azure Cosmos DB (Cassandra API).
- *Использование:* Аналитика в реальном времени (time-series данные), журналы событий, большие таблицы с миллиардами строк, рекомендательные системы, интернет вещей (IoT). Хорошо подходит для запросов по ключу строки и диапазонам ключей, агрегации по колонкам.
- *Особенности:* Отличная горизонтальная масштабируемость и производительность записи. Эффективное сжатие данных (похожие данные в колонках). Оптимизирована для чтения определенных колонок в строках. Поддержка TTL (время жизни данных). Часто используют SQL-подобный синтаксис (CQL в Cassandra).

### 4. Графовая (Graph):

- *Структура:* Данные представляются как **узлы (вершины - Vertices)** и **связи (ребра - Edges)** между ними. Узлы представляют сущности (люди, продукты, места), ребра — отношения между ними (дружит с, купил, находится в). И узлы, и ребра могут иметь свойства (атрибуты).
- *Примеры СУБД:* **Neo4j**, **Amazon Neptune**, JanusGraph, ArangoDB (мультимодельная, но сильна в графах), OrientDB, Azure Cosmos DB (Gremlin API).
- *Использование:* Социальные сети (друзья друзей), рекомендательные системы ("Купили также"), обнаружение мошенничества (выявление сложных схем), сети доставки (оптимальные маршруты), онтологии и семантические сети. Идеально для сценариев, где критичны связи и их обход (traversal).
- *Особенности:* Оптимизированы для запросов, исследующих связи (например, "найди всех друзей друзей глубиной 3"). Используют специализированные языки запросов (Cypher в Neo4j, Gremlin - стандарт Apache TinkerPop). Могут быть медленны для других типов запросов.