DB exam

Вопросы к экзамену:

1. **База данных как компонент информационной системы:**
   - Понятие базы данных (БД);
   - Системы управления базами данных (СУБД);
   - Метаданные (МД).
2. **Модель "сущность-связь":**
   - Классификации сущностей, атрибутов и связей;
   - Нотации для представления модели "сущность-связь";
   - Логическая и физическая модели данных:
     - Содержание уровней.
3. **Модели данных:**
   - Составы моделей;
   - Преимущества и недостатки различных моделей данных.
4. **Реляционная модель данных:**
   - Терминология;
   - Свойства отношений;
   - Виды ключей;
   - Реализация различных типов связей;
   - Виды целостности;
   - Операции реляционной алгебры.
5. **Структура и порядок выполнения предложения `SELECT` в SQL:**
   - План выполнения запросов;
   - Оптимизация запросов.
6. **Соединения отношений в SQL:**
   - Типы соединений (`INNER JOIN` , `LEFT JOIN` , `RIGHT JOIN` , `FULL JOIN` и др.).
7. **Нормализация реляционной модели:**
   - Избыточность данных;
   - Аномалии в данных;
   - Суть метода нормальных форм;
   - Виды зависимостей между атрибутами;
   - Нормальные формы:
     - Первая нормальная форма (`1НФ`);
     - Вторая нормальная форма (`2НФ`);
     - Третья нормальная форма (`3НФ`);
     - Бойс-Кодд нормальная форма (`BCНФ`);

        - Четвертая нормальная форма (`4НФ`);
    - Примеры нормализации.
8. **Повышение производительности:**
    - Индексы;
    - Представления.
9. **Транзакции:**
    - Понятие транзакции;
    - Свойства транзакции (`ACID`):
        - `Atomicity` (атомарность);
        - `Consistency` (согласованность);
        - `Isolation` (изоляция);
        - `Durability` (долговечность);
    - Проблемы конкурирующих транзакций:
        - Грязное чтение;
        - Потерянные изменения;
        - Неповторяющееся чтение;
        - Фантомное чтение;
    - Методы решения проблем;
    - Уровни изоляции транзакций.
10. **Идентификация и аутентификация:**
    - Понятия идентификации и аутентификации;
    - Виды контроля доступа.
11. **Распределенные базы данных:**
    - Определение;
    - Стратегии размещения данных:
        - Централизованное;
        - Реплицированное;
        - Фрагментированное;
    - Достоинства и недостатки каждой стратегии.
12. `NoSQL` **базы данных:**
    - Характеристики `NoSQL` ;
    - Виды `NoSQL` баз данных:
        - Документные;
        - Колонночные;
        - Графовые;
        - Базы типа "ключ-значение";
    - Предпосылки появления `NoSQL` .
13. **CAP-теорема:**
    - Расшифровка `CAP` :

    - Consistency (согласованность);
    - Availability (доступность);
    - Partition tolerance (устойчивость к разделению);
- Ограничения CAP-теоремы.

Ответы на билеты:

1) База данных как компонент информационной системы:

База данных (`БД`) - это организованное хранилище данных, управляемое системой управления базами данных (*СУБД*), для удобного хранения, поиска и обработки информации
Система управления базами данных (`СУБД`) - это программное обеспечение, предназначенное для создания, управления, обновления и работы с базами данных. СУБД обеспечивает взаимодействие пользователей, приложений и самой базы данных, предоставляя инструменты для выполнения некоторых типов операций (*хранение данных, извлечение данных, модификация данных, управление доступом, обеспечение целостности*)
Метаданные (`МД`) - это данные о данных, которые описывают структуру, содержание и правила работы с базой данных. Они включают информацию о таблицах, полях, типах данных, связях между таблицами, ограничениях (например, ключах и уникальности), пользовательских правах доступа и индексах. Метаданные хранятся в системных таблицах СУБД и используются для управления данными, обеспечения целостности, ускорения работы запросов и упрощения взаимодействия пользователей и приложений с базой данных

Роль базы данных в информационной системе:

1. **Хранение данных** - БД служит основным хранилищем данных, обеспечивая надежное и централизованное хранение больших объемов информации;
2. **Управление данными** - позволяет структурировать, сортировать, фильтровать и анализировать данные для их последующего использования;
3. **Обеспечение доступа** - БД обеспечивает многопользовательский доступ с разграничением прав, что особенно важно в корпоративных и распределенных системах;
4. **Поддержка принятия решений** - информационные системы используют данные из БД для анализа, прогнозирования и автоматизации принятия решений;
5. **Интеграция систем** - БД служит связующим звеном между различными компонентами и приложениями информационной системы, поддерживая стандартизированный обмен данными.

Зачем используют базы данных:

1. **Централизация данных** - все данные собираются в одном месте, упрощая их управление и обработку;
2. **Обеспечение целостности данных** - поддержка строгих ограничений (например: уникальности, связности) помогает избежать ошибок;
3. **Эффективность обработки данных** - использование индексов, запросов и транзакций позволяет быстро обрабатывать большие объемы информации;
4. **Удобство доступа** - пользователи могут работать с данными через удобные интерфейсы, такие как формы, запросы и отчеты;
5. **Масштабируемость** - БД могут расширяться с ростом объемов данных и количества пользователей.

Преимущества использования базы данных в информационной системе:

1. **Централизация управления** - уменьшение дублирования информации и конфликтов данных;
2. **Многопользовательский режим** - возможность одновременной работы большого числа пользователей;
3. **Повышение надежности**- за счет резервного копирования, отказоустойчивости и механизмов восстановления;
4. **Гибкость** - легкость изменения структуры данных под изменяющиеся требования;
5. **Безопасность** - предоставление механизмов шифрования, контроля доступа и аудита операций.

---

2) Модель "сущность связь":

Модель "сущность-связь" (Entity-Relationship Model, ER-модель) - это модель данных, предназначенная для концептуального проектирования баз данных. Она используются для описания предметной области с помощью сущностей, их атрибутов и связей между ними. *ER-модель* визуализируется через диаграммы (`ERD`), которые помогают формализовать структуру данных перед созданием базы данных

Сущности:
Сущность (entity) - это объект, обладающий именем и набором характеристик (атрибутов)

```
*Пример:*
- "Студент"
- "Курс"
- "Группа"
```

Слабая сущность - это сущность, которой недостаточно собственных атрибутов для уникальной идентификации

```
*Пример:*
Название группы уникально только в рамках одного университета (для
уникальности надо учитывать и университет)
```

Сильная сущность - это сущность, которая имеет достаточное количество собственных атрибутов для уникальной идентификации её экземпляров без зависимости от других сущностей

```
*Пример:*
Студент имеет уникальный идентификатор `ISU`, который позволяет однозначно
отличить одного студента от другого независимо от других сущностей
```

Атрибуты сущностей:
Атрибут - это характеристика сущности, включающая имя и домен значений
Типы атрибутов:

1. **Простой** - атомарное значение, например `id`;
2. **Составной** - состоит из нескольких значений, пример (паспорт).

```
{
        series: char(4),
        number: char(6),
        issue_date: timestamp
}
```

Свойства атрибутов:

- **M (Mandatory)** - обязательный атрибут;
- **O (Optional)** - необязательный атрибут;
- **PK (Primary Key)** - атрибут, образующий основной ключ;
- **Kn (Alternate Key)** - дополнительный ключ `n`.

Связи:
Связь (relation) - это отношение между двумя или более сущностями, которое описывает, как эти сущности взаимодействуют друг с другом
Характеристика связей:

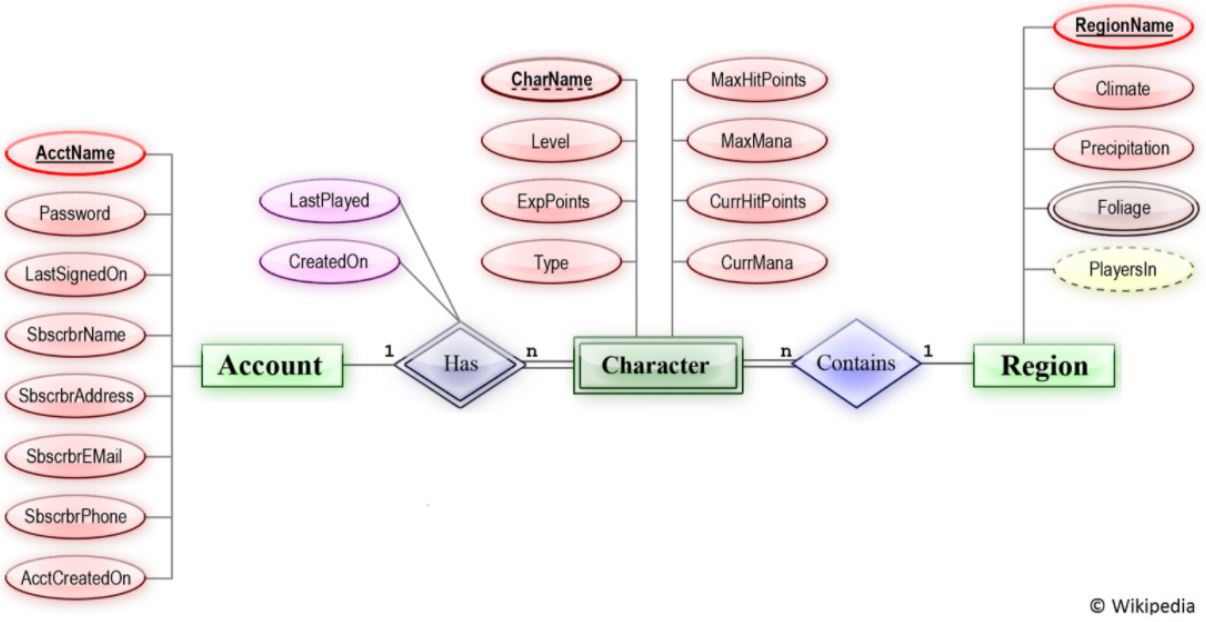
- **Имя** - определяет тип отношения между сущностями (например: "принадлежит");
- **Связываемые сущности и их роли**;
- **Тип связи** - определяется кардинальностью и модальностью концов связи.
  Типы связей:

1. **Один-к-одному (1:1)** - каждая сущность A связана с одной сущностью B;
2. **Один-ко-многим (1:N)** - одна сущность A связана с несколькими сущностями B;
3. **Многие-ко-многим (M:N)** - несколько сущностей A связаны с несколькими сущностями B. Для реализации такой связи обычно используется промежуточная сущность (сущность-связь) C.

Ассоциация - это многостороняя связь, которая может иметь неключевые атрибуты и произвольное количество концов

Графические нотации:
Нотация Питера Чена:

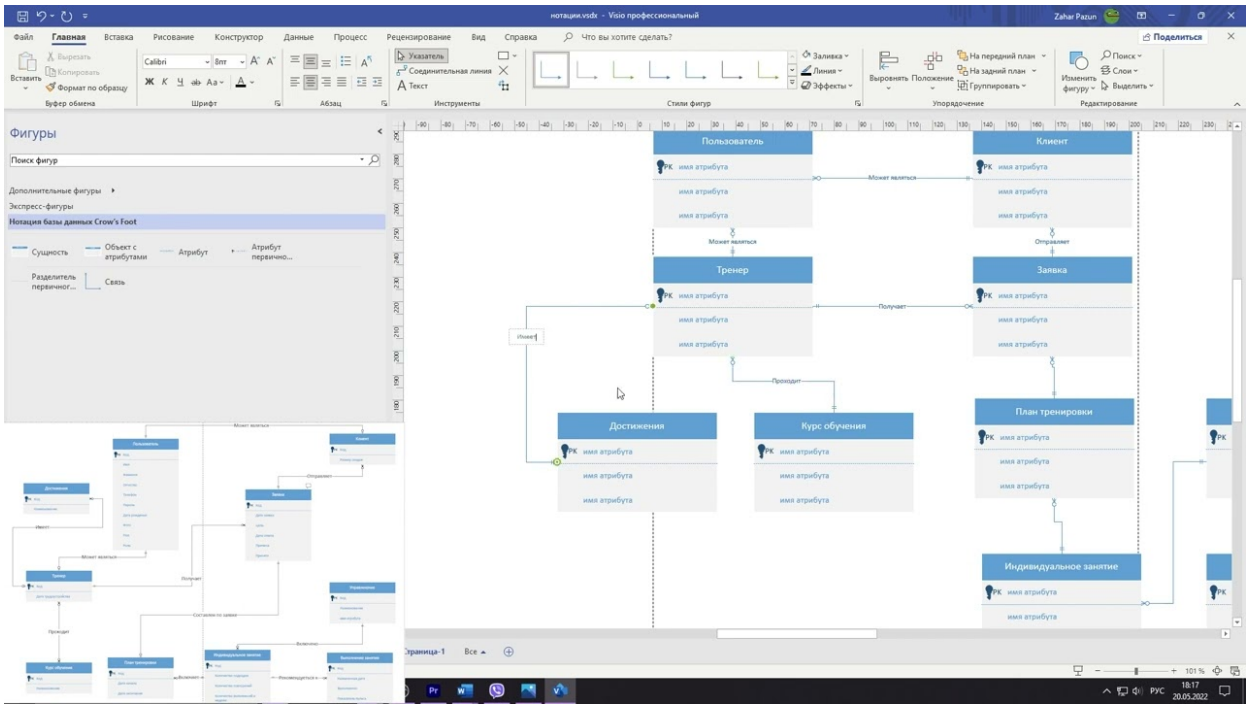
- *Сущности* - прямоугольники;
- *Связи* - ромбы;
- *Атрибуты* - овалы;
- *Обязательность* - пунктирная линия для необязательных связей.



Crow's Foot:

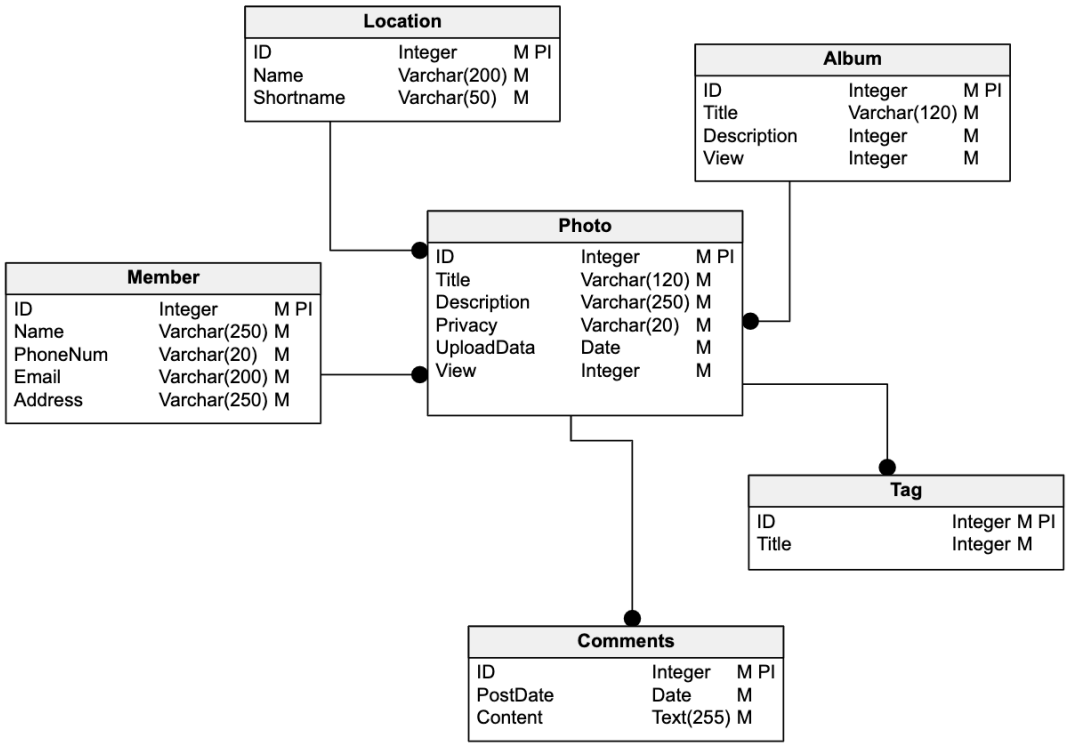
- *Сущности* - прямоугольники;
- *Связи* - линии с "лапками вороны" для обозначения кардинальности;

- *Обязательность* - круг для необязательных связей.



UML:

- Используются классы с атрибутами;
- Кардинальность указывается через диапазон (например, 1..n);
- Подходит для сложных систем с большим количеством ограничений.



Концептуальная модель данных - это высокоуровневое абстрактное описание данных и их взаимосвязей, которое используется для понимания предметной области и её структуры. Концептуальная модель фокусируется на описании сущностей, их

атрибутов и связей, но не учитывает технические или физические аспекты хранения данных

Логическая и физическая модель данных:
Логическая модель данных - это расширение концептуальной модели данных. Она формализует сущности, атрибуты, ключи и взаимосвязи, отражая бизнес-правила и требования к данным. Логическая модель не зависит от конкретной системы управления базами данных (*СУБД*)

Содержание логической модели:

- **Сущности** - определяются все основные объекты предметной области (например, "Студент", "Группа");
- **Атрибуты** - устанавливаются характеристики сущностей (например, ISU, FirstName, LastName);
- **Ключи** - определяются основные и альтернативные ключи, обеспечивающие уникальность данных;
- **Связи** - формализуются отношения между сущностями, их типы (1:1, 1:N, M:N) и кардинальность;
- **Ограничения**- указываются ограничения, такие как уникальность, обязательность атрибутов.

Цель логической модели:

- Формализовать структуру данных предметной области;
- Установить взаимосвязи и бизнес-правила;
- Ориентироваться на высокоуровневую спецификацию.

```
*Пример:*
- Сущность: студент
        - Атрибуты: ISU (PK), FirstName, LastName, BirthDate, Email
- Сущность: группа
        - Атрибуты: GroupID (PK), GroupName
- Связь: Группа (1) - Студент (N) (Студент состоит в одной группе, группа
включает множество студентов)
```

Физическая модель данных - это описание структуры базы данных с учётом конкретной СУБД. Она определяет таблицы, столбцы, типы данных, индексы, ключи, ограничения и параметры для оптимизации производительности базы данных

Содержание физической модели данных:

- **Таблицы** - определяются таблицы, соответствующие сущностям из логической модели;

- **Столбцы** - атрибуты сущностей преобразуются в столбцы с указанием типов данных (например, `VARCHAR`, `INT`, `DATE`);
- **Индексы** - создаются для повышения производительности запросов;
- **Ограничения** - реализуются ограничения, такие как первичные и внешние ключи, уникальность, проверки (`CHECK`);
- **Связи** - применяются ограничения внешнего ключа (`FOREIGN KEY`), чтобы обеспечить целостность данных;
- **Производительность** - учитываются аспекты индексации, денормализации, стратегии хранения (например, использование кластерных или некластерных индексов).

Цель физической модели данных:

- Создать физическую реализацию базы данных, соответствующую логической модели;
- Оптимизировать структуру данных для производительности в конкретной СУБД.

```
*Пример:*
- Таблица: Students
        - Поля:
                - ISU UUID PRIMARY KEY,
                - FirstName VARCHAR(50),
                - LastName VARCHAR(50),
                - BirthDate DATE,
                - Email VARCHAR(100) UNIQUE
- Таблица: Groups
        - Поля:
                - GroupID UUID PRIMARY KEY,
                - GroupName VARCHAR(50)
- Связь:
        - FOREIGN KEY (GroupID) REFERENCES Groups(GroupID)
        - Индексы для оптимизации: INDEX (LastName), INDEX (GroupID)
```

Основные отличия:
Логическая модель:

- Определяет сущности, атрибуты и связи;
- Не зависит от конкретной СУБД;
- Ориентирована на представление данных и бизнес-правил.

Физическая модель:

- Включает типы данных, индексы, ограничения, структуры хранения;
- Учитывает особенности СУБД;
- Ориентированная на реализацию и производительность

3) Модели данных: составы моделей, преимущества и недостатки:

Модели данных - это концептуальные структуры, используемые для описания данных, их организации и взаимосвязей. Они подразделяются на логические, физические и прочие модели

Типы моделей данных:
Логические:

- Иерархическая модель;
- Сетевая модель;
- Реляционная модель;
- Модель "сущность-связь";
- Объектно-ориентированная модель;
- Документная модель.
  Физические:
- Плоская модель;
- Табличная модель;
- Инвертированная модель.

Логические модели:

1. **Иерархическая модель** - данные организуются в виде дерева, где каждый узел представляет собой объект, а ветви - связи между объектами. У каждого объекта есть один родитель, и множество детей. Это напоминает структуру файловой системы
   Преимущества:
   - Естественное отображение многих реальных процессов и отношений;
   - Упрощённая навигация, если запросы следуют иерархической структуре.
     Недостатки:
   - *Дублирование данных* - если объект должен быть связан с несколькими родителями, возникают копии данных;
   - *Сложность изменения* - изменения в структуре дерева требуют больших переработок;
   - *Ограничения поиска* - сложно выполнять запросы, которые не соответствуют структуре дерева.

```
*Пример:*
Организация
├── Отдел: Продажи
│   ├── Подотдел: Региональный
```

```
│   │   └── Сотрудник: Иванов Иван Иванович
│   └── Подотдел: Международный
│       └── Сотрудник: Петров Петр Петрович
├── Проект: Разработка ПО
│   ├── Сотрудник: Иванов Иван Иванович
│   └── Связан с проектом: Тестирование ПО
└── Проект: Тестирование ПО
    ├── Сотрудник: Петров Петр Петрович
    └── Связан с проектом: Разработка ПО
```

2. **Сетевая модель** - данные организуются в виде графа, где узлы могут иметь несколько родителей и несколько детей, именно это позволяет создавать сложные связи
   Преимущества:
   - *Экономия памяти* - связи хранятся отдельно от данных;
   - *Гибкость* - позволяет моделировать сложные отношения.
     Недостатки:
   - *Медленный обход* - при больших графах поиск может быть медленным;
   - *Сложность администрирования* - управления сложными сетями требует высокой квалификации.

```
*Пример:*
- Узел: Сотрудник "Иванов Иван Иванович"
  -> Связан с узлом: Проект "Разработка ПО"
- Узел: Проект "Разработка ПО"
  -> Связан с узлом: Проект "Тестирование ПО"
- Узел: Проект "Тестирование ПО"
  -> Связан с узлом: Сотрудник "Петров Петр Петрович"
```

3. **Реляционная модель** - данные представлены в виде таблиц, строки представляют экземпляры сущностей, а столбцы их атрибуты, а связи между таблицами устанавливаются между таблицами через первичные и внешние ключи
   Преимущества:
   - Простота работа с данными через язык `SQL`;
   - Высокая степень нормализации данных;
   - Лёгкость в добавлении новых данных без изменения структуры.
     Недостатки:
   - *Проблемы масштабирования* - добавление большого количества новых атрибутов или данных может замедлить работу;
   - Ограниченные возможности для моделирования сложных объектов.

```
*Пример:*
Таблица "Сотрудники":
```

```
– ID | ФИО       | Телефон      | Отдел
– 1  | Иванов    | +7(777)...   | Продажи
```

4. **Объектно-ориентированная модель (ООП)** - данные хранятся в виде объектов, которые объединяют атрибуты (поля) и методы. Эта модель напрямую связана с объектно-ориентированным программированием
   Преимущества:
   - Естественная интеграция с языками ООП;
   - Поддержка таких механизмов, как наследование, полиморфизм, инкапсуляция.
     Недостатки:
   - *Нарушение целостности данных* - если данные изменяются вне методов объекта;
   - Сложности миграции из других моделей, особенно реляционной.

```
*Пример:*
Объект "Сотрудник":
– Поля: ID, Имя, Отдел
– Метод: "Получить контакты"
```

5. **Документная модель** - данные хранятся в виде документов, чаще всего в формате JSON, XML и т.д. Каждый документ содержит всю информацию об объекте
   Преимущества:
   - *Гибкость* - структура документа может изменяться для каждого экземпляра;
   - Удобство работы с полуструктурированными данными.
     Недостатки:
   - Сложность обеспечения целостности данных;
   - Низкая производительность при сложных запросах.

```
{
        "ID": 1,
        "Имя": "Иван",
        "Отдел": "Продажи",
        "Контакты": ["+7(777)..."]
}
```

Физические модели:

1. **Плоская модель** - данные хранятся в виде простых текстовых файлов, где каждая строка представляет запись, а столбцы разделены разделителями
   Преимущества:
   - Простота реализации;

- Отсутствие необходимости в сложной структуре.
  Недостатки:
- Дублирование данных;
- Сложность управления большими объёмами данных.

```
*Пример:*
Таблица:
- "Иванов Иван", "Продажи", "+7(777)..."
```

2. **Табличная модель** - данные организуются в таблицы, как в реляционной модели, но без строгих ограничений на типы данных и связи
   Преимущества:
   - Простота поиска данных;
   - Более гибкая структура.
     Недостатки:
   - Отсутствие строгих гарантий целостности данных;
   - Сложности при больших объёмах информации.

```
*Пример:*
Таблица "Инвентарь":

- **Колонки:** ID, Наименование, Характеристики, Количество.
- **Данные:**

| ID  | Наименование | Характеристики          | Количество |
|-----|--------------|-------------------------|------------|
| 1   | Ноутбук      | {"RAM": 16GB}           | 10         |
| 2   | Стол         | {"Материал": "Дерево"}  | 5          |
| 3   | Мышь         | {"Тип": "Беспроводная"} | 50         |
```

В реляционной модели "Характеристики" разложили бы на отдельные таблицы с чётко определёнными типами данных

---

4) Реляционная модель данных

Реляционная модель данных - это логическая модель, основанная на математическом понятии "отношение". Она описывает данные в виде таблиц, где строки (кортежи) представляют экземпляры, а столбцы - их свойства (атрибуты). На основе **реляционной модели данных** строятся реляционные базы данных, являющиеся стандартом в большинстве современных СУБД

Отношение - множество строк (кортежей), представляющее объект реального мира;
Кортеж - строка в таблице, представляющая один экземпляр объекта;
Атрибут - столбец в таблице, представляющий характеристику объекта;
Домен - множество значений, которые может принимать атрибут;
Схема отношения - структура таблицы, задающая её атрибуты и домены.

Свойства отношений:

1. Каждый атрибут имеет уникальное имя;
2. Кортежи в отношении уникальны, дублирование строк не допускается;
3. Порядок атрибутов и кортежей не имеет значения;
4. Каждое значение атрибута должно принадлежать его домену;
5. Атрибуты должны быть атомарными (не делимыми).

Виды ключей:

1. **Кандидатный ключ** - набор атрибутов, значения которых уникальны для каждого кортежа и минимальны (содержат только необходимое для уникальности);
2. **Первичный ключ (Primary Key)** - один из кандидатных ключей, выбранный для одназначной идентификации кортежей;

```
*Пример:*
В таблице "Студенты" первичным ключом может быть "ISU".
```

3. **Внешний ключ (Foreign Key)** - атрибут или набор атрибутов, указывающий на первичный ключ другой таблицы, реализующий связь между таблицами

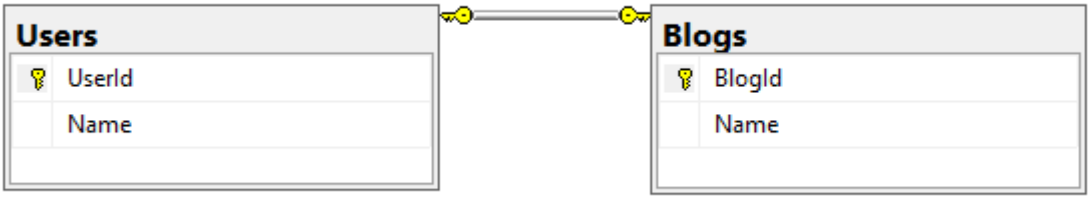
```
*Пример:*
В таблице "Записи на курсы" внешний ключ "ISU" указывает на первичный ключ таблицы "Студенты".
```

4. **Суррогатный ключ** - искусственно создаваемый уникальный идентификатор, который не имеет физического значения

Типы связей:

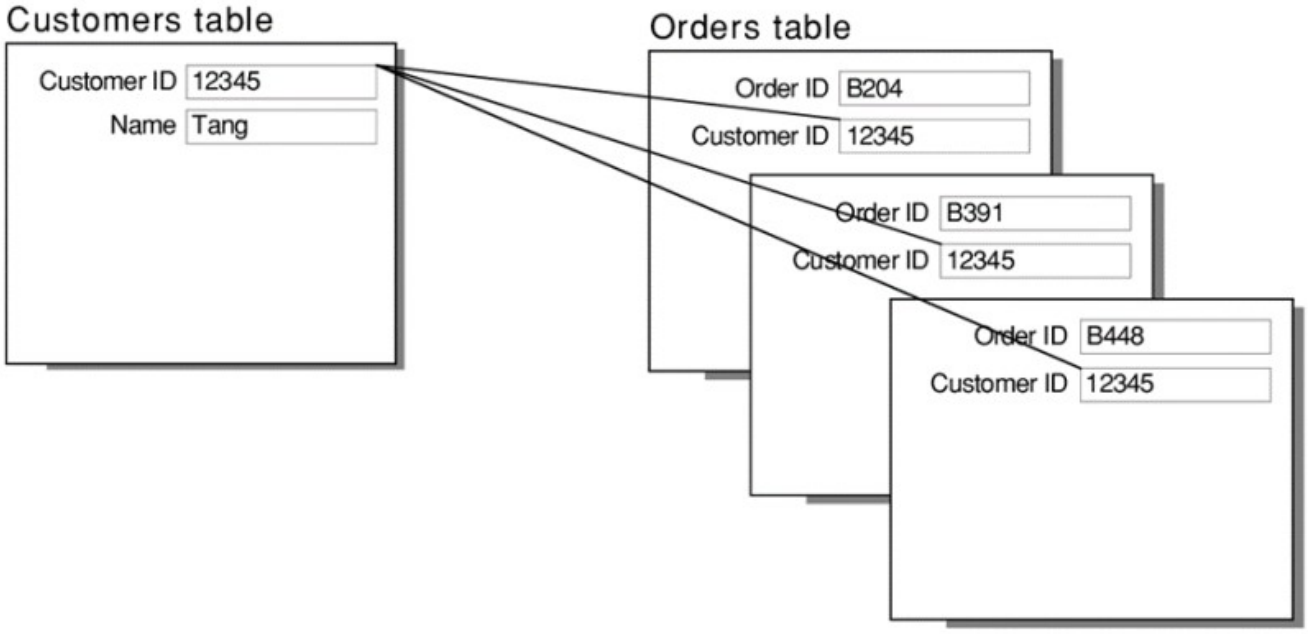
1. **Один-к-одному (1:1)** - каждая сущность A связана с одной сущностью B Реализуется через внешние ключи, ссылающиеся на первичный ключ другой таблицы

```
*Пример:*
Таблицы "Человек" и "Паспорт", где каждому человеку соответствует один паспорт.
```



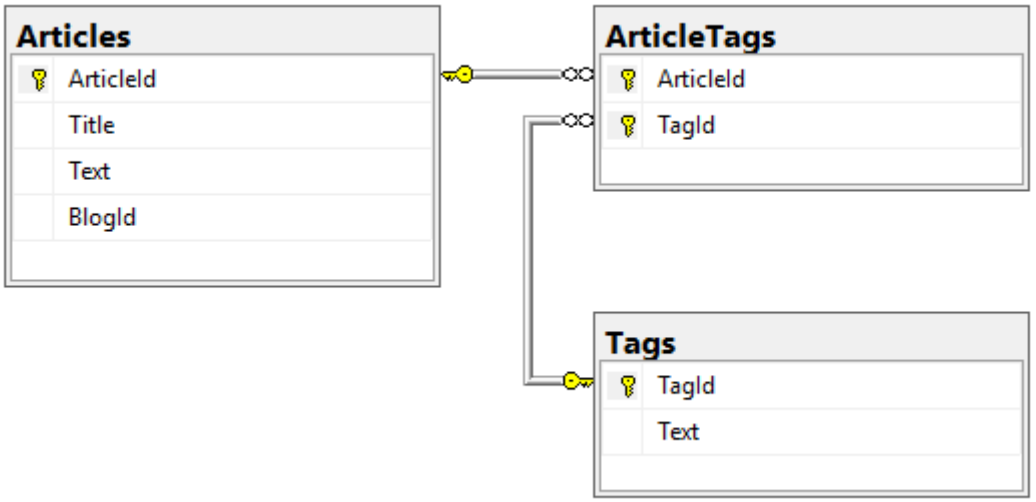
2) **Один-ко-многим (1:N)** - одна сущность A связана с несколькими сущностями B
Реализуется добавлением внешнего ключа в таблицу, представляющую множество

```
*Пример:*
Таблицы "Преподаватель" и "Курсы", где один преподаватель ведёт несколько
курсов.
```



3) **Многие-ко-многим (M:N)** - несколько сущностей A связаны с несколькими сущностями B
Реализуется через промежуточную таблицу, содержащую внешние ключи на обе таблицы

```
*Пример:*
Таблицы "Студенты" и "Курсы" связаны через таблицу "Записи на курсы".
```



Виды целостности:

1. **Сущностная целостность** - значения первичного ключа должны быть уникальными и не могут быть `NULL`
2. **Ссылочная целостность** - значения внешнего ключа должны соответствовать существующему значению первичного ключа связанной таблицы или быть `NULL`
3. **Доменная целостность** - значения атрибутов должны принадлежать их домену
4. **Бизнес-целостность** - пользовательские ограничения, отражающие бизнес-правила (например: "возраст студента не может быть отрицательным")

Преимущества реляционной модели данных:

1. **Простота** - табличный формат данных интуитивно понятен и использование SQL облегчает доступ к данным
2. **Целостность** - механизмы первичных внешних ключей обеспечивают целостность данных
3. **Независимость данных** - логическая модель отдельна от физического уровня, что упрощает изменения в структуре данных
4. **Гибкость запросов** - операции реляционной алгебры позволяют строить сложные запросы для анализа данных
5. **Поддержка транзакций** - обеспечивается согласованность данных в многопользовательских системах

Недостатки реляционной модели данных:

1. **Сложность масштабирования** - горизонтальное масштабирование реляционных баз данных сложнее, чем у `NoSQL` решений
2. **Избыточность** - для поддержания нормализации данных требуется больше таблиц, что может замедлять операции соединений
3. **Сложность работы с полуструктурированными данными** - реляционные базы данных плохо подходят для хранения JSON, XML, изображений и других сложных

структур
4. **Производительность** - при высоких нагрузках операции `JOIN` могут становится узким местом

Реляционная алгебра:

1. **Проекция** Па1 ...an (R)
   Результатом проекции является новое отношение, содержащее вертикальное подмножество исходного отношения, создаваемое путем извлечения указанных атрибутов и исключения из результата атрибутов-дубликатов.

```
SELECT Name, Age FROM Students;
```

2. **Выборка** σ(предикат)(R)
   Результатом выборки является новое отношение, которое содержит только те кортежи из исходного отношения, которые удовлетворяют заданному условию (предикату).

```
SELECT * FROM Students WHERE Age > 20;
```

3. **Объединение** R ∪ S
   Объединение двух отношений R и S определяет новое отношение, которое включает все кортежи, содержащиеся только в R, все кортежи, содержащиеся только в S, и кортежи, содержащиеся и в R, и в S, исключая дубликаты. Объединение возможно только для совместных отношений, имеющих одинаковое количество атрибутов и одинаковый домен.

```
SELECT Name, Age FROM Students
UNION
SELECT Name, Age FROM Alumni;
```

4. **Разность** R - S
   Разность двух отношений R и S определяет новое отношение, состоящее из кортежей, которые есть в R, но отсутствуют в S. Разность возможна только для совместных отношений.

```
SELECT Name, Age FROM Students
EXCEPT
SELECT Name, Age FROM Graduates;
```

5. **Пересечение** R ∩ S
   Операция пересечения определяет отношение, содержащее кортежи, которые

находятся как в R, так и в S. Пересечение возможно только для совместных отношений.

```
SELECT Name, Age FROM Students
INTERSECT
SELECT Name, Age FROM Alumni;
```

6. **Декартово произведение** R × S
   Декартово произведение определяет новое отношение, которое является результатом соединения каждого кортежа из отношения R с каждым кортежем из отношения S.

```
SELECT * FROM Students, Courses;
```

7. **Тета-соединение** R ⋈θ S
   Определяет отношение, содержащее кортежи из декартового произведения R × S, удовлетворяющие предикату F = R.a_i θ S.b_j, где θ - одна из операций сравнения { >, <, =, ... }.

```
SELECT Students.Name, Courses.CourseName
FROM Students
JOIN Courses
ON Students.GroupID = Courses.GroupID
WHERE Courses.Credits > 3;
```

8. **Экви-соединение**
   Это тета-соединение, где θ = "=".

```
SELECT Students.Name, Groups.GroupName
FROM Students
JOIN Groups
ON Students.GroupID = Groups.GroupID;
```

9. **Естественное соединение** R ⋈ S
   Это соединение по эквивалентности двух отношений, выполненное по всем общим атрибутам, с исключением одного экземпляра каждого общего атрибута в результате.

```
SELECT *
FROM Students
NATURAL JOIN Groups;
```

10. **Левое внешнее соединение** R ⟕ S
    Это естественное соединение, при котором в результирующее отношение включаются также кортежи отношения R, не имеющие совпадающих значений в общих атрибутах отношения S.

```
SELECT Students.Name, Groups.GroupName
FROM Students
LEFT JOIN Groups
ON Students.GroupID = Groups.GroupID;
```

11. **Полусоединение** R ▷θ S
    Это отношение, содержащее кортежи R, которые входят в тета-соединение R и S.

```
SELECT DISTINCT Students.Name
FROM Students
JOIN Enrollments
ON Students.StudentID = Enrollments.StudentID;
```

12. **Деление**:
    Получение отношения, в котором строки удовлетворяют условиям всех строк второго отношения

```
SELECT DISTINCT Name
FROM Enrollments E1
WHERE NOT EXISTS (
    SELECT CourseID
    FROM Courses
    WHERE NOT EXISTS (
        SELECT *
        FROM Enrollments E2
        WHERE E2.StudentID = E1.StudentID AND E2.CourseID =
Courses.CourseID
    )
);
```

---

5) Структура и порядок выполнения предложения SELECT в SQL:

Порядок команд при написании SQL-запросов:

1. **SELECT**
2. **DISTINCT**

3. **FROM**
4. **JOIN**
5. **WHERE**
6. **GROUP BY**
7. **HAVING**
8. **ORDER BY**
9. **LIMIT**

Порядок их выполнения:

1. **FROM**
2. **JOIN**
3. **WHERE**
4. **GROUP BY**
5. **HAVING**
6. **SELECT**
7. **DISTINCT**
8. **ORDER BY**
9. **LIMIT**

```sql
-- Пример:
SELECT DISTINCT e.Name, SUM(e.Salary) AS TotalSalary
FROM Employees AS e
JOIN Departments d ON e.DepartmentID = d.DepartmentID
WHERE d.DepartmentName = 'IT'
GROUP BY e.Name
HAVING SUM(e.Salary) > 5000
ORDER BY TotalSalary DESC
LIMIT 10;
```

Порядок выполнения:

1. `FROM Employees` — основная таблица для выборки данных.
2. `JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID` — объединение таблиц Employees и Departments по ключу DepartmentID.
3. `WHERE d.DepartmentName = 'IT'` — фильтрация строк по названию департамента.
4. `GROUP BY e.Name` — группировка данных по имени сотрудников.
5. `HAVING SUM(e.Salary) > 5000` — фильтрация групп, где общая сумма зарплат превышает 5000.
6. `SELECT DISTINCT e.Name, SUM(e.Salary) AS TotalSalary` — выборка уникальных имен сотрудников и их общей зарплаты.

7. `ORDER BY TotalSalary DESC` — сортировка результатов по убыванию суммы зарплат.
8. `LIMIT 10` — ограничение результата первыми 10 строками.

План выполнения запроса - это описание того, какие шаги СУБД предпримет для выполнения SQL-запроса. Это ключевой инструмент для анализа производительности запросов, позволяющий понять, как запрос будет обработан базой данных. В `PostgreSQL` для анализа плана используется команда `EXPLAIN`

План показывает:

- Какие методы доступа к данным будут использоваться (сканирование таблицы, использование индекса и т.д.);
- Какие алгоритмы соединения применяются (Nested Loop, Hash Join, Merge Join);
- Как выполняются сортировки, агрегации и группировки.

Почему важен план выполнения запроса:

1. **Оптимизация запросов** - помогает обнаружить узкие места, такие как полное сканирование таблиц или неоптимальное соединения;
2. **Понимание внутренней работы СУБД** - позволяет узнать, как СУБД данных обрабатывает запрос;
3. **Снижение времени выполнения** - определяет возможности для ускорения запросов (например: добавление индексов, изменение порядка соединений);
4. **Решение проблем производительности** - например, почему запрос выполняется дольше чем ожидалось.

Основные компоненты плана:

1. **Тип сканирования:**
   Сканирование - это процесс извлечения строк из таблицы или индекса.
   `PostgreSQL` выбирает тип сканирования в зависимости от структуры данных, размера таблицы, наличия индексов и условий запроса
   - **Seq Scan** - построчный просмотр всей таблицы без использования индексов:
     - *Когда выполняется:*
       - Если таблица слишком мала, и чтение её полностью быстрее, чем использования индекса;
       - Если запрос не содержит условий, которые могут быть оптимизированы с помощью индекса.
     - *Плюсы*:
       - Не зависит от наличия индекса;
       - Эффективно для небольших таблиц.
     - *Минусы*:

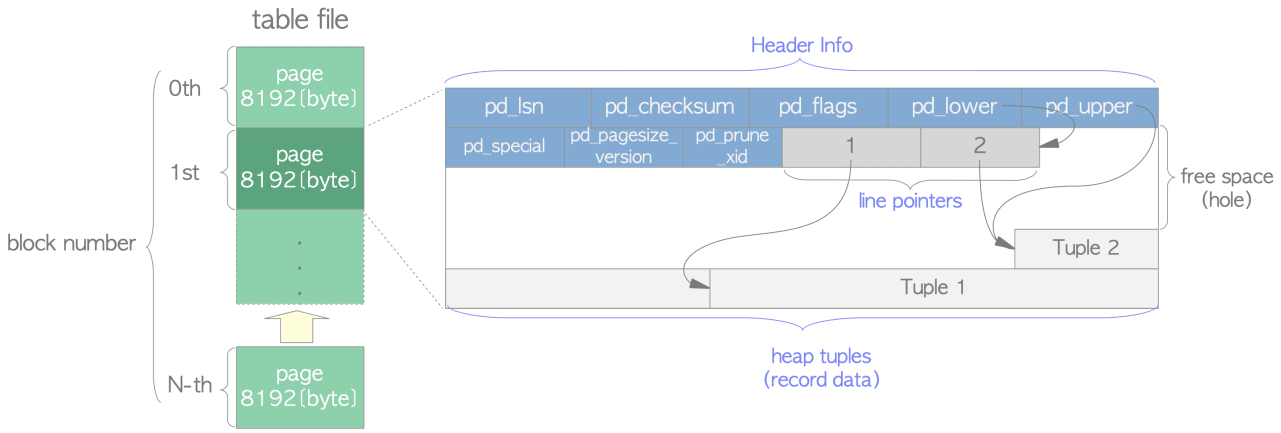
        - Неэффективно для больших таблиц, так как требует чтения всех строк, даже если удовлетворяет только несколько.
    - **Index Scan** - извлечение строк на основе значений индекса. `PostgreSQL` переходит к данным таблицы только для извлечения столбцов, не входящих в индекс:
        - *Когда выполняется:*
            - Если запрос содержи фильтры (`WHERE`), которые могут быть использованы индексом;
            - Если индекс покрывает все столбцы запроса.
        - *Плюсы:*
            - Позволяет избежать чтения всей таблицы;
            - Эффективен для точных запросов и запросов с небольшим количеством возвращаемых строк.
        - *Минусы:*
            - Может быть медленным при возвращении большого количества строк, так как требует многократного перехода от индекса к таблице.
    - **Bitmap Index Scan** - использует индекс для создания битовой карты строк, соответствующих условию
    - **Bitmap Heap Scan** - извлекает строки из таблицы на основе битовой карты
        - *Когда выполняется:*
            - Если запрос охватывает диапазон значений или использует сложные условия (например: `OR`);
            - Если требуется извлечь большое количество строк, и прямое сканирование индекса было бы слишком затратным.
        - *Плюсы:*
            - Уменьшает количество операций ввода-вывода за счёт объединения чтений;
            - Эффективен для запросов с диапазонами значений.
        - *Минусы:*
            - Требует больше памяти для хранения битовой карты;
            - Не всегда лучше, чем обычное индексное сканирование.
2. **Тип соединения (JOIN):**
   Соединение - это процесс объединения строк из двух или более таблиц на основе заданного условия
    - `Nested Loop Join (Вложенные циклы)` - этот метод перебирает каждую строку из первой таблицы и для каждой строки ищет соответствующие строки во второй таблице. Если используется индекс по столбцу соединения, то поиск во второй таблице значительно ускоряется. Обычно СУБД использует это для маленьких таблиц. Асимптотика - $O(N * M)$ где N и M количество кортежей в таблицах;

```
// Псевдокод:
package main

import "fmt"

type Row struct {
    Key   int
    Value string
}

func NestedLoopJoin(tableA, tableB []Row, joinKeyA, joinKeyB string) []Row
{
    result := []Row{}

    for _, rowA := range tableA {
        for _, rowB := range tableB {
            if rowA.Key == rowB.Key {
                result = append(result, Row{
                    Key:   rowA.Key,
                    Value: fmt.Sprintf("%s, %s", rowA.Value, rowB.Value),
                })
            }
        }
    }

    return result
}
```

(https://habrastorage.org/webt/vj/ck/ai/vjckaihodjn0a35_pyxfsdhvyry.gif);
- **`Hash Join`** - база данных сначала создает хэш-таблицу для одной из таблиц используя значения из атрибута соединения. Затем она проходит по другой таблице, выполняя поиск совпадений в хэш-таблице. Не требует индексов и хорошо работает с большим количеством данных. Отлично работает для операций `INNER JOIN` и `OUTER JOIN`. Асимптотика - `O(N + M)` где `N` и `M` количество кортежей в таблицах;

```
// Псевдокод:
package main

import "fmt"

type Row struct {
    Key   int
    Value string
}

func HashJoin(tableA, tableB []Row) []Row {
    hashTable := make(map[int]string)
```

```
    for _, rowA := range tableA {
        hashTable[rowA.Key] = rowA.Value
    }

    result := []Row{}

    for _, rowB := range tableB {
        if value, exists := hashTable[rowB.Key]; exists {
            result = append(result, Row{
                Key:   rowB.Key,
                Value: fmt.Sprintf("%s, %s", value, rowB.Value),
            })
        }
    }

    return result
}
```



- `Merge Join` - сначала обе таблицы сортируются по столбцу соединения, после чего СУБД проходит по строкам обеих таблиц в отсортированном порядке и сопоставляет значения. Если данные уже отсортированы, например, из-за индекса B-дерева или GiST, Merge Join выполняет объединение сразу. Обычно используется для `JOIN` с операторами сравнения (`>` , `<` , `>=` , `<=`) и при наличии индексов, которые предоставляют отсортированные данные. Асимптотика - `O(N + M)` , если данные отсортированы; `O(N log N + M log M)` , если требуется сортировка, где `N` и `M` - количество строк в таблицах.

```
// Псевдокод:
package main

import (
        "fmt"
        "sort"
)
```

```
type Row struct {
        Key   int
        Value string
}

func MergeJoin(tableA, tableB []Row) []Row {
        sort.Slice(tableA, func(i, j int) bool { return tableA[i].Key <
tableA[j].Key })
        sort.Slice(tableB, func(i, j int) bool { return tableB[i].Key <
tableB[j].Key })

        i, j := 0, 0
        result := []Row{}

        for i < len(tableA) && j < len(tableB) {
                if tableA[i].Key == tableB[j].Key {
                        result = append(result, Row{
                                Key:   tableA[i].Key,
                                Value: fmt.Sprintf("%s, %s",
tableA[i].Value, tableB[j].Value),
                        })
                        i++
                        j++
                } else if tableA[i].Key < tableB[j].Key {
                        i++
                } else {
                        j++
                }
        }

        return result
}
```

https://habrastorage.org/webt/w9/wz/fv/w9wzfvnufmgbrzs79i7-ct_hlwk.gif

3) **Стоимость (cost):**

Стоимость - это оценка ресурсоёмкости выполнения операции в СУБД, которая используется оптимизатором запросов для выбора наиболее эффективного плана выполнения. Она измеряется в условных единицах и включает:

- Чтение и запись с диска (I/O операции);
- Использование процессора (CPU);
- Доступная оперативная память (RAM);
- Временные структуры данных (например: временные хранилища при сортировке или хэшировании).

Основные компоненты стоимости:

1) `Startup Cost` - стоимость до выдачи первой строки:

- Включает затраты на инициализацию операции, такие как создание индексов,

настройку соединений или сортировку;
- Важна для запросов, где важно быстрое получение начальных данных (например: для запросов с `LIMIT`).
2) `Total Cost` - полная стоимость выполнения операции:
- Это сумма **Startup Cost** и стоимости извлечения всех строк
- Включает затраты на полное выполнение операций, такие как последовательное или индексное сканирование, соединения и сортировки
3) **Число строк (rows)** - ожидаемое количество строк в результате выполнения
4) **Ширина строки (width)** - средний размер строки (в байтах)

Оптимизация запросов:
Оптимизация запросов - это процесс выбора наиболее эффективного способа выполнения SQL-запроса для минимизации времени выполнения и использования ресурсов

Как оно работает:
Оптимизатор запросов PostgreSQL - это компонент отвечающий за выбор наилучшего плана запроса. Он использует статистику данных и учитывает множество факторов, таких как:

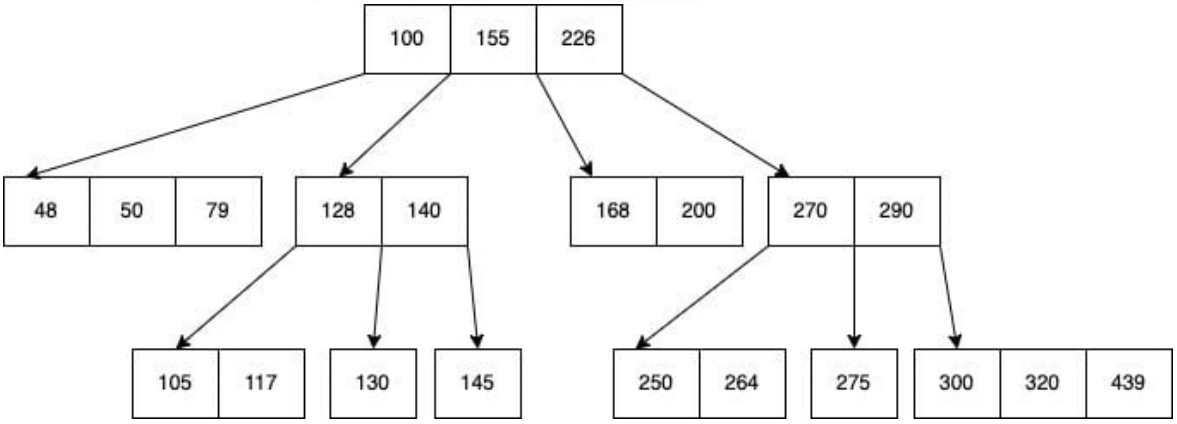
- *Размер таблиц*;
- *Наличие индексов*;
- *Распределение данных (кардинальность)*;
- *Стоимость операций (чтение диска, вычисления, сортировка, соединения)*.

Этапы работы оптимизатора:

1. **Анализ запроса** - разбивает запрос на компоненты: `FROM`, `WHERE`, `JOIN`, `GROUP BY` и т.д, а также проверяет синтаксис и семантику
2. **Генерация возможных планов выполнения** - создаются альтернативные планы с разными стратегиями, например:
    - Использование индекса или последовательного сканирования;
    - Выбора метода соединения: `Nested Loop`, `Hash Join` или `Merge Join`;
    - Порядок обработки таблиц в соединениях.
3. **Оценка стоимости каждого плана:**
    - Оптимизатор использует статистику (`pg_statistic`) для оценки:
        - Количества строк (кардинальность);
        - Ширины строк (объёма данных);
        - Вероятности попадания под условия фильтрации.
    - Рассчитывается стоимость для каждого плана.
4. **Выбор наилучшего плана:**
    - Оптимизатор выбирает план с минимальной стоимостью и передает его исполнителю

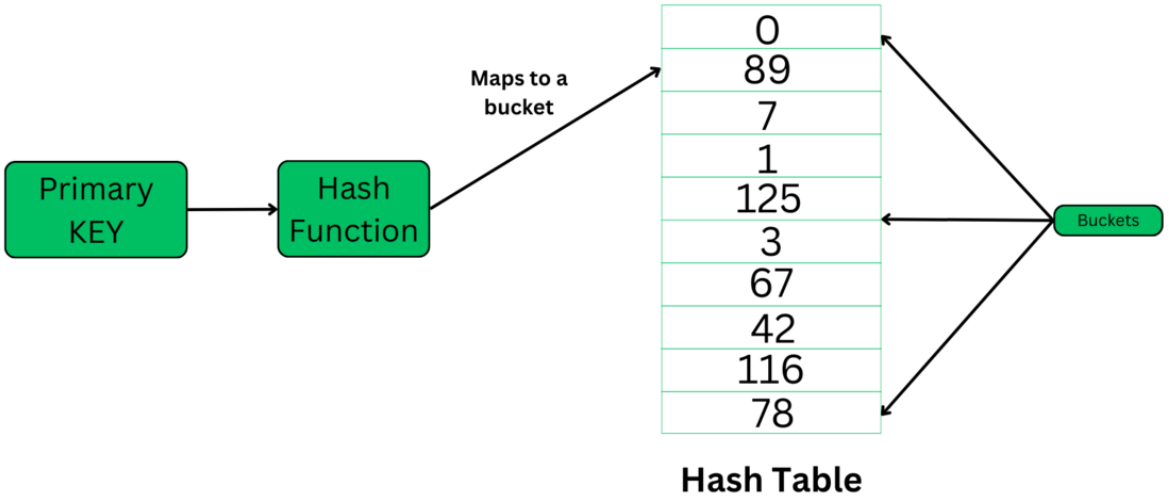
Стратегия оптимизации:

1. **Использование индексов** - индексы ускоряют поиск данных, уменьшая количество операций чтения с диска;
   Индексы в БД - это структура данных (`key-value`), которая ускорят выполнение запросов к БД, создавая отдельную структуры для быстрого поиска значений по атрибутам;
   Типы индексов:
   - **B-дерево** - сбалансированное дерево, где данные организованы по иерархии. Каждый узел содержит несколько ключей и указатели на дочерни узлы. Подходит для поиска по диапазону и точного поиска, т.к поддерживает упорядоченность данных. Хорошо работает на большом количестве данных
     - **Поиск**: `O(logN)`
     - **Вставка**: `O(logN)`
     - **Удаление**: `O(logN)`
     - **Использование**: (`=` , `<` , `>` , `<=` , `>=` , `BETWEEN`)

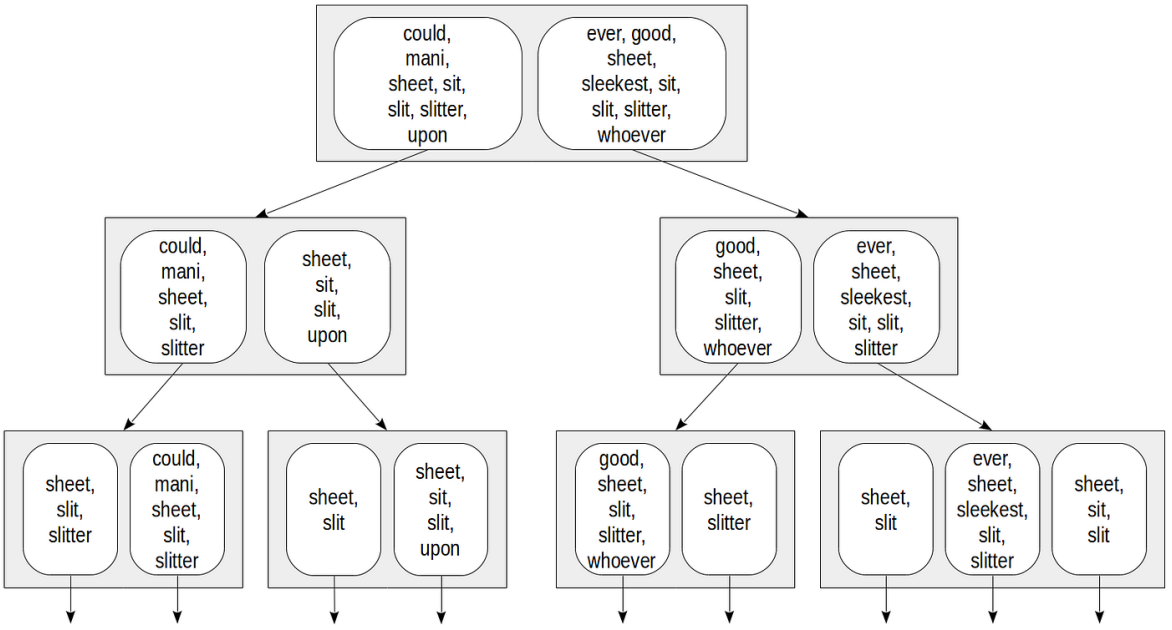
     

   - **Хэш** - данный тип индексов использует хэш-функции в определенный хэш. Для каждого значения создаётся хэш, по которому находится место хранения строки. Например, если искомое значение - `123` , хэш-функция сопоставит его с конкретным местом в индексе, и система найдёт строки с нужным значением сразу. Быстрый для поиска равенства, но требует больше места для хранения, особенно падает скорость если встречаются коллизии
     - **Поиск**: `O(1)` в среднем, для точных совпадений, но при коллизии `O(N)`
     - **Вставка**: `O(1)` , при коллизии `O(N)`
     - **Удаление**: `O(logN)` , при коллизии `O(N)`

- **Использование**: `=` и не поддерживает диапазонные запросы

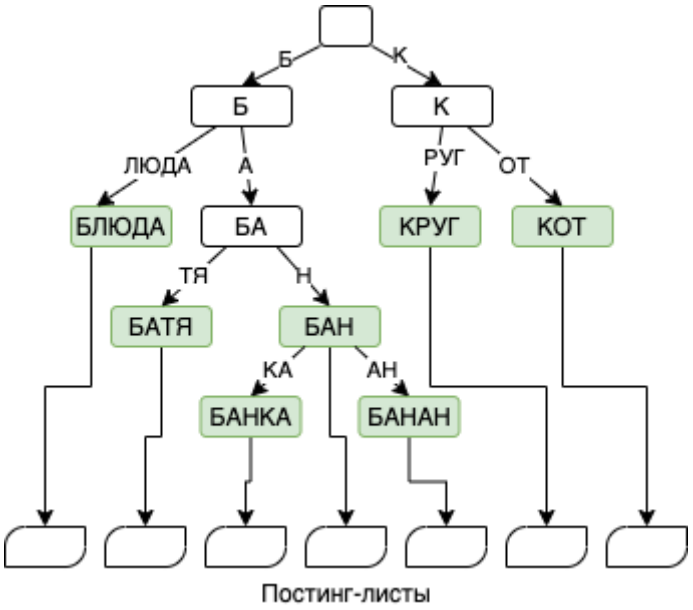


- **GiST (Generalized Search Tree, обобщённое поисковое дерево)** - это индекс, используемый для работы со сложными типами данных, такими как геометрия, текст, массивы. GiST организован в виде сбалансированного дерева, где каждый узел представляет диапазон значений и связан с предикатом, проверяющим принадлежность значений диапазону
  - **Поиск**: `O(log N)`
  - **Вставка**: `O(log N)`, но может быть перебалансировка
  - **Удаление**: `O(log N)`, но может быть перебалансировка
  - **Использование**: Подходит для индексации нестандартных данных



- **Инвертированный индекс** - структура, используемая для быстрого поиска данных, особенно в полнотекстовом поиске. Каждый уникальный токен или слово указывается в индексе, и ему сопоставляются ссылки на документы или строки, содержащие этот токен. Обычно применяется в документоориентированных базах и поисковых движках
  - **Поиск**: `O(log N)` для быстрого поиска уникального токена и `O(M)` для извлечения всех строк или документов, где `M` - количество совпадений
  - **Вставка**: `O(log N)` для добавления нового токена и `O(1)` для добавления ссылки на существующий токен

- **Удаление**: `O(log N)` для поиска и `O(1)` для удаления ссылки
- **Использование**: Для полнотекстового поиска, фильтрации данных и ускоренного доступа к строкам с часто встречающимися словами



Постинг-листы

2. **Избегать** `SELECT *` - нужно выбирать только необходимые столбцы, это сокращает объём передаваемых данных и снижает нагрузку на сеть;
3. **Фильтрация данных на ранних этапах** - условия в `WHERE` выполняются до `GROUP BY` и `HAVING`. Нужно фильтровать строки как можно раньше;
4. **Использовать** `LIMIT` - по хорошему, стоит ограничивать количество возвращаемых строк, если запросу не нужно возвращать весь результат;
5. **Анализ медленных запросов** - использовать команды `EXPLAIN` для анализа плана выполнения и `EXPLAIN ANALYZE` с которым запрос действительно выполняется и можно увидеть фактическое выполнение запроса;
6. **Использование временных таблиц** - для сложных запросов можно разбить выполнение на этапы с использованием временных таблиц.

---

6) Соединения отношений в SQL:

SQL-соединения используются для объединения данных из двух или более таблиц на основе условий. Различные типы соединений позволяют гибко управлять результатом: от выбора только совпадающих строк до полного покрытия всех строк в обеих таблицах

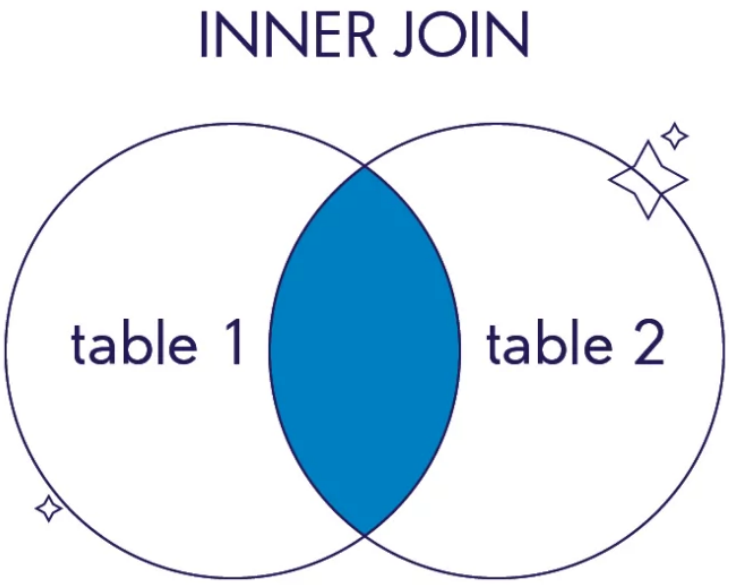
Типы соединений:

1. **Внутреннее соединение (`INNER JOIN`)** - возвращает только те строки, где существует совпадение между связанными столбцами в обеих таблицах

```
-- Синтаксис
SELECT column_name(s)
```

```
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

```
-- Пример
SELECT Students.ISU, Students.Name, Groups.GroupName
FROM Students
INNER JOIN Groups
ON Students.Group = Groups.GroupID;
```

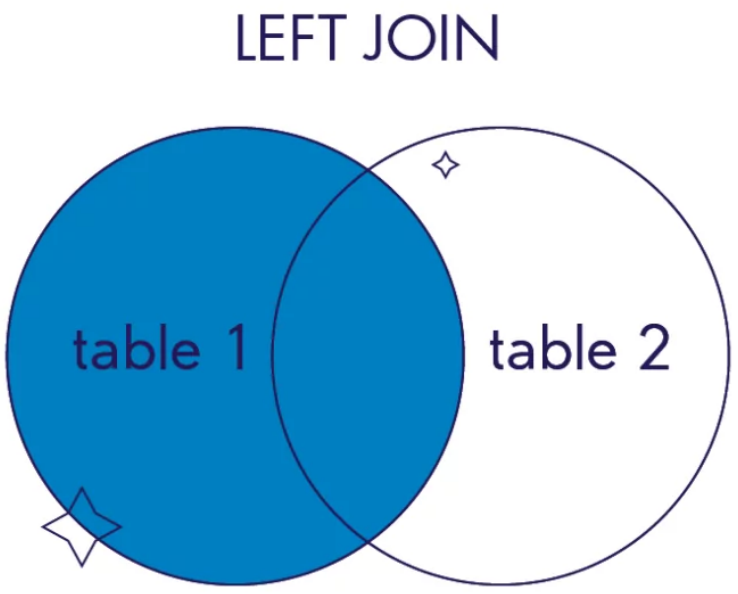


---

2. **Левое внешнее соединение (`LEFT JOIN`)** - возвращает все строки из левой таблицы и совпадающие строки из правой таблицы. Если совпадений нет, возвращаются `NULL`

```
-- Синтаксис
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

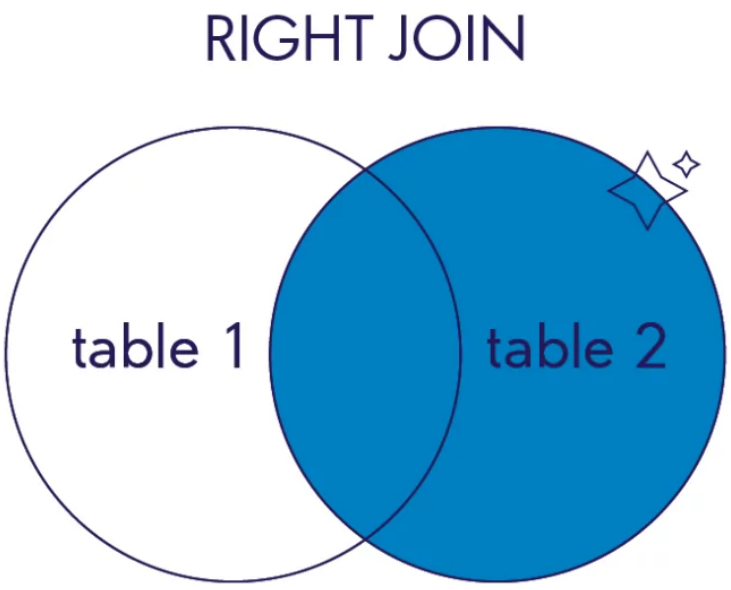
```
-- Пример
SELECT Students.ISU, Students.Name, Groups.GroupName
FROM Students
LEFT JOIN Groups
ON Students.Group = Groups.GroupID;
```



3. **Правое внешнее соединение (`RIGHT JOIN`)** - возвращает все строки из правой таблицы и совпадающие строки из левой таблицы. Если совпадений нет, то возвращаются `NULL`

```
-- Синтаксис
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

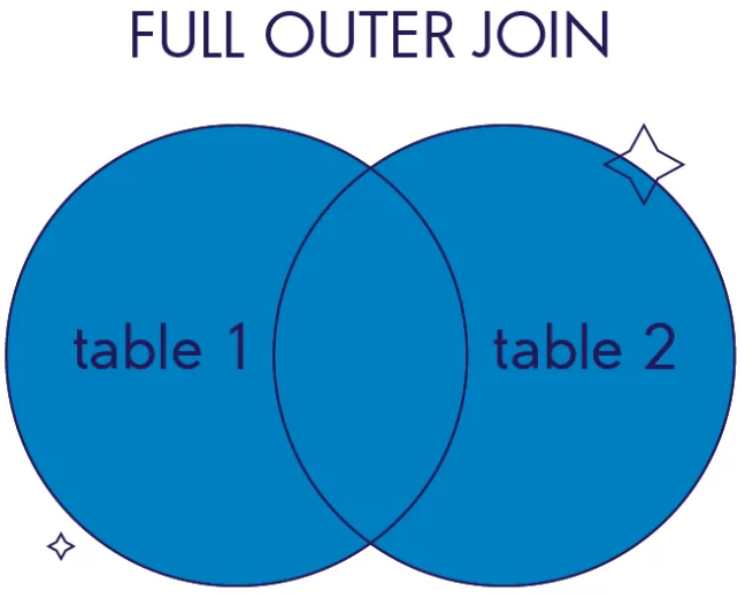
```
-- Пример
SELECT Groups.GroupName, Students.Name
FROM Students
RIGHT JOIN Groups
ON Students.Group = Groups.GroupID;
```



4. **Полное внешнее соединение (`FULL OUTER JOIN`)** - возвращает все строки из обеих таблиц. Если совпадений нет, возвращается `NULL`

```
-- Синтаксис
SELECT column_name(s)
FROM table1
FULL JOIN table2
ON table1.column_name = table2.column_name;
```

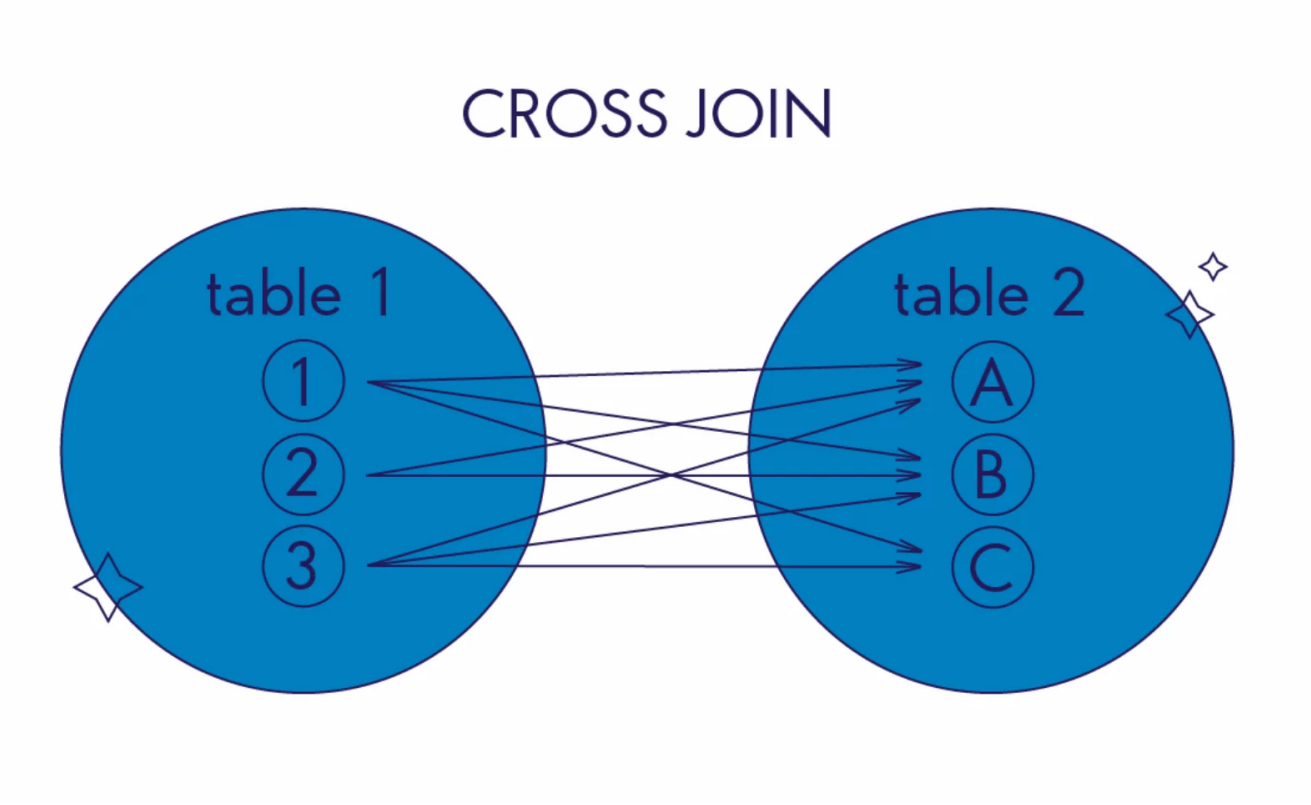
```
-- Пример
SELECT Students.Name, Groups.GroupName
FROM Students
FULL JOIN Groups
ON Students.Group = Groups.GroupID;
```



5. **Перекрестное соединение (`CROSS JOIN`)** - возвращает декартово произведение таблиц: каждая строка первой таблицы соединяется с каждой строкой второй таблицы

```
-- Синтаксис
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```

```
-- Пример
SELECT Students.Name, Groups.GroupName
FROM Students
CROSS JOIN Groups;
```



6. **Самосоединение (`SELF JOIN`)** - соединяет таблицу с самой собой: используется для сравнения строк внутри одной таблицы

```
-- Синтаксис
SELECT a.column_name, b.column_name
FROM table a, table b
WHERE a.common_column = b.common_column;
```

```
-- Пример
SELECT s1.Name AS Student, s2.Name AS Classmate
FROM Students s1
JOIN Students s2
ON s1.Group = s2.Group
WHERE s1.ISU != s2.ISU;
```

Тип соединения (JOIN):
Соединение - это процесс объединения строк из двух или более таблиц на основе заданного условия
- **`Nested Loop Join (Вложенные циклы)`** - этот метод перебирает каждую строку из первой таблицы и для каждой строки ищет соответствующие строки во второй таблице. Если используется индекс по столбцу соединения, то поиск во второй таблице значительно ускоряется. Обычно СУБД использует это для маленьких таблиц. Асимптотика - `O(N * M)` где `N` и `M` количество кортежей в таблицах;

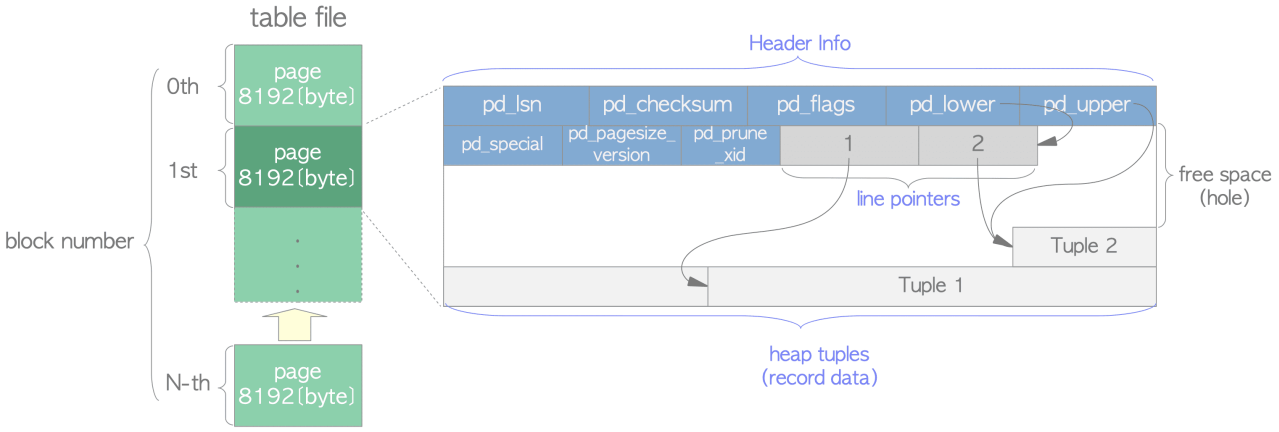
```
// Псевдокод:
package main
```

```
import "fmt"

type Row struct {
    Key   int
    Value string
}

func NestedLoopJoin(tableA, tableB []Row) []Row {
    result := []Row{}

    for _, rowA := range tableA {
        for _, rowB := range tableB {
            if rowA.Key == rowB.Key {
                result = append(result, Row{
                    Key:   rowA.Key,
                    Value: fmt.Sprintf("%s, %s", rowA.Value, rowB.Value),
                })
            }
        }
    }

    return result
}
```

(https://habrastorage.org/webt/vj/ck/ai/vjckaihodjn0a35_pyxfsdhvyry.gif);
- **`Hash Join`** - база данных сначала создает хэш-таблицу для одной из таблиц используя значения из атрибута соединения. Затем она проходит по другой таблице, выполняя поиск совпадений в хэш-таблице. Не требует индексов и хорошо работает с большим количеством данных. Отлично работает для операций `INNER JOIN` и `OUTER JOIN`. Асимптотика - `O(N + M)` где `N` и `M` количество кортежей в таблицах;

```
// Псевдокод:
package main

import "fmt"

type Row struct {
    Key   int
    Value string
}

func HashJoin(tableA, tableB []Row) []Row {
    hashTable := make(map[int]string)

    for _, rowA := range tableA {
        hashTable[rowA.Key] = rowA.Value
    }
```

```
    result := []Row{}

    for _, rowB := range tableB {
        if value, exists := hashTable[rowB.Key]; exists {
            result = append(result, Row{
                Key:   rowB.Key,
                Value: fmt.Sprintf("%s, %s", value, rowB.Value),
            })
        }
    }

    return result
}
```



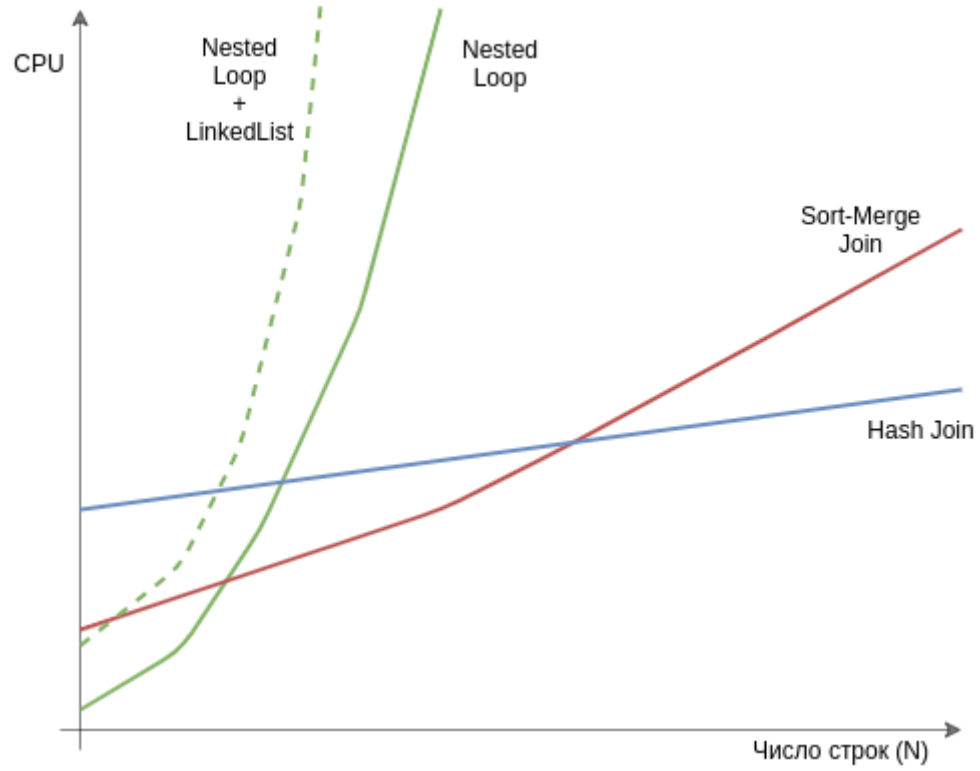
- `Merge Join` - сначала обе таблицы сортируются по столбцу соединения, после чего СУБД проходит по строкам обеих таблиц в отсортированном порядке и сопоставляет значения. Если данные уже отсортированы, например, из-за индекса B-дерева или GiST, Merge Join выполняет объединение сразу. Обычно используется для `JOIN` с операторами сравнения (`>`, `<`, `>=`, `<=`) и при наличии индексов, которые предоставляют отсортированные данные. Асимптотика - `O(N + M)`, если данные отсортированы; `O(N log N + M log M)`, если требуется сортировка, где `N` и `M` - количество строк в таблицах.

```
// Псевдокод:
package main

import (
        "fmt"
        "sort"
)

type Row struct {
        Key   int
        Value string
}
```

```
func MergeJoin(tableA, tableB []Row) []Row {
        sort.Slice(tableA, func(i, j int) bool { return tableA[i].Key <
tableA[j].Key })
        sort.Slice(tableB, func(i, j int) bool { return tableB[i].Key <
tableB[j].Key })

        i, j := 0, 0
        result := []Row{}

        for i < len(tableA) && j < len(tableB) {
                if tableA[i].Key == tableB[j].Key {
                        result = append(result, Row{
                                Key:   tableA[i].Key,
                                Value: fmt.Sprintf("%s, %s",
tableA[i].Value, tableB[j].Value),
                        })
                        i++
                        j++
                } else if tableA[i].Key < tableB[j].Key {
                        i++
                } else {
                        j++
                }
        }

        return result
}
```

https://habrastorage.org/webt/w9/wz/fv/w9wzfvnufmgbrzs79i7-ct_hlwk.gif

А вот общее сравнение алгоритмов:



Сравнение типов соединений:

Тип соединения	Совпадающие строки	Несовпадающие строки из левой таблицы	Несовпадающие строки из правой таблицы	Декартово произведен
INNER JOIN	+	-	-	-
LEFT JOIN	+	+	-	-
RIGHT JOIN	+	-	+	-
FULL JOIN	+	+	+	-
CROSS JOIN	-	-	-	+
SELF JOIN	+ (внутри одной таблицы)	-	-	-

---

7) Нормализация реляционной модели:

Избыточность – это наличие повторяющихся или лишних данных, что может негативно сказаться на работе базы данных:

- Увеличение объема хранения данных;
- Сложности в поддержании целостности;

- Ухудшение производительности.

Дублирование – это частный случай избыточности, когда одни и те же данные повторяются в разных местах. Например, имя студента хранится в нескольких таблицах

Аномалии:
Аномалии данных - это проблемы, возникающие при внесении, обновлении или удалении данных в таблице, приводящие к нарушению целостности и избыточности данных.

Пример таблицы студентов

ФИО	Номер группы	Факультет
Иванов И.И.	М3210	ФИТИП
Петров П.П.	М3211	ФИТИП
Сидоров С.С.	М3212	ФИТИП
Кузнецов К.К.	М3213	ФИТИП
Смирнов С.С.	М3214	ФИТИП

Основные виды аномалий:

- **Аномалия модификации**
  Изменение данных в одной записи может потребовать внесения изменений в другие записи

```
Пример: Если изменить название факультета для группы М3212, то необходимо внести изменения для всех студентов этой группы. Если не внести их для одного из студентов, данные окажутся несогласованными
```

- **Аномалия удаления**
  Удаление одной записи может случайно удалить важную информацию

```
Пример: При удалении всех студентов группы М3212 мы потеряем информацию о принадлежности этой группы к факультету ФИТИП, хотя это свойство группы, а не конкретного студента.
```

- **Аномалия добавления**
  Невозможно добавить данные без указания дополнительных сведений, которые могут быть избыточными

```
Пример: Чтобы добавить нового студента в группу М3212, нужно указать
факультет ФИТИП, даже если это значение повторяется для всех студентов
этой группы.
```

Суть метода нормальных форм:
Нормализация - преобразования отношения к виду, отвечающему нормальной форме, для:

- Уменьшение избыточности;
- Устранение аномалий.

Нормальные формы - это стандартизированные правила для организации данных в реляционных таблицах, которые помогают минимизировать избыточность и избежать аномалий

Пример для видов зависимостей между атрибутами и нормальных форм:

ФИО	Номер группы	Форма обучения	ОП	Факультет
Иванов И.И.	М3210	Контракт	ИС	ФИТИП
Петров П.П.	М3211	Бюджет	ИС	ФИТИП
Сидоров С.С.	М3212	Бюджет	ИС	ФИТИП
Кузнецов К.К.	М3213	Контракт	ИС	ФИТИП
Смирнов С.С.	М3214	Бюджет	ИС	ФИТИП

Функциональная зависимость между атрибутами - это связь, при которой каждому значению атрибута *X* соответствует ровно одно значение атрибута *Y*.

$$R : X \to Y$$

1. **Полная функциональная зависимость** - явление, когда неключевой атрибут зависит от всего составного ключа

```
Пример:
Форма обучения, ОП и Факультет зависят от составного ключа ФИО + Номер
группы:
```

$$R : (ФИО, Номер\ Группы) \to Форма\ обучения,\ ОП,\ Факультет$$

2) **Частичная функциональная зависимость** - зависимость от части составного ключа

```
Пример:
Факультет и ОП зависят только от Номера группы, а не от полного ключа ФИО
```

```
+ Номер группы
Нет необходимости учитывать ФИО для определения факультета и образовательной программы, так как номер группы уникально определяет эти атрибуты
```

$$R : Номер\ Группы \rightarrow Факультет,\ Форма\ обучения$$

3. **Транзитивная функциональная зависимость** - зависимость через промежуточный атрибут

```
Пример:
Факультет транзитивно зависит от Номера группы через ОП
Поэтому, факультет можно определить через ОП, которая, в свою очередь, зависит от номера группы
```

$$R : Факультет\ и\ Номер\ группы \rightarrow\ ОП\ \Longrightarrow\ \ Факультет$$

Нормальные формы:
1-я Нормальная Форма - таблица находится в **первой нормальной форме** (1НФ), если все ее атрибуты являются атомарными, то есть неделимыми
Текущая таблица находится в 1-ой нормальной форме

2-я Нормальная Форма - таблица находится во **второй нормальной форме**, если находится в первой нормальной форме и все неключевые атрибуты функционально полностью зависят от первичного ключа (отсутсвуют частичные функциональные зависимости)

Чтобы таблица соответствовала **2-ой нормальной форме** необходимо устранить **частичные функциональные зависимости**. В данном случае "ОП" и "Факультет" зависят только от "Номера группы", а не от составного ключа "ФИО + Номер группы". Поэтому нужно декомпозировать таблицу, разделив информацию на две таблицы

Таблица 1: Студент

ФИО	Номер группы	Форма обучения
Иванов И.И.	М3210	Контракт
Петров П.П.	М3211	Бюджет
Сидоров С.С.	М3212	Бюджет
Кузнецов К.К.	М3213	Контракт
Смирнов С.С.	М3214	Бюджет

Таблица 2: Группа

Номер группы	ОП	Факультет
М3210	ИС	ФИТИП
М3211	ИС	ФИТИП
М3212	ИС	ФИТИП
М3213	ИС	ФИТИП
М3214	ИС	ФИТИП

3-я Нормальная Форма - таблица находится в **третьей нормальной форме**, если она находится во второй нормальной форме и все неключевые атрибуты независимы друг от друга и функционально полностью зависят от первичного ключа (отсутствуют транзитивные зависимости)

Чтобы таблица соответствовала **3-ей нормальной форме** необходимо устранить **транзитивные зависимости**. Чтобы устранить транзитивную зависимость, нужно создать еще одну таблицу, связав атрибут "ОП" и "Факультет"

Таблица 1: Студент (без изменений)

ФИО	Номер группы	Форма обучения
Иванов И.И.	М3210	Контракт
Петров П.П.	М3211	Бюджет
Сидоров С.С.	М3212	Бюджет
Кузнецов К.К.	М3213	Контракт
Смирнов С.С.	М3214	Бюджет

Таблица 2: Группа (обновленная)

Номер группы	ОП
М3210	ИС
М3211	ИС
М3212	ИС
М3213	ИС
М3214	ИС

Таблица 3: Образовательная программа

ОП	Факультет
ИС	ФИТИП

Нормальная форма Бойса-Кодда (BCNF) - таблица находится в **нормальной форме Бойса-Кодда**, если детерминанты всех функциональных зависимостей являются потенциальными ключами.
Разделение на 3 таблицы уже принадлежит **нормальной форме Бойса-Кодда**

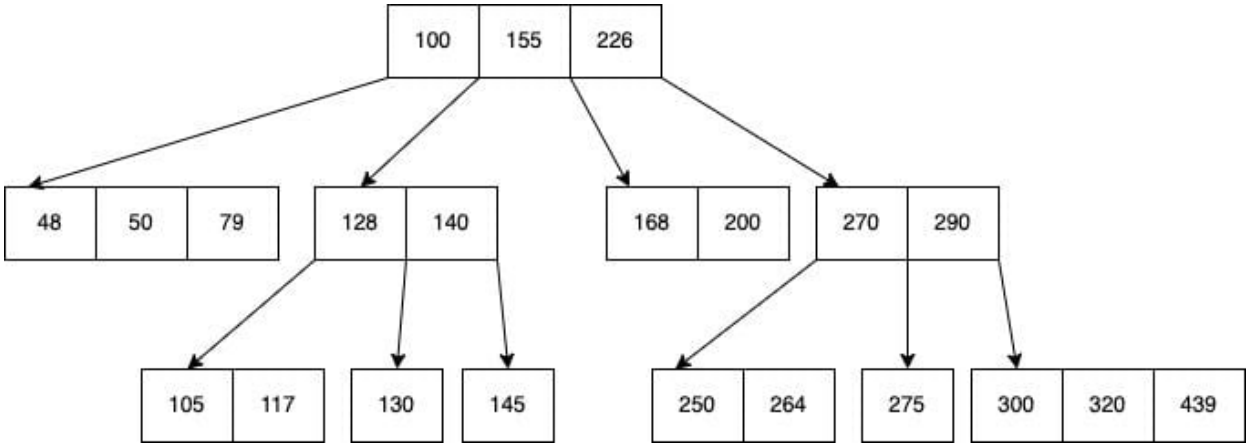
4-я Нормальная Форма - таблица находится в **4-ой нормальная форме** если находится **нормальной форме Бойса-Кодда** и отошение не содержит нетривиальных многозначных значений

---

8) Повышение производительности:

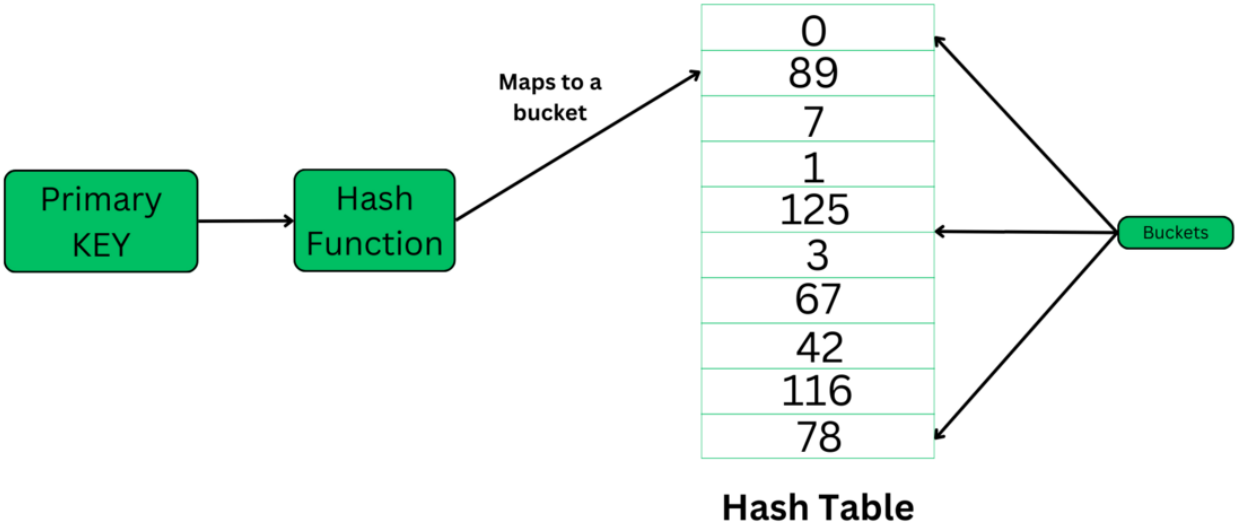
Индексы в БД - это структура данных (`key-value`), которая ускорят выполнение запросов к БД, создавая отдельную структуры для быстрого поиска значений по атрибутам;
Типы индексов:

- **B-дерево** - сбалансированное дерево, где данные организованы по иерархии. Каждый узел содержит несколько ключей и указатели на дочерние узлы. Подходит для поиска по диапазону и точного поиска, т.к поддерживает упорядоченность данных. Хорошо работает на большом количестве данных
  - **Поиск**: `O(logN )`
  - **Вставка**: `O(logN)`
  - **Удаление**: `O(logN)`
  - **Использование**: (`=` , `<` , `>` , `<=` , `>=` , `BETWEEN`)

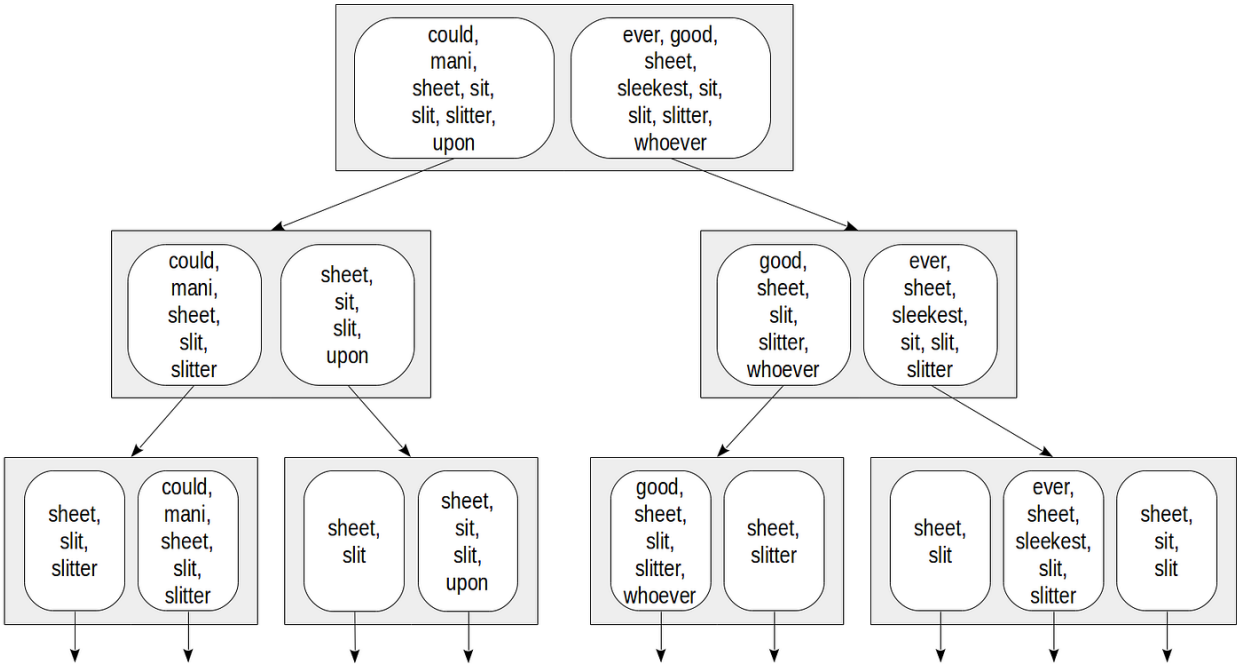


- **Хэш** - данный тип индексов использует хэш-функции в определенный хэш. Для каждого значения создаётся хэш, по которому находится место хранения строки. Например, если искомое значение - `123` , хэш-функция сопоставит его с конкретным местом в индексе, и система найдёт строки с нужным значением

сразу. Быстрый для поиска равенства, но требует больше места для хранения, особенно падает скорость если встречаются коллизии
- **Поиск**: `O(1)` в среднем, для точных совпадений, но при коллизии `O(N)`
- **Вставка**: `O(1)`, при коллизии `O(N)`
- **Удаление**: `O(1)`, при коллизии `O(N)`
- **Использование**: `=` и не поддерживает диапазонные запросы

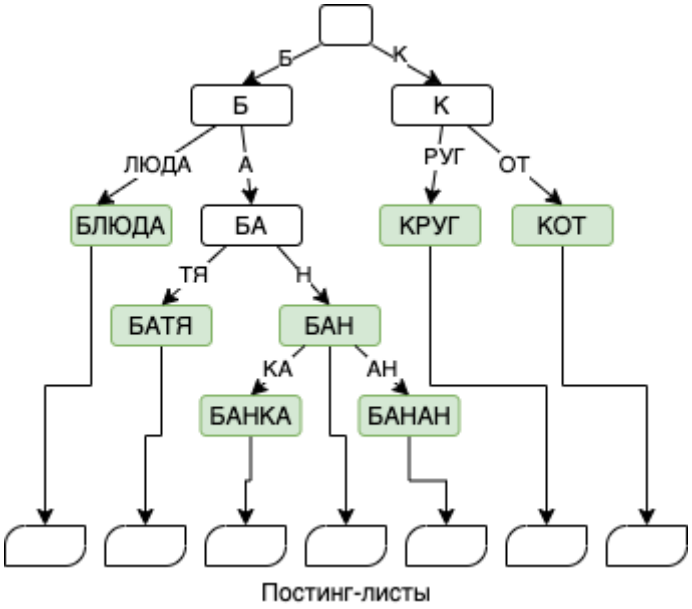


- **GiST (Generalized Search Tree, обобщённое поисковое дерево)** - это индекс, используемый для работы со сложными типами данных, такими как геометрия, текст, массивы. GiST организован в виде сбалансированного дерева, где каждый узел представляет диапазон значений и связан с предикатом, проверяющим принадлежность значений диапазону
  - **Поиск**: `O(log N)`
  - **Вставка**: `O(log N)`, но может быть перебалансировка
  - **Удаление**: `O(log N)`, но может быть перебалансировка
  - **Использование**: Подходит для индексации нестандартных данных



- **Инвертированный индекс** - структура, используемая для быстрого поиска данных, особенно в полнотекстовом поиске. Каждый уникальный токен или слово указывается в индексе, и ему сопоставляются ссылки на документы или строки,

содержащие этот токен. Обычно применяется в документоориентированных базах и поисковых движках
- **Поиск**: `O(log N)` для быстрого поиска уникального токена и `O(M)` для извлечения всех строк или документов, где `M` - количество совпадений
- **Вставка**: `O(log N)` для добавления нового токена и `O(1)` для добавления ссылки на существующий токен
- **Удаление**: `O(log N)` для поиска и `O(1)` для удаления ссылки
- **Использование**: Для полнотекстового поиска, фильтрации данных и ускоренного доступа к строкам с часто встречающимися словами



Зачем нужны индексы:

- Ускорение выполнение операций `SELECT`, `WHERE`, `JOIN`, `GROUP BY`, `ORDER BY`;
- Минимизация операций последовательного чтения данных;
- Оптимизация поиска, особенно в таблицах с больши объемом данных.

Преимущества и недостатки индексов:
Преимущества:

- Ускоряют выполнения запросов;
- Уменьшают нагрузку на диск;
- Поддерживают сортировку и упрощают агрегатные запросы.
  Недостатки:
- Занимают дополнительное место;
- Замедляют операции `INSERT`, `UPDATE`, `DELETE`
- Требуют обновления при изменении данных

Представления (`Views`):
Представления в реляционных базах данных — это виртуальные таблицы, созданные на основе SQL-запросов. Они упрощают доступ к данным, повышают безопасность и могут быть использованы для оптимизации

Виды представлений:

1. **Обычные представления** - создаются на основе запросов. Данные извлекаются в реальном времени
   Преимущества - гибкость, минимальные затраты на хранение;
   Недостатки - зависит от того, насколько быстро база данных может обработать исходные данные в реальном времен.

```
-- Пример
CREATE VIEW active_students AS
SELECT * FROM students WHERE status = 'active';
```

2. **Материализованные представления** - сохраняют результат запроса в виде физической таблицы
   Преимущества - мгновенный доступ к данным;
   Недостатки - дополнительное место, требует ручного обновления.

```
-- Пример
CREATE MATERIALIZED VIEW cached_students AS
SELECT * FROM students WHERE status = 'active';
```

3. **Представление замены** - это представление, которое не хранит данные физически, а только запрос, необходимый для получения данных. При обращении к такому представлению запрос выполняется в реальном времени, и данные извлекаются из исходных таблиц

Плюсы представлений:

- **Упрощение работы** - позволяют сократить сложные запросы
- **Повышение безопасности** - скрывают ненужные данные
- **Оптимизация запросов** - материализвованные представления ускоряют обработку
  Минусы представлений:
- **Производительность** - табличные представления зависят от исходных данных, что может замедлить запросы
- **Ограничение обновления** - не все представления можно модифицировать напрямую
- **Ресурсы хранения** - материализованные представления требуют памяти

---

9) Транзакции:

Транзакция — это последовательность операций, выполняемых с базой данных как единое целое. Транзакция либо полностью выполняется, либо не выполняется вовсе. Завершение транзакции может происходить через:

- `COMMIT` - подтверждение изменений;
- `ROLLBACK` - откат всех изменений.

```
-- Пример
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

Если одна из операций внутри транзакции завершится с ошибкой, изменения будут отменены через `ROLLBACK`

ACID - это набор свойств, которые должны обеспечиваться транзакциями в базе данных для гарантии надежности и целостности данных:

1. **Атомарность (Atomicity)** - транзакция выполняется полностью или не выполняется вовсе. Если одна часть операции не удается, все изменения откатываются;

```
Пример: Перевод денег между счетами. Если списание с одного счета прошло успешно, но зачисление на другой — нет, вся операция отменяется
```

2. **Согласованность (Consistency)** - после завершения транзакции данные должны оставаться в согласованном состоянии, соблюдая все правила и ограничения базы данных;

```
Пример: После перевода денег сумма всех счетов должна оставаться неизменной
```

3. **Изолированность (Isolation)** - параллельные транзакции не должны влиять друг на друга. Результат одной транзакции виден другим только после её завершения;

```
Пример: Если одна транзакция изменяет данные, другая не должна их видеть, пока изменения не подтверждены
```

4. **Долговечность (Durability)** - после завершения транзакции все изменения сохраняются и остаются неизменными даже при сбоях системы

```
Пример: После перевода денег изменения остаются сохраненными даже в случае
```

```
сбоя базы данных
```

Плюсы:

- Обеспечивает точность и надежность данных;
- Подходит для критичных транзакционных операций;
- Гарантирует, что все операции с данными согласованы.
  Минусы:
- Плохо масштабируется в распределённых системах;
- Менее эффективна при работе с большими объёмами данных или высокой нагрузке.

Проблемы конкурирующих транзакций:

1. **Грязное чтение** - происходит, когда одна транзакция читает данные, изменённые, но не подтверждённые (`commit`) другой транзакцией, если вторая транзакция откатит изменения (`rollback`), первая транзакция будет работать с несуществующими или неверными данными;

```
Пример: Транзакция A изменяет баланс, транзакция B читает этот баланс до `COMMIT` транзакции A
```

2. **Неповторяемое чтение** - происходит, когда одна транзакция читает одни и те же данные несколько раз, но между чтениями другая транзакция изменяет или удаляет эти данные;

```
Пример: Транзакция A читает данные, затем транзакция B обновляет эти данные. При повторном чтении данные изменены
```

3. **Фантомное чтение** - происходит, когда одна транзакция читает набор строк, удовлетворяющих некоторому условию, но другая транзакция добавляет или удаляет строки, которые также удовлетворяют этому условию;

```
Пример: Транзакция A читает все строки с условием `salary > 5000`. Транзакция B добавляет новую строку, удовлетворяющую этому условию
```

4. **Потерянное обновление** - происходит, когда несколько транзакций одновременно изменяют одно и то же значение, сохраняется только последнее изменение, в то время как предыдущее теряется без уведомления.

```
Пример: Транзакция A и B обновляют один и тот же баланс. Сохраняются изменения только последней транзакции
```

Уровни изолированности транзакций - это набор правил, определяющих, каким образом данные, изменяемые одной транзакцией, становятся видимыми для других параллельно выполняемых транзакций:

1. **READ UNCOMMITTED** - самый низкий уровень изолированности, транзакция может видеть изменения, сделанные другой транзакцией, даже если они не подтверждены (возможны *грязные чтения, неповторяемые чтения и фантомные чтения*)
2. **READ COMMITTED** - транзакция видит только те данные, которые были подтверждены (commit) другими транзакциями (предотвращаются *грязные чтения, но возможны неповторяемые чтения и фантомные чтения*)
3. **REPEATABLE READ** - гарантирует, что данные, считанные в транзакции, не изменятся в течение её выполнения, повторное чтение тех же данных возвращает одинаковые результаты (предотвращает *грязные и неповторяемые чтения, но возможны фантомные чтения*)
4. **SERIALIZABLE** - самый высокий уровень изолированности, транзакции выполняются так, как будто они идут последовательно, одна за другой (предотвращаются все проблемы: *грязные чтения, неповторяемые чтения и фантомные чтения*)

Проблема транзакции	**READ UNCOMMITTED**	**READ COMMITTED**	**REPEATABLE READ**	**SERIALIZABLE**
Грязное чтение	Возможна	Исключена	Исключена	Исключена
Неповторяемое чтение	Возможна	Возможна	Иключена	Исключена
Фантомное чтение	Возможна	Возможна	Возможна	Исключена
Потерянное обновление	Возможна	Возможна	Возможна	Исключена

Методы решения проблем конкурирующих транзакций:

1. **Использование уровней изоляции**;
2. **Механизмы блокировки**;
3. **Использование MVCC**.

Блокировки транзакций:
Блокировки — это механизм синхронизации доступа к данным в СУБД. Они используются для обеспечения согласованности данных при параллельном выполнении транзакций

Типы блокировок:

1. **Строковые блокировки** - блокируют отдельные строки таблицы, используются для обеспечения согласованности данных на уровне записей (блокировки строк менее затратны, чем блокировки таблиц, но при высокой конкуренции могут вызвать задержки)
2. **Табличные блокировки** - блокируют доступ к таблице целиком, используются для операций, затрагивающих структуру таблицы или большого объёма данных (замедляют выполнение запросов, так как блокируют доступ ко всей таблице, они используются только при необходимости изменения структуры таблицы или при массовых операциях)
3. **Блокировки на уровне индексов** - PostgreSQL и некоторые другие СУБД блокирует индексы во время операций, таких как `CREATE INDEX` или массовое обновление
4. **Deadlocks (взаимные блокировки)** - возникают, когда две транзакции блокируют ресурсы, которые нужны друг другу для завершения

Как блокировки влияют на производительность?

1. **Задержки из-за ожидания блокировок** - если транзакция долго удерживает блокировку, другие транзакции вынуждены ждать
2. **Конфликты блокировок** - высокая конкуренция за ресурсы может вызвать взаимные блокировки
3. **Исключение параллелизма** - табличные блокировки ограничивают возможность выполнения параллельных операций

MVCC (MultiVersion Concurrency Control):
MVCC (Многоверсионное управление параллелизмом) — это метод управления параллелизмом в базах данных, который позволяет нескольким транзакциям одновременно читать и изменять данные без конфликтов. Вместо блокировки данных MVCC сохраняет несколько версий строк, чтобы каждая транзакция могла работать с согласованным набором данных, не ожидая завершения других транзакций

Как работает MVCC в PostgreSQL?
В PostgreSQL MVCC реализован через сохранение нескольких версий строк в таблице

Основные принципы работы MVCC:

1. **Хранение версий строк** - PostgreSQL сохраняет несколько версий строки, каждая из которых относится к определённой транзакции, у каждой версии строки есть два специальных атрибута:
   - **xmin**: идентификатор транзакции, создавшей строку
   - **xmax**: идентификатор транзакции, удалившей или изменившей строк
2. **Чтение данных** - транзакция видит только те версии строк, которые были актуальны на момент её начала: версии строк, созданные более поздними транзакциями, игнорируются, удалённые строки не видны

3. **Изменение данных** - при изменении строки PostgreSQL не обновляет её напрямую, а создаёт новую версию с обновлёнными данными
4. **Удаление данных** - при удалении строки она не удаляется физически, а помечается как “неактуальная” для новых транзакций
5. **Очистка старых версий** - PostgreSQL периодически выполняет **автотранзакцию VACUUM**, чтобы удалять устаревшие версии строк и освобождать место

Преимущества и недостатки MVCC:
Плюсы:

- *Высокая производительность при параллельной работе*
- *Изолированность транзакций*
- *Минимизация блокировок*
- *Поддержка временных запросов*
  Минусы:
- *Повышенные затраты на хранение старых строк*
- *Необходимость очистки устаревших данных*
- *Сложность настройки*
- *Проблемы с долгими транзакциями (удаление старых версий)*

---

10) Идентификация и аутентификация:

Идентификация — это процесс распознавания пользователя или системы на основе предоставленных идентификаторов. Она служит первым шагом в системе контроля доступа, предшествуя аутентификации и авторизации

Как работает процесс идентификации:

1. **Подача идентификатора** - пользователь или система предоставляет уникальный идентификатор (логин, имя пользователя, идентификатор устройства)
2. **Сопоставление с базой данных** - система проверяет наличие предоставленного идентификатора в базе данных пользователей
3. **Переход к аутентификации** - после успешной идентификации начинается процесс проверки личности (аутентификация)

Как идентификация влияет на безопасность данных:

1. **Точность определения пользователя** - идентификация обеспечивает уникальность доступа. Пользователь с уникальным идентификатором не может быть спутан с другим
2. **Привязка действий к пользователю** - все операции, выполненные в системе, связываются с идентификатором пользователя. Это упрощает аудит и выявление

несанкционированных действий
3. **Предотвращение доступа анонимных пользователей**

Проблемы идентификации:

1. **Подмена идентификаторов** - злоумышленник может попытаться выдать себя за другого пользователя, используя его идентификатор
2. **Недостаточная уникальность идентификаторов** - если идентификаторы не уникальны, разные пользователи могут быть ошибочно распознаны как один и тот же человек

Аутентификация - это процесс проверки личности пользователя. Он подтверждает, что пользователь действительно является тем, за кого себя выдаёт
Методы аутентификации:

1. **Аутентификация через пароль** - проверяется личность пользователя с помощью пароля
   - *Плюсы*: простота реализации
   - *Минусы*: уязвимость к атакам методом перебора паролей
2. **Двухфакторная аутентификация (2FA)** - использует комбинацию двух факторов: *пароля* (что пользователь знает) и *временного кода*, отправленного на устройство пользователя (что пользователь имеет)
   - *Плюсы*: повышает безопасность, даже если пароль будет украден
   - *Минусы:* требует настройки дополнительного оборудования или приложений
3. **Аутентификация с использованием сертификатов** - пользователь предоставляет цифровой сертификат, подтверждающий его личность
   - *Плюсы:* высокий уровень безопасности
   - *Минусы:* сложность настройки
4. **Биометрическая аутентификация** - использование уникальных биометрических данных (отпечатки пальцев, сканирование лица или сетчатки)
   - *Плюсы:* очень сложно подделать биометрические данные
   - *Минусы*: требует специализированного оборудования

Авторизация - это процесс определения, какие действия пользователь может выполнять после успешной аутентификации для этого используются различные уровни доступов

Уровни доступа управляют тем, кто и какие действия может выполнять в базе данных. Это первый слой защиты, который предотвращает несанкционированные изменения или просмотр данных

Ключевые элементы уровней доступа:

1. **Мандатная схема** - это политика, в рамках которой к данным и пользователям присваиваются классификационные уровни доступа
   Уровни доступа располагаются по возрастанию защищённости, например: S0 (общедоступные данные) до S4 (самые защищённые данные)
   Правила выполнения операций:
   - *Чтение* (пользователь может читать данные, если его уровень доступа выше или равен уровню данных)
   - *Изменение* (пользователь может изменять данные, если уровень данных равен уровню пользователя)
   - *Добавление* (пользователь может добавлять данные, если их уровень ниже уровня пользователя)
2. **Дискреционная схема:**
   Роли - логические сущности, которым присваиваются привилегии
   Типы ролей:
   - **Административные роли** (роли, которые предоставляют полный контроль над базой данных или ее компонентами):
     1. *Суперпользователь* - имеет полный доступ ко всем объектам базы данных, может выполнять любые операции с базой данных
     2. *Администратор базы данных* - управляет схемами, таблицами индексами и другими объектами базы данных
   - **Пользовательские роли** (роль, которым предоставляют доступ к данным на уровне чтения, записи или подключения):
     1. *Чтение* - имеет доступ только на чтение
     2. *Чтение и запись* - имеет доступ только на запись
     3. *Только подключение* - позволяет подключиться к базе данных, но не выполнять никакие SQL-запросы без доп привелегий
   - **Роли для разработчиков и аналитиков**:
     1. *Роль разработчика* - доступ к созданию и изменению объектов
     2. *Роль аналитика* - доступ к чтению и сложным аналитическим запросам
   - **Системные роли для выполнения системных задач** (специальные роли для выполнения системных задач)
   - **Роль для мониторинга** (предоставляет доступ к системным таблицам и представлениям, связанными с состоянием базы данных)
     Группы - позволяют объединить пользователей, чтобы назначить им одинаковые права
     Привилегии - определяют какие действия пользователь или роль могут выполнить в базе данных (позволяет ограничить или предоставить права на выполнение операций с базой данных)

---

11) Распределенные Базы Данных:

Распределенные базы данных (РБД) — это системы, в которых данные распределены между несколькими физически разделенными узлами (серверами), но взаимодействие с ними осуществляется так, как будто это единая база данных. Распределенные БД предназначены для повышения производительности, масштабируемости и доступности данных в сложных системах

Основные характеристики распределенных баз данных:

1. **Логическая целостность** - пользователи взаимодействуют с данными как с единой системой, несмотря на их физическое распределение;
2. **Физическое распределение** - данные хранятся на разных серверах или географически разнесенных узлах;
3. **Автономность узлов** - каждый узел может работать автономно. Сбой одного из узлов не нарушает работу всей системы
4. **Согласованность данных** - узлы синхронизируются, чтобы обеспечить актуальность данных для всех пользователей

Горячее хранение данных - это хранение данных, которые требуют быстрого доступа и часто используются в повседневной работе
Ключевые характеристики:

1. **Высокая скорость доступа** - используются быстрые хранилища, данные хранятся в системах с минимальной задержкой
2. **Частое использование** - предназначено для данных, которые постоянно запрашиваются
3. **Высокая стоимость** - горячее хранение требует дорого оборудования и инфраструктуры

Холодное хранение данных - это хранение данных, которые используются редко, но должны быть доступны при необходимости
Ключевые характеристики:

1. **Низкая скорость доступа** - данные хранятся в более медленных системах, таких как HDD и тд
2. **Редкое использование** - холодное хранение предназначено для архивов, исторических данных или резервных копий
3. **Низкая стоимость** - используются экономичные решения для хранения больших объёмов данных

Стратегии размещения данных:

1. **Централизованное размещение данных** предполагает хранение всех данных на одном сервере, а все пользователи и системы получают доступ к этим данным через сеть
   Преимущества:

- Простота управления;
- Легкость обеспечения согласованности данных;
- Минимальная сложность настройки.
  Недостатки:
- Ограниченная масштабируемость;
- Высокая нагрузка на центральный сервер;
- Уязвимость к отказу сервера (отсутствие отказоустойчивости).

2. **Реплицированное размещение** подразумевает копирование данных на несколько узлов (реплик), что обеспечивает их доступность в разных точках сети
   Преимущества:
   - Высокая доступность данных (данные доступны даже при отказе одного из узлов);
   - Ускорение чтения, так как пользователи могут обращаться к ближайшей реплике;
   - Отказоустойчивость.
     Недостатки:
   - Высокие затраты на хранение из-за дублирования данных;
   - Сложность синхронизации данных между репликами (особенно при асинхронной репликации);
   - Возможность конфликтов при записи (особенно в системах с мастер-мастер репликацией)
3. **Фрагментированное размещение** делит данные на отдельные части (фрагменты), которые хранятся на разных серверах. Каждый сервер хранит только часть общей базы данных
   Преимущества:
   - Эффективное использование ресурсов (каждый сервер хранит только нужную часть данных);
   - Лучшая масштабируемость;
   - Снижение сетевой нагрузки (запросы обрабатываются локально на нужном сервере).

Репликация - процесс создания копий данных на нескольких узлах или серверах.
Основная цель репликации - повысить доступность данных, обеспечить отказоустойчивость и ускорить чтение, так как пользователи могут обращаться к ближайшей копии данных

Виды репликации:

1. **Синхронная репликация:**
   Изменение данных на одном сервере (Primary) немедленно применяется на реплике (Replica). Запрос считается завершенным только после того, как данные обновлены на всех репликах

2. **Асинхронная репликация:**
   Данные сначала обновляются на основном сервере, а затем отправляются на реплики. Запрос считается завершенным после записи на основной сервер, не дожидаясь завершения репликации
3. **Полурепликация:**
   Промежуточный вариант между синхронной и асинхронной. Основной сервер дожидается подтверждения репликации хотя бы от одной реплики перед завершением запроса

Типы репликаций:

1. **Мастер-Слейв репликация:**
   Один сервер (мастер) принимает все запросы на запись, а остальные сервера (слейвы) обслуживают запросы только на чтение
   - *Плюсы*: хорошо подходит для систем с высокой нагрузкой на чтение
   - *Минусы*: все операции записи проходят через один сервер, нет возможности масштабировать операции записи
2. **Мастер-Мастер репликация:**
   Несколько серверов одновременно выполняют роль мастера, то есть могут обрабатывать запросы как на чтение, так и на запись. Узлы синхронизируют изменения друг с другом, чтобы была согласованность данных
   - *Плюсы:* высокая отказоустойчивость, если один мастер выходит из строя, остальные продолжают работать; возможность распределенной записи данных
   - *Минусы:* конфликты при записи, особенно в асинхронных системах
3. **Каскадная репликация:**
   Реплики не получают данные напрямую от мастера, а через другие реплики (это создает некую иерархическую структуру: **мастер -> реплика №1 -> реплика №2**)
   - *Плюсы:* снижает нагрузку на мастер, особенно в системах с большим количеством реплик; репликация остается прозрачной для конечных пользователей
   - *Минусы:* увеличивается время доставки данных до конечных узлов; при отказе промежуточного узла, вся цепочка ниже него теряет доступ к новым данным
4. **Логическая репликация:**
   Логическая репликация позволяет реплицировать не всю базу данных, а только определенные таблицы или даже изменения (например новые строки). Это делается с помощью логических потоков, которые в свою очередь передают данные на основе изменений в транзакциях
   - *Плюсы:* высокая гибкость (можно реплицировать только нужные данные); подходит для сценариев миграции или интеграции
   - *Минусы:* производительность может быть ниже, чем у физической репликации,

из-за необходимости обработки изменений на уровне транзакций
Сравнительная таблица репликаций:

Типы репликации	Консистентность	Масштабируемость	Пример
Мастер-Слейв	Высокая	Только для чтения	Системы с высокой нагрузкой на чтение
Мастер-Мастер	Средняя/Низкая	Для записи и чтения	Системы с распределенной записью
Каскадная	Средняя	Для чтения	Снижение нагрузки на мастер
Логическая	Настраиваемая	Для определенных данных	Миграция данных

Шардирование - это процесс разделения больших объёмов данных на более мелкие, независимые части, которые называются шардами
Основная цель шардирования - распределить нагрузку между серверами, чтобы повысить производительность и обеспечить масштабируемость системы

Принципы шардирования:

1. **Горизонтальное деление данных** - в отличие от репликации, где все сервера содержат копии данных, шардирование разделяет данные между серверами. Каждый сервер хранить в себе только часть от всей базы (пример: пользователи с ID 1-1000 хранятся на одном сервере, а с ID 1001 - 2000 на другом)
2. **Единый интерфейс** - для пользователя шардированная система выглядит как одна база данных. Шарды скрыты от конечного пользователя, и запросы автоматически направляются в нужный шард
3. **Ключ шардирования** - шардирование требует выбора ключа, по которому данные будут распределяться

Методы шардирования:

1. **Диапазонное шардирование:**
   Данные распределяются по диапазонам значений ключа (пример: пользователи с ID 1-1000 хранятся на одном сервере, а с ID 1001 - 2000 на другом)
   - *Плюсы:* простое управление данными; легко масштабировать, добавляя новые диапазоны
   - *Минусы:* может быть неравномерная нагрузка, если большинство запросов будет приходится на определенный диапазон
2. **Хеширование:**
   Ключ шардирования пропускается через хеш-функцию, чтобы определить в какой

шард отправлять данные (пример: хэш-функция возвращает числа от 0 до 9, данные отправляются в 10 серверов)
    - *Плюсы:* равномерное распределение нагрузки между серверами
    - *Минусы:* сложнее добавлять новые серверы, так как нужно перераспределять данные
3. **Географическое шардирование:**
   Данные распределяются по географическим регионам (пример: данные пользователей из Европы хранятся в одном центре, из Азии — в другом)
    - *Плюсы:* уменьшение задержек из-за пинга
    - *Минусы:* требуется обеспечить репликацию и согласованность между регионами

Сравнительная таблица шардирования:

Типы шардирования	Консистентность	Масштабируемость	Пример
Диапазонное	Высокая	Простая, но ограниченная	Пользователи с ID 1–1000 на одном сервере, ID 1001–2000 — на другом
Хеширование	Высокая	Высокая, но сложно и дорого из-за перераспределения	Данные распределяются по серверам через хэш-функцию
Географическое	Высокая в регионе	Простая	Пользователи из Европы обслуживаются в европейском дата-центре

---

12) NoSQL базы данных:

NoSQL базы данных — это классы систем управления базами данных, которые отличаются от традиционных реляционных СУБД своей гибкостью в структуре данных, масштабируемостью и возможностями работы с большими объёмами информации

Причины появления NoSQL баз данных:

1. **Масштабируемость традиционных реляционных баз данных** - реляционные базы данных (например, PostgreSQL, MySQL) изначально проектировались для вертикального масштабирования, а современные системы с большими объёмами

данных требуют горизонтального масштабирования, что сложно для реляционных баз
2. **Гибкость в моделировании данных** - реляционные базы данных требуют заранее определённой схемы, а современные приложения требуют динамических изменений структуры данных
3. **Высокая доступность и отказоустойчивость** - системы, такие как социальные сети и интернет-магазины, требуют высокой доступности данных даже при сбоях, а NoSQL базы данных предлагают удобную репликацию и шардирование по узлам для обеспечения отказоустойчивости

Основные характеристики:

1. **Гибкость схемы** - данные не требуют фиксированной схемы, поэтому такие решения позволяют хранить разные типы данных (например, JSON-документы, ключ-значение, графы)
2. **Горизонтальное масштабирование** - данные могут быть распределены между несколькими серверами, а новые узлы добавляются без сложной настройки
3. **Высокая производительность** - оптимизированы для быстрого выполнения операций чтения и записи и часто отказываются от сложных операций (например, `JOIN`) для повышения скорости
4. **Консистентность и доступность** - многие NoSQL базы данных поддерживают модели, основанные на теореме CAP, но все равно большинство NoSQL баз данных жертвуют строгой консистентностью в пользу доступности

Типы NoSQL баз данных:

1. **Документо-ориентированные базы данных** - хранят данные в виде документов (например JSON, XML и тд), где документ представляет собой самостоятельную запись, содержащую структуру данных (ключи и значения), которая может быть вложенной
   Ключевые особенности:
   1. *Гибкая структура* - документы могут иметь разные схемы (например, один документ может содержать поле age, а другой — нет)
   2. *Поддержка вложенных данных* - документы могут включать вложенные структуры (например, массивы или вложенные документы)
   3. *Идеальны для динамических данных* - позволяют легко изменять структуру данных без необходимости миграции схемы
2. **Колоночные базы данных** - хранят данные в формате столбцов, а не строк, оптимизированы для работы с большими объёмами данных, где чаще выполняются запросы по конкретным столбцам
   Ключевые особенности:
   1. *Высокая производительность* - отлично справляются с аналитическими запросами, где нужны агрегаты по столбцам

    2. *Горизонтальное масштабирование* - данные распределяются между узлами
    3. *Экономия памяти* - хранение столбцов позволяет лучше сжимать данные
3. **Ключ-значение базы данных** - хранят данные в виде пар “ключ-значение”, ключи должны быть уникальными, а значения могут быть любыми (например, строки, числа, JSON)
   Ключевые особенности:
    1. *Высокая производительность* - оптимизированы для быстрого поиска по ключу
    2. *Простота* - простой интерфейс для чтения и записи данных
    3. *Идеально для кешей* - быстрое хранение и извлечение временных данных
4. **Графовые базы данных** - оптимизированы для хранения графов, которые состоят из узлов (nodes) и связей (edges). Узлы представляют сущности (например, пользователи), а связи — их отношения (например, дружба)
   Ключевые особенности:
    1. *Моделирование сложных связей* - подходят для работы с данными, где важны связи между объектами
    2. *Быстрые запросы к графам* - например, поиск пути между двумя узлами
    3. *Гибкая структура* - узлы и связи могут иметь свойства

Когда лучше использовать NoSQL:

1. **Большие объёмы данных** - NoSQL базы данных лучше справляются с обработкой и хранением больших объёмов данных, особенно если данные распределены между несколькими узлами
2. **Динамическая структура данных** - если структура данных часто меняется или заранее неизвестна
3. **Высокая нагрузка на чтение/запись** - NoSQL базы обеспечивают высокую производительность для операций записи и чтения
4. **Горизонтальное масштабирование** - когда система должна масштабироваться путём добавления новых серверов, NoSQL базы данных более эффективны
5. **Требования высокой доступности**

Когда лучше использовать реляционные базы данных:

1. **Строгая консистентность данных** - если критически важно, чтобы все транзакции строго соблюдали ACID-свойства, реляционные базы являются лучшим выбором
2. **Сложные запросы и связи** - когда данные требуют сложных запросов, таких как `JOIN` между таблицами, реляционные базы данных более эффективны
3. **Фиксированная структура данных** - если структура данных стабильна и мало меняется
4. **Небольшой объём данных** - реляционные базы данных оптимизированы для работы с меньшими объёмами данных, где не требуется горизонтальное

масштабирование

BASE - это подход, который часто используется в распределённых системах, таких как NoSQL базы данных. BASE-системы жертвуют строгой согласованностью в угоду доступности и масштабируемости

1. **Базовая доступность (Basically Available)** - система гарантирует, что запросы к базе данных всегда получат ответ (даже если данные могут быть частично устаревшими)
2. **Мягкое состояние (Soft State)** - состояние системы может меняться со временем, даже без поступления новых запросов, поскольку данные синхронизируются между узлами
3. **Постепенная согласованность (Eventual Consistency)** - данные в конечном итоге будут согласованными во всех узлах системы, но это может занять некоторое время

Плюсы:

- Высокая производительность и доступность
- Идеально подходит для распределенных систем
- Хорошо справляется с большими данными
  Минусы:
- Нет строгой согласованности
- Требуется много усилий для работы со старыми данными из-за времени на синхронизацию

Критерий	**ACID**	**BASE**
Согласованность	Система строго соблюдает согласованность данных	Постепенная согласованность: данные синхронизируются между узлами с некоторой задержкой
Доступность	Доступность может быть ограничена для поддержания согласованности	Высокая доступность: система всегда отвечает на запросы, даже если данные устарели
Масштбируемость	Ограниченная, чаще вертикальное масштабирование	Высокая, горизонтальное масштабирование (добавление новых узлов)
Подход к отказам	При сбое транзакция полностью откатывается, данные остаются согласованными	Система продолжает работать, данные постепенно синхронизируются

---

13) CAP:

CAP теорема - это концепция из теории распределённых систем, которая утверждает, что в любой распределённой системе невозможно одновременно достичь трёх свойств:

1. **Consistency (Консистентность)**: все узлы системы видят одно и то же состояние данных одновременно, любая транзакция немедленно синхронизируется между всеми узлами
2. **Availability (Доступность)**: система отвечает на запросы даже при отказах
3. **Partition Tolerance (Устойчивость к разделению сети)**: система продолжает работать, даже если связь между узлами нарушена

Суть CAP теоремы:
В условиях распределённых систем невозможно одновременно достичь всех трёх свойств. Система должна делать компромисс, выбирая два из трёх:

- **CP**: согласованность и устойчивость к разделению
- **AP**: доступность и устойчивость к разделению
- **CA**: согласованность и доступность (невозможно)

Дополнительная информация:

Отказы:

Отказ - это ситуация, при которой система перестает функционировать в нормальном режиме из-за сбоев оборудования, по, сети или человеческого фактора. Отказы могут быть:

- **Частичными** (пример: в распределенной системы выходит из строя один сервер)
- **Полными** (пример: полностью перестал отвечать дата-центр)

Основная цель решения проблем с отказами — минимизировать их влияние на систему, обеспечить доступность данных и сохранить консистентность

Основные подходы к решению проблемы с отказами:

1. **Репликация** - создание копий данных на нескольких узлах
    - *Преимущества* - обеспечивает отказоустойчивость и доступность данных; реплики могут использоваться как разервные копии
2. **Автоматическое восстановление** - переключение запросов на резервный узел или сервер при отказе основного. Специальные сервисы мониторинга автоматически обнаруживают сбой и перенаправляют трафик

    - *Преимущества* - минимизирует время простоя; ускоряет восстановления системы
3. **Кворумы** - система принимает решения (например об обновлении или чтении данных) на основе согласия определенного числа узлов. В распределенных системах реплицируются данные на несколько узлов, однако возникает проблема гарантии, что данные остаются согласованными, если часть узлов недоступна или не синхронизирована, это решается через кворум
    - *Преимущества* - повышает надежность в распределенных системах; защищает от потерь данных при частичных отказах
4. **RAID массивы для хранения данных** - технология, которая объединяет несколько физических дисков в массив
   Типы RAID для отказоустойчивости:
    - **RAID 1 (зеркалирование)** - данные дублируются на два и более диска
    - **RAID 5 (с четностью)** - данные разбиваются на блоки, которые распределяются между всеми дисками. Контрольные суммы (четность) также записываются на диски, но равномерно распределяются между ними. Используется минимум 3 диска
    - **RAID 6 (с двойной четностью)** - похож на RAID 5, но поддерживает отказ двух дисков одновременно
    - **RAID 10 (зеркалирование + разделение данных)** - это комбинация RAID 1 и RAID 0 (разделение данных по дискам для повышения производительности)

Дополнительная информация:

Триггер — это объект базы данных, который автоматически выполняет заданное действие в ответ на определённое событие, такое как вставка, обновление или удаление данных в таблице. Триггеры используются для автоматизации обработки данных и обеспечения согласованности

Ключевые особенности триггеров:

- Выполняются автоматически при возникновении заданного события
- Связываются с таблицей или представлением
- Запускают выполнение функции, содержащей логику триггера

Типы триггеров:

1. **До события** - выполняются перед выполнением операции, могут модифицировать данные или отменить выполнение операции
2. **После события** - выполняются после успешного завершения операции, используются для записи в журналы, отправки уведомлений или других задач,

которые не влияют на основные данные
3. **Вместо события** - используются для замены стандартного поведения операции, обычно применяются для работы с представлениями

Плюсы и минусы триггеров:
Плюсы:

- **Автоматизация задач** - снижение дублирования кода в приложениях
- **Согласованность данных** - обеспечение контроля над изменениями данных
- **Журналирование** - упрощение отслеживания изменений
  Минусы:
- **Сложность отладки** - ошибки в работе триггеров могут быть сложными для диагностики
- **Потенциальное снижение производительности** - избыточные триггеры могут замедлить выполнение операций
- **Скрытая логика** - логика, реализованная в триггерах, может быть неочевидной для разработчиков

Хранимая процедура — это объект базы данных, содержащий набор SQL-операторов и логики, которые выполняются на стороне сервера. Хранимые процедуры используются для выполнения сложных операций, таких как обработка данных, управление транзакциями или автоматизация задач

Функция - это объект базы данных, который выполняет заданные операции и возвращает результат. PostgreSQL поддерживает множество типов функций, которые могут быть использованы для выполнения вычислений, обработки данных и других задач

Типы функций:

1. **Скалярные функции** - возвращают одно значение
2. **Функции возвращающие таблицы** - возвращают набор строк, который можно использовать в запросах
3. **Агрегатные функции** - используются для выполнения агрегатных вычислений
4. **Функции-триггеры** - создаются для использования с триггерами

Отличие хранимых процедур и функций:

Критерий	Хранимые процедуры	Функции
Возврат значений	Не возвращают значения, но могут изменять параметры	Возвращают скалярное значение, таблицу или набор строк
Использование в запросах	Нельзя вызывать в запросах	Можно вызывать в запросах, как и встроенные в СУБД функции

Критерий	Хранимые процедуры	Функции
Управление траназкциями	Поддерживает явное управление транзакциями	Не поддерживает управление транзакциями и выполняются в рамках одной транзакции
Цель использования	Для выполнения сложных операций и управления процессами	Для вычислений, получения данных, преобразований

Различия между триггерами и хранимыми процедурами:

Критерий	Триггеры	Хранимые процедуры
Исполнение	Выполняются автоматически в ответ на события	Выполняются вручную
Назначение	Реакция на изменения в таблице	Выполнение сложных операций и задач
Управление траназакциями	Не поддерживает управление транзакциями	Поддерживает управление транзакциями
Примеренение	Автоматизация действий на уровне таблицы или строки	Выполнение заданий, не связанных напрямую с событиями

Домен - это множество допустимых значений атрибута
Отношение - это множество упорядоченных кортежей, где каждое значение берется из соответствующего домена
Схема отношения — это строка заголовков
Степень отношения — это количество его атрибутов
Кардинальность отношения — это количество его кортежей

Свойство отношений:

- **Уникальность имени таблицы** - каждая таблица имеет уникальное имя в БД, что позволяет однозначно ее идентифицировать
- **Уникальность кортежей и атрибутов** - каждый кортеж и атрибут в таблице уникален, что исключает возможность дублирования данных
- **Неупорядоченность кортежей и атрибутов** - порядок атрибутов и кортежей не имеет значения, так как данные идентифицируются по именам атрибутов и извлекаются независимо от порядка
- **Атомарность значений** - каждая ячейка в таблице содержит одно неделимое значение
- **Однородность доменов** - значения атрибутов берутся из одного и того же домена (типа данных), чтобы сохранить единообразие

- **Целостность данных** - значения атрибутов должны соответствовать определенным правилам и ограничениям, например, `NOT NULL` для обязательных полей или ограничение уникальности для первичного ключа