# Boundary-Control-Theory-2

---

## Список тем и вопросов

- Распределенные базы данных
- Типы масштабирований
- Репликация
- Шардирование
- Консистентность и согласованность
- Отказы
- Идентификация
- Аутентификация и авторизация
- Уровни доступа
- Шифрование данных
- NoSQL базы данных
- Общие характеристики NoSQL баз данных
- Типы NoSQL баз данных
- Сравнения NoSQL с реляционными БД
- Индексы в реляционных базах данных
- Оптимизация запросов в базах данных
- Транзакции в БД
- Проблемы параллельных транзакций
- Уровни изолированности транзакций
- Блокировки транзакций
- MVCC
- CAP теорема
- ACID
- BASE
- Сравнение ACID и BASE
- Нормальные формы
- Триггеры
- Хранимые процедуры и функции
- Горячее и холодное хранение данных
- Стратегия хранения данных
- Статистика в PostgreSQL
- Табличное представление

# Теория, ответы на вопросы

## Распределенные базы данных:

**Распределённые базы данных (РБД)** — это система хранения данных, в которой информация распределена между несколькими физически разделёнными узлами (серверами или компьютерами), но взаимодействие с ней осуществляется так, как будто это единая система
**Главная цель** таких баз данных — обеспечить масштабируемость, доступность и устойчивость к отказам, особенно в больших и распределённых системах

**Основные характеристики распределённых баз данных:**

1. **Логическая целостность**:
   Пользователь взаимодействует с базой данных как с единой системой, даже если данные физически распределены между узлами
2. **Физическое распределение**:
   Данные хранятся на разных серверах или в разных географических зонах. Это позволяет повысить производительность и устойчивость системы
3. **Автономность узлов**:
   Каждая часть распределённой БД может работать автономно. В случае сбоя одного из узлов система продолжает функционировать
4. **Обеспечение согласованности**:
   Узлы синхронизируют данные между собой, чтобы пользователи видели актуальную информацию

**Типы распределенных БД:**

- **NoSQL базы данных:**
  1. Документо-ориентированные базы данных (MongoDB)
  2. Ключ-Значение (Redis)
  3. Колонночные базы данных (Cassandra)
  4. Графовые базы данных (Neo4j)
- **Реляционные распределенные базы данных:**
  Используют SQL и обеспечивают ACID транзакции (CockroachDB)

**Гомогенная** - на всех узлах одна и та же СУБД
**Гетерогенные** - на разных узлах могут быть разные СУБД

## Типы масштабирований:

**Горизонтальное масштабирование** - увеличение мощности за счет добавления новых серверов (узлов)
**Вертикальное масштабирование** - улучшение одного сервера (докинуть оперативки, поменять проц и тд)

---

## Репликация:

**Репликация** - процесс созданий 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:**

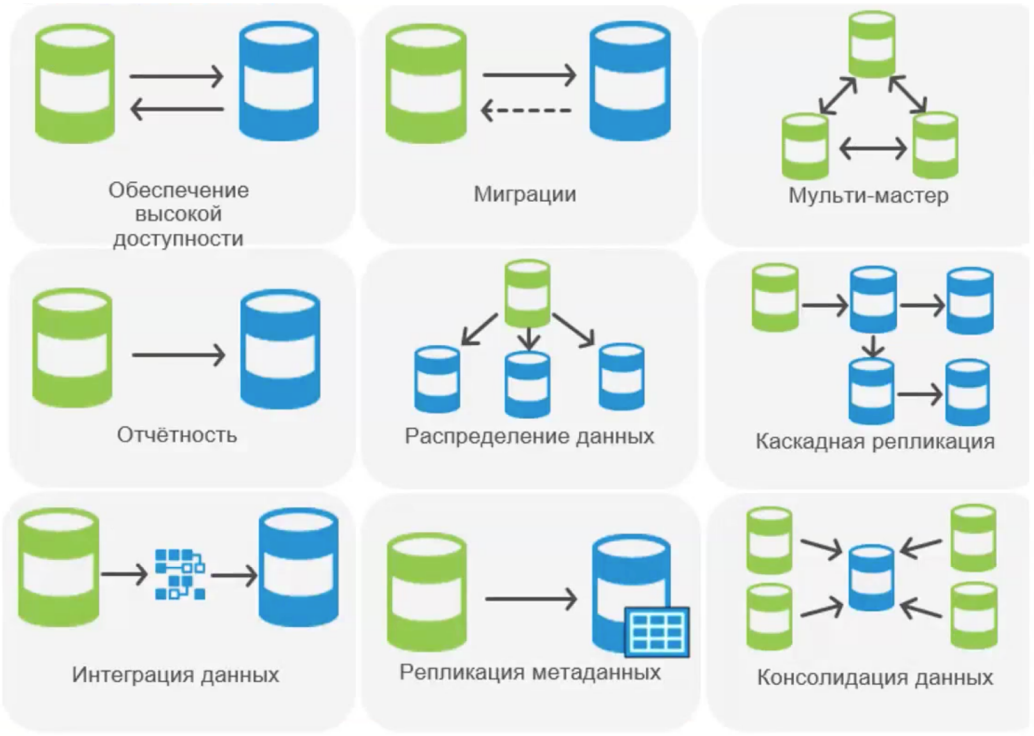
1. **Синхронная репликация:**
   Изменение данных на одном сервере (Primary) немедленно применяется на реплике (Replica). Запрос считается завершенным только после того, как данные обновлены на всех репликах
2. **Асинхронная репликация:**
   Данные сначала обновляются на основном сервере, а затем отправляются 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 подходить 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 измения 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:

**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 диапозон

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* | Средняя | Высокая | Данные распределяются по серверам через хэш-функцию |
| *Географическое* | Высокая в регионе | Простая | Пользователи из Европы обслуживаются в европейском дата-центре |

---

## Консистентность и согласованность:

**Консистентность данных** - означает, что все узлы системы видят одни и те же данные в один момент времени

**CAP теорема** говорит, что распределенная система может обеспечивать одновременно только два из трех 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

**Подходы для обеспечения консистентности:**

1. **Сильная консистентность** - после выполнения операции все узлы обновляются синхронно (подходит для финансовых транзакций)
2. **Слабая консистентность** - узлы обновляются асинхронно, возможны задержки (подходит для соц сетей и систем логирования)
3. **Согласованность в итоге** - со временем узлы придут к одинаковому состоянию, даже если временно данные могут различаться (применяется в системах, где важна высокая доступность)

---

## 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)

---

## Идентификация

**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 и авторизация:

**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
5. **Одноразовые токены (OTP)** - временные коды или токены, которые пользователь получает через SMS, email или специальное приложение
   - *Плюсы:* повышенная безопасность
   - *Минусы:* зависимость от внешних устройств или сервисов

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

---

# Уровни доступа:

**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. **Дискреционная схема** - это политика, в рамках которой поддерживаются списки, содержащие ифнормацию о том, кто и к чему имеет доступ
   **Основные понятия:**
   1) *Объекты* - набор данных или прав для них
   2) *Пользователи* - пользователи системы, которые обладают правами на доступ к объектам
   3) Группы пользователей - объединения пользователей, которые унаследуют общие права. Группам можно назначать дополнительные права, распространяющиеся на всех участников группы
2. **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)
3. **Классическая 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)

---

## Шифрование данных

**Шифрование данных** — это процесс преобразования данных в такой формат, который может быть прочитан только при наличии ключа для расшифровки. Шифрование защищает данные как при их передаче по сети, так и при хранении на дисках, предотвращая их утечку или несанкционированный доступ

**Виды шифрования:**

1. **Шифрование при передаче** - защищает данные при передаче между клиентом и сервером или между узлами системы
2. **Шифрование при хранение** - защищает данные, хранящиеся на носителях (диски, файловые системы, базы данных)
3. **Конечное шифрование** - данные шифруются на стороне отправителя и расшифровываются только получателем

---

## 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

---

## Общие характеристики NoSQL баз данных:

**Основные характеристики:**

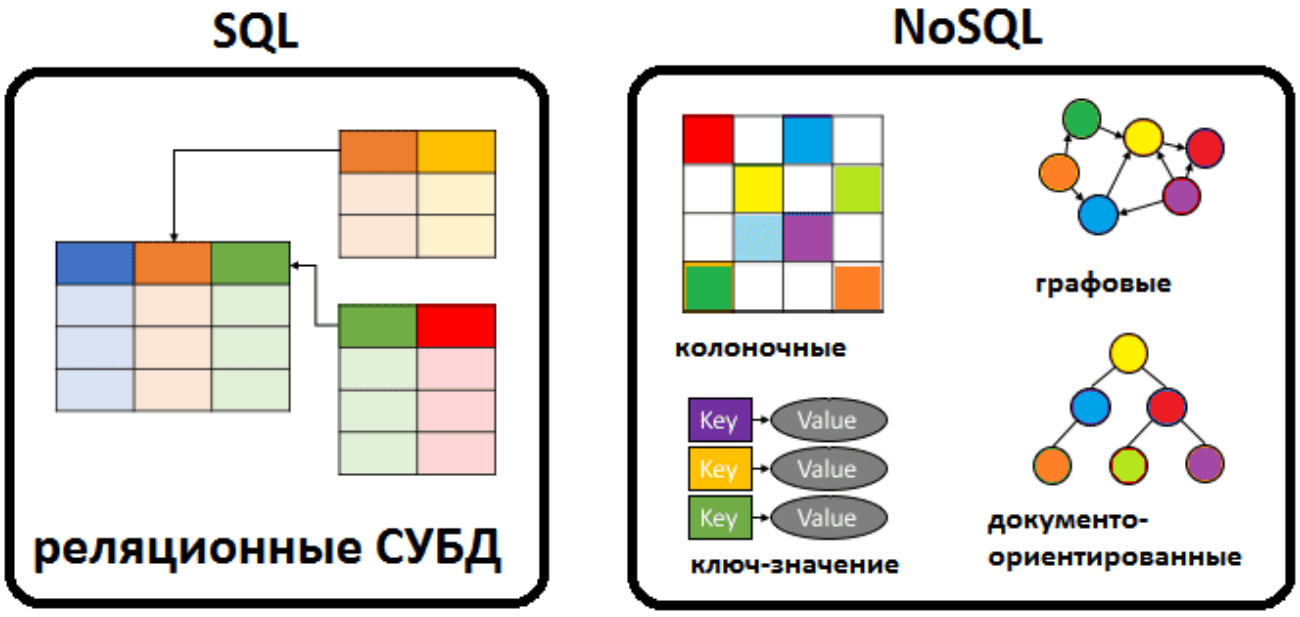
1. **Гибкость схемы** - данные не требуют фиксированной схемы, поэтому такие решения позволяют хранить разные типы данных (например, JSON-документы, ключ-значение, графы)
2. **Горизонтальное масштабирование** - данные могут быть распределены между несколькими серверами, а новые узлы добавляются без сложной 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 зачения 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




---

## Сравнения с NoSQL реляционными БД:

**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

---

## Индексы в реляционных базах данных:

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

| Ключ | Значение |
| --- | --- |
| Значение поля | Адрес строки (или блока строк) |

**Зачем нужны индексы?**
**Индексы** повышают скорость выполнения запросов, позволяя базе данных быстро находить строк по значениям в индексируемых атрибутах

**Первичный индекс** — это индекс, построенный по первичному ключу при условии, что исходный файл отсортирован по нему же

**Индекс кластеризации** — это индекс, построенный по ключевому или неключевому полю при условии, что исходный файл отсортирован по нему же

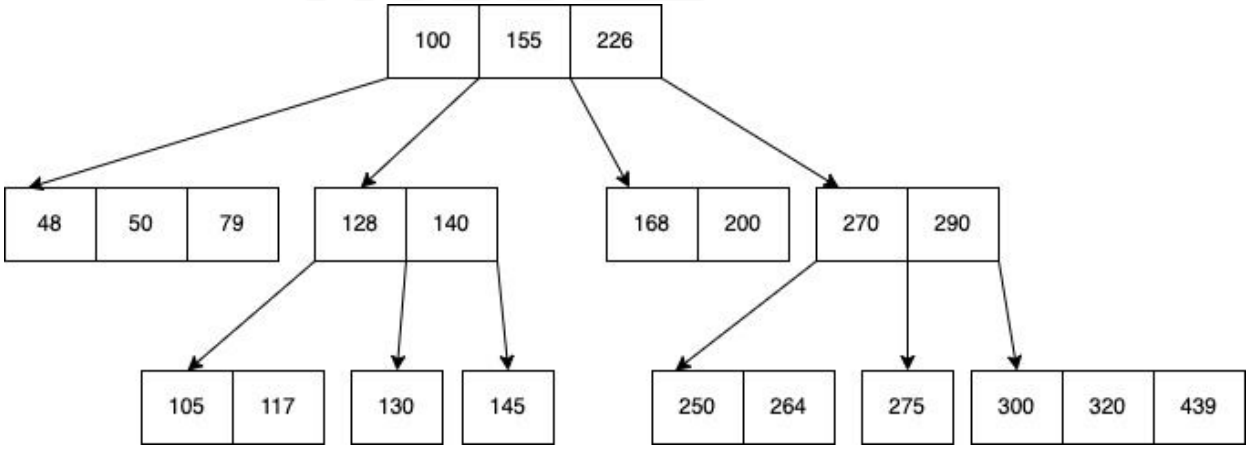
**Вторичный индекс** — это индекс, построенный по неключевому полю при условии, что исходный файл не отсортирован

**Плотный индекс** - содержит в себе ссылку на каждую строку в таблице

**Разреженный индекс** - хранит в себе ссылки только на первые строки

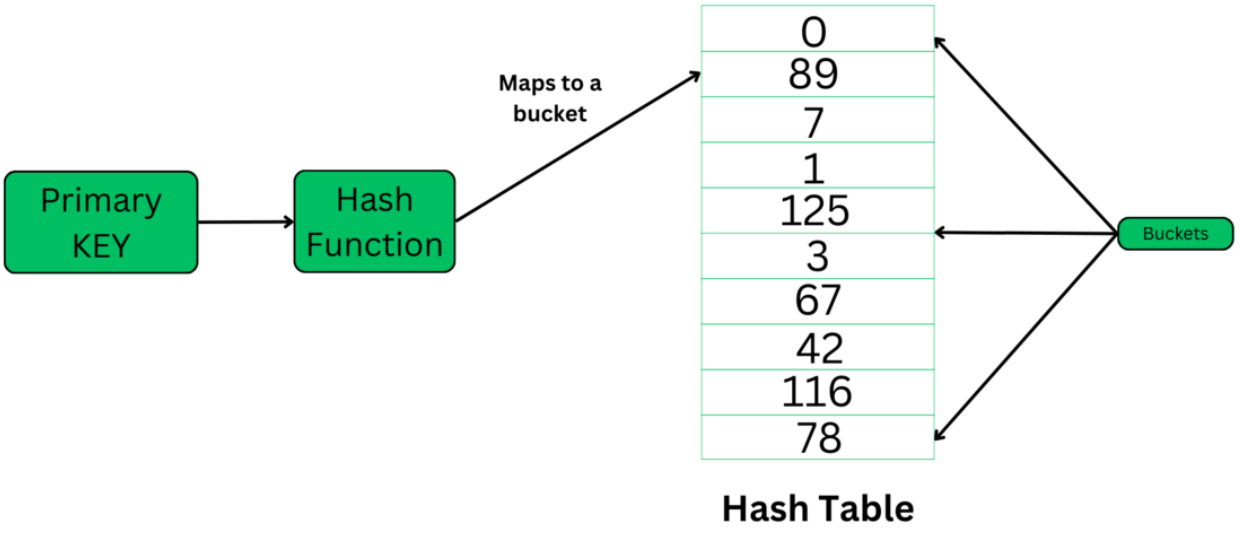
**Основные типы индексов:**

1. **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`)

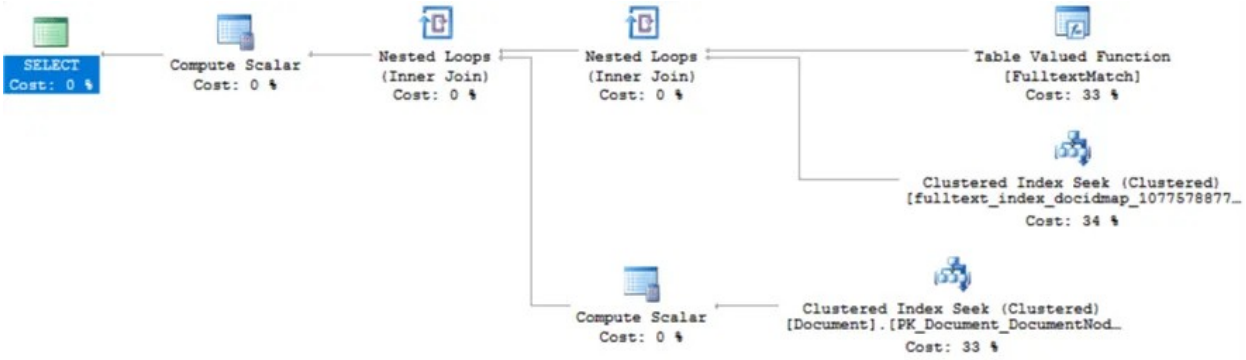



2. **Хэш** - данный тип индексов использует хэш-функции в определенный хэш. Для каждого значения создаётся хэш, по которому находится место хранения строки. Например, если искомое значение - `123`, хэш-функция сопоставит его с конкретным местом в индексе, и система найдёт строки с нужным значением сразу.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

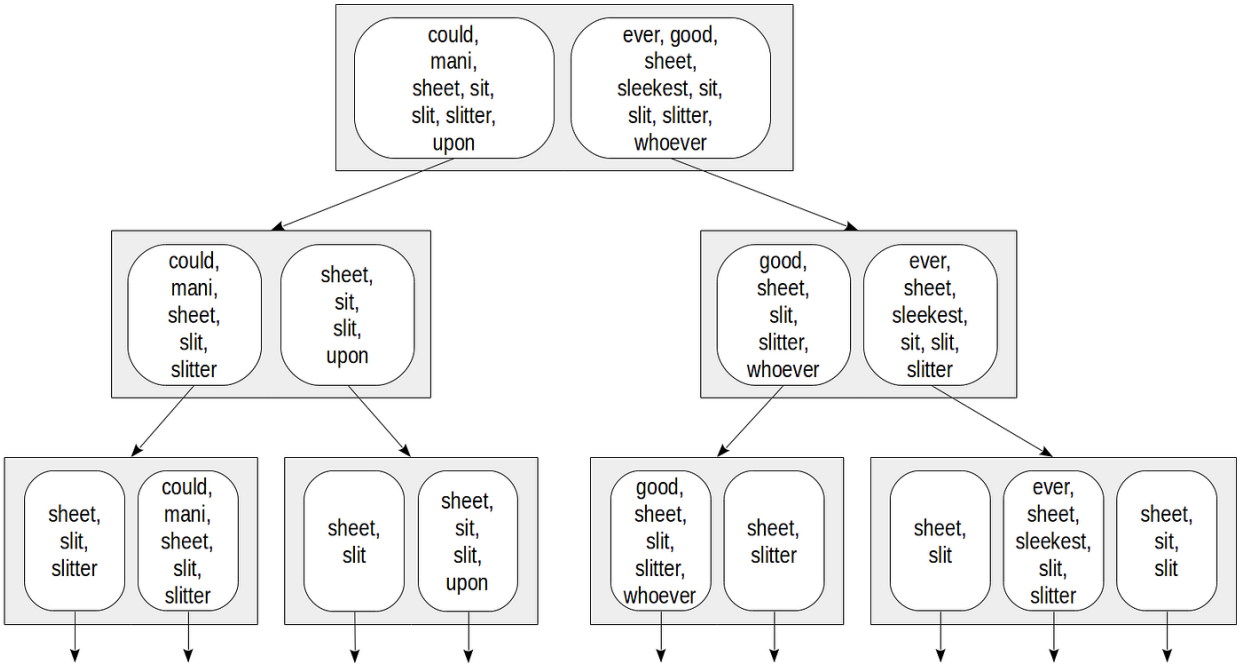



3. **Полнотекстовое индексирование** - метод индексации, ускоряющий текстовые запросы в базе данных. Текст разбивается на токены (слова), которые затем сохраняются в индекс
   - **Поиск**: `O(log N)` для извлечения токенов и `O(M)` для обработки результатов, где `M` - количество совпадений.
   - **Вставка**: `O(log N)`
   - **Удаление**: `O(log N)`
   - **Использование**: Для поиска по большим текстовым данным и базам с активными текстовыми запросами

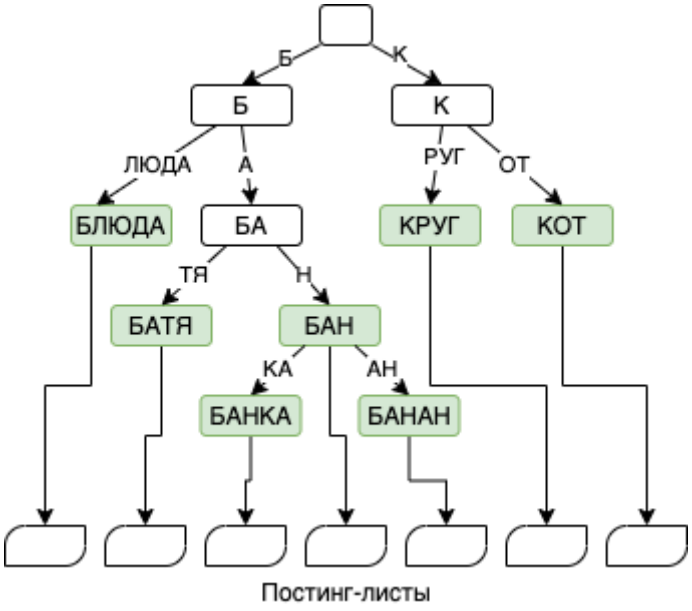



4. **GiST (Generalized Search Tree, обобщённое поисковое дерево)** - это индекс, используемый для работы со сложными типами данных, такими как геометрия, текст, массивы. GiST организован в виде сбалансированного дерева, где каждый узел представляет диапазон значений и связан с предикатом, проверяющим 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




5. **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



# Оптимизация запросов в базах данных:

**Оптимизация запросов** — это процесс выбора наиболее эффективного плана выполнения SQL-запроса для минимизации времени выполнения и использования ресурсов. В современных системах управления базами данных (СУБД) за это отвечает **оптимизатор запросов**

**Как работает оптимизация запросов?**

1. **Анализа запроса** - SQL-запрос разбивается на отдельные компоненты (таблицы, условия, сортировка, группировка) и оптимизатор анализирует синтаксис и семантику запроса, чтобы понять, какие данные нужны
2. **Генерация возможных планов выполнения** - оптимизатор создаёт множество альтернативных стратегий выполнения запроса (например, выбор индекса, порядок соединения таблиц) и каждый план описывает последовательность операций
3. **Оценка стоимости каждого плана** - оптимизатор выбирает план с минимальной стоимостью и передаёт его исполнителю

**Факторы, влияющие на выбор плана:**

1. **Наличие индексов** - использование индексов значительно ускоряет доступ к данным
2. **Размеры таблиц** - если таблица мала, полное сканирование может быть быстрее, чем использование индекса
3. **Типы соединений (JOIN)** - оптимизатор выбирает лучший способ выполнения соединений:
    - *Nested Loop*
    - *Hash Join*
    - *Merge Join*
4. **Статистика данных** - оптимизатор использует статистику о данных (распределение значений, количество строк, кардинальность) для оценки эффективности различных операций

---

# Транзакции в БД:

**Транзакции** - это последовательность, набор операций, которые выполняются как единое целое. Если хотя бы одна операция из набора не выполнится, все изменения отменяются. Это гарантирует целостность данных. Результатом каждой транзакции может быть либо `commit`, либо `rollback` (Собственная интерпретация + интернет)

**Транзакция** — это последовательность действий с базой данных, в которой либо все действия выполняются успешно, либо не выполняется ни одно из них. Результатом каждой транзакции может быть либо `commit`, либо `rollback` (Маятин)

---

## Проблемы параллельных транзакций:

**Проблемы 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
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
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
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

| 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 |

---

## Уровни изолированности транзакций:

**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*)

---

## 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:

**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)*

---

## CAP теорема

**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)

---

## ACID:

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

3. **Изолированность (Isolation)** - параллельные транзакции не должны влиять друг на друга. Результат одной транзакции виден другим только после её завершения
4. **Долговечность (Durability)** - после завершения транзакции все изменения сохраняются и остаются неизменными даже при сбоях системы

**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

---

# BASE:

**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 с провляется с большими данными
  **Минусы:**
- Нет строгой согласованности
- Требуется много усилий для работы со старыми данными из-за времени на синхронизацию

---

## Сравнение ACID и BASE:

| 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 |

---

## Нормальные формы:

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

**Нормализация** - преобразования отношения к виду, отвечающему нормальной форме

**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
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 |
|---|---|
| M3210 | ИС |
| M3211 | ИС |
| M3212 | ИС |
| M3213 | ИС |
| M3214 | 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

---

## Триггеры:

**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

---

## Хранимые процедуры и функции

**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 |
| *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 |

---

## Горячее и холодное хранение данных

**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

---

## Стратегия хранения данных:

**Стратегия хранения данных** - это план распределения данных между горячим и холодным хранилищами, основанный на частоте использования, времени доступа и стоимости хранения

**Основные этапы разработки стратегии:**

1. **Классификация данных** - разделение данных на категории (часто используемы, редко используемы, бэкапы, архивы и тд)
2. **Определение требований к данным** - какие данные должны быть доступно прямо сейчас, а какие могут быть закинуты в архив
3. **Выбор технологий хранения** - нужно использовать горячее хранилище для часто используемых данных, а редко используемые данные в холодное хранилище
4. **Автоматизация переноса данных** - нужно, чтобы было настроенно автоматическое перемещение данных между горячими и холодными хранилищами в зависимости от их возраста или частоты использования

---

## Статистика в PostgreSQL:

**Статистика в PostgreSQL** — это набор данных, собираемых сервером базы данных для анализа состояния таблиц, индексов и запросов

**Какие данные собираются?**

1. **Общая информация о таблицах и индексах** - количество строк в таблицах, количество пустых или удалённых строк, объём данных таблицы и её индексов
2. **Распределение значений в столбцах** - минимальное и максимальное значение, среднее значение и дисперсия, частота появления значений

3. **Частотность запросов** - данные о количестве запросов к таблицам и индексам, информация о выполнении операций чтения, записи и обновления
4. **Информация о транзакциях** - количество выполненных транзакций, конфликты транзакцйий
5. **Системная статистика**

*Статистика PostgreSQL хранится в системных представлениях*

---

## Представление:

**Представление** - это результат выполнения операций над таблицами для создания нового отношения. Представления упрощают доступ к данным, повышают безопасность и могут оптимизировать производительность

**Виды представлений:**

1. **Табличное представление** - это виртуальная таблица, созданная на основе некого SQL запроса.

```
CREATE VIEW student_info AS
SELECT s.student_id, s.name, g.group_name
FROM students s
JOIN groups g ON s.group_id = g.group_id;
```

**Таблица студентов:**

| student_id | name | group_id |
|---|---|---|
| 1 | Вася Пупкин | M3212 |

**Таблица групп:**

| group_id | educational_program |
|---|---|
| M3212 | ИС |

**Результат представления:**

| student_id | name | educational_program |
|---|---|---|
| 1 | Вася Пупкин | ИС |

2. **Материализованные представления** - это представление, в котором результат выполнения запроса сохраняется физически в базе данных

```
CREATE MATERIALIZED VIEW cached_student_info AS
SELECT s.student_id, s.name, g.group_name
FROM students s
JOIN groups g ON s.group_id = g.group_id;
```

**Таблица студентов:**

| student_id | name | group_id |
|---|---|---|
| 1 | Вася Пупкин | M3212 |

**Таблица групп:**

| group_id | educational_program |
|---|---|
| M3212 | ИС |

**Результат представления:**

| student_id | name | educational_program |
|---|---|---|
| 1 | Вася Пупкин | ИС |

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** - материализованные 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