Базы Данных

Содержание

Лекция 1. Вводная лекция

Почему PostgreSQL?

- **Популярность** — большое сообщество, много документации и готовых решений.
- **MS SQL Server ушёл из России** — миграция на PostgreSQL стала практической необходимостью для многих компаний.
- **Open source** — бесплатный и с открытым исходным кодом.
- **Лёгкость запуска на любом устройстве** — в отличие от MS SQL Server, который тяжело запустить где-то кроме Windows.
- **Нетребовательна к ресурсам** — MS SQL Server не запустить на сервере с 2 ГБ ОЗУ, в то время как PostgreSQL может работать даже в очень скромных условиях (условно, от ~16 МБ).

Правила курса

- Сервера с mock-данными для практики.
- Методичка (ликбез по SQL).

Защита лабораторных работ

- Тестовый запрос, который вы не видели, — пишете его за отведённое время.
- 5 минут на запрос: если корректен — сдали.
- Пары дробятся на слоты по 25 минут.
- Запись через Telegram-бота (или Google-таблицу, если не успеют).
- Приходим ко времени, есть ~20 минут на написание теста.

Дедлайны

Жёстких дедлайнов нет — только мягкие, со снятием баллов: - Сдал вовремя — полный балл. - Не сдал — штрафы по арифметической прогрессии (-1, -1, -2, -3, …). - Сдавать желательно равномерно: количество людей на слот ограничено.

Распределение баллов

Компонент	Баллы
Лабораторные работы	60
Рубежное тестирование	20 (диапазон может меняться, обсуждается)

Компонент	Баллы
Экзамен	20

Рубежка одна, поэтому баллы могут перераспределить (например, 10 на рубежку и 30 на экзамен).

Разбалловка лаб

`6 + 8 + 10 + 12 + 12 + 12`

- Лабы 1 и 2 идут вместе.
- Лабы 3 и 4 — скорее всего, тоже вместе.
- Освободившиеся «лабы» могут уйти на другие СУБД, например MongoDB.

Полезная ссылка: https://info.sqlwars.ru

Что такое данные?

- **Данные** — нет контекста, нет осмысленности. Своего рода «атомы» — измеряются в байтах.
- К данным добавляется контекст — получается **информация**.
- **Знания** — субъективное понятие: по одной и той же информации каждый делает свой вывод.

  Данные — ресурс. Информация — продукт. Знания — сила.

База данных

Аналогия с библиотекой: - **Книги** — данные. - **Полки и каталоги** — структура базы данных. - **Библиотекарь** — Система Управления Базами Данных (СУБД).

Какую структуру данных лучше использовать? — всё зависит от специфики данных.

> База данных — это компонент информационной системы.

Почему бы не писать просто в файл?

- **Безопасность.** Если есть доступ к ОС, к файлу легко добраться напрямую.

- **Дубликаты.** Как исключить повторы? Сложно и долго.
- Плюс отсутствие транзакционности, многопользовательского доступа, индексов и т.д.

Требования к БД

- **Структура.**
- **Целостность.**
- **Многопользовательский доступ** (в т.ч. параллельный).
- **Безопасность** — даже при доступе к системному уровню данные защищены.
- **Язык запросов** — более-менее одинаков для всех БД (SQL).

  Нас не интересует, что под капотом у СУБД. Мы говорим, *что* нужно сделать, а *как* — это её забота.

Модель данных

> Архитектурный «чертёж» будущего здания, где кирпичи — это данные.

Формальное описание того, как мы будем хранить данные, как они связаны между собой и какие у них правила (ограничения).

Уровни моделирования

- **Концептуальный уровень**
  - Сущности и связи.
  - Модель для общения с заказчиками.
- **Логический уровень**
  - Для реляционной модели: проектирование без привязки к конкретной СУБД.
- **Физический уровень**
  - На какие диски будут положены данные?
  - Какие индексы создать для ускорения поиска?
  - Как партиционировать данные?

Реляционные базы данных

Виды связей: - one-to-one - one-to-many - many-to-one - many-to-many

> Избыточность приемлема ради оптимизации.

Лекция 2. Модели данных

Основные понятия

- **База данных** — организованная совокупность данных, хранящаяся в ЭВМ и отображающая некоторую предметную область.
- **Модель данных** — набор концепций, правил и структур, которые определяют, как хранятся данные в базе данных.
- **СУБД** — ПО для создания, манипулирования и совместного использования баз данных.
- **ERD** (Entity Relation Diagram) — показывает сущности и их связи.

У ERD есть разные нотации: - Воронья лапка (Crow’s foot) - E-диаграмма (нотация Чена) - IDEF1X и другие

Сущности и атрибуты

- **Сущности** — объекты.
- **Атрибуты** — свойства сущностей:
  - простые и составные;
  - обязательные и необязательные.

Модели данных

> Это решение определяет, насколько система будет производительной, масштабируемой, гибкой и в конечном итоге успешной. Неверный выбор модели данных подобен выбору неправильного фундамента для здания.

Реляционная

Самая популярная модель, основанная на реляционной алгебре.

Особенности: - Организует данные в виде двумерных таблиц. - Использует SQL для запросов. - Поддерживает **ACID**-свойства — гарантии инвариантности данных.

Преимущества: - Целостность и непротиворечивость данных за счёт ключей, ограничений и транзакций. - SQL как стандарт облегчает разработку, поддерж-

ку приложений и миграцию данных. - Нормализация устраняет избыточность и аномалии.

Недостатки: - Изменение структуры таблиц на работающей системе — сложная и дорогая операция. - Горизонтальное масштабирование сложнее, чем у некоторых NoSQL-решений. - Сложные `JOIN`-запросы на огромных таблицах могут выполняться медленно. - Неэффективна для хранения иерархических (древовидных) или графовых данных.

Применение: - **OLTP**-системы (Online Transaction Processing) — запросы на каждое действие пользователя. - Системы с высокими требованиями к целостности — везде, где есть деньги, всё должно быть «чётенько». - Когда данные чётко структурированы.

> OLTP vs OLAP: OLTP — много мелких онлайн-транзакций; OLAP (Online Analytical Processing) — аналитика, агрегации, отчёты по большим объёмам.

Семейства: SQL, NoSQL, NewSQL.

Колоночная

Особенности: - Вместо хранения записи целиком, данные организуются по столбцам. - Относятся к семейству NoSQL (хотя есть и реляционные представители). - Идеальна для **OLAP**-запросов. - По сути та же двумерная таблица, но «горизонталь» и «вертикаль» поменяны местами. - Много оптимизаций (компрессия, векторизованные вычисления).

Преимущества: - Высокая производительность аналитических запросов. - Эффективное сжатие данных (в столбце много однотипных значений). - Улучшенная производительность ввода-вывода. - При горизонтальном масштабировании — почти линейное увеличение производительности.

Недостатки: - Операции, характерные для OLTP, работают менее эффективно. - При частых изменениях данных возникают сложности в синхронизации и обеспечении целостности между столбцами.

Применение: big data, аналитика, BI.

Реализации: - Cassandra - ScyllaDB - Google BigQuery - ClickHouse (колоночная реляционная СУБД)

Документоориентированные

Особенности: - Данные хранятся в виде документов, а не строк таблицы. - Относятся к семейству NoSQL. - Документы обычно в формате BSON (a-la бинарный JSON). - Объединяются в коллекции (аналог таблиц, но без обязательного единого формата для всех записей). - Документы могут содержать вложенные объекты и массивы.

> **Пример: спроектируем хранение сериала.** В реляционной модели — куча таблиц и `FOREIGN KEY` (сериал → сезоны → серии → актёры …). В документоориентированной — просто массивы внутри одного документа. Если же использовать ссылки вместо вложенных объектов — теряем целостность ссылок (возможны «висячие» ссылки) — рэйс-кондишены и проблемы.

Преимущества: - Возможность изменения структуры документов без модификации общей схемы БД. - Документы в JSON/BSON естественно ложатся на объекты приложения.

Недостатки: - Ограниченная целостность данных. - Сложность запросов. - Потенциальная избыточность.

Применение: - Хранение логов. - OLTP-системы. - Разработка MVP. - Неструктурированные данные. - Обработка больших объёмов данных.

Реализации: - MongoDB - Firebase Realtime Database - Elasticsearch (поисковый движок) - Amazon DynamoDB - CouchDB

Графовая

Особенности: - Основные объекты — **узлы, рёбра, свойства**, а не таблицы. - Относятся к семейству NoSQL.

Преимущества: - Запросы на поиск путей, обход графа и анализ связей выполняются очень быстро. - Нет жёстко фиксированной схемы. - Легко добавлять новые типы связей и узлов без пересмотра всей структуры.

Недостатки: - Узко специализированы — за пределами «графовых» задач уступают другим моделям.

Реализации: - Neo4j - Dgraph

Ключ-значение

Особенности: - Простейшая модель: ключ → значение.

Преимущества: - Операции получения, вставки и удаления выполняются за $O(1)$. - Быстрые и нетребовательные к ресурсам.

Недостатки: - Не подходят для сложных запросов, агрегаций или выборок.

Применение: - Кеширование данных. - Хранение сессионных данных. - Реализация очередей. - Хранение конфигурационных данных.

Реализации: Redis, Memcached, etcd.

Временные ряды (Time Series)

Особенности: - Каждое значение привязано к моменту времени.

Преимущества: - Оптимизация под высокую скорость записи. - Простая агрегация и интерполяция. - Хранение больших объёмов данных. - Быстрый доступ к данным за определённые временные интервалы.

Применение: - Мониторинг и логирование. - Финансовые рынки. - Интернет вещей (IoT).

Реализации: InfluxDB, TimescaleDB, Prometheus.

Векторная

Особенности: - Данные сохраняются в виде векторов вещественных чисел (эмбеддингов).

Преимущества: - Быстро выполняют запросы поиска похожих векторов благодаря специальным индексам и алгоритмам (например, HNSW — Hierarchical Navigable Small World).

Применение: - Рекомендательные системы. - Поиск по смыслу (семантический поиск). - Кластеризация.

Реализации: - Pinecone - Milvus - Weaviate - pgvector (расширение PostgreSQL)

Объектные хранилища данных

Особенности: - Предназначены для работы с файлами: документами, изображениями, видео, аудио. - Файлы располагаются в каталогах и подкаталогах. - Хранят помимо файла метаданные.

Преимущества: - Интеграция с ОС и стандартными программами для работы с файлами. - Позволяют выстраивать сложные модели доступа. - Позволяют разворачивать CDN (Content Delivery Network).

Недостатки: - Поиск осуществляется по именам файлов и метаданным, а не по содержимому.

Применение: хранение слабоструктурированных данных, медиаконтент, бэкапы.

Реализации: - Amazon S3 - MinIO

Лекция 3. Атрибуты и ключи

Термины

- **Отношение** — таблица в БД.
- **Кортеж** — строка значений в таблице.
- **Атрибут** — столбец в таблице.

Свойства

- Отношения не упорядочены между собой.
- Атрибуты внутри отношения не упорядочены.
- Каждый кортеж уникален.

У каждого атрибута (столбца) должно быть уникальное имя в рамках своей таблицы.

У каждого кортежа есть по одному значению каждого атрибута, и это значение принадлежит домену атрибута в рамках таблицы.

> **INFO: Домен атрибута** Множество допустимых значений данного атрибута, определяющее его тип данных и накладывающее ограничения на значения.

Основные виды атрибутов

Вид атрибута	Описание	Пример
Простой	Неделимый, нельзя разделить на более мелкие части	Имя, цена
Составной	Делимый, состоит из нескольких податрибутов	Адрес (страна, город, улица, дом)
Однозначный	Имеет одно значение для каждой записи	Дата рождения, ID пользователя
Многозначный	Может иметь несколько значений	Номера телефонов, e-mail’ы человека
Производный	Вычисляется из других атрибутов	Стоимость со скидкой = `цена − цена × процент_скидки`

Основные виды доменов

Вид домена	Пример
Тип данных + ограничения	`VARCHAR(50)` (имя), `DECIMAL(10, 2)` (цена), ограничения `UNIQUE`, `NOT NULL`
Целостность данных	Связывание таблиц через `FOREIGN KEY` → `PRIMARY KEY` гарантирует связность и целостность данных
Согласованность	E-mail должен содержать `@`, возраст ≥ 0 и т.д.

Характеристики

- **Степень отношения** — количество атрибутов.
- **Кардинальность отношения** — количество кортежей в таблице.
- **Кардинальность атрибута** — количество *уникальных* значений в атрибуте.
- **Селективность** — количество (или доля) строк, удовлетворяющих выражению `WHERE`.

Реляционные ключи

- **Суперключ** — набор атрибутов, уникально идентифицирующий каждый кортеж.
- **Потенциальный ключ** — минимальный суперключ (нельзя выбросить ни одного атрибута без потери уникальности).
- **Первичный ключ (Primary Key)** — выбранный потенциальный ключ.
- **Внешний ключ (Foreign Key)** — ссылается на PK из другой таблицы.
- **Альтернативный ключ** — потенциальный ключ, не выбранный первичным.

Типы реляционных ключей

- **Простой** — состоит из одного атрибута.
- **Составной** — из нескольких атрибутов.
- **Естественный** — атрибут, который сущность содержит сама по себе и который имеет смысл (фамилия, если есть гарантия уникальности).
- **Суррогатный** — атрибут, не имеющий смысла для сущности (например, `id`).

IMPORTANT: > Главный недостаток естественного ключа — он иногда может меняться (например, фамилия после замужества).

TIP: > Лучше всего использовать суррогатный ключ: - целочисленный (`SERIAL`, `BIGSERIAL`), или - UUID (Universal Unique Identifier).

Типы данных

- Целочисленные (`smallint`, `int`, `bigint`)
- С плавающей точкой (`decimal`, `real`)
- Строки (`varchar`, `char`, `text`)
- Дата и время (`date`, `time`, `timestamp`, `interval`)
- Логический (`bool`)
- Геометрические (`point`, `path`, `line`)
- JSON
- `NULL`

Тип данных	Пояснение
`smallint`	Целое небольшого диапазона ($-32768 \ldots 32767$)
`int`	Стандартное целое, 4 байта ($-2^{31} \ldots 2^{31}-1$)
`bigint`	Очень большие целые числа, 8 байт
`decimal`	Число с фиксированной точностью
`real`	Число с плавающей точкой (аналог `float`)
`varchar`	Строка переменной длины с ограничением
`char`	Строка фиксированной длины
`text`	Длинный текст без ограничения длины
`date` / `time`	Дата / время
`timestamp`	Отметка времени (дата + время)
`interval`	Промежуток времени
`bool`	`true` / `false`
`point`	Геометрическая точка
`path`	Геометрический путь из нескольких точек
`line`	Линия бесконечной длины
`JSON`	Структурированные данные в формате JSON

DML

> **INFO: DML — Data Manipulation Language** Язык манипулирования данными.

Виды	Описание
`SELECT`	Получение данных из таблиц
`INSERT`	Добавляет новые кортежи в таблицу
`UPDATE`	Обновляет существующие кортежи
`DELETE`	Удаляет существующие записи по условию
`MERGE`	Выполняет вставку и обновление одновременно: • запись существует → обновляем • записи нет → добавляем новую

DDL

> **INFO: DDL — Data Definition Language** Язык описания (определения) данных.

Виды	Описание
`CREATE TABLE`	Создаёт таблицу
`ALTER TABLE`	Изменяет структуру существующей таблицы
`DROP`	Удаляет таблицу из БД
`TRUNCATE`	Удаляет все кортежи, оставляя структуру таблицы для дальнейшего использования

Лекция 4. Реляционная модель

Реляционная модель опирается на: - **реляционную алгебру**; - **реляционное исчисление**.

Один из первых языков, основанных на реляционной алгебре, — **ISBL**.

> **Эдгар Франк «Тед» Кодд** (1923–2003) — британский учёный, работавший в IBM. Его работы заложили основы теории реляционных баз данных: именно он создал реляционную модель данных.
>
> **Кристофер Дейт** дал наиболее распространённую трактовку реляционной модели. По его мнению, модель состоит из трёх частей: **структурной**, **манипуляционной** и **целостной**.

Типы операций

- **Теоретико-множественные** — `UNION`, `INTERSECT`, разность, декартово произведение.
- **Специальные реляционные** — выборка, проекция, соединение, деление.

Выборка (унарная операция)

Применяется к одному отношению и определяет результирующее отношение, содержащее только те кортежи, которые удовлетворяют заданному условию (предикату).

$$\sigma_{\text{предикат}}(R)$$

```sql
SELECT * FROM "Персоны" WHERE "Возраст" >= 34;
```

Проекция (унарная операция)

Применяется к одному отношению и определяет новое отношение, содержащее **вертикальное подмножество** исходного отношения: извлекаются значения указанных атрибутов, а строки-дубликаты исключаются.

$$\Pi_{a_1,\ldots,a_n}(R)$$

```sql
SELECT "Возраст", "Вес" FROM "Персоны";
```

Объединение

Объединение двух отношений R и S определяет новое отношение, включающее все кортежи, содержащиеся в R, все кортежи, содержащиеся в S, и кортежи, содержащиеся одновременно в обоих (с исключением дубликатов).

R и S должны быть **совместимы по объединению**: одинаковое количество атрибутов и одинаковые домены соответствующих пар. Имена атрибутов могут не совпадать — главное, чтобы совпадали домены. Для совместимости можно предварительно применить проекцию.

$$R \cup S$$

```sql
SELECT * FROM "Персоны" UNION SELECT * FROM "Сотрудники";
```

Пересечение

Отношение, содержащее кортежи, присутствующие как в R, так и в S. Отношения должны быть совместимы по объединению.

$$R \cap S$$

Разность

Кортежи, имеющиеся в R, но отсутствующие в S. Отношения должны быть совместимы по объединению.

$$R - S$$

Декартово произведение

Новое отношение, являющееся результатом конкатенации (сцепления) каждого кортежа из R с каждым кортежем из S.

$$R \times S$$

В чистом виде применяется редко, но служит основой для построения соединений.

Деление

Определяет отношение из множества кортежей R, определённых на атрибуте C, соответствующих комбинации всех кортежей S, где C — множество атрибутов, имеющихся в R, но отсутствующих в S.

Соединение

- **Тета-соединение (θ-join).** Определяет отношение, содержащее кортежи из декартова произведения R и S, удовлетворяющие предикату F вида $R.a_i\ \theta\ S.b_i$, где θ — операция сравнения (<, <=, =, и т.д.).

$$R \bowtie_F S$$

- **Эквисоединение (Equi-join).** Если используются только сравнения по равенству.
- **Естественное соединение (Natural join).** Эквисоединение двух отношений, выполненное по всем общим атрибутам, из результата которого исключается по одному экземпляру каждого общего атрибута.

$$R \bowtie S$$

Критика Дейта

Избыточность

- Достаточно всего пяти базовых операций.
- Пересечение, соединение и деление выводятся из остальных.

Недостаточность

- Не хватает операций:
  - переименования атрибутов;
  - вычисления атрибутов;
  - агрегирующих функций (`SUM`, `AVG`, и т.д.);

- присваивания результатов временным отношениям.

Синтаксический порядок SELECT

(Порядок написания и порядок логического исполнения отличаются — это важно помнить.)

Выбор типа данных для времени

- **TIMESTAMPTZ** (timestamp with time zone)
  - Если важны события и их последовательность — логи, аудиты.
  - При запросе смотрит на клиентскую тайм-зону.
  - Сам не хранит тайм-зону — хранит UTC, постоянно конвертирует.
- **TIMESTAMP** (без зоны)
  - Подходит для дат из прошлого, для которых не важен часовой пояс.

  Пример. Сервер на UTC-3, клиент на UTC+3. Открываем 3 разных клиента (в разных часовых поясах) и запрашиваем текущее время. С TIMESTAMPTZ каждый клиент увидит время в своей локальной зоне — момент времени один и тот же, представление разное.

INSERT

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING [ WITH ( { OLD | NEW } AS output_alias [, ...] ) ]
                { * | output_expression [ [ AS ] output_name ] } [, ...] ];

-- conflict_target:
--   ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [
--       [ WHERE index_predicate ]
--   ON CONSTRAINT constraint_name
--
-- conflict_action:
--   DO NOTHING
```

```
--   DO UPDATE SET { column_name = { expression | DEFAULT } | ... } [, ...]
--               [ WHERE condition ]
```

При конфликте обработка идёт в `ON CONFLICT`: - `ON CONFLICT DO NOTHING` — игнорируем конфликт, ничего не вставляем и не обновляем. - `ON CONFLICT DO UPDATE SET ...` — обновляем существующую запись (паттерн **UPSERT**).

UPDATE

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } | ... } [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING [ WITH ( { OLD | NEW } AS output_alias [, ...] ) ]
                { * | output_expression [ [ AS ] output_name ] } [, ...] ];
```

Лекция 5. Нормализация

Определение

Нормализация — процесс организации данных в базе с двумя основными целями: устранение избыточности данных и обеспечение целостности данных.

Избыточность и аномалии

- **Аномалия добавления (вставки).** Невозможно добавить данные об одном объекте, не добавив данные о другом. *Пример: чтобы добавить студента, посещающего «Математику», приходится добавлять и саму «Математику», если её ещё нет в БД.*
- **Аномалия удаления.** При удалении одних данных мы невольно теряем другие. *Пример: если только один студент посещал «Математику», то, удаляя его, мы потеряем и всю информацию о предмете.*
- **Аномалия обновления.** Чтобы изменить значение, его приходится обновлять в нескольких местах. *Пример: если имя студента дублируется в нескольких строках, фиксить нужно каждое вхождение.*

Функциональная зависимость

Функциональная зависимость $A \to B$ означает, что каждому значению атрибута A соответствует **ровно одно** значение атрибута B. Зная A, мы однозначно определяем B.

- НомерПаспорта $\to$ ФИО — зная номер паспорта, мы найдём только одного человека.
- НомерЗаказа $\to$ ДатаЗаказа, Сумма — зная номер заказа, узнаём его дату и сумму.

Первая нормальная форма (1НФ)

Переменная отношения находится в 1НФ тогда и только тогда, когда в любом допустимом значении отношения каждый кортеж содержит **только одно значение** каждого из атрибутов.

Проблема. Пусть у человека более одного телефона. - Создавать атрибуты `phone1`, `phone2`, …? Непонятно, сколько столбцов делать, и `UNIQUE` уже не сработает (одинаковые номера могут оказаться в разных столбцах). - Складывать все номера в одну ячейку? Поиск конкретного номера и `UNIQUE`-проверки становятся медленными и неудобными.

Решение. Несколько раз запишем одного и того же человека, но в каждом кортеже — другой номер телефона (отдельная таблица «человек—телефон»).

> **Формальное определение (википедия).** Отношение находится в 1НФ тогда и только тогда, когда: - в отношении нет повторяющихся групп (атрибутов с одинаковым смыслом); - все атрибуты атомарны; - у отношения есть ключ.

Вторая нормальная форма (2НФ)

> Переменная отношения находится во 2НФ тогда и только тогда, когда она находится в 1НФ **и каждый неключевой атрибут неприводимо зависит от каждого её потенциального ключа** (требования уникальности и минимальности ключа).

Пример: пусть есть составной потенциальный ключ `(ИСУ, Предмет)`. Неключевой атрибут «Имя» зависит только от `ИСУ`, то есть от части ключа — частичная зависимость.

Решение. Декомпозиция: разделяем таблицу так, чтобы все неключевые атрибуты полностью зависели от ключа.

> Невозможно перейти ко 2НФ, не приведя сначала отношение к 1НФ.

Третья нормальная форма (3НФ)

> Переменная отношения находится в 3НФ тогда и только тогда, когда она находится во 2НФ **и отсутствуют транзитивные функциональные зависимости неключевых атрибутов от ключа**.

Пример: «Человек → Кафедра → Номер_телефона_кафедры». Номер не зависит от человека напрямую, только от кафедры.

Решение. Декомпозиция: вынести «Кафедру» в отдельную таблицу с её собственным телефоном.

Нормальная форма Бойса-Кодда (НФБК / BCNF)

Переменная отношения находится в НФБК, когда она находится в 3НФ **и ключевые атрибуты составного ключа не зависят от неключевых атрибутов** (точнее: для каждой нетривиальной ФЗ $X \to Y$ детерминант X является суперключом).

Таблицы с простым первичным ключом, находящиеся в 3НФ, автоматически соответствуют НФБК.

Высшие нормальные формы

- **4НФ.** Борется с многозначными зависимостями — когда один атрибут определяет множество независимых значений других атрибутов.
- **5НФ.** Борется с зависимостями соединения без потерь — когда отношение можно без искажений восстановить только из трёх и более проекций.

Многозначные зависимости

X многозначно определяет Y тогда и только тогда, когда для каждого значения X существует набор значений Y, и этот набор **не зависит** от значений других атрибутов Z.

Если зафиксировать значение X, то получим **независимые** друг от друга множества соответствующих значений Y и Z.

Многозначная зависимость возникает, когда у одной сущности есть две или более независимых характеристики «многие-ко-многим».

Четвёртая нормальная форма (4НФ)

Переменная отношения находится в 4НФ, если она находится в НФБК и не содержит нетривиальных многозначных зависимостей.

Пример 4НФ Преподаватель Иванов ведёт два предмета (Математика, Физика) и у него два студента (Петров, Сидоров). Чтобы отразить все парные комбинации, приходится создать 4 строки.

- **Аномалия вставки.** Чтобы добавить Иванову третьего студента, нужно вставить **две** новых строки: одну для Математики, другую для Физики.

- **Аномалия удаления.** Если Сидоров перестал изучать Физику у Иванова — какую строку удалить? Удаление одной строки нарушит целостность данных.

Решение. Декомпозиция на два отношения: `(Преподаватель, Предмет)` и `(Преподаватель, Студент)`.

Пятая нормальная форма (5НФ)

> Отношение находится в 5НФ, если каждая нетривиальная зависимость соединения в нём следует из его потенциальных ключей.
>
> Отношение в 5НФ не может быть без потерь декомпозировано на несколько меньших отношений и потом корректно восстановлено. Любая такая попытка приведёт либо к потере данных, либо к появлению «фантомных» данных.

Пример 5НФ Бизнес-правило: «Если поставщик S поставляет деталь P, и деталь P используется в проекте J, и поставщик S работает над проектом J, то поставщик S поставляет деталь P для проекта J».

Это **циклическое ограничение**: таблица должна быть замкнута относительно этого правила. Если есть пары (S, P), (P, J) и (S, J), то должна существовать и тройка (S, P, J).

Аномалия. Если декомпозировать таблицу на любые две проекции (например, `(Поставщик, Деталь)` и `(Деталь, Проект)`) и обратно их соединить — получим фиктивные (ложные) строки.

Решение. Перепроектирование отношений.

Пример: `staff_property_inspection`

Отчёт об осмотрах объектов недвижимости сотрудниками.

Проблема. Повторяющиеся группы: адрес объекта и имя сотрудника дублируются для каждого осмотра.

property_no	property_address	inspection_date	inspection_time	comments	staff_no	staff_name	car_registration
PG4	6 Lawrence St...	18-Oct-00	10:00	Need to replace	SG37	Ann Beech	M231JGR
PG4	6 Lawrence St...	22-Apr-01	09:00	In good order	SG14	David Ford	M533HDR
PG4	6 Lawrence St...	1-Oct-01	12:00	Damp rot in bathroom	SG14	David Ford	N721HFR

1НФ

- Устранены повторяющиеся группы. Первичный ключ — `(property_no, inspection_date)`.
- `property_address` частично зависит только от `property_no` — это **нарушает 2НФ**.

property_no	inspection_date	inspection_time	property_address	comments	staff_no	staff_name	car_registration
PG4	18-Oct-00	10:00	6 Lawrence St...	Need to replace	SG37	Ann Beech	M231JGR
PG4	22-Apr-01	09:00	6 Lawrence St...	In good order	SG14	David Ford	M533HDR
PG4	1-Oct-01	12:00	6 Lawrence St...	Damp rot in bathroom	SG14	David Ford	N721HFR

2НФ

Устранили частичную зависимость, вынеся адрес в отдельную таблицу. Неключевые атрибуты зависят от всего первичного ключа. Атрибут `staff_name` **транзитивно** зависит от первичного ключа через `staff_no`, что нарушает 3НФ.

property_no	inspection_date	inspection_time	comments	staff_no	staff_name	car_registration
PG4	18-Oct-00	10:00	Need to replace	SG37	Ann Beech	M231JGR
PG4	22-Apr-01	09:00	In good order	SG14	David Ford	M533HDR
PG4	1-Oct-01	12:00	Damp rot in bathroom	SG14	David Ford	N721HFR

property_no	property_address
PG4	6 Lawrence St…
PG16	5 Novar Dr…

3НФ

Устранены транзитивные зависимости. Зависимость `(staff_no, inspection_date) → car_registration` нарушает НФБК, так как детерминант не является потенциальным ключом.

property_no	inspection_date	inspection_time	comments	staff_no	car_registration
PG4	18-Oct-00	10:00	Need to replace	SG37	M231JGR
PG4	22-Apr-01	09:00	In good order	SG14	M533HDR
PG4	1-Oct-01	12:00	Damp rot in bathroom	SG14	N721HFR

staff_no	staff_name
SG37	Ann Beech
SG14	David Ford

НФБК

property_no	inspection_date	inspection_time	comments	staff_no
PG4	18-Oct-00	10:00	Need to replace	SG37
PG4	22-Apr-01	09:00	In good order	SG14

property_no	inspection_date	inspection_time	comments	staff_no
PG4	1-Oct-01	12:00	Damp rot in bathroom	SG14

staff_no	car_registration	inspection_date
SG37	M231JGR	18-Oct-00
SG14	M533HDR	22-Apr-01
SG14	N721HFR	1-Oct-01

DELETE

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING [ WITH ( { OLD | NEW } AS output_alias [, ...] ) ]
                { * | output_expression [ [ AS ] output_name ] } [, ...] ];
```

ON DELETE

Поведение при удалении строки, на которую ссылается внешний ключ:

- **RESTRICT** — запрещает удаление, если есть зависимые строки (немедленная проверка).
- **NO ACTION** — то же, но проверка откладывается до конца транзакции (по умолчанию).
- **SET NULL** — в зависимых строках значение внешнего ключа становится NULL.
- **SET DEFAULT** — в зависимых строках устанавливается значение по умолчанию.
- (Также есть **CASCADE** — каскадное удаление зависимых строк.)

Лекция 6. Индексы

Определение

- **Индексы** — объекты базы данных, предназначенные для **ускорения поиска и выборки** строк из таблицы за счёт минимизации количества данных, которые нужно просматривать.
- Индексы также служат для поддержания некоторых **ограничений целостности** (например, `UNIQUE`, `PRIMARY KEY`).

Как работает индекс?

Устанавливает соответствие между **ключом** (например, значением проиндексированного столбца) и **строками таблицы**, в которых этот ключ встречается. Строки идентифицируются с помощью **TID (tuple id)**, который состоит из: - номера блока файла; - позиции строки внутри блока.

Увеличение затрат на запись

При любой операции над проиндексированными данными — будь то вставка, удаление или обновление строк таблицы — индексы, созданные для этой таблицы, должны быть перестроены, причём в рамках той же транзакции.

> Платим записью ради ускорения чтения. Перебарщивать с индексами вредно — есть риск замедлить запись больше, чем выиграть на чтении.

Типы индексов в PostgreSQL

- **B-tree**
- **Hash**
- **GIN**
- **GiST**
- **SP-GiST**
- **BRIN**

Hash-индекс

- **Поддерживает только поиск по равенству.**

- По мере увеличения количества индексируемых строк одна из корзин расщепляется на две (динамическое хеширование).
- Элементы корзин упорядочены по хеш-кодам ключей, подходящие идентификаторы эффективно находятся двоичным поиском.
- До версии PostgreSQL 10 хеш-индексы **не журналировались** (не попадали в WAL), что делало их небезопасными при сбоях.

Ограничения

- Кластеризация таблицы по хеш-индексу **не предусмотрена**.
- Хеш-функция не сохраняет порядок — к хеш-индексу неприменимы свойства упорядоченности.
- **Не может участвовать в сканировании только индекса** (Index Only Scan): не сохраняет ключ индексации и требует перепроверки по таблице.
- **Не работает с `NULL`**: операция «равно» не имеет смысла для `NULL`.
- Поиск значений из массива не реализован.

B-tree

Свойства

- **Сбалансировано**: все листья находятся на одной глубине. Поэтому поиск любого значения занимает одинаковое время.
- **Сильно ветвисто**: каждый узел содержит много элементов (часто сотни). За счёт этого глубина B-деревьев получается небольшой даже для очень больших таблиц.
- **Упорядочено**: данные в индексе отсортированы по возрастанию как между узлами, так и внутри каждого узла. Узлы одного уровня связаны двунаправленным списком.

Что поддерживает B-tree

- Поиск по равенству;
- Поиск по неравенству;
- Поиск по диапазону;
- Поиск по префиксу (для строк, например `LIKE 'abc%'`);
- Оптимизация сортировки (`ORDER BY`);

- Уникальность (`UNIQUE`).

Особенности

- **Дубликаты «схлопываются»** в одну индексную запись, содержащую ключ и список табличных идентификаторов.
- Уникальные индексы могут содержать дубликаты ключей из-за **многоверсионности** (MVCC), поскольку индекс хранит ссылки на все версии табличных строк.

GiST (Generalized Search Tree)

GiST — **не конкретный тип индекса, а фреймворк**, который позволяет создавать пользовательские реализации для различных видов данных (геометрия, диапазоны, полнотекстовый поиск и др.).

R-дерево

- Идея R-дерева: плоскость разбивается на прямоугольники, которые в сумме покрывают все индексируемые точки.
- Индексная запись хранит **ограничивающий прямоугольник**, а предикат можно сформулировать так: точка лежит внутри данного ограничивающего прямоугольника.

Типичный запрос, ускоряемый таким индексом, — получить все точки, входящие в заданную область.

k-NN (k-Nearest Neighbors)

GiST позволяет эффективно искать k ближайших соседей к заданной точке.

RD-дерево для полнотекстового поиска

- Задача: выбрать из набора документов те, которые соответствуют поисковому запросу.
- Для целей поиска документ приводится к специальному типу `tsvector`, содержащему лексемы и их позиции в документе.

Полнотекстовый поиск Чтобы полнотекстовый поиск работал быстро, его нужно поддержать индексом. Поскольку индексируются не сами документы, а значения `tsvector`, есть два варианта: - построить индекс по выражению с приведением типа; - создать отдельный столбец типа `tsvector` и индексировать его.

Идея RD-дерева R-дерево как таковое не годится для индексации документов, поскольку к ним неприменимо понятие ограничивающего прямоугольника. Используется модификация — **RD-дерево** (Russian Doll, «матрёшка»).

Вместо ограничивающего прямоугольника это дерево использует **ограничивающее множество**, то есть множество, содержащее все элементы дочерних множеств.

Сигнатурное дерево Используется **фильтр Блума**. Каждую лексему можно представить **сигнатурой** — битовой строкой определённой длины, в которой все биты равны нулю, кроме одного, который равен единице. Номер установленного бита определяется значением хеш-функции от лексемы.

GIN (Generalized Inverted Index)

Аналогия

Предметный указатель в конце книги: собраны все важные термины, и для каждого приведён список страниц. Чтобы указателем было удобно пользоваться, он составляется по алфавиту. Так и GIN полагается на то, что элементы составных значений можно упорядочить, и в качестве основной структуры использует B-tree.

Назначение

Тип индексной структуры, предназначенной для работы с **коллекциями данных** и для быстрого поиска по составным структурам. Особенно эффективен, когда одно поле содержит множество значений (массивы, JSON, `tsvector`).

Лекция 7. Транзакции

ACID

- **A — Atomicity (Атомарность).** Транзакция выполняется полностью или не выполняется совсем.
- **C — Consistency (Согласованность).** После завершения транзакции БД остаётся в согласованном состоянии, то есть не возникает нарушений ограничений целостности. Ограничения целостности:
  - `NOT NULL`
  - `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK` — проверка условий
  - **Exclusion constraints** — расширенные ограничения
- **I — Isolation (Изоляция).** Параллельные транзакции выполняются так, как если бы они были последовательными, предотвращая конфликты данных.
- **D — Durability (Долговечность).** После фиксации (`COMMIT`) изменения остаются в базе данных, даже если система выйдет из строя.

Уровни изоляции транзакций

- **Read Uncommitted** — позволяет читать даже незавершённые изменения других транзакций. *(Есть в стандарте SQL, но в PostgreSQL фактически работает как Read Committed.)*
- **Read Committed** *(по умолчанию в PostgreSQL)* — транзакция видит только зафиксированные (`COMMIT`-нутые) изменения других транзакций.
- **Repeatable Read** — транзакция видит данные в том виде, в каком они были на момент её начала, даже если другие транзакции вносят изменения.
- **Serializable** — максимальный уровень изоляции. Гарантирует, что параллельные транзакции выполняются так, как если бы они шли последовательно.

Аномалии

- **Потерянное обновление (Lost Update).** Две транзакции читают одну и ту же строку, затем одна обновляет её, а после этого вторая тоже обновляет — не учитывая изменений первой.

- **Грязное чтение (Dirty Read).** Транзакция читает ещё не зафиксированные изменения, сделанные другой транзакцией.
- **Неповторяющееся чтение (Non-Repeatable Read).** Транзакция дважды читает одну и ту же строку и получает разные значения, потому что другая транзакция успела её изменить и закоммитить между чтениями.
- **Фантомное чтение (Phantom Read).** Транзакция дважды читает набор строк по одному и тому же условию, а в промежутке между чтениями вторая транзакция добавляет строки, удовлетворяющие этому условию (и фиксирует изменения). Тогда первая транзакция получает разные **наборы** строк.

Serializable

Уровень Serializable должен предотвращать **вообще все аномалии**. Это означает, что разработчику приложения не надо думать об одновременном выполнении: если транзакции выполняют корректные последовательности операторов, работая в одиночку, данные будут согласованы и при их одновременной работе.

Аномалии сериализации

- **Циклическая зависимость данных (Write Skew).**

Почему именно такие уровни в стандарте?

Разница между уровнями изоляции стандарта объясняется как раз количеством необходимых блокировок:

- Транзакция блокирует изменяемые строки **от изменения**, но не от чтения → **Read Uncommitted**.
- Транзакция блокирует изменяемые строки **и от чтения, и от изменения** → **Read Committed**.
- Транзакция блокирует и читаемые, и изменяемые строки и от чтения, и от изменения → **Repeatable Read**.
- С Serializable проблема: невозможно заблокировать строку, которой ещё нет. Из-за этого остаётся возможность фантомного чтения. Поэтому для реализации Serializable обычных блокировок не хватает — нужно блокировать не строки, а **условия** (предикаты). Такие блокировки и были названы **предикатными**.

Write-Ahead Logging (WAL)

WAL — стратегия, при которой все изменения данных сначала записываются в журнал (лог), и только потом применяются к основным файлам базы данных. Это даёт долговечность (`Durability` в ACID): даже если сервер упадёт, изменения можно восстановить по журналу.

Структура WAL-записи

- **Header** (24 байта)
- **Page ID** (8 байт)
- **Old Data** (размер страницы)
- **New Data** (размер страницы)
- **Checksum** (8 байт)

Ключевые параметры

```
fsync = on
synchronous_commit = on
wal_buffers = 64MB
```

`synchronous_commit`

- `on` — `COMMIT` дожидается, пока WAL гарантированно сброшен на диск (максимальная надёжность).
- `off` — `COMMIT` возвращается сразу; данные могут быть потеряны при крахе (но БД останется консистентной).
- `local` — ждём fsync только локально (без подтверждения реплик).
- `remote_write` — ждём, пока WAL дошёл до памяти standby-реплики.
- `remote_apply` — ждём, пока WAL применён на standby (самый строгий режим).

Checkpoint

Checkpoint — процесс, при котором все изменяемые данные из буферного кэша сбрасываются на диск, создавая согласованное состояние базы данных. После чекпоинта старые WAL-записи становятся не нужны для восстановления.

Лекция 8. Таблицы и хранение данных

Хеш-таблица как движок хранения

- Данные хранятся в файле, в который можно только дописывать в конец (**append-only log**).
- В оперативной памяти хранится хеш-таблица (словарь): **ключ → смещение (offset) в файле данных**.

Запись: добавляем пару «ключ-значение» в конец файла и обновляем хеш-таблицу. **Чтение:** по ключу находим смещение в хеш-таблице, переходим по нему в файле и читаем значение.

Сегменты

- Разбиваем журнал на сегменты фиксированного размера.
- При заполнении сегмента — закрываем его и начинаем новый.
- Запускаем фоновый процесс **уплотнения (compaction)** и **слияния (merging)** сегментов.

Уплотнение (compaction)

- Удаление дубликатов ключей.
- Сохранение только самой последней версии значения для каждого ключа.

Слияние (merging)

- Объединение нескольких небольших сегментов в один новый.
- Позволяет уменьшить общее количество сегментов.

Алгоритм чтения

- Поиск начинается в хеш-таблице самого свежего сегмента.
- Если ключ не найден — проверяется следующий по старшинству сегмент.
- И так далее, пока ключ не будет найден или не будут проверены все сегменты.

Преимущества append-only подхода

- **Высокая производительность записи.** Последовательная запись на диск намного быстрее случайной.

- **Надёжность.** Упрощение конкурентного доступа и восстановления после сбоев — нет риска «полузаписанных» данных.
- **Борьба с фрагментацией.** Процесс слияния сегментов естественным образом переписывает данные в компактном виде.

Ограничения хеш-движка

- **Ключи должны помещаться в ОЗУ.** Хеш-таблица на диске медленная из-за случайных чтений, дорогого расширения и сложности разрешения коллизий.
- **Неэффективные диапазонные запросы.** Нельзя быстро найти все ключи от `00000` до `99999` — нужно обращаться к хеш-таблице для каждого ключа в диапазоне отдельно.

SS-Table (Sorted String Table)

Особенности

- Данные **отсортированы по ключу**.
- Каждый ключ встречается **только один раз** (обеспечивается уплотнением).
- Порядок записи значений не важен — приоритет у более новых значений.

Слияние

- Алгоритм аналогичен **сортировке слиянием** (merge sort).
- Работает даже когда данные не помещаются в оперативной памяти.
- При конфликте ключей берётся значение из самого нового сегмента.

Разреженный индекс

- Не нужно хранить все ключи — только некоторые ориентиры.
- *Пример:* известны смещения для `"handbag"` и `"handsome"` → `"handiwork"` находится между ними.
- Можно быстро просканировать небольшой диапазон.

MemTable

- Сбалансированное дерево в оперативной памяти (красно-чёрное, AVL).

- Данные сохраняются отсортированными по ключу.
- Быстрая вставка в любом порядке.

Надёжность

- Все операции немедленно записываются в журнал (**WAL**).
- Журнал неупорядочен — используется только для восстановления.
- После записи MemTable в SS-таблицу журнал удаляется.

Алгоритм работы (LSM-tree)

- **Запись.** Добавляется в MemTable и параллельно пишется в журнал для надёжности.
- **Чтение.** Поиск в MemTable → последнем сегменте → предыдущем → и т.д.
- **Фоновые процессы.** При превышении лимита MemTable записывается в SS-таблицу. Регулярное слияние и уплотнение сегментов.

Фильтр Блума

- Эффективная проверка **отсутствия** ключа.
- Избегание ненужных чтений с диска.
- Ответ: «возможно есть» / «точно нет».

Write-Ahead Log

- Файл только для добавления (append-only).
- Быстрая последовательная запись.
- Используется для восстановления после сбоев.

Защёлки (latches)

- Облегчённые версии блокировок.
- Защита структур данных при одновременном доступе.
- Высокая производительность (быстрее обычных блокировок, но используются для коротких критических секций).

LSM-tree vs B-tree

- **LSM-деревья:** обычно быстрее при записи.
- **B-деревья:** обычно быстрее при чтении.

Модель доступа

Дискреционная модель доступа (DAC)

> Модель управления доступом, при которой **владельцы объектов** (таблиц, схем, представлений и т.д.) в базе данных самостоятельно определяют, кто и какие действия может выполнять с этими объектами.

Роли

- В PostgreSQL понятие **«пользователь»** является частным случаем **роли**. Каждая роль может иметь или не иметь возможность входа в систему (login privilege). Если у роли установлено свойство `LOGIN`, она может использоваться как учётная запись пользователя.
- Роли могут быть **членами других ролей**. Это позволяет организовывать группы пользователей — например, создать роль `developers`, которой будут принадлежать все разработчики, и назначить этой роли общие привилегии.

Команды и атрибуты

```
CREATE ROLE name [attr1, attr2, ...];
ALTER ROLE developer WITH PASSWORD 'newsecret';
```

Атрибуты ролей: - `LOGIN` — возможность входа в систему. - `SUPERUSER` — суперпользователь, обходящий все проверки прав. - `CREATEDB` — может создавать базы данных. - `CREATEROLE` — может создавать другие роли. - `PASSWORD 'secret'` — пароль.

Наследование привилегий

- Роль, являющаяся членом другой роли, по умолчанию автоматически наследует все её привилегии.

```
GRANT role1 TO role2;
REVOKE role1 FROM role2;
```

Привилегии таблиц

- SELECT
- INSERT
- UPDATE
- REFERENCES
- DELETE
- TRUNCATE
- TRIGGER

Привилегии БД

- CREATE
- CONNECT
- TEMPORARY

Привилегии схем

- CREATE
- USAGE

Права доступа к схеме влияют на возможность создания новых объектов, но сами по себе **не контролируют доступ** к уже существующим объектам. Для этого необходимо назначать привилегии непосредственно на уровне объектов.

Лекция 9. Этапы выполнения запроса

Лексический и синтаксический разбор

- Сервер принимает SQL-текст запроса и анализирует его на наличие синтаксических ошибок.
- Результат — **дерево разбора (parse tree)**, представляющее структуру запроса.

Семантический разбор

- Заменяет представления (`VIEW`) на их определения и обрабатывает правила, определённые для таблиц.
- Проверяет права пользователя на объекты.
- Вся необходимая для семантического анализа информация хранится в **системном каталоге**.
- Семантический анализатор получает от синтаксического дерево разбора и перестраивает его, дополняя ссылками на конкретные объекты БД, указанием типов данных и другой информацией.

Планирование

- Любой запрос можно выполнить разными способами.
- План выполнения также представляется в виде дерева, но его узлы содержат не логические, а **физические операции** над данными.
- Текстовое представление плана выводит команда `EXPLAIN`.
- **Планировщик** (planner) принимает переписанное дерево запроса и генерирует один или несколько вариантов выполнения.
- **Оптимизатор** оценивает стоимость каждого варианта плана, используя статистику о таблицах, и выбирает наиболее выгодный.

Планы

- Количество возможных планов **экспоненциально** зависит от количества соединяемых таблиц.
- Для сокращения пространства перебора традиционно используется **динамическое программирование** в сочетании с эвристиками.
- Точное решение задачи оптимизации **не гарантирует**, что найденный план действительно будет лучшим (оценки могут быть неточны).

Выбор лучшего плана: общие и частные планы

При выполнении **подготовленных запросов** (`PREPARE`) план может быть сохранён и повторно использован:

- **Общий план (generic plan)** — строится один раз, без учёта конкретных значений параметров.
- **Частный план (custom plan)** — создаётся при каждом выполнении и учитывает конкретные значения параметров.

При первом выполнении подготовленного запроса PostgreSQL планирует его с учётом переданных параметров (частный план). Однако, если запрос выполняется многократно, система может перейти к общему плану. Если общий план оказывается более выгодным по суммарным затратам, PostgreSQL будет использовать его для последующих вызовов.

Выполнение

- После выбора оптимального плана **исполнитель** запускает последовательность операций, предусмотренных планом, последовательно проходя по узлам.
- Доступ к данным осуществляется через **буферный менеджер**, который минимизирует количество операций ввода-вывода на диск.
- Если запрос рассчитан на параллельное выполнение, результат собирается из нескольких рабочих процессов и объединяется в итоговый набор данных.

Статистика

> **Статистика** — набор данных, который собирается системой для оценки распределения значений в таблицах и столбцах, что затем используется планировщиком запросов для выбора наиболее эффективного плана.

Хранение

- PostgreSQL хранит собранную статистику в системной таблице `pg_statistic`.
- Прямой доступ к `pg_statistic` обычно не требуется — для просмотра статистики используют представление `pg_stats`.
- `pg_stats` — публичное представление, предоставляющее информацию из `pg_statistic` в удобном виде, без избыточных деталей, доступных только

суперпользователям.
- Базовая статистика уровня отношения хранится в таблице `pg_class`.

Какие данные собираются

- `reltuples` — число кортежей в отношении.
- `n_distinct` — оценка количества уникальных значений.
- `most_common_vals` и `most_common_freqs` — список наиболее часто встречающихся значений и их частоты.
- `histogram_bounds` — границы гистограммы распределения значений, используются для оценки селективности диапазонных запросов.
- `correlation` — оценка корреляции между порядком значений в столбце и порядком их физического хранения.
- `null_frac` — доля неопределённых (`NULL`) значений.

Сбор статистики

- Статистика собирается при `ANALYZE` — ручном или автоматическом. Также базовая статистика рассчитывается при выполнении некоторых операций (`VACUUM FULL`, `CLUSTER`, `CREATE INDEX`, `REINDEX`) и уточняется при очистке.
- Для анализа случайно выбираются `300 × default_statistics_target` строк. Размер выборки, достаточной для построения статистики заданной точности, слабо зависит от объёма анализируемых данных, поэтому размер таблицы не учитывается.

`pg_stats`: использование частых значений и корреляции

- Для уточнения оценки при неравномерном распределении собирается статистика по наиболее часто встречающимся значениям и частоте их появления (`most_common_vals` / `most_common_freqs`).
- Список частых значений используется и для оценки селективности условий с неравенствами. Например, для условия `столбец < значение`: надо найти в `most_common_vals` все значения, меньшие искомого, и просуммировать их частоты из `most_common_freqs`.
- Поле `correlation` показывает корреляцию между **физическим** расположением данных и **логическим** порядком (в смысле операций сравнения):
  - значения хранятся строго по возрастанию → корреляция близка к +1;
  - по убыванию → к -1;

  - чем более хаотично расположены данные на диске — тем ближе к `0`.
- Корреляция используется для оценки стоимости индексного сканирования.

Табличные методы доступа

Последовательное сканирование (Seq Scan)

- Полностью читается файл основного слоя таблицы. На каждой прочитанной странице проверяется видимость каждой версии строки.
- Чтение происходит через буферный кеш; чтобы большие таблицы не вытесняли полезные данные, используется **кольцевой буфер**. Другие процессы, одновременно сканирующие ту же таблицу, присоединяются к кольцу и тем самым экономят операции дисковых чтений. Поэтому в общем случае сканирование может стартовать не с начала файла.
- Последовательное сканирование — самый эффективный способ прочитать всю таблицу или значительную её часть. То есть оно хорошо работает при **низкой селективности** условия.

Оценка стоимости В оценке стоимости оптимизатор учитывает две составляющие — дисковый ввод-вывод и ресурсы процессора: - **Стоимость ввода-вывода** = число страниц в таблице × стоимость чтения одной страницы (при последовательном чтении). - **Стоимость процессора** учитывает обработку каждой версии строки (определяется параметром `cpu_tuple_cost`). - Соотношение по умолчанию подходит для HDD-дисков. Для SSD имеет смысл существенно уменьшить значение параметра `random_page_cost` (значение `seq_page_cost`, как правило, не трогают, оставляя единицу в качестве опорного значения). - Если на сканируемую таблицу наложены условия, они отображаются в плане под узлом `Seq Scan` в секции `Filter`. Оценка числа строк учитывает селективность этих условий, а оценка стоимости — затраты на их вычисление.

Parallel Seq Scan

- Чтение выполняется несколькими параллельно работающими процессами.
- Процессы синхронизируются между собой через специально отведённый участок общей памяти, чтобы не прочитать одну и ту же страницу дважды.

Gather и Gather Merge

- **`Gather`** — собирает результаты от параллельных рабочих процессов.
- **`Gather Merge`** — параллельные процессы выполняют сортировку локально, а `Gather Merge` объединяет уже отсортированные потоки, гарантируя отсортированный итоговый результат без дополнительной сортировки на этапе объединения.

Сканирование только индекса (Index Only Scan)

- Операция представляется в плане узлом `Index Only Scan`.
- На оценку влияет доля табличных страниц, отмеченных в **карте видимости** (visibility map): только для них можно не обращаться к самой таблице.

Сканирование по битовой карте (Bitmap Scan)

- Ограничение обычного индексного сканирования: при уменьшении корреляции увеличивается количество обращений к страницам, а характер чтения меняется с последовательного на случайный.
- Bitmap Scan строит битовую карту нужных страниц, а затем читает страницы по порядку — это позволяет вернуться к последовательному характеру чтения.

Сравнение методов доступа

- Зависимость стоимости различных методов доступа от селективности условий — это «вилка»: при высокой селективности выигрывает индексное сканирование, при низкой — последовательное.
- Стоимость **индексного сканирования сильно зависит от корреляции**. При идеальной корреляции индексное сканирование эффективно даже при довольно большой доле выбираемых строк.

Агрегация

- **Hash Aggregation** — строит хеш-таблицу по ключам группировки.
- **Group Aggregate**:
  - входные данные **должны быть отсортированы** по ключам группировки;

- данные читаются в отсортированном порядке, и когда встречается новое значение ключа, текущая группа завершается;
- **не требует** хранения всей хеш-таблицы в памяти.

CTE (Common Table Expressions)

Идея: если какой-то вложенный подзапрос используется в запросе несколько раз, его можно определить как **общее табличное выражение** (`WITH ... AS (...)`) и ссылаться на него столько раз, сколько потребуется. В этом случае PostgreSQL вычисляет результаты один раз и повторно использует их при повторных обращениях.

Shared Buffers

Оперативная память, где PostgreSQL хранит копии страниц таблиц и индексов.

Когда запрос требует доступ к данным, сервер сначала ищет их в этом кэше. Если страница найдена (**cache hit**), запрос выполняется быстрее, чем при обращении к диску (**cache miss**).

`EXPLAIN`

Команда `EXPLAIN` показывает план выполнения запроса. Что в нём важно:

- **Тип операций (узлов) плана** (`Seq Scan`, `Index Scan`, `Hash Join`, ...).
- **Оценка стоимости** (`cost`) в формате `cost=START_COST..END_COST`.
- **Оценка количества строк** (`rows`).
- **Размер данных** (`width`).
- **Параллельное выполнение**.

При `EXPLAIN ANALYZE` дополнительно: - **Фактическое время выполнения** — время, затраченное на каждую операцию. - **Фактическое количество строк** — сколько строк реально обработано на каждом этапе. - **Число циклов / итераций**.

Виды и способы соединений

Соединение хешированием (Hash Join)

- Реализация использует **динамически расширяемую хеш-таблицу** с разрешением коллизий цепочками (как для буферного кеша).
- Очень эффективно для больших наборов данных. При достаточном объёме оперативной памяти требует **однократного просмотра** двух наборов данных — то есть имеет **линейную сложность**.

Соединение хешированием в параллельных планах

Соединение слиянием (Merge Join)

- Работает для наборов данных, отсортированных по ключу соединения, и возвращает отсортированный же результат.
- Входной набор может оказаться уже отсортированным в результате индексного сканирования или быть отсортирован явно.
- Используются алгоритмы: быстрая сортировка, частичная пирамидальная сортировка, **внешняя сортировка слиянием** (если данные не помещаются в память).

Соединение вложенным циклом (Nested Loop Join)

На эффективность Nested Loop Join влияют несколько условий: - кардинальность внешнего набора строк; - наличие метода доступа ко внутреннему набору, позволяющего эффективно получить нужные строки; - повторные обращения к одним и тем же строкам внутреннего набора.

Полная стоимость соединения складывается из: - стоимости получения всех строк внешнего набора; - однократной стоимости первоначального получения всех строк внутреннего набора (в ходе которого выполняется материализация); - $(N-1)$-кратной стоимости повторного получения строк внутреннего набора; - стоимости обработки каждой строки результата.

Сортировки

- PostgreSQL использует **quicksort**, чтобы выполнить операцию в оперативной памяти. Размер выделенной памяти контролируется параметром

`work_mem`.

- Если объём данных превышает значение `work_mem`, PostgreSQL может производить сортировку с временным сохранением промежуточных результатов на диске (**external merge sort**).
- В новых версиях PostgreSQL поддерживается параллельное выполнение операций сортировки.

Лекция 10. Безопасная система

Что такое безопасная система?

Безопасная система — это система, которая защищена от несанкционированного доступа, использования, раскрытия, нарушения, модификации или уничтожения, тем самым обеспечивая **конфиденциальность, целостность и доступность** данных и функций.

CIA Triad

- **Confidentiality** — конфиденциальность.
- **Integrity** — целостность.
- **Availability** — доступность.

Определение пользователя

- **Идентификация** — процесс, когда информационная система определяет, существует конкретный пользователь или нет, с помощью **идентификатора** (логин, e-mail, номер телефона и т.п.).
- **Аутентификация** — процесс, когда пользователь вводит ключ (пароль, пин-код и т.п.), подтверждая своё право на доступ к учётной записи.
- **Авторизация** — процесс определения того, какие действия позволено совершать аутентифицированному пользователю.

Факторы идентификации (аутентификации)

- **То, что субъект знает** (пароль, PIN, ответ на секретный вопрос).
- **То, что субъекту принадлежит** (телефон, ключ, токен).
- **То, что является неотъемлемой характеристикой субъекта** (биометрия: отпечаток, лицо, голос).

Многофакторная аутентификация

Если для входа требуются пароль и, например, ответ на секретный вопрос (который тоже является «знанием»), то это **не многофакторная аутентификация**, а лишь двухэтапная проверка в рамках одного фактора. Настоящая MFA — это сочетание факторов **из разных категорий**.

Авторизация

Дискреционная модель доступа (DAC)

- Каждый объект имеет **владельца**.
- Владелец полностью контролирует доступ к своему объекту.
- Права доступа определяются на основе **списков контроля доступа (ACL)**.

```
SELECT * FROM pg_class;
```

Виды привилегий

- `SYSTEM PRIVILEGES`
- `DATABASE PRIVILEGES`
- `SCHEMA PRIVILEGES`
- `TABLE PRIVILEGES`
- `COLUMN PRIVILEGES`
- `ROW-LEVEL PRIVILEGES`

Владелец объекта

- В PostgreSQL каждый объект БД имеет владельца, который создаёт этот объект. Владелец обладает полным контролем над объектом.
- По умолчанию владелец может **передавать свои права** другим ролям.

Пользователи и группы

Реализованы через **концепцию ролей**. В этой системе каждая роль может выступать как в роли пользователя, так и в роли группы пользователей.

Роли

- В PostgreSQL «пользователь» — частный случай роли. Каждая роль может иметь или не иметь возможность входа в систему (`LOGIN`). Если у роли установлено `LOGIN`, она может использоваться как учётная запись.
- Роли могут быть **членами других ролей** (например, роль `developers` для всех разработчиков с общими привилегиями).

```
CREATE ROLE name [attr1, attr2, ...];
ALTER ROLE developer WITH PASSWORD 'newsecret';
```

Атрибуты: - `LOGIN` - `SUPERUSER` - `CREATEDB` - `CREATEROLE` - `PASSWORD 'secret'`

Наследование привилегий

Роль, являющаяся членом другой роли, по умолчанию автоматически наследует все привилегии родительской роли.

```
GRANT role1 TO role2;
REVOKE role1 FROM role2;
```

Смена владельца

```
ALTER {obj} OWNER TO {role1};
```

Сменить владельца может сам владелец и роли, в которые он входит.

Привилегии таблиц

- `SELECT`
- `INSERT`
- `UPDATE`
- `REFERENCES`
- `DELETE`
- `TRUNCATE`
- `TRIGGER`

Привилегии БД

- `CREATE`
- `CONNECT`
- `TEMPORARY`

Привилегии схем

- `CREATE`
- `USAGE`

Права доступа к схеме влияют на возможность создания новых объектов, но сами по себе **не контролируют** доступ к уже существующим объектам. Для этого назначаются привилегии непосредственно на уровне объектов.

Схема `public`

- Создаётся по умолчанию при создании новой базы данных.
- Объекты в ней доступны всем пользователям, если не настроены отдельные ограничения.
- При создании объектов без явного указания схемы они автоматически создаются в `public`.

Стандартные БД (шаблоны)

- При создании новой БД без явного указания шаблона используется база `template1`.
- **Template** — «чистая» база данных, которая не изменялась после установки PostgreSQL. Она содержит минимальное количество объектов и служит эталоном.

Мандатная модель доступа (MAC)

Mandatory Access Control (MAC) используется в системах, где требуется высший уровень безопасности и централизованный контроль. В отличие от дискреционной модели, где владелец решает, кому предоставить доступ, в MAC доступ определяется **системными политиками**.

Ключевые принципы: - **«Нет права давать права»** — пользователь не может перераспределять доступ по своему усмотрению. - **Протоколирование ВСЕХ действий.** - **Защита от копирования.**

Шифрование БД

- **Прозрачное шифрование** (Transparent Data Encryption, TDE) — шифрование на уровне файлов БД.
- **На уровне столбцов** — шифруются конкретные чувствительные поля.
- **На уровне приложения** — данные шифруются ещё до отправки в БД.

Угроза действий привилегированных пользователей

Защита от злоупотреблений администраторами: - **Принцип разделения обязанностей** — критические действия требуют участия нескольких ролей. - **Принцип наименьших привилегий** — каждому даётся ровно столько прав, сколько нужно для выполнения работы.

Хранимые процедуры и функции

Подпрограммы PL/pgSQL

Использование `OUT`-аргумента

Условные операторы

- `IF` — возвращает или не возвращает значение в зависимости от ветки.
- `CASE` *(в форме оператора)* — **не возвращает** значение, выбирает блок инструкций для выполнения.

`IF`

Циклы

- `FOR`
- `WHILE`
- `LOOP`

Циклы поддерживают операторы `EXIT` и `CONTINUE`.

`FOR`

Триггеры

> Триггеры используются, когда мы хотим выполнить действия **в момент изменения данных**.

Опции триггера: - **`BEFORE` / `AFTER`** — выполняется перед действием или после. Если используем `BEFORE`, можно менять данные. - **`FROM` (точнее `ON`)** — на какую таблицу вешается триггер. - **`FOR ROW` / `FOR STATEMENT`** — на каждую изменённую строку или

на весь запрос целиком. У STATEMENT-триггера нет информации об изменённых данных. - **WHEN** — содержит условие, определяющее, будет ли вызываться функция.

Специальные переменные триггера

- NEW — содержит новую строку.
- OLD — содержит старую строку.
- TG_NAME — название триггера.
- TG_OP — название операции (INSERT, UPDATE, DELETE).
- TG_TABLE_NAME — название таблицы.

Пример триггера

Плюсы хранимых процедур и триггеров

- Инкапсуляция функциональности.
- Изоляция пользователей от таблиц (доступ только через процедуры).
- В некоторых случаях ускорение выполнения (за счёт предкомпиляции).

Минусы

- Повышение нагрузки на БД.
- Миграция между СУБД вызывает проблемы — **обратной совместимости нет совсем**.

Состояние, Strict, Diagnostics

Функция vs Процедура

Функция	Процедура
Имеет возвращаемый тип и возвращает значение	Может не возвращать значение или возвращать через OUT-параметры
Использование DML-запросов внутри **невозможно** (только SELECT)	Использование DML-запросов **возможно**
Вызов хранимой процедуры из функции **невозможен**	Управление транзакциями (COMMIT, ROLLBACK) возможно внутри процедуры

Функция	Процедура
Вызов функции внутри SELECT-запросов **возможен**	Вызов хранимой процедуры из SELECT-запросов **невозможен**

Создание функции

Особенности функций: - Могут состоять из нескольких операторов SQL. - Значение возвращается через RETURN. - **Нельзя** использовать операторы управления транзакциями (BEGIN, COMMIT, ROLLBACK). - **Нельзя** использовать служебные команды (например, CREATE INDEX).

Типы параметров

- IN — входной (по умолчанию).
- OUT — выходной.
- INOUT — и входной, и выходной.
- RETURN — возвращаемое значение (для функций).

Привилегии доступа

- **SECURITY DEFINER** — функция выполняется с правами **создателя** (definer).
- **SECURITY INVOKER** — функция выполняется с правами **вызывающего** (invoker), по умолчанию.

  Это **перегрузка процедур**, а не функций; в процедурах нет RETURN, только IN и OUT.

Категории изменчивости

- **VOLATILE** — возвращаемое значение может произвольно меняться на одних и тех же входных. (По умолчанию.)
- **STABLE** — значение не меняется в рамках одного оператора, функция не может менять таблицы. Оптимизатор может кешировать результаты в пределах команды.
- **IMMUTABLE** — значение не меняется, функция детерминирована, функция не может менять таблицы. Может вызываться на этапе планирования запроса.

PL/pgSQL

PL/pgSQL — загружаемый процедурный язык для системы управления базами данных PostgreSQL: - может быть использован для создания функций, процедур и триггеров; - добавляет структуры управления к языку SQL; - наследует все пользовательские типы, функции, процедуры и операторы; - прост в использовании.

Структура блока

- Метка (опционально);
- Секция объявления переменных;
- Операторы;
- Обработка исключительных ситуаций.

Форма блока

Лекция 11. Распределённая база данных

В этой части конспекта курсив — комментарии лектора, остальной текст — материал презентации.

> *Когда строим какую-либо реляционную систему, у неё три части: - клиентская; - бэкенд; - база данных.*
>
> *Нужно масштабироваться. Можно делать это вертикально, но упрёмся в производительность. Тогда проще сделать много серверов — БД распределена на несколько устройств.*

Определение

> **Распределённая база данных** — это набор логически взаимосвязанных данных, которые **физически распределены** между несколькими узлами в некоторой компьютерной сети.

Мотивация

- Масштабируемость;
- высокая доступность и отказоустойчивость;
- производительность.

Модели распределения данных

- **Репликация.** *Если репликация, то соединённые ноды имеют одни и те же данные. Одна сломалась — в другой всё осталось (отказоустойчивость). Повышается производительность чтения.*
- **Шардинг.** *Если шардинг, то в разных нодах разные данные. Нода упала — половина данных недоступна.*
- **Replication + Sharding** — комбинация.

А партиционирование?

- **Партиционирование** — это **«вертикальное»** разделение таблицы по строкам **внутри одной базы данных** (на одном сервере), а не между разными узлами. Не путать с шардингом.

Репликация

По способу синхронизации: - **Синхронная** — `COMMIT` дожидается подтверждения от реплик. Высокая консистентность, но выше латентность. - **Асинхронная** — реплики отстают от мастера. Быстро, но возможна потеря самых свежих изменений при падении мастера.

Архитектуры репликации

- **Master-Slave (Primary-Replica).** *Есть одна нода-мастер, остальные — её слейвы. Мастер может писать и читать, слейвы — только читать.*
- **Multi-Master.** *Любая реплика тоже мастер — может принимать запись.*
- **Peer-to-Peer.** *Каждая нода — полноценный, независимый организм.*

Стратегии шардирования

- **Шардирование по диапазону** *(самый простой вариант — например, `id 1..1000` на одну ноду, `1001..2000` на другую).*
- **Шардирование по хешу.**
- **Шардирование по справочнику** (lookup-таблица: ключ → шард).
- **Геошардинг** (по географическому признаку).

Проблемы шардирования

- **Сложность запросов** (`JOIN`-ы поперёк шардов).
- **Транзакции** — требуют дорогих **Two-Phase Commit**.
- **Решардинг** — *перераспределение данных при изменении числа шардов очень дорого.*
- **Роутинг запросов** — нужен **координатор**.
- **Неподходящие ключи шардирования** — могут привести к перекосу нагрузки.
- *Кайфанёшь от `SELECT COUNT(*)`* — приходится считать по всем шардам.

Типы распределённых СУБД

- **Однородные (Homogeneous)** — все узлы используют одну СУБД и схему.
- **Неоднородные (Heterogeneous)** — узлы могут использовать разные СУБД.

Two-Phase Commit (2PC)

Протокол атомарной фиксации распределённой транзакции:

1. **Фаза подготовки (Prepare).** Координатор просит всех участников приготовиться зафиксировать транзакцию; они отвечают «готов» или «не готов».
2. **Фаза решения (Commit / Abort).** Если все согласны — координатор рассылает `COMMIT`, иначе — `ROLLBACK`.

Недостатки 2PC

- Протокол требует **двух раундов** сетевого взаимодействия.
- Каждая транзакция ждёт ответов от **всех** участников.
- **Координатор — единая точка отказа.**

  *Альтернатива 2PC: делаем «маленькие» операции на уровне микросервисов (паттерн **SAGA**: длинная транзакция как последовательность локальных, с компенсирующими действиями при сбое).*

BASE

> **BASE** — свойство системы, которое **жертвует немедленной строгой согласованностью** в обмен на высокую доступность и производительность.

Расшифровка: - **Basically Available** — система отвечает на любой запрос, но ответ может содержать ошибку или несогласованные данные. - **Soft state** — состояние системы может меняться со временем из-за изменений конечной согласованности. - **Eventual consistency** — система в конечном итоге станет согласованной. Она будет продолжать принимать данные и не будет проверять каждую транзакцию на согласованность.

CAP-теорема

CAP-теорема (теорема Брюера) утверждает, что в любой распределённой системе, обеспечивающей хранение данных, невозможно одновременно гарантировать более **двух из трёх** свойств:

- **Consistency** — информация на разных узлах согласована.

- **Availability** — система отвечает на запросы в любой момент времени.
- **Partition tolerance** — связи между узлами могут обрываться (система продолжает работать при сетевых разделениях).

CP

- При разрыве связи между узлами система **блокирует запись** на часть узлов, чтобы гарантировать, что данные останутся согласованными.
- Примеры: **MongoDB, Redis, ZooKeeper**.

AP

- При разрыве связи все узлы остаются доступными для чтения и записи. Это может привести к тому, что разные узлы будут иметь разные версии данных.
- Примеры: **Cassandra, ScyllaDB**.

CA

Такая система может существовать только при отсутствии сетевых сбоев (то есть фактически — нераспределённые системы или системы в одном дата-центре с надёжной сетью).

Пример — сервис «Позвони, напомню!»

Идея сервиса — система хранения фактов с разрешёнными действиями: - запрос на запись факта; - запрос на изменение факта; - запрос на чтение факта.

Продолжение разбора кейса — на Хабре

PACELC

PACELC расширяет CAP, описывая поведение системы не только во время сетевых сбоев, но и в нормальных условиях работы:

- **If (P)artition** — если произошло сетевое разделение, выбираем между **A**vailability и **C**onsistency.
- **(E)lse** — иначе (в нормальной работе) выбираем между **L**atency (низкой задержкой) и **C**onsistency.

То есть даже без сетевых сбоев приходится платить либо консистентностью, либо скоростью ответа.