Databases

- Модели данных баз данных с примерами промышленных СУБД.
- Нормализация (нормальные формы) в реляционной модели данных.
- Первичный ключ, внешний ключ, отношения.
- Язык SQL (SELECT, WHERE, (LEFT) JOIN, GROUP BY, HAVING).
- Язык SQL (DML: INSERT, UPDATE, DELETE). Варианты синтаксиса для множественного обновления данных.
- Язык SQL (Триггеры, процедуры, функции, а также курсоры, циклы, условные операторы, временные таблицы).
- Индексы в РСУБД, виды индексов.
- Сбалансированное дерево, как пример индекса РСУБД.
- Устройство любого не b-tree индекса РСУБД (полнотекстовый, пространственный, колончатый, и др.).
- Оптимизация запросов. Планы запросов.
- Транзакции. ACID.
- Уровни изоляции транзакций. Модели конкурентного доступа.
- Блокировки и взаимоблокировки.
- Безопасность БД.
- Архитектура ИС (Клиент-Сервер, Трехзвенная) с точки зрения подключения к СУБД.

?????? ?????? ??? ?????? ? ????????? ????????????? ????

▪

Модель данных — это способ логического описания, организации и хранения информации в базе данных. Она определяет структуру данных, связи между ними и методы доступа.

Существует несколько основных моделей:

Реляционная модель представляет данные в виде таблиц со строками и столбцами. Связи между таблицами обеспечиваются ключами — первичными и внешними. Примеры: PSQL, MySQL, Oracle, MSSQL.

Иерархическая модель строит данные как дерево: один родитель → много детей. Пример: IBM IMS.

Сетевая модель расширяет иерархическую, позволяя множество связей между записями. Пример: IDMS.

NoSQL объединяет несколько подходов, оптимизированных под разные задачи:

- **Key-Value** — быстрый доступ к данным (Redis).
- **Документная** — данные хранятся как документы (JSON) с индивидуальной структурой (MongoDB).
- **Графовая** — данные представлены узлами и рёбрами, удобно для взаимосвязанных объектов (Neo4j).

?????????? (?????????? ?????) ? ?????????? ?????? ??????.

Нормализация — процесс удаления избыточных данных, устранение аномалий.

????

- Повышение производительности
- Повышение удобства управления данными

Избыточность данных — когда одни и те же данные хранятся в нескольких местах базы данных.

Процесс нормализации — последовательный процесс приведения базы данных к эталонному виду, переход от одной нормальной формы к следующей.

Нормальная форма базы данных — набор правил и критериев, которым должна соответствовать база данных.

UNF (????????? ?????, ??????? ?????????? ?????)

Перед приведением БД к нормальной форме нужно привести её к табличному виду, чтобы он соответствовал базовым принципам реляционной теории:

- строки в таблицах не должны быть пронумерованы
- порядок столбцов не имеет значения

?????? ??????? ? UNF

BlockID	Area	Operator	Holes	Depths	Explosives	Charges
B1	A1	Op1	H1,H2	10,12	ANFO, Emulsion	50,60

Проблемы:

- В ячейках содержится несколько значений
- Данные дублируются

1NF (?????? ??????????? ?????)

Правила 1NF:

- Ячейки таблицы хранят **атомарные значения** — одно не составное значение в каждой ячейке
- Строки таблицы не должны дублироваться
- Столбцы должны содержать данные одного типа

?????? ??????? ? 1NF

BlockID	Area	Operator	HoleID	Depth	Explosive	Charge
B1	A1	Op1	H1	10	Березит	50
B1	A1	Op1	H2	12	Амонит	60

2NF (?????? ??????????? ?????)

Правила 2NF:

- Таблица должна иметь ключ, по которому можно идентифицировать каждую строку
- Все неключевые столбцы должны зависеть от **полного ключа** (для составного ключа)
- Выделяем отдельные сущности, чтобы убрать дублирование данных и частичные зависимости

??????? ?????? ? 2NF

BlastBlock

BlockID	AreaID	OperatorID
B1	A1	O1

Area

AreaID	AreaName
A1	Pit-1

Operator

OperatorID	OperatorName
O1	Опер главный

BlastHole

HoleID	BlockID	Depth
H1	B1	10
H2	B1	12

Charging

HoleID	ExplosiveID	Charge
H1	E1	50
H2	E2	60

ExpAgent

ExplosiveID	ExplosiveName
E1	Березит
E2	Амонит

3NF (?????? ??????????? ?????)

Правила 3NF:

- Нет транзитивных зависимостей
  (одно поле не зависит от другого неключевого поля)

Примеры в наших таблицах:

- `AreaName` зависит только от `AreaID`
- `OperatorName` зависит только от `OperatorID`
- `ExplosiveName` зависит только от `ExplosiveID`

BCNF (?????????? ????? ?????–?????)

Правила BCNF:

- Любой детерминант (столбец или комбинация столбцов, от которых что-то зависит) должен быть кандидатом в ключ

> То есть, не должно быть зависимостей, где неключевой столбец определяет другой столбец.

?????????? ????, ???????? ????, ??????????.

Первичный ключ — это поле или совокупность полей таблицы, которое однозначно идентифицирует каждую строку. Значение первичного ключа не должно повторяться и, как правило, не может быть `NULL`. Именно первичный ключ позволяет надежно ссылаться на конкретную запись.

Внешний ключ — это поле или группа полей в одной таблице, которое ссылается на первичный ключ другой таблицы. Внешние ключи используются для обеспечения ссылочной целостности данных и описывают связи между таблицами.

????????? ????? ?????????

В реляционных системах управления базами данных (РСУБД) таблицы связываются между собой с помощью первичных (PK) и внешних (FK) ключей. Тип связи определяет, сколько записей одной таблицы может соответствовать записям другой.

????? «???? ?? ??????»

Одна запись в родительской таблице может быть связана с множеством записей в дочерней таблице.

Пример. Одна область (*Area*) может содержать множество блоков (*BlastBlocks*). В таблице `BlastBlock` присутствует колонка `AreaId`, которая является внешним ключом и ссылается на первичный ключ таблицы областей.

????? «???? ? ??????»

Каждая запись в одной таблице связана ровно с одной записью в другой таблице.

Пример. Один сотрудник связан с одним служебным пропуском. В одной из таблиц внешний ключ имеет ограничение уникальности и ссылается на первичный ключ другой таблицы.

????? «?????? ?? ??????»

Одна запись в таблице может быть связана со множеством записей в другой таблице, и наоборот.

Такую связь невозможно реализовать напрямую, поэтому используется промежуточная таблица. Она содержит два внешних ключа — по одному на каждую из связанных таблиц.

Пример. Студент может посещать множество курсов, и на один курс могут быть записаны многие студенты. Промежуточная таблица хранит пары идентификаторов студентов и курсов.

???? SQL (SELECT, WHERE, (LEFT) JOIN, GROUP BY, HAVING).

DDL ? DML — ???????????? ????? SQL

DDL (Data Definition Language, язык описания данных) служит для создания и модификации структуры БД, т.е. для создания/изменения/удаления таблиц и связей.

DML (Data Manipulation Language, язык манипулирования данными) позволяет осуществлять манипуляции с данными таблиц, т.е. с ее строками. Он позволяет делать выборку данных из таблиц, добавлять новые данные в таблицы, а также обновлять и удалять существующие данные.

SQL

язык запросов для управления данными в РСУБД.

SELECT

оператор для выбора данных из одной или нескольких таблицы

```
SELECT <columns> FROM <table>
```

WHERE

оператор для фильтрации результатов

```
SELECT <columns> FROM <table> WHERE <condition>
```

GROUP BY

оператор для группировки по определенному столбцу

```
SELECT <columns> FROM <table> GROUP BY <column>
SELECT Department, COUNT(*) as EmployeeCount FROM Employees GROUP BY Department
```

HAVING

позволяет ставить условия на группы, которые были получены после GROUP BY.

> Это как WHERE, но для сгруппированных данных

```
SELECT Country, COUNT(*) AS cnt
FROM Customers
GROUP BY Country
HAVING COUNT(*) > 10;
```

WHERE — фильтрует исходные строки перед группировкой. **HAVING** — фильтрует уже сгруппированные результаты.

LEFT JOIN

выводит все строки из левой таблицы, даже если в правой нет совпадений. Если нет совпадающих данных — связанные поля будут NULL.

```
SELECT c.CustomerName, o.OrderID
FROM Customers c
LEFT JOIN Orders o
  ON c.CustomerID = o.CustomerID;
```

Последовательность логической обработки SQL-запроса

```
FROM / JOIN → WHERE → GROUP BY → HAVING → SELECT
```

То есть:

- сначала выбираются таблицы и объединяются,
- потом фильтруются строки,
- затем группируются,
- после фильтруются группы,

- и только потом выдается результат.

???? SQL (DML: INSERT, UPDATE, DELETE). ???????? ??????????? ??? ?????????????? ??????????? ??????.

DDL ? DML — ???????????? ????? SQL

DDL (Data Definition Language, язык описания данных) служит для создания и модификации структуры БД, т.е. для создания/изменения/удаления таблиц и связей.

DML (Data Manipulation Language, язык манипулирования данными) позволяет осуществлять манипуляции с данными таблиц, т.е. с ее строками. Он позволяет делать выборку данных из таблиц, добавлять новые данные в таблицы, а также обновлять и удалять существующие данные.

INSERT

Оператор используется для добавления данных в таблицу.

Добавление одной строки:

```
INSERT INTO Employees (Name, Department, Salary)
VALUES ('Ivan', 'IT', 1000);
```

Добавление нескольких строк (множественная вставка):

```
INSERT INTO Employees (Name, Department, Salary)
VALUES
  ('Ivan', 'IT', 1000),
  ('Anna', 'HR', 900),
  ('Petr', 'IT', 1100);
```

Вставка данных из другой таблицы:

```
INSERT INTO Employees (Name, Department, Salary)
SELECT Name, Department, Salary
FROM NewEmployees;
```

UPDATE

Оператор используется для изменения существующих данных.

Обновление нескольких строк по условию:

```
UPDATE Employees
SET Salary = Salary * 1.1
WHERE Department = 'IT';
```

> В SQL нет отдельного оператора для обновления «одной» или «нескольких» строк — количество обновляемых строк определяется условием WHERE.

Множественное обновление нескольких столбцов:

```
UPDATE Employees
SET
  Salary = 1200,
  Department = 'Finance'
WHERE Name = 'Ivan';
```

Обновление с использованием подзапроса:

```
UPDATE Employees
SET Salary = (
    SELECT AVG(Salary)
    FROM Employees
)
WHERE Department = 'HR';
```

Обновление с JOIN (диалектозависимо, например MySQL / PostgreSQL):

```
UPDATE Employees e
SET Salary = s.NewSalary
```

```
FROM SalaryChanges s
WHERE e.EmployeeID = s.EmployeeID;
```

DELETE

Оператор используется для удаления данных.

Удаление строк по условию:

```
DELETE FROM Employees
WHERE Department = 'HR';
```

Удаление всех строк таблицы:

```
DELETE FROM Employees;
```

Удаление с использованием подзапроса:

```
DELETE FROM Employees
WHERE Salary < (
    SELECT AVG(Salary)
    FROM Employees
);
```

Summary:

- **INSERT** — добавляет одну или несколько строк
- **UPDATE** — изменяет одну или несколько строк (множественное обновление задаётся через WHERE)
- **DELETE** — удаляет одну или несколько строк

Множественное обновление данных в SQL достигается:

- обновлением всех строк, подходящих под WHERE
- обновлением нескольких столбцов за один запрос
- использованием подзапросов и JOIN

???? SQL (????????, ?????????, ???????, ? ????? ???????, ?????, ???????? ?????????, ????????? ???????).

???????

хранимая процедура, которая автоматически выполняется в ответ на определённые события в базе данных

События:

- INSERT
- UPDATE
- DELETE

Время срабатывания:

- BEFORE
- AFTER

Пример триггера:

```
CREATE TRIGGER <name_trigger>
BEFORE INSERT ON <table_name>
FOR EACH ROW
BEGIN
    SET NEW.created_at = NOW();
END;
```

???????? ?????????

набор SQL-операторов, сохранённых в БД и выполняемых по явному вызову

Особенности:

- могут принимать входные и выходные параметры (необязательно);
- возвращают данные через OUT-параметры;
- выполняются на стороне базы данных;
- могут содержать управляющие конструкции (циклы, условия, курсоры);
- могут изменять данные.

Пример:

```
CREATE PROCEDURE add_user(IN p_name VARCHAR(50))
BEGIN
    INSERT INTO users(name) VALUES (p_name);
END;

-- Вызов процедуры
CALL add_user('Ivan');
```

???????

подпрограмма, которая обязательно возвращает значение

Особенности:

- может использоваться в SQL-запросах и выражениях;
- как правило, не изменяет данные;
- всегда возвращает одно значение.

Пример:

```
CREATE FUNCTION get_user_count()
RETURNS INT
BEGIN
    RETURN (SELECT COUNT(*) FROM users);
END;

SELECT get_user_count();
```

??????

механизм для построчной обработки результата запроса, когда невозможно обработать данные одним SQL-запросом

Используется:

- чаще всего в хранимых процедурах;
- при необходимости последовательной обработки строк.

Основные шаги работы с курсором:

1. объявление курсора
2. открытие
3. чтение строк (FETCH)
4. закрытие
5. освобождение ресурсов

Пример:

```
-- Объявление
DECLARE cur CURSOR FOR
SELECT id FROM users;

-- Открытие
OPEN cur;

-- Чтение строк
FETCH NEXT FROM cur;

-- Закрытие
CLOSE cur;

-- Освобождение ресурсов
DEALLOCATE cur;
```

?????

В SQL (в хранимых программах) используются следующие циклы:

- WHILE (проверка условия происходит перед выполнением тела цикла)

```
DECLARE counter INT DEFAULT 1;
```

```
WHILE counter <= 5 DO
    INSERT INTO numbers(value) VALUES (counter);
    SET counter = counter + 1;
END WHILE;
```

- REPEAT ... UNTIL (проверка условия происходит после выполнения тела цикла)

```
DECLARE counter INT DEFAULT 1;

REPEAT
    INSERT INTO numbers(value) VALUES (counter);
    SET counter = counter + 1;
UNTIL counter > 5
END REPEAT;
```

- LOOP (базовый бесконечный цикл, выход осуществляется вручную через LEAVE)

```
DECLARE counter INT DEFAULT 1;

my_loop: LOOP
    IF counter > 5 THEN
        LEAVE my_loop;
    END IF;
    INSERT INTO numbers(value) VALUES (counter);
    SET counter = counter + 1;
END LOOP my_loop;
```

□

???????? ?????????

Для ветвления логики применяются:

- IF ... ELSE

```
DECLARE user_age INT DEFAULT 20;

IF user_age >= 18 THEN
    SET @status = 'Adult';
ELSE
    SET @status = 'Minor';
END IF;
```

- CASE

```
SELECT
    name,
    CASE
        WHEN salary < 1000 THEN 'Low'
        WHEN salary BETWEEN 1000 AND 3000 THEN 'Medium'
        ELSE 'High'
    END AS salary_level
FROM employees;
```

????????? ???????

таблица для временного хранения данных

```
-- Создание временной таблицы
CREATE TEMPORARY TABLE temp_users (
    id INT,
    name VARCHAR(50)
);

-- Вставка данных
INSERT INTO temp_users VALUES (1, 'Ivan'), (2, 'Anna');

-- Использование временной таблицы
SELECT * FROM temp_users;

-- После окончания сессии таблица автоматически удаляется
```

Особенности:

- существует только в рамках текущего соединения;
- автоматически удаляется после завершения сессии;
- используется для хранения промежуточных результатов;
- недоступна другим пользователям.

??????? ? ?????, ???? ????????.

Индекс — специальная структура данных, предназначенная для ускорения поиска и сортировки данных в таблицах. Создается по одному или нескольким столбцам таблицы и хранит:

- значение индексируемых полей
- указатели на соответствующие строки таблицы

> Аналогия: индекс в книге позволяет быстро найти нужную страницу, не просматривая всю книгу

Назначение индексов:

- ускорение SELECT, JOIN
- быстрое выполнение WHERE, ORDER BY, GROUP BY
- обеспечение уникальности

Недостатки:

- замедление INSERT, UPDATE, DELETE
- дополнительная память
- нуждаются в обслуживании (перестроение, обновление)

????????????? ????????

?? ???????????

- Уникальный индекс — запрещает дублирование значений, автоматически создаётся для PK.
- Неуникальный индекс — допускает повторяющиеся значения, используется только для ускорения поиска.

?? ?????????? ????????

Кластеризованный индекс

- определяет физический порядок хранения строк в таблице
- данные таблицы хранятся отсортированными по ключу индекса
- один на таблицу
- обычно создаётся для PRIMARY KEY

Некластеризованный индекс

- хранит:
  - ключ индекса
  - указатель на строку (или на кластеризованный индекс)
- не влияет на физический порядок строк
- может быть несколько на одну таблицу

?? ??????? ????????

- Однопольный индекс — создается по одному столбцу
- Составной (многопольный) индекс — создается по нескольким столбцам

> Порядок столбцов имеет значение

?? ????????? ??????

- **B-tree**
  - самый распространённый тип
  - эффективен для диапазонных запросов (<, >, BETWEEN)
  - используется по умолчанию во многих СУБД
- **Hash**
  - использует хеш-таблицу
  - эффективен для точного поиска (=)
  - не подходит для диапазонных запросов и сортировки
- **Bitmap**
  - использует битовые карты
  - эффективен, когда мало уникальных значений
  - часто используется в аналитических системах
- **GiST, GIN, SP-GiST** (PostgreSQL)
  - применяются для нестандартных данных:
  - массивы
  - JSON
  - полнотекстовый поиск

- геоданные

??????? ??????

Таблица индексов имеет структуру (ключ, номер записи в таблице) в отсортированном виде по ключу, при этом она разделена на блоки.

?????????????? ??????, ??? ?????? ??????? ?????.

> В реляционных СУБД большие таблицы могут содержать тысячи записей, поиск элемента без индекса стоит O(n). Из-за этого используют индексы, сбалансированное дерево является одним из самых популярных индексов

Сбалансированное дерево - структура данных, в которой высота левого и правого поддерева каждого узла отличается не более чем на единицу.

Основные свойства

1. Высота дерева минимальна
2. Все листьях находятся примерно на одном уровне
3. При вставке или удалении структура автоматически перестраивается

> Сложность в сбалансированном дереве всегда составляет O(log(n))

Сбалансированные деревья не используются напрямую, в базах данных(postgresql, mysql, oracle) используются **B-Tree или B+Tree**

Особенности

1. Каждый узел хранит несколько ключей
2. Каждый узел может иметь множество потомков
3. Дерево получается низким(3-4 уровня для миллиона записей)

B-Tree

Пример B-Tree

> Количество узлов выходящих из родителя равно **количество узлов родителя + 1**

>

> Поиск выполняется следующим образом - B-Tree начинает с верхнего узла и сравнивает значения(сначала меньше/больше, а потом равно ли оно), например возьмем **7**. B-Tree начинает слева направо и смотрит:
>
> - 7<3 - нет, значит идем дальше направо
> - 7>3 - да, значит идем дальше направо
> - 7>6 - да, значит идем дальше направо
> - 7>10 - нет, значит нужная нам ветка между 6 и 10, т.к. 6<7<10
> - также смотрим в узле между 6 и 10(789), слева направо
> - 7=7 - да, нашли

Пример использования

1. Пользователь делает поле, например **name**, индексом
2. B-Tree записывает в себя отсортированный список имён и распределяет их по дереву
3. При запросе вида `SELECT * FROM users WHERE name='hyilo'` происходит следующее
   1. B-Tree смотрит на верхний узел и сравнивает то, что мы хотим найти с записями
   2. До того момента пока B-Tree не найден элемент он будет проходить по узлам

      > ВАЖНО! B-Tree оперирует знаками <,>,= это значит что индексироваться могут и строки, и числа

B+Tree ???????? ????? ?????, ??? ? B-Tree, ?? ? ???? ???????? ????????? ?????? ?? ???????, ?? ??????? ????? ???????? ??? - ?????? ??????????

```
       [30 | 50]
      /    |    \
[10 | 20] [35 | 40] [55 | 60]
```

> Поиск выполняется следующим образом - B+Tree начинает с верхнего узла и сравнивает значения меньше/больше, например возьмем **27**. B+Tree начинает слева направо и смотрит:
>
> - 27<30 - да, значит идем ниже
> - 27<10 - нет, значит проверяем дальше этот узел
> - 10<27<20 - нет, значит точно верно что 20<27
> - дальше идут такие же проверки, пока не найдем значение

Сложности

Операция	Без индекса	С B-Tree
Поиск	O(n)	O(log n)
Вставка	O(n)	O(log n)
Удаление	O(n)	O(log n)

??????? ?????? ????????

> У дерева есть строгие правила по узлам, для разных структур данных оно может иметь разный порядок(m), в большинстве случаях берется m=4

B-Tree порядка m означает, что в одном узле:

- минимум ⌈m/2⌉ − 1 ключей
- максимум m − 1 ключей

Количество детей:

- от ⌈m/2⌉ до m

Пример

m	Кол-во узлов	Кол-во детей
4	1-3	2-4
6	2-5	3-6
8	3-7	4-6

B-Tree

1. ????? ?????

Новое значение всегда вставляется **в листовой узел**.

1. Начинает поиск с корня
2. Сравнивает вставляемый ключ с ключами в узле
3. По диапазону выбирает нужную ветку
4. Повторяет, пока не дойдёт до листа

2. ??????? ? ????

- Ключ вставляется в отсортированном порядке
- Структура дерева не меняется

Пример: Лист `[10, 20, 40]`, вставляем `30` → `[10, 20, 30, 40]`

3. ???????????? ????

Если после вставки в узле стало слишком много ключей

1. Узел делится на два
2. Средний ключ поднимается в родительский узел
3. Левые ключи остаются в левом узле, правые — в правом

Пример:

```
[10 | 20 | 30 | 40 | 50]   ← переполнение
```

Средний ключ `30` поднимается вверх:

```
          [30]
         /    \
   [10 | 20]  [40 | 50]
```

Если родительский узел тоже переполняется после добавления среднего ключа, то операция повторяется **рекурсивно вверх**.

Если переполняется корень:

- Создаётся новый корень
- Высота дерева увеличивается на 1

B+Tree

Работает практически также

- Средний ключ **копируется в родителя**
- Все реальные данные остаются в листьях

??????????? ?????? ?? b-tree ??????? ????? (??????????????, ??????????????, ???????????, ? ??.).

Колончатые индексы устроены принципиально иначе, чем традиционные строковые индексы (например, B-Tree), и оптимизированы для аналитических запросов, где часто затрагивается лишь небольшое подмножество столбцов таблицы, но при этом обрабатывается большое количество строк.

Ключевой момент: в отличие от обычного индекса, это фактически колоночная организация таблицы, а не отдельная структура поверх таблицы.

> Вместо того чтобы хранить строки таблицы подряд, данные разбиваются по столбцам.

```
Таблица:
id | name  | age
1  | Alice | 30
2  | Bob   | 25
3  | Carl  | 35

Колоночное хранение:
id:   [1, 2, 3]
name: ["Alice", "Bob", "Carl"]
age:  [30, 25, 35]
```

- Это позволяет эффективно читать только нужные столбцы при выполнении запроса, экономя I/O.
- Колоночные индексы/структуры особенно эффективны для: GROUP BY, COUNT, SUM, WHERE по отдельным столбцам.

- Медленнее работают при запросах, где нужно получить или обновить целые строки, так как строка распределена по разным местам на диске.

?????????? (Chunking) ? ???????????

- Данные в каждом столбце разбиваются на блоки или сегменты (chunks/segments/partitions).
- Обычно в одном блоке находится определенное количество строк (например, 1000 строк).
- Такой блок — минимальная единица чтения, сжатия и индексирования.

??????

- Блоки данных сжимаются по отдельности.
- Поскольку в одном столбце значения однородны, применяются эффективные алгоритмы:
  - **Dictionary encoding** — если значений мало (например, пол: M/F), они заменяются на индексы.
  - **Run-Length Encoding (RLE)** — если значение повторяется подряд, оно заменяется на (значение, количество).
  - **Delta encoding** — для чисел или дат, где хранится разница между соседними значениями.

??????? ?????

- В отличие от строковых индексов, порядок строк в колоночной системе может быть изменён для оптимизации.
- Строки часто сортируются по какому-то столбцу (например, дате или ID), чтобы улучшить сжатие и ускорить диапазонные запросы.

?????????? ????????. ????? ????????.

???? ???????

описание того, каким образом СУБД будет выполнять SQL-запрос, включая порядок операций, способы доступа к данным и оценки затрат.

План **строится оптимизатором запросов** на основе:

- структуры запроса;
- статистики таблиц;
- доступных индексов;
- системных ресурсов.

План запроса **используется** для:

- выбора наиболее эффективного способа выполнения;
- анализа медленных запросов;
- поиска проблем с индексами;
- оценки нагрузки на систему;
- сравнения вариантов оптимизации.

???? ?????? ????????

?? ?????? ??????????

Тип плана	Описание
Логический	Описывает *что* нужно сделать (SELECT, JOIN, WHERE)
Физический	Описывает *как* именно выполняется запрос

?? ??????? ?????????

План	Характеристика
Оценочный (Estimated)	Строится до выполнения
Фактический (Actual)	Получается после выполнения с замерами

????????? ????? ???????

План обычно представлен в виде дерева операций, где:

- листья — чтение данных;
- узлы — обработка;
- корень — итоговый результат.

???????? ???????? ?????

???????? ??????? ? ??????

Операция	Назначение
`Seq Scan`	Полное сканирование таблицы
`Index Scan`	Чтение через индекс
`Index Only Scan`	Только индекс, без таблицы
`Bitmap Index Scan`	Поиск подходящих строк
`Bitmap Heap Scan`	Чтение строк по битовой карте

???????? ?????????? ?????? (JOIN)

Тип JOIN	Описание
`Nested Loop`	Вложенные циклы
`Hash Join`	Хэш-таблица в памяти
`Merge Join`	Слияние отсортированных данных

Выбор зависит от:

- размера таблиц;
- наличия индексов;
- условий соединения.

???????? ????????? ??????

Операция	Функция
`Filter`	Применение условий
`Sort`	Сортировка
`Aggregate`	Агрегация (`SUM`, `COUNT`)
`GroupAggregate`	Группировка
`HashAggregate`	Агрегация через хэш

Операция	Функция
`Limit`	Ограничение строк
`Materialize`	Временное хранение

??????? ????? ???????

????????? (Cost)

```
cost = startup_cost .. total_cost
```

Метрика	Значение
`Startup Cost`	Цена получения первой строки
`Total Cost`	Общая оценка стоимости

> Стоимость — условная величина, не время

?????????? ?????

Метрика	Описание
`rows` (estimated)	Ожидаемое число строк
`rows` (actual)	Фактическое число строк

> Большая разница → проблемы со статистикой

?????

Показатель	Назначение
`Planning Time`	Время построения плана
`Execution Time`	Время выполнения

????????????? ????????

Метрика	Описание
`CPU`	Загрузка процессора
`I/O`	Дисковые операции

Метрика	Описание
`Memory`	Использование памяти
`Loops`	Количество повторений

??????????? ???????? ?? ?????

Основные методы оптимизации:

- добавление индексов;
- уменьшение числа строк;
- корректные JOIN;
- отказ от SELECT *;
- обновление статистики (ANALYZE);
- устранение лишних сортировок.

??????????. ACID.

Транзакция — это набор операций, выполняющихся как единое целое. Обеспечивает целостность данных.

???????? ACID

1. **Atomicity (Атомарность)**
   - Все операции транзакции выполняются полностью или ни одна.
   - Пример: Если одна операция в транзакции не удалась, вся транзакция откатывается.
2. **Consistency (Согласованность)**
   - Данные остаются в корректном состоянии после выполнения транзакции.
   - Пример: Если на счету недостаточно средств для перевода, транзакция не выполнится.
3. **Isolation (Изолированность)**
   - Транзакции не мешают друг другу, выполняются так, как если бы они шли последовательно.
   - Пример: Если две транзакции пытаются изменить одни и те же данные, они выполняются последовательно.
4. **Durability (Долговечность)**
   - Результаты транзакции сохраняются даже после сбоя системы.
   - Пример: После завершения транзакции данные не потеряются, даже если сервер упадет.

?????? ???????? ???????????. ?????? ????????????? ???????.

PostgreSQL поддерживает несколько уровней изоляции, которые определяют, как транзакции видят изменения других транзакций. Уровни изоляции регулируют конкурентный доступ к данным.

?????? ????????

1. Read Uncommitted (Чтение незафиксированных данных)
   - Самый низкий уровень изоляции.
   - Транзакции могут видеть незафиксированные изменения других транзакций.
2. Read Committed (Чтение зафиксированных данных)
   - Транзакции видят только зафиксированные изменения.
   - Уровень по умолчанию в PostgreSQL.
3. Repeatable Read (Повторяемое чтение)
   - Гарантирует, что данные, прочитанные в транзакции, не изменятся другими транзакциями до ее завершения.
   - Полезно для предотвращения фантомного чтения.
4. Serializable (Сериализуемый)
   - Самый строгий уровень изоляции.
   - Гарантирует, что параллельные транзакции выполняются так, как если бы они шли последовательно.

Уровень изоляции	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted			
Read Committed			
Repeatable Read			
Serializable			

> **Dirty Read** (Грязное чтение) — транзакция читает незафиксированные изменения другой транзакции. Если та откатится, данные окажутся неверными.

> **Non-Repeatable Read** (Неповторяемое чтение) — данные, прочитанные транзакцией, могут измениться другой транзакцией до её завершения.
>
> **Phantom Read** (Фантомное чтение) — при повторном выполнении запроса могут появляться или исчезать новые строки, добавленные или удалённые другими транзакциями.

Пример установки уровня изоляции:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Операции внутри транзакции
SELECT * FROM accounts WHERE user_id = 1;
COMMIT;
```

□

?????? ????????????? ???????

1. Оптимистический (версионированный)
   - Проверка конфликтов после выполнения действия.
   - Пример: пользователь изменяет данные и пытается сохранить изменения. Если версия данных совпадает с текущей — операция проходит, иначе — конфликт, который обрабатывается повторным выполнением или откатом.
2. Пессимистический (с блокировками)
   - Проверка конфликтов до выполнения действия.
   - Потоки сериализуются на защищаемой области данных (блокируются).
3. Частично-оптимистический (смешанный)
   - Комбинирует оба подхода: часть операций проверяется заранее, часть — после выполнения.

?????????? ? ??????????????.

Блокировка — запрет другим транзакциям доступа к объекту для предотвращения коллизий и обеспечения целостности данных.

?? ??????? ????????

- Строчная (Row-level) — блокируется только одна строка, остальные строки доступны.
- Гранулярная (Table/Page) — блокируется вся таблица или страница.
- Предикатная (Range/Predicate) — блокировка по диапазону значений, предотвращает фантомные чтения (Serializable).

?? ?????????

- Shared (S, разделяемая) — для чтения. Несколько транзакций могут читать.
- Exclusive (X, исключительная) — для изменения. Только одна транзакция может писать, другие не имеют доступа.

Intent Lock (?????????? ?????????)

- IS (Intent Shared) / IX (Intent Exclusive) — блокировка таблицы для сигнализации, что внутри есть блокировки строк.

?? ?????? ??????????

- Пессимистическая — блокировки до модификации, предотвращает конфликты заранее.
- Оптимистическая — проверка конфликта перед коммитом; откат при изменении данных другими транзакциями.

?????????? ?????????? (2PL)

1. Growing (Фаза захвата) — берутся блокировки.

2. Shrinking (Фаза освобождения) — отпускаются блокировки.

Гарантирует сериализуемость транзакций.

?????????????? (Deadlock)

Условия:

1. Mutual Exclusion — ресурс эксклюзивен
2. Hold and Wait — держит ресурс и ждёт другой
3. No Preemption — нельзя отобрать ресурс
4. Circular Wait — замкнутый круг ожиданий

Решения:

- Обнаружение: строится граф ожиданий, откат одной транзакции при цикле
- Предотвращение: Wait-Die, Wound-Wait
- Игнорирование: таймаут N секунд

????????? ??????????

Если слишком много мелких блокировок (строки), СУБД заменяет их на блокировку таблицы для экономии памяти.

?????????? ??.

???? ? ?????????????

- Доступ к БД только через приватную сеть или VPN
- Firewall: открыт только порт СУБД
- БД не должна быть напрямую в интернете

????? ? ??

- Права доступа на уровне ОС для файлов БД
- Регулярные обновления безопасности ОС
- Шифрование диска (LUKS/BitLocker/TPM) для защиты данных «на физическом носителе»

???????????? ? ???????

- Принцип минимальных прав (Least Privilege)
- Только нужные права: SELECT / INSERT / UPDATE / DELETE
- Не использовать root / postgres в приложениях
- Не хранить пароли в коде

????? ? ???????????

- Аудит: кто, когда, что делал, с какими объектами
- Логирование: подключения, ошибки, подозрительная активность
- Профайлер / планы запросов: выявление тяжёлых запросов → предотвращение DoS или неправильного доступа

SQL-????????

- П араметризованные запросы / prepared statements

??????

- Цель: ошибки пользователей, сбои, атаки
- Типы бэкапов:
  1. Полный — вся база
  2. Инкрементальный — изменения с последнего бэкапа
  3. Дифференциальный — изменения с последнего полного бэкапа
- Методы:
  - SQL-дампы — перенос структуры и данных
  - Файловый — точная копия файлов
  - Стриминг (WAL) — откат к любому моменту времени
  - Практика: хранить отдельно, шифровать, проверять восстановление

??????

- Разделение сред: dev / stage / prod
- 2FA для админов
- Отключить пользователей и базы по умолчанию
- Отключить анонимные подключения
- SSL / TLS: шифрование трафика (пароли и PII)

> Безопасность БД = сеть + доступы + шифрование + аудит + бэкапы + защита данных + ограничения ресурсов

?????????? ?? (??????-??????, ????????????) ? ????? ?????? ????????????? ? ????.

?????????? ??????-?????? (2-???????)

Клиент напрямую обращается к СУБД, формирует SQL и получает данные. Может быть толстый или тонкий клиент.

??????? ? ?????? ??????

Тип клиента	Характер	Где логика	Обработка запросов к СУБД
Толстый клиент	Fat Client	Бизнес-логика + UI на клиенте	Клиент формирует SQL и напрямую обращается к СУБД
Тонкий клиент	Thin Client	Логика минимальна	Клиент формирует простые SQL-запросы и напрямую обращается к СУБД

?????

??????? ?????? (2-???????)

```
Пользователь → (логика, запросы SQL) Клиент
           → (SQL) → Сервер СУБД
                     ← результаты
```

?????? ?????? (2-???????)

```
Пользователь → UI-только клиент
           → минимальная логика на клиенте
           → SQL → Сервер СУБД
                     ← результат
```

????? ??????

```
[ Клиент ]
     |
  SQL запрос
     |
```

```
[ Сервер СУБД ]
```


??????????? ??????????? (3-tier)

Это расширение клиент-серверной модели. Вводится промежуточный уровень приложения (API), который обрабатывает бизнес-логику и взаимодействует с СУБД вместо клиента. Клиент общается только с API.

????? (3 ??????)

```
[ Клиент (UI) ]
      |
   API/HTTP
      |
[ Сервер приложений ]
      |
 JDBC/ODBC/ORM
      |
[ Сервер СУБД ]
```

Уровень	Что делает	К СУБД подключается?
Клиент	Интерфейс пользователя	Нет прямого доступа
Сервер приложений / API	Бизнес-логика и обработка данных	Да
Сервер СУБД	Хранит данные и отвечает на запросы	—

> Клиент никогда не общается напрямую с СУБД — только через API сервера приложений.

Преимущества:

- Лучшая безопасность
- Обновления логики на сервере без изменения клиентов
- Меньшая нагрузка на сеть
- Масштабируемость
- Разделение ответственности

????????? ??????????

Характеристика	2-звенная	3-звенная

Кол-во уровней	2	3
К клиенту можно подключиться к БД?	Да	Нет
Бизнес-логика	На клиенте	На сервере приложений
Обновление UI	Труднее	Легче
Масштабируемость	Ниже	Выше
Безопасность доступа к данным	Ниже	Выше
СУБД-API	Прямой	Через сервер приложений