SQL THEORY

UNION, UNION ALL, INTERSECT, INTERSECT ALL, EXCEPT и EXCEPT ALL.

UNION

- Объединяет результаты двух SELECT-запросов. (построчно!)
- **Убирает дубликаты** автоматически.
- Количество и тип столбцов в обоих SELECT должны совпадать.

SELECT name FROM hr.employees

UNION

SELECT name FROM sales.employees;

- Результат = все уникальные имена из обеих таблиц.

UNION ALL

- Объединяет результаты двух SELECT, **сохраняя все дубликаты**.

SELECT name FROM hr.employees

UNION ALL

SELECT name FROM sales.employees;

- Результат = все имена, включая повторяющиеся.
- **Быстрее**, чем UNION, потому что не требуется удаление дубликатов.

INTERSECT

- Возвращает строки, **которые есть в обоих SELECT**.
- Убирает дубликаты.

SELECT name FROM hr.employees

INTERSECT

SELECT name FROM sales.employees;

- Результат = имена, которые есть и в HR, и в Sales.

INTERSECT ALL

- Как INTERSECT, но **сохраняет количество повторений**.

- Например, если имя встречается 2 раза в первом SELECT и 3 раза во втором, результат будет 2 раза.
- Поддерживается не во всех СУБД (например, PostgreSQL поддерживает).

```
SELECT name FROM hr.employees

INTERSECT ALL

SELECT name FROM sales.employees;
```

EXCEPT

- Возвращает строки из первого SELECT, **которых нет во втором SELECT**.
- Убирает дубликаты.

```
SELECT name FROM hr.employees

EXCEPT

SELECT name FROM sales.employees;
```

- Результат = имена сотрудников HR, которых нет в Sales.

EXCEPT ALL

- Как EXCEPT, но **сохраняет количество повторений**.
- Например, если имя встречается 3 раза в первом SELECT и 1 раз во втором, оно появится **2 раза**.
- Не поддерживается во всех СУБД.

```
SELECT name FROM hr.employees

EXCEPT ALL

SELECT name FROM sales.employees;
```

Важные правила

1. Количество и тип столбцов в обеих SELECT должны совпадать.
2. Порядок столбцов важен.
3. Можно комбинировать эти операции, но желательно использовать скобки для читаемости:

```
(SELECT name FROM hr.employees

 INTERSECT

 SELECT name FROM sales.employees)
```

UNION

SELECT name FROM contractors;

💡 **Итоговое различие**

Операция	Удаляет дубликаты	Что делает
UNION	Да	Объединяет, уникальные
UNION ALL	Нет	Объединяет, все строки
INTERSECT	Да	Пересечение двух наборов
INTERSECT ALL	Нет	Пересечение с учётом повторений
EXCEPT	Да	Разность: в первом, нет во втором
EXCEPT ALL	Нет	Разность с учётом повторов

JOINS

Что такое JOIN

JOIN позволяет объединять строки из двух и более таблиц **по общему полю** (ключу).

- Обычно используют для связи id, foreign key, primary key.

Основные типы JOIN

2.1. INNER JOIN

- Возвращает только **те строки, которые совпадают в обеих таблицах**.
- Если совпадений нет — строка не попадёт в результат.

SELECT e.name, d.department_name

FROM employees e

INNER JOIN departments d

ON e.department_id = d.id;

- Результат = только сотрудники, у которых есть соответствующий отдел.

2.2. LEFT JOIN (LEFT OUTER JOIN)

- Возвращает **все строки из левой таблицы**, даже если нет совпадений в правой.
- Столбцы правой таблицы будут NULL, если совпадений нет.

SELECT e.name, d.department_name

FROM employees e

LEFT JOIN departments d

ON e.department_id = d.id;

- Результат = все сотрудники, отделы там, где они есть; где нет — NULL.

2.3. RIGHT JOIN (RIGHT OUTER JOIN)

- Аналогично LEFT JOIN, но возвращает **все строки из правой таблицы**.

SELECT e.name, d.department_name

FROM employees e

RIGHT JOIN departments d

ON e.department_id = d.id;

- Результат = все отделы, сотрудники там, где есть; где нет — NULL.

2.4. FULL OUTER JOIN

- Объединяет LEFT и RIGHT JOIN.
- Возвращает все строки из обеих таблиц.
- Если нет совпадения — соответствующие столбцы будут NULL.

SELECT e.name, d.department_name

FROM employees e

FULL OUTER JOIN departments d

ON e.department_id = d.id;

2.5. CROSS JOIN

- Декартово произведение двух таблиц.
- Каждая строка левой таблицы соединяется со **всеми строками правой таблицы**.

- Обычно используется редко, только когда нужен полный набор комбинаций.

SELECT e.name, d.department_name

FROM employees e

CROSS JOIN departments d;

- Результат = все возможные комбинации сотрудников и отделов.

2.6. SELF JOIN

- JOIN таблицы с самой собой.
- Полезно для иерархий или поиска связей внутри одной таблицы.

SELECT e1.name AS employee, e2.name AS manager

FROM employees e1

LEFT JOIN employees e2

ON e1.manager_id = e2.id;

Основные моменты

1. ON определяет условие соединения (e.department_id = d.id).
2. INNER JOIN → только совпадения
3. LEFT/RIGHT JOIN → все строки из одной таблицы + совпадения
4. FULL JOIN → все строки из обеих таблиц
5. CROSS JOIN → все комбинации

💡 **Итог**: JOIN позволяет **объединять данные из нескольких таблиц**, сохраняя нужные связи, и выбирать только те строки, которые соответствуют логике запроса

Window Functions

Window Functions –
это функция, которая выполняет агрегирование или вычисление по «окну» строк, но при этом не группирует весь результат в одну строку, как обычные агрегатные функции (SUM, AVG).

То есть, каждая строка сохраняется, но рядом с ней можно получить агрегированные данные по группе или диапазону строк.

```
SELECT employee_id, department_id, salary,
RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank
FROM employees;
```

синтаксис 1

```
SELECT sum(salary) OVER w, avg(salary) OVER w
 FROM empsalary
 WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

синтаксис 2(выносим определение окна в конец для многократного использования)

```
<window_function>() OVER (
   [PARTITION BY column1, column2, ...]  -- деление на группы (не обязательно)
   [ORDER BY column3, ...]               -- порядок внутри окна
   [ROWS BETWEEN ...]                    -- рамки окна (не обязательно)
)
```

ВИДЫ ОКОННЫХ ФУНКЦИЙ

🔹 1. **Агрегатные оконные функции**

- Работают как обычные агрегатные функции, но не «склеивают» строки, а сохраняют их.
- Примеры:

Функция	Что делает
SUM()	Скользящая сумма по окну
AVG()	Среднее значение по окну
MIN()	Минимум по окну
MAX()	Максимум по окну
COUNT()	Количество строк в окне

Пример:

SUM(salary) OVER (PARTITION BY department_id)

→ сумма зарплат в каждом департаменте для каждой строки.

🔹 2. **Функции ранжирования (Ranking Functions)**

- Присваивают порядковый номер или ранг строкам внутри окна.

Функция	Что делает
ROW_NUMBER()	Уникальный номер строки в окне
RANK()	Ранг строки, пропуская позиции при равных значениях
DENSE_RANK()	Ранг без пропуска номеров при равных значениях
NTILE(n)	Делит строки на n равных частей (квартиль, процентиль)

🔹 3. **Функции смещения (Offset Functions)**

- Позволяют получать значения из соседних строк в окне.

Функция	Что делает
LAG(expr, n)	Значение expr из **предыдущей строки** (по умолчанию 1)
LEAD(expr, n)	Значение expr из **следующей строки**
FIRST_VALUE(expr)	Первое значение в окне
LAST_VALUE(expr)	Последнее значение в окне

🔷 4. **Функции скользящих агрегатов (Cumulative / Moving)**

- Считают агрегаты по скользящему окну (например, кумулятивная сумма).
- Обычно задаются через ROWS BETWEEN ... или RANGE BETWEEN ....

Пример:

SUM(salary) OVER (PARTITION BY department_id ORDER BY salary ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

→ кумулятивная сумма зарплат в департаменте до текущей строки.

✅ Кратко:

Вид	Примеры	Назначение
Агрегатные	SUM, AVG, COUNT, MIN	Скользящие агрегаты, сохраняют строки
Ранжирующие	ROW_NUMBER, RANK	Нумерация и ранжирование строк
Смещения	LAG, LEAD, FIRST_VALUE	Доступ к соседним значениям
Кумулятивные	SUM(... ROWS BETWEEN ...)	Кумулятивные и скользящие вычисления

Procedures and Functions

Функция (User Defined Function, UDF) –
это объект базы данных, который принимает параметры и **обязательно возвращает одно значение**.

- Основная цель: **вычисления** или преобразование данных.
- Может использоваться в SELECT, WHERE, HAVING, ORDER BY.
- Всегда возвращает значение через RETURN.
- Параметры только входные (IN)

```
CREATE FUNCTION AnnualSalary(monthly_salary NUMERIC(10,2))  RETURNS NUMERIC(12,2)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN monthly_salary * 12;
END;
$$;
```

Код функции и SQL — это разные вещи

- **SQL — это декларативный язык: вы описываете, *что* сделать с данными.**
  Примеры: SELECT, INSERT, UPDATE, DELETE.
- **PL/pgSQL (или другой procedural language) — это процедурный язык: вы описываете *как* выполнять логику.**
  Примеры: переменные, условия, циклы, обработка ошибок:

Создание функции

```
SELECT first_name, AnnualSalary(salary) AS yearly_salary
FROM employees;
```

Использование

Хранимая процедура (Stored Procedure) –
это объект базы данных, в котором хранится набор SQL-инструкций.

- Основная цель: **выполнение действий** (модификация данных, бизнес-логика).
- Может принимать параметры (IN, OUT, INOUT).
- Может возвращать **несколько значений через OUT-параметры** или вообще ничего.
- Нельзя использовать прямо в SELECT (вызывается отдельно через CALL или EXEC).

```
    CREATE PROCEDURE IncreaseSalary (IN dept_id INT)
BEGIN
  UPDATE employees
  SET salary = salary * 1.10
  WHERE department_id = dept_id;
END;
```

Создание функции

```
CALL IncreaseSalary(5);
```

Использование

🔑 Основные отличия процедур и функций

Критерий	Процедура	Функция
Тип объекта	**Объект БД для выполнения действий INSERT, UPDATE, DELETE**	**Объект БД для вычислений**
Возврат значения	**Может вернуть 0, 1 или несколько значений через OUT**	**Всегда возвращает одно значение через RETURN**
Использование в SQL	**Нельзя в SELECT**	**Можно в SELECT, WHERE, HAVING**
Параметры	**IN, OUT, INOUT**	**Только IN**
Основное назначение	**Изменение и управление данными**	**Получение результата (вычисление)**

Triggers

Триггер(Trigger) –
Это объект базы, который выполняет процедуру (или функцию) автоматически при событии на таблице: INSERT, UPDATE или DELETE.

Пример триггера в PostgreSQL

Шаг 1. Таблицы

```
CREATE TABLE sales.orders (
    id SERIAL PRIMARY KEY,
    employee_id INT,
    product_id INT,
    quantity INT,
    order_date TIMESTAMP DEFAULT NOW()
);


CREATE TABLE sales.stock (
    product_id INT PRIMARY KEY,
    quantity INT
);
```

Шаг 2. Процедура

```
CREATE PROCEDURE update_stock(prod_id INT, qty INT)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE sales.stock
    SET quantity = quantity - qty
    WHERE product_id = prod_id;
END;
$$;
```

Шаг 3. Триггер-функция

```
    CREATE FUNCTION trigger_update_stock()
RETURNS TRIGGER AS $$
BEGIN
  -- вызываем процедуру при добавлении нового заказа
  CALL update_stock(NEW.product_id, NEW.quantity);
  RETURN NEW;  -- обязательно для AFTER INSERT
END;
$$ LANGUAGE plpgsql;
```

Триггерная функция vs обычная функция

- Обычная функция (RETURNS NUMERIC или RETURNS INT)
  - Должна вернуть значение типа, указанного в RETURNS.
  - Использует RETURN для возврата этого значения.
- Триггерная функция (RETURNS TRIGGER)
  - Возвращает специальное значение: NEW или OLD.
    - NEW — новая строка после вставки/обновления
    - OLD — старая строка при обновлении/удалении
  - СУБД использует это возвращаемое значение для завершения операции на строке.

Шаг 4. Создание триггера

```
    CREATE TRIGGER after_order_insert
AFTER INSERT ON sales.orders
FOR EACH ROW
EXECUTE FUNCTION trigger_update_stock();
```

Как это работает:

1. Вставляем заказ:

INSERT INTO sales.orders(employee_id, product_id, quantity)

VALUES (1, 101, 5);

1. PostgreSQL **автоматически вызывает триггер** after_order_insert
2. Триггер выполняет функцию trigger_update_stock()
3. Функция внутри вызывает процедуру update_stock, которая обновляет таблицу stock

✅ Всё происходит автоматически, без явного вызова функции/процедуры пользователем.

Типы триггеров

Тип	Описание
BEFORE	Срабатывает до действия (можно изменить или отменить действие)
AFTER	Срабатывает после действия (для логов, уведомлений, обновления других таблиц)
INSTEAD OF	Для представлений, заменяет действие

Плюсы использования триггеров

1. **Автоматизация** — не нужно писать отдельные запросы каждый раз
2. **Целостность данных** — гарантирует выполнение правил (например, обновление запасов)
3. **Логирование и аудит** — можно автоматически записывать изменения в отдельную таблицу
4. **Сложная бизнес-логика** — объединение нескольких операций в одном событии

Cursor

Cursor –
это специальный объект базы данных, который позволяет построчно обрабатывать результат выполнения SQL-запроса.
Обычно SQL работает с целыми наборами данных ("set-based" подход), но иногда возникает ситуация, когда нужно пройтись по строкам результата по одной и выполнить какие-то действия для каждой строки. Вот тут и используется курсор.

Когда нужен курсор:

- нужно пошагово обрабатывать строки результата запроса;

- требуется выполнять разные действия для каждой строки (например, обновления в зависимости от условий);
- невозможно реализовать задачу простым **set-based** запросом (JOIN, UPDATE с подзапросами и т.п.).

Основные этапы работы с курсором

Объявление курсора – определяется SQL-запрос, результаты которого будут построчно обрабатываться.

```
DECLARE my_cursor CURSOR FOR

SELECT id, name FROM employees;
```

Открытие курсора – выполняется запрос и формируется набор данных.

```
OPEN my_cursor;
```

Чтение строк – построчно извлекаются данные.

```
FETCH NEXT FROM my_cursor INTO @id, @name;
```

Обработка данных – выполняются нужные действия для каждой строки.

Закрытие курсора – освобождаются ресурсы.

```
CLOSE my_cursor;

DEALLOCATE my_cursor;
```

Допустим, нужно пройтись по всем сотрудникам и вывести их имена (упрощённый пример):

```
DECLARE @id INT, @name NVARCHAR(100);

DECLARE emp_cursor CURSOR FOR
SELECT id, name FROM employees;

OPEN emp_cursor;

FETCH NEXT FROM emp_cursor INTO @id, @name;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Сотрудник: ' + @name;

    FETCH NEXT FROM emp_cursor INTO @id, @name;
END;

CLOSE emp_cursor;
DEALLOCATE emp_cursor;
```

Пример использования курсора

Важно ⚠️

- **Курсоры медленные** по сравнению с обычными SQL-операциями, потому что обрабатывают строки по одной.
- Если задачу можно решить "наборным" способом (через JOIN, UPDATE, CASE, GROUP BY и т.п.), **лучше избегать курсоров**.
- Курсоры чаще используют в хранимых процедурах или сложной бизнес-логике.

📌 Когда реально возникает потребность в построчном анализе?

1. **Сложная бизнес-логика на уровне БД**
   Например: нужно пройти по каждой транзакции и для неё выполнить несколько разных действий (создать запись в логе, обновить баланс, проверить лимиты, возможно вызвать другую процедуру). Это уже не просто "сумма", а серия разных шагов.
2. **Динамическое выполнение SQL**
   Когда строки содержат условия/данные, из которых нужно собрать и выполнить разные SQL-запросы.
   Например: в таблице хранятся названия таблиц или колонок, и для каждой строки нужно построить отдельный SELECT.
3. **Обработка данных "по порядку"**
   Есть задачи, где результат следующей строки зависит от обработки предыдущей.
   Пример: считать накопительный баланс (running total) не с помощью оконных функций, а вручную.
   Или расчёт процентов/штрафов по каждой транзакции в зависимости от остатка на счёте.
4. **Интеграции и сторонние системы**
   Иногда хранимые процедуры с курсорами используют для итеративной передачи данных во внешние системы (например, в ERP или API), потому что те принимают данные "по одному объекту", а не пакетами.
5. **Миграции и массовые обновления с разной логикой**
   Если строки сильно отличаются по условиям обработки, проще пройти по ним курсором, чем лепить гигантский CASE с десятками условий.

📌 Когда курсор не нужен

- **подсчёт балансов (SUM ... CASE)**
- **массовые обновления (UPDATE ... JOIN)**
- **выборка данных с условиями (SELECT ... WHERE)**
- **агрегации (GROUP BY, оконные функции ROW_NUMBER, SUM() OVER(...))**

⚡ Итог:
Курсор — это "последний вариант", когда другого способа нет или слишком сложно. В 80–90% задач его заменяют простые SQL-запросы.

Explain/Analyze Execution Plan (join types, index scan/table scan)

Explain/Analyze Execution Plan –
это «пошаговый рецепт», как СУБД (Postgres, MySQL, SQL Server и т. д.) собирается выполнять твой SQL-запрос:

- какие таблицы она читает,
- как именно читает (по индексу или полностью),
- как соединяет таблицы (join),
- где сортирует или фильтрует.

🔹 Типы сканов (чтения таблицы)

1. **Table/Seq Scan (последовательное чтение)**
   - Пробегает всю таблицу построчно.
   - Быстро для маленьких таблиц или когда надо почти все строки.
   - Плохо для больших таблиц, если фильтр очень узкий.
2. **Index Scan (чтение по индексу)**
   - Идёт по индексу и достаёт нужные строки.
   - Хорошо, когда условие выборки очень избирательное (WHERE id = 123).
3. **Index Only Scan**
   - Если все нужные колонки есть в индексе → таблицу вообще не трогает.
   - Обычно самый быстрый вариант.

🔹 Типы join (соединений таблиц)

1. **Nested Loop Join (вложенные циклы)**
   - Берёт строку из первой таблицы → ищет подходящие во второй.
   - Хорошо для маленьких таблиц или когда есть индекс по условию.
   - Очень медленно, если обе таблицы большие и индекса нет.
2. **Hash Join**
   - Строит хеш-таблицу из маленькой таблицы и по ней ищет совпадения из большой.
   - Отлично работает для равенств (ON a.id = b.id).
3. **Merge Join**
   - Обе таблицы сортируются по ключу → потом проходят «синхронно» сверху вниз.
   - Хорошо для больших отсортированных наборов.

🔹 Что ещё можно встретить

- **Sort** – явная сортировка (дорого, если нет индекса).
- **Aggregate** – подсчёт SUM, COUNT и т. д.
- **Filter** – применение условий WHERE.
- **Limit** – ограничение количества строк.

🔹 Как анализировать план

1. Смотри, **сколько строк предполагалось** (Estimated rows) и **сколько реально вернулось** (Actual rows).
   - Если сильно отличаются → статистика устарела, нужно ANALYZE (Postgres) или обновить статистику в СУБД.
2. Если видишь **Seq Scan на огромной таблице** → возможно, нужен индекс.
3. Если видишь **Nested Loop с большими таблицами** → это часто проблема → лучше Hash/Merge Join.
4. Самые «дорогие» шаги (по времени/стоимости) обычно внизу → там и ищем узкие места.

```
EXPLAIN ANALYZE

SELECT c.name, o.total

FROM customers c

JOIN orders o ON c.id = o.customer_id

WHERE c.country = 'US';
```

Пример

```
План может быть такой:

Hash Join

 -> Seq Scan on orders

 -> Hash

    -> Index Scan on customers (idx_country)
```

Расшифровка:

- Index Scan on customers → быстро нашёл клиентов по стране.
- Seq Scan on orders → читает всю таблицу заказов (может стоить индекс по customer_id).
- Hash Join → соединяет по ID через хеш.

Разница между EXPLAIN и EXPLAIN ANALYZE

🔹 EXPLAIN

- **Не выполняет сам запрос.**
- **Показывает план, который построил оптимизатор:**
  - **какие типы сканов и соединений будут использоваться,**
  - **порядок выполнения,**
  - **оценки (cost, предполагаемое количество строк).**
- **Это «прогноз» работы, без фактического исполнения.**

Используй EXPLAIN → когда хочешь просто увидеть стратегию выполнения, без изменения данных.

🔹 EXPLAIN ANALYZE

- **Реально выполняет запрос**.
- Показывает не только план, но и:
  - **фактическое время** выполнения каждого шага (actual time).
  - **реальное количество строк** (actual rows).
  - сколько раз выполнялась операция (loops).

Используй EXPLAIN ANALYZE → когда нужно проверить, насколько оценки оптимизатора совпадают с реальностью и где «узкие места» в запросе.

CTE (Recursive CTE)

CTE –

Это временный результат (подзапрос), которому даётся имя.

Он существует только в рамках одного запроса, но выглядит как «виртуальная таблица».

```
WITH big_orders AS (
   SELECT customer_id, SUM(amount) AS total
   FROM orders
   GROUP BY customer_id
   HAVING SUM(amount) > 1000
)
SELECT c.name, b.total
FROM customers c
JOIN big_orders b ON c.id = b.customer_id;
```

пример

📌 Применения:

- чтобы упростить длинные запросы (разбить на шаги),
- использовать один и тот же подзапрос несколько раз,
- сделать код более читаемым.

Рекурсивный CTE —

это особый CTE, который **ссылается сам на себя**, чтобы строить результаты **итеративно**.
Идеально подходит для:

- деревьев (категории, папки),
- иерархий (сотрудники и начальники),
- графов (например, маршруты).

```
WITH RECURSIVE cte_name AS (
    -- Базовый запрос (anchor member)
    SELECT ...

    UNION ALL

    -- Рекурсивный запрос (recursive member)
    SELECT ...
    FROM cte_name
    JOIN ...
)
SELECT * FROM cte_name;
```

🔹 Автоматическое повторение

- СУБД сама «запускает» рекурсивную часть многократно.
- Каждая итерация использует **только что добавленные строки** из CTE.
- Как только итерация **не возвращает новых строк**, рекурсия останавливается.

В нашем примере:

1. **Итерация 1:**
   - Базовый уровень: John (id=1)
   - Рекурсивная часть ищет сотрудников с manager_id = 1 → Alice (2), Bob (3)
2. **Итерация 2:**
   - Берём новые строки из предыдущей итерации: Alice, Bob
   - Рекурсивная часть ищет сотрудников с manager_id = 2 или 3 → Charlie (4), David (5), Eva (6)
3. **Итерация 3:**
   - Новые строки: Charlie, David, Eva
   - Рекурсивная часть ищет сотрудников с manager_id = 4,5,6 → новых нет
   - Рекурсия останавливается

🔹 Итог

рекурсивная часть повторяется автоматически.

- Не нужно ничего писать вручную, СУБД сама выполняет итерации.
- СТРОКИ, найденные на каждом шаге, добавляются в результирующий CTE.
- Рекурсия заканчивается, когда новые строки не появляются.

DELETE / TRUNCATE

DELETE → когда нужно удалить **конкретные строки** или использовать триггеры.

DELETE FROM table_name

WHERE condition;

Особенности:

1. Можно удалить **выборочно** (по условию WHERE).
2. Каждая удалённая строка фиксируется в журнале транзакций (лог), что позволяет откатить транзакцию (ROLLBACK).
3. Триггеры ON DELETE срабатывают.
4. Обычно **медленнее**, особенно на больших таблицах, из-за логирования каждой строки.

TRUNCATE → когда нужно **очистить таблицу полностью**, быстро и эффективно.

TRUNCATE TABLE table_name;

Особенности:

1. **Удаляет все строки сразу** — без возможности указать условие.
2. Обычно **очень быстро**, потому что:
   - не логируется удаление каждой строки отдельно,
   - может сбрасывать автогенерируемые счётчики (SERIAL/IDENTITY).
3. В большинстве СУБД триггеры **не срабатывают**.
4. В PostgreSQL и некоторых других СУБД TRUNCATE нельзя откатить, если не в транзакции.

WHERE / HAVING

🔹 1. WHERE

Что делает: фильтрует строки **до группировки** (до GROUP BY) в запросе.

Синтаксис:

SELECT *

FROM table_name

WHERE condition;

Примеры:

-- выбрать всех сотрудников с зарплатой > 50000

SELECT *

FROM employees

WHERE salary > 50000;

Особенности:

- Работает на уровне **отдельных строк**.
- Нельзя использовать агрегатные функции (SUM, COUNT, AVG) напрямую, например так не получится:

WHERE SUM(salary) > 100000 -- ❌ ошибка

🔹 2. HAVING

Что делает: фильтрует **уже агрегированные группы** (после GROUP BY).

Синтаксис:

```
SELECT column, AGG_FUNC(column)

FROM table_name

GROUP BY column

HAVING AGG_FUNC(column) condition;
```

Примеры:

```
-- выбрать отделы, где суммарная зарплата > 200000

SELECT department_id, SUM(salary) AS total_salary

FROM employees

GROUP BY department_id

HAVING SUM(salary) > 200000;
```

Особенности:

- Работает **только с агрегатными данными** или после группировки.
- Можно использовать агрегатные функции (SUM, COUNT, AVG, MAX, MIN).

🔹 Важное отличие

Свойство	WHERE	HAVING
Когда фильтрует	До группировки (по строкам)	После группировки (по группам)
Можно агрегаты?	Нет	Да
Пример	WHERE salary > 50000	HAVING SUM(salary) > 200000

🔹 Комбинирование WHERE и HAVING

Часто используют вместе:

```
SELECT department_id, SUM(salary) AS total_salary

FROM employees

WHERE hire_date > '2020-01-01'   -- фильтруем строки до группировки

GROUP BY department_id

HAVING SUM(salary) > 200000;    -- фильтруем группы после агрегирования
```

- Сначала WHERE → оставляет только нужные строки.
- Потом GROUP BY → формирует группы.
- Потом HAVING → оставляет только группы, удовлетворяющие условию.

MERGE / UPSERT

🔹 1. UPSERT

UPSERT — это комбинация **INSERT + UPDATE**:

- Если строка с таким ключом **ещё не существует**, вставляем её (INSERT).
- Если строка уже есть, обновляем её (UPDATE).

Пример (PostgreSQL / MySQL 8+):

```
INSERT INTO employees (id, name, salary)

VALUES (1, 'John', 50000)

ON CONFLICT (id)         -- ключ, по которому проверяем существование

DO UPDATE SET

  name = EXCLUDED.name,

  salary = EXCLUDED.salary;
```

Что происходит:

- Если id = 1 ещё нет → создаётся новая строка
- Если id = 1 есть → обновляется имя и зарплата

🔹 2. MERGE

MERGE — более универсальный оператор для **сравнения двух таблиц** и синхронизации данных.

- Можно **вставлять**, **обновлять** и даже **удалять** данные в одной операции.
- Чаще используется в SQL Server, Oracle, DB2.

Синтаксис:

```
MERGE INTO target_table t

USING source_table s

ON t.id = s.id

WHEN MATCHED THEN

  UPDATE SET t.name = s.name, t.salary = s.salary

WHEN NOT MATCHED THEN

  INSERT (id, name, salary) VALUES (s.id, s.name, s.salary);
```

Что происходит:

- ON t.id = s.id — сравниваем строки по ключу
- WHEN MATCHED — если совпало, обновляем
- WHEN NOT MATCHED — если нет совпадения, вставляем

🔹 Основные отличия UPSERT vs MERGE

Свойство	UPSERT	MERGE
Основная цель	Вставить или обновить одну таблицу	Синхронизировать две таблицы
Может удалять?	❌	✅ (можно с WHEN NOT MATCHED BY SOURCE)
Синтаксис	INSERT ... ON CONFLICT / ON DUPLICATE KEY UPDATE	MERGE INTO ... USING ...
Подходит для	Обновление по уникальному ключу	Комплексная синхронизация (ETL, DW)

🔹 Когда использовать

- **UPSERT** → удобно для одной таблицы, когда есть уникальный ключ.
- **MERGE** → удобно для сравнения двух таблиц и синхронизации данных (например, данные из staging → main).

PIVOT, CROSSTAB

🔹 1. PIVOT

PIVOT — оператор SQL для **транспонирования строк в столбцы**.

- Чаще всего используется в **SQL Server** и Oracle.
- Преобразует **категории из строк** в **столбцы**, обычно с агрегатной функцией.

Пример (SQL Server)

Есть таблица sales:

year	product	amount
2023	A	100
2023	B	200
2024	A	150
2024	B	250

Сделаем PIVOT, чтобы получить сумму продаж по продуктам в столбцах:

```
SELECT *
FROM sales
PIVOT (
    SUM(amount)
    FOR product IN ([A], [B])
) AS p;
```

Результат:

year	A	B
2023	100	200
2024	150	250

Особенности PIVOT:

- Нужно заранее знать **значения, которые будут столбцами** (IN ([A], [B])).
- Используется агрегатная функция (SUM, COUNT, MAX и т.д.).

🔹 2. CROSSTAB

CROSSTAB — аналог PIVOT в **PostgreSQL**, доступный через модуль tablefunc.

- Создаёт **кросс-таблицу**, преобразуя строки в столбцы.
- Результат похож на PIVOT, но синтаксис немного другой.

Пример (PostgreSQL)

```
-- Нужно сначала подключить модуль

CREATE EXTENSION IF NOT EXISTS tablefunc;


SELECT *

FROM crosstab(

    'SELECT year, product, amount FROM sales ORDER BY 1,2',

    'SELECT DISTINCT product FROM sales ORDER BY 1'

) AS ct(year int, A int, B int);
```

Результат такой же:

year	A	B
2023	100	200
2024	150	250

Особенности CROSSTAB:

- Первый аргумент — SQL-запрос с исходными данными.
- Второй аргумент — список категорий (будущие столбцы).
- Гибкость аналогична PIVOT, но синтаксис PostgreSQL специфичен.

🔹 Отличия PIVOT vs CROSSTAB

Свойство	PIVOT (SQL Server/Oracle)	CROSSTAB (PostgreSQL)
СУБД	SQL Server, Oracle	PostgreSQL (tablefunc)
Синтаксис	PIVOT (AGG_FUNC FOR col IN (...))	crosstab(sql1, sql2)
Нужно знать значения для столбцов	Да	Да
Используется агрегатная функция	Да	Да
Гибкость	Средняя	Высокая, можно писать SQL-запросы для динамики

🔹 Когда использовать

- **PIVOT/CROSSTAB** → когда нужно превратить строки с категориями в столбцы и агрегировать значения.
- Полезно для: отчётов, BI, сводных таблиц.

set theory:

🌐 Операции над множествами, диаграммы Эйлера-Венна

relation algebra

Что такое реляционная алгебра

Реляционная алгебра — это формальный язык для работы с **реляционными таблицами (отношениями)**.

- Определяет **операции над таблицами** (отношениями).
- Операции возвращают **новую таблицу** (новое отношение).
- Основу составляют **операции выборки, объединения, проекции, соединений, разности** и др.

💡 SQL во многом реализует реляционную алгебру на практике.

Основные операции реляционной алгебры

2.1. Проекция (π)

- Выбирает **определённые столбцы** из таблицы.
- Аналог SELECT column1, column2 в SQL.

Пример:

π_name,salary (Employees)

→ таблица с только столбцами name и salary.

2.2. Селекция (σ)

- Выбирает **строки по условию**.
- Аналог WHERE в SQL.

Пример:

σ_salary > 5000 (Employees)

→ все сотрудники с зарплатой > 5000.

2.3. Объединение (∪)

- Объединяет **две таблицы с одинаковыми столбцами**.
- Убирает дубликаты.
- Аналог UNION в SQL.

2.4. Разность (−)

- Возвращает строки, которые **есть в первой таблице, но нет во второй**.
- Аналог EXCEPT в SQL.

2.5. Декартово произведение (×)

- Каждая строка первой таблицы соединяется со всеми строками второй таблицы.
- Аналог CROSS JOIN в SQL.

2.6. Соединение (Join, ⋈)

- Комбинирует строки двух таблиц **по условию совпадения атрибутов**.
- В SQL: INNER JOIN, LEFT JOIN, RIGHT JOIN.

Пример:

Employees ⋈ Employees.department_id = Departments.id Departments

2.7. Переименование (ρ)

- Меняет имя таблицы или столбца для удобства.
- SQL использует AS для этого:

SELECT name AS employee_name FROM Employees;

Дополнительные операции

- **INTERSECTION (∩)** — строки, которые есть в обеих таблицах (INTERSECT).
- **DIVISION (÷)** — используется для запросов «для всех» (например, найти студентов, сдавших все экзамены).

Связь с SQL

Реляционная алгебра	SQL пример
σ (Selection)	SELECT * FROM Employees WHERE salary > 5000
π (Projection)	SELECT name, salary FROM Employees
∪ (Union)	SELECT ... UNION SELECT ...
− (Difference)	SELECT ... EXCEPT SELECT ...
× (Cartesian)	SELECT * FROM A CROSS JOIN B
⋈ (Join)	SELECT * FROM A INNER JOIN B ON ...

💡 **Итог:**
Реляционная алгебра — это **теоретический язык операций над таблицами**, а SQL — его практическая реализация.
Она даёт **структуру и правила**, по которым работают реляционные базы.