SELECT столбцы
FROM таблица
WHERE условия
GROUP BY столбцы_группировки
HAVING условия_для_групп
ORDER BY сортировка

WHERE — фильтрация строк до группировки
Назначение: Отбирает строки из таблицы, которые удовлетворяют указанным условиям.
Особенности:
Работает с отдельными строками, а не с группами.
Не может использовать агрегатные функции (SUM, COUNT, AVG), так как они применяются после группировки.

GROUP BY — это оператор SQL, который группирует строки с одинаковыми значениями в указанных столбцах. После группировки можно применять агрегатные функции (например, COUNT, SUM, AVG, MAX, MIN) для анализа данных внутри каждой группы. Можно группировать данные по комбинации столбцов:
SELECT department, position, COUNT(*) as count
FROM employees
GROUP BY department, position;

В SELECT можно указывать только:
столбцы из GROUP BY,
агрегатные функции (например, SUM, COUNT).
Если добавить столбец, не входящий в GROUP BY, возникнет ошибка.

CUBE расширяет обычную группировку, создавая все возможные комбинации группируемых столбцов, включая итоги. NULL в столбцах группировки означает, что итог рассчитан без учета этого столбца.

ROLLUP создает иерархические итоги (например, для иерархии "год → месяц → день"). ROLLUP (a, b, c) → группы: (a,b,c), (a,b), (a), (). NULL в столбце означает, что это итоговая строка для предыдущего уровня иерархии.
GROUPING(column) возвращает:
0, если столбец участвует в текущей группировке,
1, если столбец не участвует (итоговая строка).

Электроника	Смартфоны	500
Электроника	Ноутбуки	1000
Одежда	Мужская	300
Одежда	Женская	400

Запрос:

```sql
SELECT
    category,
    subcategory,
    SUM(revenue) AS total_revenue
FROM
    products
GROUP BY ROLLUP (category, subcategory);
```

Результат:

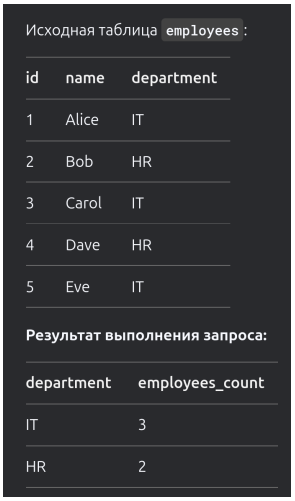
category	subcategory	total_revenue	
Электроника	Смартфоны	500	-- Детализация
Электроника	Ноутбуки	1000	-- Детализация
Электроника	NULL	1500	-- Итог по категории "Электроника"
Одежда	Мужская	300	-- Детализация
Одежда	Женская	400	-- Детализация
Одежда	NULL	700	-- Итог по категории "Одежда"
NULL	NULL	2200	-- Общий итог

Ключевое слово DISTINCT в SQL используется для возврата уникальных значений из столбцов в результатах запроса. Оно помогает исключить дубликаты, оставив только неповторяющиеся строки. DISTINCT применяется ко всем столбцам, перечисленным после SELECT. Возвращает только уникальные комбинации значений этих столбцов.

HAVING — фильтрация групп после группировки
Назначение: Отбирает группы, которые удовлетворяют условию.
Особенности:

Работает с агрегированными данными (например, COUNT(*), AVG(salary)).
Всегда используется после GROUP BY.
SELECT department, AVG(salary) as avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 60000;

Комбинация GROUP BY и агрегатных функций в SQL позволяет анализировать данные на уровне групп, а не отдельных строк. Агрегатные функции (COUNT, SUM, AVG, MAX, MIN) применяются к каждой группе отдельно.
SELECT department, COUNT(*) as employees_count
FROM employees
GROUP BY department;



Исходная таблица `employees`:

id	name	department
1	Alice	IT
2	Bob	HR
3	Carol	IT
4	Dave	HR
5	Eve	IT

Результат выполнения запроса:

department	employees_count
IT	3
HR	2

Агрегатные функции в SQL — это функции, которые выполняют вычисления над набором значений и возвращают единственное значение. Они используются для анализа данных, например, для подсчета количества записей, вычисления суммы, среднего значения, поиска максимума или минимума. Агрегатные функции нельзя использовать в WHERE. Для фильтрации групп применяйте HAVING.
SELECT department, AVG(salary) as avg_salary, COUNT(*) as employees_count
FROM employees
WHERE hire_date >= '2020-01-01' – Сотрудники, принятые после 2020
GROUP BY department – Группировка по отделам
HAVING AVG(salary) > 60000 – Отделы с средней зарплатой > 60k
ORDER BY avg_salary DESC; – Сортировка от высокой к низкой зарплате

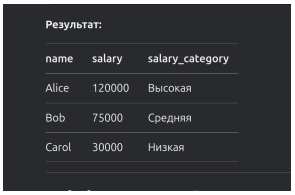
Если что-то сравнивается с null, то результат - null.

– Найти имена, начинающиеся на "Ан"
SELECT * FROM users WHERE name LIKE 'Ан%'; (LIKE - с учётом регистра, ILIKE - нет)

SIMILAR TO - для сравнения с регулярками POSIX.

Конструкция CASE WHEN THEN в SQL позволяет выполнять условные проверки и возвращать различные значения в зависимости от выполнения условий. Это аналог оператора if-else в других языках программирования.
CASE
WHEN условие1 THEN результат1
WHEN условие2 THEN результат2
...
ELSE результат_по_умолчанию
END

SELECT name, salary,
CASE
WHEN salary > 100000 THEN 'Высокая'
WHEN salary BETWEEN 50000 AND 100000 THEN 'Средняя'
ELSE 'Низкая'
END AS salary_category
FROM employees;



CAST(выражение AS целевой_тип) - преобразование типов
выражение::целевой_тип - тоже (чисто для PostgreSQL)
SELECT '3.14'::FLOAT; – Результат: 3.14

Запросы с несколькими таблицами:
1. СУБД формирует строки декартова произведения таблиц, перечисленных во фразе FROM.
2. СУБД проверяет, удовлетворяют ли данные в сформированной строке условиям из фразы WHERE.
3. Если данные в строке удовлетворяют условию из фразы WHERE, то СУБД включает в ответ на запрос те поля строки, которые соответствуют столбцам, перечисленным во фразе SELECT.

INNER JOIN
Описание:
Объединяет строки из двух таблиц только при наличии совпадений по условию. Соединения (JOIN) часто более читаемы и лучше оптимизируются СУБД, особенно для эквисоединений.
SELECT users.name, orders.amount
FROM users
INNER JOIN orders ON users.id = orders.user_id;
Результат:
Только пользователи с заказами.

LEFT OUTER JOIN (или LEFT JOIN)
Описание:
Возвращает все строки из левой таблицы и совпадающие строки из правой. Если совпадений нет, в правой части — NULL.
SELECT users.name, orders.amount
FROM users
LEFT JOIN orders ON users.id = orders.user_id;
Результат:
Все пользователи, включая тех, у кого нет заказов (в amount будет NULL). Если в таблице groups есть несколько строк с одинаковым id, соответствующим id юзера, то в итоговой выборке каждому юзеру будет сопоставлена каждая из найденных групп. Это приведет к дублированию строк студента для каждого совпадения в users.

RIGHT OUTER JOIN (или RIGHT JOIN)
Возвращает все строки из правой таблицы и совпадающие строки из левой. Если совпадений нет, в левой части — NULL.
SELECT users.name, orders.amount
FROM users
RIGHT JOIN orders ON users.id = orders.user_id;
Результат:
Все заказы, включая те, у которых нет пользователя (в name будет NULL).

FULL OUTER JOIN (или FULL JOIN)
Возвращает все строки из обеих таблиц. Если совпадений нет, недостающие части заполняются NULL (комбинация LEFT JOIN и RIGHT JOIN).
SELECT users.name, orders.amount
FROM users
FULL JOIN orders ON users.id = orders.user_id;
Результат:
Все пользователи и все заказы, включая несовпадающие записи.

CROSS JOIN
Создает декартово произведение строк из двух таблиц (каждая строка из первой объединяется с каждой строкой из второй). Обычно не является желаемым результатом, так как создает много “бессмысленных” комбинаций строк и очень ресурсоёмко.

Вложенный подзапрос (subquery) — предложение SELECT, которое заключено в круглые скобки и вложено в WHERE/HAVIN часть другого SQL- предложения. Обрабатываются "снизу вверх": первым обрабатывается вложенный подзапрос самого нижнего уровня.

```
SELECT Surname
FROM STUDENT
WHERE CityName IN (
    SELECT City FROM CITIES
    WHERE CITIES.Country = 'Россия'
);
```

Получаем список городов в России, используем их для выполнения внешнего запроса

Аналогично:
SELECT Surname
FROM STUDENT
JOIN CITIES ON CityName = City
WHERE CITIES.Country = 'Россия'

Сначала выполняется JOIN (объединение таблиц).
Затем WHERE (фильтрация строк).
Потом GROUP BY (группировка).
После этого HAVING (фильтрация групп).
И только в конце — SELECT.

а) Некоррелированный подзапрос
Не зависит от данных внешнего запроса.
Выполняется один раз перед внешним запросом.

б) Зависит от данных внешнего запроса.
Выполняется для каждой строки внешнего запроса.
SELECT name
FROM employees e
WHERE salary > (SELECT AVG(salary)
FROM employees
WHERE department = e.department);

IN (аналогично ANY, SOME)
Подзапрос должен возвращать ровно один столбец.
Результат выражения сравнивается с каждым значением, возвращённым подзапросом.
Результатом выражения IN будет «true», если значение выражения соответствует хотя бы одному значению, которое вернул подзапрос;
Если выражение: NULL или соответствие в правой части не найдено, и есть хотя бы одно NULL-значение в результате подзапроса, то результат - NULL.

```
SELECT *
    FROM STUDENT
        WHERE StudentId = ANY
            ( SELECT StID FROM EXAMS );
```

EXISTS
EXISTS принимает оператор SELECT (подзапрос)
Если подзапрос возвращает хотя бы одну строку, то результатом EXISTS будет «true», а если не возвращает ни одной — «false».
Если подзапрос возвращает NULL, результат - «true».
Подзапрос SELECT 1 — это технический прием, так как важен сам факт наличия строк, а не их содержимое. Если студент имеет несколько экзаменов с оценкой <60, он все равно выведется один раз (дубликаты фамилий не создаются, если они не дублируются в таблице STUDENT). Использование EXISTS эффективнее, чем IN, так как проверка останавливается при нахождении первой подходящей строки.

```
SELECT Surname
FROM STUDENT
WHERE EXISTS (
      SELECT 1 FROM EXAM WHERE
        STUDENT.StudentID = EXAM.StID
        AND EXAM.Result < 60
);
```

ALL (эквивалент NOT IN, <> - не равно)
Подзапрос должен возвращать ровно один столбец
Значение выражения сравнивается со значением в каждой строке результата подзапроса с помощью оператора, который должен возвращать логическое значение.
Результатом ALL будет «true», если условие истинно для всех строк (ANY - для какой-то) (и когда подзапрос не возвращает строк), и «false», если находятся строки, для которых оно ложно.
Результат будет NULL, если сравнение не возвращает «false» ни для одной из строк, но как минимум для одной результат сравнения при применении оператора NULL.
SELECT Name, Salary
FROM Employees
WHERE Salary > ALL (
SELECT Salary
FROM Employees
WHERE Department = 'IT'
);

TRUNCATE() или TRUNC() — отбрасывание знаков: обрезает число до указанного количества знаков после запятой (без округления, FORMAT() — с округлением).
numeric - числа с заданной точностью

Представления (VIEWs) — это виртуальные таблицы (по сути это именованны подзапрос), которые хранят результат выполнения запроса. Они ведут себя как обычные таблицы, но данные в них не хранятся физически, а формируются динамически при каждом обращении. Вместо написания сложного SQL-запроса каждый раз можно создать представление и обращаться к нему, как к обычной таблице. Можно ограничить доступ к определённым данным (по сути через представления можно давать ограниченный доступ к таблице кому-то, давая её 'копию'), создавая представления, которые показывают только нужные столбцы или строки.

```
CREATE VIEW VIEW_NAME
[ ( ColumnName [, ...] ) ]
AS подзапрос
...
```

CREATE VIEW active_users AS
SELECT id, name, email FROM users WHERE status = 'active';
SELECT * FROM active_users;

```
CREATE VIEW PICTStudents2 AS
    (PICTId, pSurname) AS
    SELECT StudentID, Surname FROM STUDENT
    WHERE GroupID IN (
        SELECT GroupID FROM GROUP
         WHERE GroupName LIKE 'P3%'
    );


SELECT PICTid FROM PICTStudents2;
```

Материализованные представления хранят результат запроса в виде таблицы и могут быть обновлены вручную:
CREATE MATERIALIZED VIEW user_stats AS

SELECT status, COUNT(*) FROM users GROUP BY status;
Обновление:
REFRESH MATERIALIZED VIEW user_stats;
Материализованное представление быстрее при чтении, но не обновляется автоматически (нужно выполнять REFRESH). Обычные представления нельзя изменять (INSERT, UPDATE, DELETE), но с INSTEAD OF триггерами можно сделать их редактируемыми.

В PostgreSQL представления можно изменять с помощью команды:
CREATE OR REPLACE VIEW имя_представления AS новый_запрос;
Этот метод позволяет обновлять представление без удаления и повторного создания, что удобно при обновлении логики запросов. Новое представление полностью заменит старое. Если на представление ссылаются другие объекты (например, другие представления), то замена может вызвать ошибки. Материализованные представления нельзя заменить через CREATE OR REPLACE.

PL/pgSQL (Procedural Language/PostgreSQL) — это процедурный язык программирования, встроенный в PostgreSQL. Он позволяет писать функции, триггеры и хранимые процедуры, расширяя возможности SQL логикой программирования. Код PL/pgSQL обычно пишется внутри функций или анонимных блоков. Общий шаблон:
DECLARE
– Объявление переменных (опционально)
BEGIN
– Основной код
– SQL-запросы, циклы, условия, обработка ошибок
EXCEPTION
– Обработка ошибок (опционально)
END;

Последовательности (Sequences) в SQL — это объекты базы данных, предназначенные для генерации уникальных числовых значений. Они часто используются для создания автоинкрементных идентификаторов (например, первичных ключей) или других последовательностей чисел.
CREATE SEQUENCE sequence_name
INCREMENT BY increment – шаг приращения (по умолчанию 1)
START WITH start – начальное значение (по умолчанию 1)
MINVALUE min – минимальное значение
MAXVALUE max – максимальное значение
CYCLE | NO CYCLE – перезапуск при достижении максимума (без цикла - ошибка)
CACHE cache; – кэширование значений для повышения производительности

Получить следующее значение:
SELECT nextval('seq');
Текущее значение:
SELECT currval('seq');
Сбросить последовательность:
ALTER SEQUENCE user_id_seq RESTART WITH 50;
Последовательности часто используются для автоинкрементных полей:
CREATE TABLE users (
id INT PRIMARY KEY DEFAULT nextval('user_id_seq'),
name TEXT
);

smallserial - serial на основе smallint (2 байта)
serial - на основе интов (4 байта)
bigserial - на основе бигинт (8)

CREATE SEQUENCE user_id_seq
START WITH 1000
INCREMENT BY 1
MINVALUE 1000
MAXVALUE 9999
CYCLE
CREATE TABLE users (
user_id INT DEFAULT nextval('user_id_seq') PRIMARY KEY,
name VARCHAR(50) NOT NULL
);

INSERT INTO users (name) VALUES ('Alice'); – user_id = 1000

INSERT INTO users (name) VALUES ('Bob'); – user_id = 1001

Функция
Возвращает одно значение (число, строку, таблицу и т.д.) на основе входных параметров.
Используется в SQL-запросах
Процедура
Выполняет действия с данными (вставка, обновление, удаление) или бизнес-логику.
Не возвращает значение напрямую (может использовать OUT-параметры). Вызывается через CALL.
Транзакция
Обеспечивает атомарность группы операций: либо все выполняются, либо ни одна.
Начинается с BEGIN, завершается COMMIT или ROLLBACK.
Гарантирует целостность данных при сбоях.

USING (colum_list): Упрощенная запись для эквисоединения, когда столбцы, по которым идет соединение, имеют одинаковые имена в обеих таблицах. Столбцы из списка column_list включаются в результат только один раз.

Предполагая, что в обеих таблицах есть колонка gr_id
FROM students JOIN groups USING (gr_id)
Эквивалентно ON students.gr_id = groups.gr_id, но в SELECT будет только одна колонка gr_id.

NATURAL JOIN: Еще более короткая запись. Автоматически соединяет таблицы по всем столбцам с одинаковыми именами. Совпадающие столбцы включаются в результат только один раз.
FROM students NATURAL JOIN groups

Ограничение UNIQUE гарантирует, что все значения в столбце (или комбинации столбцов) уникальны.
CREATE TABLE orders (
order_id INT,
product_id INT,
UNIQUE (order_id, product_id) – Комбинация order_id и product_id должна быть уникальной
);

ALTER TABLE users
ADD CONSTRAINT unique_email UNIQUE (email);

ALTER TABLE users
DROP CONSTRAINT unique_email;

DELETE - удаляет строки из таблицы
DROP - полностью удаляет объект
TRUNCATE - очищает таблицу